



Algorithms

FOURTH EDITION

PART I

ROBERT SEDGEWICK | KEVIN WAYNE

Algorithms

FOURTH EDITION

PART I

This page intentionally left blank

Algorithms

FOURTH EDITION

PART I

Robert Sedgewick
and
Kevin Wayne

Princeton University

◆ Addison-Wesley

Upper Saddle River, NJ • Boston • Indianapolis • San Francisco
New York • Toronto • Montreal • London • Munich • Paris • Madrid
Capetown • Sydney • Tokyo • Singapore • Mexico City

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and the publisher was aware of a trademark claim, the designations have been printed with initial capital letters or in all capitals.

The authors and publisher have taken care in the preparation of this book, but make no expressed or implied warranty of any kind and assume no responsibility for errors or omissions. No liability is assumed for incidental or consequential damages in connection with or arising out of the use of the information or programs contained herein.

For information about buying this title in bulk quantities, or for special sales opportunities (which may include electronic versions; custom cover designs; and content particular to your business, training goals, marketing focus, or branding interests), please contact our corporate sales department at (800) 382-3419 or corpsales@pearsoned.com.

For government sales inquiries, please contact governmentsales@pearsoned.com.

For questions about sales outside the United States, please contact international@pearsoned.com.

Visit us on the Web: informit.com/aw

Copyright © 2014 Pearson Education, Inc.

All rights reserved. Printed in the United States of America. This publication is protected by copyright, and permission must be obtained from the publisher prior to any prohibited reproduction, storage in a retrieval system, or transmission in any form or by any means, electronic, mechanical, photocopying, recording, or likewise. To obtain permission to use material from this work, please submit a written request to Pearson Education, Inc., Permissions Department, One Lake Street, Upper Saddle River, New Jersey 07458, or you may fax your request to (201) 236-3290.

ISBN-13: 978-0-13-379869-2

ISBN-10: 0-13-379869-0

First digital release, February 2014

To Adam, Andrew, Brett, Robbie
and especially Linda

To Jackie and Alex

CONTENTS

Note: This is an online edition of Chapters 1 through 3 of Algorithms, Fourth Edition, which contains the content covered in our online course Algorithms, Part I.

Preface *ix*

1. Fundamentals **3**

1.1	Basic Programming Model	8
	Primitive data types • Loops and conditionals • Arrays • Static methods • Recursion • APIs • Strings • Input and output • Binary search	
1.2	Data Abstraction	64
	Objects • Abstract data types • Implementing ADTs • Designing ADTs	
1.3	Bags, Queues, and Stacks	120
	APIs • Arithmetic expression evaluation • Resizing arrays • Generics • Iterators • Linked lists	
1.4	Analysis of Algorithms	172
	Running time • Computational experiments • Tilde notation • Order-of-growth classifications • Amortized analysis • Memory usage	
1.5	Case Study: Union-Find	216
	Dynamic connectivity • Quick find • Quick union • Weighted quick union	

2. Sorting243

2.1	Elementary Sorts	244
	Rules of the game • Selection sort • Insertion sort • Shellsort	
2.2	Mergesort	270
	Abstract in-place merge • Top-down mergesort • Bottom-up mergesort • N lg N lower bound for sorting	
2.3	Quicksort	288
	In-place partitioning • Randomized quicksort • 3-way partitioning	
2.4	Priority Queues	308
	Priority queue API • Elementary implementations • Binary heap • Heapsort	
2.5	Applications	336
	Comparators • Stability • Median and order statistics	

3. Searching361

3.1	Symbol Tables	362
	Symbol table API • Ordered symbol table API • Dedup • Frequency counter • Sequential search • Binary search	
3.2	Binary Search Trees	396
	Basic implementation • Order-based methods • Deletion	
3.3	Balanced Search Trees	424
	2-3 search trees • Red-black BSTs • Deletion	
3.4	Hash Tables	458
	Hash functions • Separate chaining • Linear probing	
3.5	Applications	486
	Set data type • Whitelist and blacklist filters • Dictionary lookup • Inverted index • File indexing • Sparse matrix-vector multiplication	

Chapters 4 through 6, which correspond to our online course Algorithms, Part II, are available as Algorithms, Fourth Edition, Part II.

For more information, see <http://algs4.cs.princeton.edu>.

This page intentionally left blank

PREFACE

This book is intended to survey the most important computer algorithms in use today, and to teach fundamental techniques to the growing number of people in need of knowing them. It is intended for use as a textbook for a second course in computer science, after students have acquired basic programming skills and familiarity with computer systems. The book also may be useful for self-study or as a reference for people engaged in the development of computer systems or applications programs, since it contains implementations of useful algorithms and detailed information on performance characteristics and clients. The broad perspective taken makes the book an appropriate introduction to the field.

THE STUDY OF ALGORITHMS AND DATA STRUCTURES is fundamental to any computer-science curriculum, but it is not just for programmers and computer-science students. Everyone who uses a computer wants it to run faster or to solve larger problems. The algorithms in this book represent a body of knowledge developed over the last 50 years that has become indispensable. From N-body simulation problems in physics to genetic-sequencing problems in molecular biology, the basic methods described here have become essential in scientific research; from architectural modeling systems to aircraft simulation, they have become essential tools in engineering; and from database systems to internet search engines, they have become essential parts of modern software systems. And these are but a few examples—as the scope of computer applications continues to grow, so grows the impact of the basic methods covered here.

In Chapter 1, we develop our fundamental approach to studying algorithms, including coverage of data types for stacks, queues, and other low-level abstractions that we use throughout the book. In Chapters 2 and 3, we survey fundamental algorithms for sorting and searching; and in Chapters 4 and 5, we cover algorithms for processing graphs and strings. Chapter 6 is an overview placing the rest of the material in the book in a larger context.

Distinctive features The orientation of the book is to study algorithms likely to be of practical use. The book teaches a broad variety of algorithms and data structures and provides sufficient information about them that readers can confidently implement, debug, and put them to work in any computational environment. The approach involves:

Algorithms. Our descriptions of algorithms are based on complete implementations and on a discussion of the operations of these programs on a consistent set of examples. Instead of presenting pseudo-code, we work with real code, so that the programs can quickly be put to practical use. Our programs are written in Java, but in a style such that most of our code can be reused to develop implementations in other modern programming languages.

Data types. We use a modern programming style based on data abstraction, so that algorithms and their data structures are encapsulated together.

Applications. Each chapter has a detailed description of applications where the algorithms described play a critical role. These range from applications in physics and molecular biology, to engineering computers and systems, to familiar tasks such as data compression and searching on the web.

A scientific approach. We emphasize developing mathematical models for describing the performance of algorithms, using the models to develop hypotheses about performance, and then testing the hypotheses by running the algorithms in realistic contexts.

Breadth of coverage. We cover basic abstract data types, sorting algorithms, searching algorithms, graph processing, and string processing. We keep the material in algorithmic context, describing data structures, algorithm design paradigms, reduction, and problem-solving models. We cover classic methods that have been taught since the 1960s and new methods that have been invented in recent years.

Our primary goal is to introduce the most important algorithms in use today to as wide an audience as possible. These algorithms are generally ingenious creations that, remarkably, can each be expressed in just a dozen or two lines of code. As a group, they represent problem-solving power of amazing scope. They have enabled the construction of computational artifacts, the solution of scientific problems, and the development of commercial applications that would not have been feasible without them.

Booksite An important feature of the book is its relationship to the online booksite `algs4.cs.princeton.edu`. This site is freely available and contains an extensive amount of material about algorithms and data structures, for teachers, students, and practitioners, including:

An online synopsis. The text is summarized in the booksite to give it the same overall structure as the book, but linked so as to provide easy navigation through the material.

Full implementations. All code in the book is available on the booksite, in a form suitable for program development. Many other implementations are also available, including advanced implementations and improvements described in the book, answers to selected exercises, and client code for various applications. The emphasis is on testing algorithms in the context of meaningful applications.

Exercises and answers. The booksite expands on the exercises in the book by adding drill exercises (with answers available with a click), a wide variety of examples illustrating the reach of the material, programming exercises with code solutions, and challenging problems.

Dynamic visualizations. Dynamic simulations are impossible in a printed book, but the website is replete with implementations that use a graphics class to present compelling visual demonstrations of algorithm applications.

Course materials. A complete set of lecture slides is tied directly to the material in the book and on the booksite. A full selection of programming assignments, with check lists, test data, and preparatory material, is also included.

Online course. A full set of lecture videos and self-assessment materials provide opportunities for students to learn or review the material on their own and for instructors to replace or supplement their lectures.

Links to related material. Hundreds of links lead students to background information about applications and to resources for studying algorithms.

Our goal in creating this material was to provide a complementary approach to the ideas. Generally, you should read the book when learning specific algorithms for the first time or when trying to get a global picture, and you should use the booksite as a reference when programming or as a starting point when searching for more detail while online.

Use in the curriculum The book is intended as a textbook in a second course in computer science. It provides full coverage of core material and is an excellent vehicle for students to gain experience and maturity in programming, quantitative reasoning, and problem-solving. Typically, one course in computer science will suffice as a prerequisite—the book is intended for anyone conversant with a modern programming language and with the basic features of modern computer systems.

The algorithms and data structures are expressed in Java, but in a style accessible to people fluent in other modern languages. We embrace modern Java abstractions (including generics) but resist dependence upon esoteric features of the language.

Most of the mathematical material supporting the analytic results is self-contained (or is labeled as beyond the scope of this book), so little specific preparation in mathematics is required for the bulk of the book, although mathematical maturity is definitely helpful. Applications are drawn from introductory material in the sciences, again self-contained.

The material covered is a fundamental background for any student intending to major in computer science, electrical engineering, or operations research, and is valuable for any student with interests in science, mathematics, or engineering.

Context The book is intended to follow our introductory text, *An Introduction to Programming in Java: An Interdisciplinary Approach*, which is a broad introduction to the field. Together, these two books can support a two- or three-semester introduction to computer science that will give any student the requisite background to successfully address computation in any chosen field of study in science, engineering, or the social sciences.

The starting point for much of the material in the book was the Sedgewick series of *Algorithms* books. In spirit, this book is closest to the first and second editions of that book, but this text benefits from decades of experience teaching and learning that material. Sedgewick's current *Algorithms in C/C++/Java, Third Edition* is more appropriate as a reference or a text for an advanced course; this book is specifically designed to be a textbook for a one-semester course for first- or second-year college students and as a modern introduction to the basics and a reference for use by working programmers.

Acknowledgments This book has been nearly 40 years in the making, so full recognition of all the people who have made it possible is simply not feasible. Earlier editions of this book list dozens of names, including (in alphabetical order) Andrew Appel, Trina Avery, Marc Brown, Lyn Dupré, Philippe Flajolet, Tom Freeman, Dave Hanson, Janet Incerpi, Mike Schidlowsky, Steve Summit, and Chris Van Wyk. All of these people deserve acknowledgement, even though some of their contributions may have happened decades ago. For this fourth edition, we are grateful to the hundreds of students at Princeton and several other institutions who have suffered through preliminary versions of the work, and to readers around the world for sending in comments and corrections through the booksite.

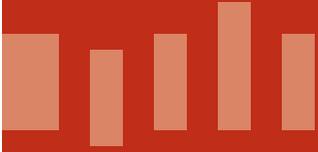
We are grateful for the support of Princeton University in its unwavering commitment to excellence in teaching and learning, which has provided the basis for the development of this work.

Peter Gordon has provided wise counsel throughout the evolution of this work almost from the beginning, including a gentle introduction of the “back to the basics” idea that is the foundation of this edition. For this fourth edition, we are grateful to Barbara Wood for her careful and professional copyediting, to Julie Nahil for managing the production, and to many others at Pearson for their roles in producing and marketing the book. All were extremely responsive to the demands of a rather tight schedule without the slightest sacrifice to the quality of the result.

Robert Sedgewick
Kevin Wayne

Princeton, New Jersey
January 2014

ONE



Fundamentals

1.1	Basic Programming Model.	8
1.2	Data Abstraction.	64
1.3	Bags, Queues, and Stacks	120
1.4	Analysis of Algorithms	172
1.5	Case Study: Union-Find.	216

The objective of this book is to study a broad variety of important and useful *algorithms*—methods for solving problems that are suited for computer implementation. Algorithms go hand in hand with data structures—schemes for organizing data that leave them amenable to efficient processing by an algorithm. This chapter introduces the basic tools that we need to study algorithms and data structures.

First, we introduce our basic programming model. All of our programs are implemented using a small subset of the Java programming language plus a few of our own libraries for input/output and for statistical calculations. SECTION 1.1 is a summary of language constructs, features, and libraries that we use in this book.

Next, we emphasize data abstraction, where we define abstract data types (ADTs) in the service of modular programming. In SECTION 1.2 we introduce the process of implementing an ADT in Java, by specifying an applications programming interface (API) and then using the Java class mechanism to develop an implementation for use in client code.

As important and useful examples, we next consider three fundamental ADTs: the bag, the *queue*, and the stack. SECTION 1.3 describes APIs and implementations of bags, queues, and stacks using arrays, resizing arrays, and linked lists that serve as models and starting points for algorithm implementations throughout the book.

Performance is a central consideration in the study of algorithms. SECTION 1.4 describes our approach to analyzing algorithm performance. The basis of our approach is the *scientific method*: we develop hypotheses about performance, create mathematical models, and run experiments to test them, repeating the process as necessary.

We conclude with a case study where we consider solutions to a connectivity problem that uses algorithms and data structures that implement the classic *union-find* ADT.

Algorithms When we write a computer program, we are generally implementing a *method* that has been devised previously to solve some problem. This method is often independent of the particular programming language being used—it is likely to be equally appropriate for many computers and many programming languages. It is the method, rather than the computer program itself, that specifies the steps that we can take to solve the problem. The term *algorithm* is used in computer science to describe a finite, deterministic, and effective problem-solving method suitable for implementation as a computer program. Algorithms are the stuff of computer science: they are central objects of study in the field.

We can define an algorithm by describing a procedure for solving a problem in a natural language, or by writing a computer program that implements the procedure, as shown at right for *Euclid's algorithm* for finding the greatest common divisor of two numbers, a variant of which was devised over 2,300 years ago. If you are not familiar with Euclid's algorithm, you are encouraged to work EXERCISE 1.1.24 and EXERCISE 1.1.25, perhaps after reading SECTION 1.1. In this book, we use computer programs to describe algorithms. One important reason for doing so is that it makes easier the task of checking whether they are finite, deterministic, and effective, as required. But it is also important to recognize that a program in a particular language is just one way to express an algorithm. The fact that many of the algorithms in this book have been expressed in multiple programming languages over the past several decades reinforces the idea that each algorithm is a method suitable for implementation on any computer in any programming language.

Most algorithms of interest involve organizing the data involved in the computation. Such organization leads to data structures, which also are central objects of study in computer science. Algorithms and data structures go hand in hand. In this book we take the view that data structures exist as the byproducts or end products of algorithms and that we must therefore study them in order to understand the algorithms. Simple algorithms can give rise to complicated data structures and, conversely, complicated algorithms can use simple data structures. We shall study the properties of many data structures in this book; indeed, we might well have titled the book *Algorithms and Data Structures*.

English-language description

Compute the greatest common divisor of two nonnegative integers p and q as follows: If q is 0, the answer is p . If not, divide p by q and take the remainder r . The answer is the greatest common divisor of q and r .

Java-language description

```
public static int gcd(int p, int q)
{
    if (q == 0) return p;
    int r = p % q;
    return gcd(q, r);
}
```

Euclid's algorithm

When we use a computer to help us solve a problem, we typically are faced with a number of possible approaches. For small problems, it hardly matters which approach we use, as long as we have one that correctly solves the problem. For huge problems (or applications where we need to solve huge numbers of small problems), however, we quickly become motivated to devise methods that use time and space efficiently.

The primary reason to learn about algorithms is that this discipline gives us the potential to reap huge savings, even to the point of enabling us to do tasks that would otherwise be impossible. In an application where we are processing millions of objects, it is not unusual to be able to make a program millions of times faster by using a well-designed algorithm. We shall see such examples on numerous occasions throughout the book. By contrast, investing additional money or time to buy and install a new computer holds the potential for speeding up a program by perhaps a factor of only 10 or 100. Careful algorithm design is an extremely effective part of the process of solving a huge problem, whatever the applications area.

When developing a huge or complex computer program, a great deal of effort must go into understanding and defining the problem to be solved, managing its complexity, and decomposing it into smaller subtasks that can be implemented easily. Often, many of the algorithms required after the decomposition are trivial to implement. In most cases, however, there are a few algorithms whose choice is critical because most of the system resources will be spent running those algorithms. These are the types of algorithms on which we concentrate in this book. We study fundamental algorithms that are useful for solving challenging problems in a broad variety of applications areas.

The sharing of programs in computer systems is becoming more widespread, so although we might expect to be using a large fraction of the algorithms in this book, we also might expect to have to implement only a small fraction of them. For example, the Java libraries contain implementations of a host of fundamental algorithms. However, implementing simple versions of basic algorithms helps us to understand them better and thus to more effectively use and tune advanced versions from a library. More important, the opportunity to reimplement basic algorithms arises frequently. The primary reason to do so is that we are faced, all too often, with completely new computing environments (hardware and software) with new features that old implementations may not use to best advantage. In this book, we concentrate on the simplest reasonable implementations of the best algorithms. We do pay careful attention to coding the critical parts of the algorithms, and take pains to note where low-level optimization effort could be most beneficial.

Choosing the best algorithm for a particular task can be a complicated process, perhaps involving sophisticated mathematical analysis. The branch of computer science that comprises the study of such questions is called *analysis of algorithms*. Many of the

algorithms that we study have been shown through analysis to have excellent theoretical performance; others are simply known to work well through experience. Our primary goal is to learn reasonable algorithms for important tasks, yet we shall also pay careful attention to comparative performance of the methods. We should not use an algorithm without having an idea of what resources it might consume, so we strive to be aware of how our algorithms might be expected to perform.

Summary of topics As an overview, we describe the major parts of the book, giving specific topics covered and an indication of our general orientation toward the material. This set of topics is intended to touch on as many fundamental algorithms as possible. Some of the areas covered are core computer-science areas that we study in depth to learn basic algorithms of wide applicability. Other algorithms that we discuss are from advanced fields of study within computer science and related fields. The algorithms that we consider are the products of decades of research and development and continue to play an essential role in the ever-expanding applications of computation.

Fundamentals (CHAPTER 1) in the context of this book are the basic principles and methodology that we use to implement, analyze, and compare algorithms. We consider our Java programming model, data abstraction, basic data structures, abstract data types for collections, methods of analyzing algorithm performance, and a case study.

Sorting algorithms (CHAPTER 2) for rearranging arrays in order are of fundamental importance. We consider a variety of algorithms in considerable depth, including insertion sort, selection sort, shellsort, quicksort, mergesort, and heapsort. We also encounter algorithms for several related problems, including priority queues, selection, and merging. Many of these algorithms will find application as the basis for other algorithms later in the book.

Searching algorithms (CHAPTER 3) for finding specific items among large collections of items are also of fundamental importance. We discuss basic and advanced methods for searching, including binary search trees, balanced search trees, and hashing. We note relationships among these methods and compare performance.

Graphs (CHAPTER 4) are sets of objects and connections, possibly with weights and orientation. Graphs are useful models for a vast number of difficult and important problems, and the design of algorithms for processing graphs is a major field of study. We consider depth-first search, breadth-first search, connectivity problems, and several algorithms and applications, including Kruskal's and Prim's algorithms for finding minimum spanning tree and Dijkstra's and the Bellman-Ford algorithms for solving shortest-paths problems.

Strings (CHAPTER 5) are an essential data type in modern computing applications. We consider a range of methods for processing sequences of characters. We begin with faster algorithms for sorting and searching when keys are strings. Then we consider substring search, regular expression pattern matching, and data-compression algorithms. Again, an introduction to advanced topics is given through treatment of some elementary problems that are important in their own right.

Context (CHAPTER 6) helps us relate the material in the book to several other advanced fields of study, including scientific computing, operations research, and the theory of computing. We survey event-driven simulation, B-trees, suffix arrays, maximum flow, and other advanced topics from an introductory viewpoint to develop appreciation for the interesting advanced fields of study where algorithms play a critical role. Finally, we describe search problems, reduction, and NP-completeness to introduce the theoretical underpinnings of the study of algorithms and relationships to material in this book.

THE STUDY OF ALGORITHMS IS INTERESTING AND EXCITING because it is a new field (almost all the algorithms that we study are less than 50 years old, and some were just recently discovered) with a rich tradition (a few algorithms have been known for hundreds of years). New discoveries are constantly being made, but few algorithms are completely understood. In this book we shall consider intricate, complicated, and difficult algorithms as well as elegant, simple, and easy ones. Our challenge is to understand the former and to appreciate the latter in the context of scientific and commercial applications. In doing so, we shall explore a variety of useful tools and develop a style of *algorithmic thinking* that will serve us well in computational challenges to come.



1.1 BASIC PROGRAMMING MODEL

OUR STUDY OF ALGORITHMS is based upon implementing them as programs written in the Java programming language. We do so for several reasons:

- Our programs are concise, elegant, and complete descriptions of algorithms.
- You can run the programs to study properties of the algorithms.
- You can put the algorithms immediately to good use in applications.

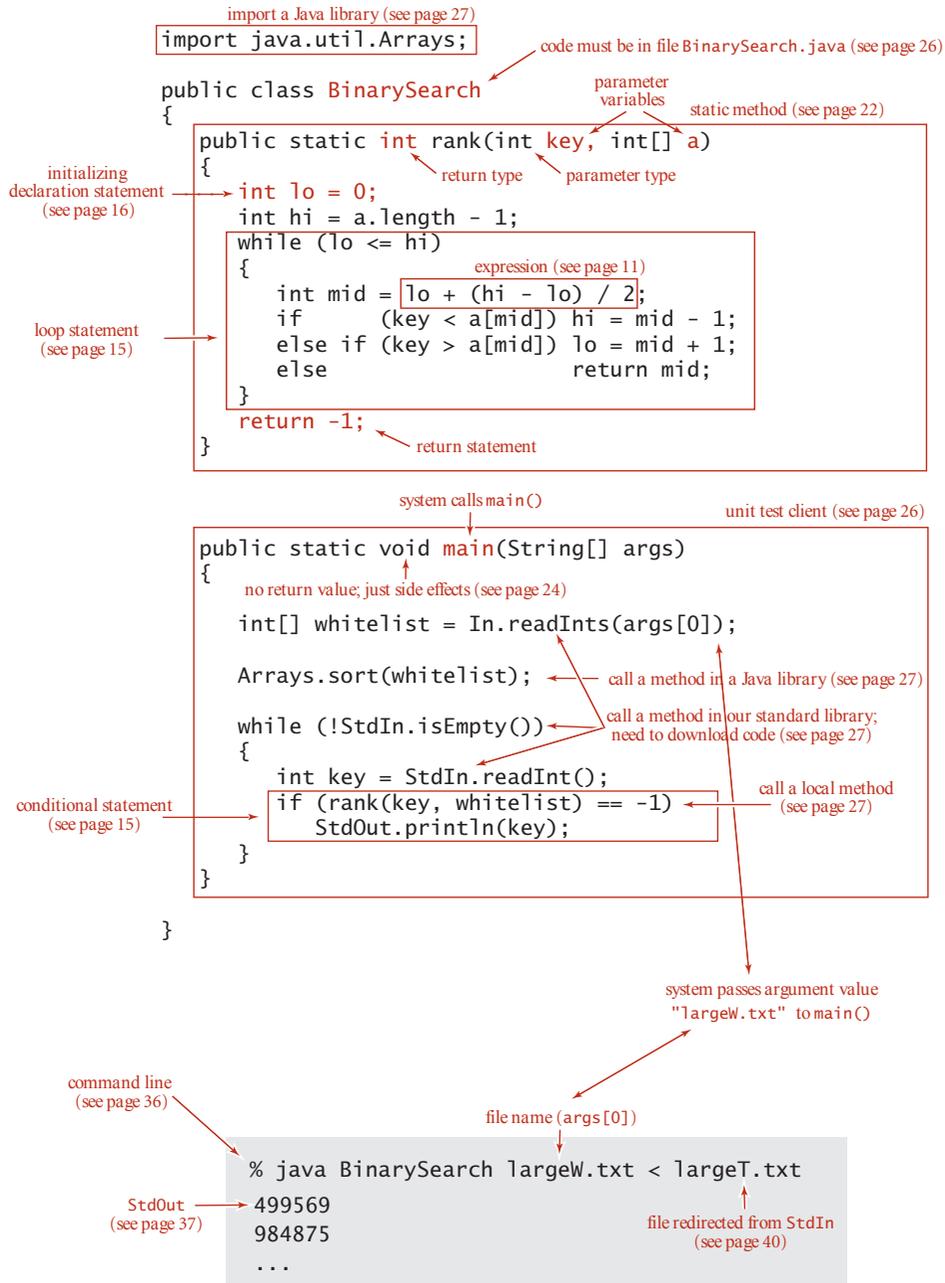
These are important and significant advantages over the alternatives of working with English-language descriptions of algorithms.

A potential downside to this approach is that we have to work with a specific programming language, possibly making it difficult to separate the idea of the algorithm from the details of its implementation. Our implementations are designed to mitigate this difficulty, by using programming constructs that are both found in many modern languages and needed to adequately describe the algorithms.

We use only a small subset of Java. While we stop short of formally defining the subset that we use, you will see that we make use of relatively few Java constructs, and that we emphasize those that are found in many modern programming languages. The code that we present is complete, and our expectation is that you will download it and execute it, on our test data or test data of your own choosing.

We refer to the programming constructs, software libraries, and operating system features that we use to implement and describe algorithms as our programming model. In this section and SECTION 1.2, we fully describe this programming model. The treatment is self-contained and primarily intended for documentation and for your reference in understanding any code in the book. The model we describe is the same model introduced in our book *An Introduction to Programming in Java: An Interdisciplinary Approach*, which provides a slower-paced introduction to the material.

For reference, the figure on the facing page depicts a complete Java program that illustrates many of the basic features of our programming model. We use this code for examples when discussing language features, but defer considering it in detail to page 46 (it implements a classic algorithm known as *binary search* and tests it for an application known as *whitelist filtering*). We assume that you have experience programming in some modern language, so that you are likely to recognize many of these features in this code. Page references are included in the annotations to help you find answers to any questions that you might have. Since our code is somewhat stylized and we strive to make consistent use of various Java idioms and constructs, it is worthwhile even for experienced Java programmers to read the information in this section.



Anatomy of a Java program and its invocation from the command line

Basic structure of a Java program A Java program (class) is either a library of *static methods* (functions) or a *data type definition*. To create libraries of static methods and data-type definitions, we use the following seven components, the basis of programming in Java and many other modern languages:

- Primitive data types precisely define the meaning of terms like integer, real number, and boolean value within a computer program. Their definition includes the set of possible values and operations on those values, which can be combined into expressions like mathematical expressions that define values.
- *Statements* allow us to define a computation by creating and assigning values to variables, controlling execution flow, or causing side effects. We use six types of statements: declarations, assignments, conditionals, loops, calls, and returns.
- Arrays allow us to work with multiple values of the same type.
- *Static methods* allow us to encapsulate and reuse code and to develop programs as a set of independent modules.
- *Strings* are sequences of characters. Some operations on them are built into Java.
- *Input/output* sets up communication between programs and the outside world.
- Data abstraction extends encapsulation and reuse to allow us to define non-primitive data types, thus supporting object-oriented programming.

In this section, we will consider the first five of these in turn. Data abstraction is the topic of the next section.

Running a Java program involves interacting with an operating system or a program development environment. For clarity and economy, we describe such actions in terms of a virtual terminal, where we interact with programs by typing commands to the system. See the booksite for details on using a virtual terminal on your system, or for information on using one of the many more advanced program development environments that are available on modern systems.

For example, `BinarySearch` is two static methods, `rank()` and `main()`. The first static method, `rank()`, is four statements: two declarations, a loop (which is itself an assignment and two conditionals), and a return. The second, `main()`, is three statements: a declaration, a call, and a loop (which is itself an assignment and a conditional).

To invoke a Java program, we first compile it using the `javac` command, then run it using the `java` command. For example, to run `BinarySearch`, we first type the command `javac BinarySearch.java` (which creates a file `BinarySearch.class` that contains a lower-level version of the program in Java bytecode). Then we type `java BinarySearch` (followed by a whitelist file name) to transfer control to the bytecode version of the program. To develop a basis for understanding the effect of these actions, we next consider in detail primitive data types and expressions, the various kinds of Java statements, arrays, static methods, strings, and input/output.

Primitive data types and expressions A data type is a set of values and a set of operations on those values. We begin by considering the following four primitive data types that are the basis of the Java language:

- *Integers*, with arithmetic operations (`int`)
- Real numbers, again with arithmetic operations (`double`)
- Booleans, the set of values { `true`, `false` } with logical operations (`boolean`)
- *Characters*, the alphanumeric characters and symbols that you type (`char`)

Next we consider mechanisms for specifying values and operations for these types.

A Java program manipulates variables that are named with *identifiers*. Each variable is associated with a data type and stores one of the permissible data-type values. In Java code, we use expressions like familiar mathematical expressions to apply the operations associated with each type. For primitive types, we use identifiers to refer to variables, operator symbols such as `+` `-` `*` `/` to specify operations, literals such as `1` or `3.14` to specify values, and expressions such as `(x + 2.236)/2` to specify operations on values. The purpose of an expression is to define one of the data-type values.

term	examples	definition
<i>primitive data type</i>	<code>int double boolean char</code>	a set of values and a set of operations on those values (built into the Java language)
<i>identifier</i>	<code>a abc Ab\$ a_b ab123 lo hi</code>	a sequence of letters, digits, <code>_</code> , and <code>\$</code> , the first of which is not a digit
<i>variable</i>	[<i>any identifier</i>]	names a data-type value
<i>operator</i>	<code>+</code> <code>-</code> <code>*</code> <code>/</code>	names a data-type operation
<i>literal</i>	<code>int</code> <code>1 0 -42</code> <code>double</code> <code>2.0 1.0e-15 3.14</code> <code>boolean</code> <code>true false</code> <code>char</code> <code>'a' '+' '9' '\n'</code>	source-code representation of a value
<i>expression</i>	<code>int</code> <code>lo + (hi - lo)/2</code> <code>double</code> <code>1.0e-15 * t</code> <code>boolean</code> <code>lo <= hi</code>	a literal, a variable, or a sequence of operations on literals and/or variables that produces a value

Basic building blocks for Java programs

To define a data type, we need only specify the values and the set of operations on those values. This information is summarized in the table below for Java's `int`, `double`, `boolean`, and `char` data types. These data types are similar to the basic data types found in many programming languages. For `int` and `double`, the operations are familiar arithmetic operations; for `boolean`, they are familiar logical operations. It is important to note that `+`, `-`, `*`, and `/` are overloaded—the same symbol specifies operations in multiple different types, depending on context. The key property of these primitive operations is that *an operation involving values of a given type has a value of that type*. This rule highlights the idea that we are often working with approximate values, since it is often the case that the exact value that would seem to be defined by the expression is not a value of the type. For example, `5/3` has the value 1 and `5.0/3.0` has a value very close to 1.666666666666667 but neither of these is exactly equal to `5/3`. This table is far from complete; we discuss some additional operators and various exceptional situations that we occasionally need to consider in the Q&A at the end of this section.

type	set of values	operators	typical expressions	
			expression	value
<code>int</code>	integers between -2^{31} and $+2^{31} - 1$ (32-bit two's complement)	<code>+</code> (add)	<code>5 + 3</code>	8
		<code>-</code> (subtract)	<code>5 - 3</code>	2
		<code>*</code> (multiply)	<code>5 * 3</code>	15
		<code>/</code> (divide)	<code>5 / 3</code>	1
		<code>%</code> (remainder)	<code>5 % 3</code>	2
<code>double</code>	double-precision real numbers (64-bit IEEE 754 standard)	<code>+</code> (add)	<code>3.141 - .03</code>	3.111
		<code>-</code> (subtract)	<code>2.0 - 2.0e-7</code>	1.9999998
		<code>*</code> (multiply)	<code>100 * .015</code>	1.5
		<code>/</code> (divide)	<code>6.02e23 / 2.0</code>	3.01e23
<code>boolean</code>	true or false	<code>&&</code> (and)	<code>true && false</code>	false
		<code> </code> (or)	<code>false true</code>	true
		<code>!</code> (not)	<code>!false</code>	true
		<code>^</code> (xor)	<code>true ^ true</code>	false
<code>char</code>	characters (16-bit)	[arithmetic operations, rarely used]		

Primitive data types in Java

Expressions. As illustrated in the table at the bottom of the previous page, typical expressions are *infix*: a literal (or an expression), followed by an operator, followed by another literal (or another expression). When an expression contains more than one operator, the order in which they are applied is often significant, so the following precedence conventions are part of the Java language specification: The operators `*` and `/` (and `%`) have higher precedence than (are applied before) the `+` and `-` operators; among logical operators, `!` is the highest precedence, followed by `&&` and then `||`. Generally, operators of the same precedence are applied left to right. As in standard arithmetic expressions, you can use parentheses to override these rules. Since precedence rules vary slightly from language to language, we use parentheses and otherwise strive to avoid dependence on precedence rules in our code.

Type conversion. Numbers are automatically promoted to a more inclusive type if no information is lost. For example, in the expression `1 + 2.5`, the `1` is promoted to the double value `1.0` and the expression evaluates to the `double` value `3.5`. A cast is a type name in parentheses within an expression, a directive to convert the following value into a value of that type. For example `(int) 3.7` is `3` and `(double) 3` is `3.0`. Note that casting to an `int` is truncation instead of rounding—rules for casting within complicated expressions can be intricate, and casts should be used sparingly and with care. A best practice is to use expressions that involve literals or variables of a single type.

Comparisons. The following operators compare two values of the same type and produce a `boolean` value: *equal* (`==`), *not equal* (`!=`), *less than* (`<`), *less than or equal* (`<=`), *greater than* (`>`), and *greater than or equal* (`>=`). These operators are known as mixed-type operators because their value is `boolean`, not the type of the values being compared. An expression with a boolean value is known as a boolean expression. Such expressions are essential components in conditional and loop statements, as we will see.

Other primitive types. Java's `int` has 2^{32} different values by design, so it can be represented in a 32-bit machine word (many machines have 64-bit words nowadays, but the 32-bit `int` persists). Similarly, the `double` standard specifies a 64-bit representation. These data-type sizes are adequate for typical applications that use integers and real numbers. To provide flexibility, Java has five additional primitive data types:

- 64-bit integers, with arithmetic operations (`long`)
- 16-bit integers, with arithmetic operations (`short`)
- 16-bit characters, with arithmetic operations (`char`)
- 8-bit integers, with arithmetic operations (`byte`)
- 32-bit single-precision real numbers, again with arithmetic operations (`float`)

We most often use `int` and `double` arithmetic operations in this book, so we do not consider the others (which are very similar) in further detail here.

Statements A Java program is composed of statements, which define the computation by creating and manipulating variables, assigning data-type values to them, and controlling the flow of execution of such operations. Statements are often organized in blocks, sequences of statements within curly braces.

- Declarations create variables of a specified type and name them with identifiers.
- Assignments associate a data-type value (defined by an expression) with a variable. Java also has several implicit assignment idioms for changing the value of a data-type value relative to its current value, such as incrementing the value of an integer variable.
- Conditionals provide for a simple change in the flow of execution—execute the statements in one of two blocks, depending on a specified condition.
- Loops provide for a more profound change in the flow of execution—execute the statements in a block as long as a given condition is true.
- Calls and returns relate to static methods (see page 22), which provide another way to change the flow of execution and to organize code.

A program is a sequence of statements, with declarations, assignments, conditionals, loops, calls, and returns. Programs typically have a nested structure: a statement among the statements in a block within a conditional or a loop may itself be a conditional or a loop. For example, the `while` loop in `rank()` contains an `if` statement. Next, we consider each of these types of statements in turn.

Declarations. A declaration statement associates a variable name with a type at compile time. Java requires us to use declarations to specify the names and types of variables. By doing so, we are being explicit about any computation that we are specifying. Java is said to be a strongly typed language, because the Java compiler checks for consistency (for example, it does not permit us to multiply a `boolean` and a `double`). Declarations can appear anywhere before a variable is first used—most often, we put them at the point of first use. The scope of a variable is the part of the program where it is defined. Generally the scope of a variable is composed of the statements that follow the declaration in the same block as the declaration.

Assignments. An assignment statement associates a data-type value (defined by an expression) with a variable. When we write `c = a + b` in Java, we are not expressing mathematical equality, but are instead expressing an action: set the value of the variable `c` to be the value of `a` plus the value of `b`. It is true that `c` is mathematically equal to `a + b` immediately after the assignment statement has been executed, but the point of the statement is to change the value of `c` (if necessary). The left-hand side of an assignment statement must be a single variable; the right-hand side can be an arbitrary expression that produces a value of the type.

Conditionals. Most computations require different actions for different inputs. One way to express these differences in Java is the `if` statement:

```
if (<boolean expression>) { <block statements> }
```

This description introduces a formal notation known as a *template* that we use occasionally to specify the format of Java constructs. We put within angle brackets (`< >`) a construct that we have already defined, to indicate that we can use any instance of that construct where specified. In this case, `<boolean expression>` represents an expression that has a boolean value, such as one involving a comparison operation, and `<block statements>` represents a sequence of Java statements. It is possible to make formal definitions of `<boolean expression>` and `<block statements>`, but we refrain from going into that level of detail. The meaning of an `if` statement is self-explanatory: the statement(s) in the block are to be executed if and only if the boolean expression is `true`. The `if-else` statement:

```
if (<boolean expression>) { <block statements> }
else                       { <block statements> }
```

allows for choosing between two alternative blocks of statements.

Loops. Many computations are inherently repetitive. The basic Java construct for handling such computations has the following format:

```
while (<boolean expression>) { <block statements> }
```

The `while` statement has the same form as the `if` statement (the only difference being the use of the keyword `while` instead of `if`), but the meaning is quite different. It is an instruction to the computer to behave as follows: if the boolean expression is `false`, do nothing; if the boolean expression is `true`, execute the sequence of statements in the block (just as with `if`) but then check the boolean expression again, execute the sequence of statements in the block again if the boolean expression is `true`, and continue as long as the boolean expression is `true`. We refer to the statements in the block in a loop as the *body* of the loop.

Break and continue. Some situations call for slightly more complicated control flow than provided by the basic `if` and `while` statements. Accordingly, Java supports two additional statements for use within `while` loops:

- The `break` statement, which immediately exits the loop
- The `continue` statement, which immediately begins the next iteration of the loop

We rarely use these statements in the code in this book (and many programmers never use them), but they do considerably simplify code in certain instances.

Shortcut notations There are several ways to express a given computation; we seek clear, elegant, and efficient code. Such code often takes advantage of the following widely used shortcuts (that are found in many languages, not just Java).

Initializing declarations. We can combine a declaration with an assignment to initialize a variable at the same time that it is declared (created). For example, the code `int i = 1;` creates an `int` variable named `i` and assigns it the initial value 1. A best practice is to use this mechanism close to first use of the variable (to limit scope).

Implicit assignments. The following shortcuts are available when our purpose is to modify a variable's value relative to its current value:

- Increment/decrement operators: `++i` is the same as `i = i + 1`; both have the value `i` in an expression. Similarly, `--i` is the same as `i = i - 1`. The code `i++` and `i--` are the same except that the expression value is the value before the increment/decrement, not after.
- Other compound operators: Prepending a binary operator to the `=` in an assignment is equivalent to using the variable on the left as the first operand. For example, the code `i/=2;` is equivalent to the code `i = i/2;` Note that `i += 1;` has the same effect as `i = i+1;` (and `i++`).

Single-statement blocks. If a block of statements in a conditional or a loop has only a single statement, the curly braces may be omitted.

For notation. Many loops follow this scheme: initialize an index variable to some value and then use a `while` loop to test a loop continuation condition involving the index variable, where the last statement in the `while` loop increments the index variable. You can express such loops compactly with Java's `for` notation:

```
for (<initialize>; <boolean expression>; <increment>)  
{  
    <block statements>  
}
```

This code is, with only a few exceptions, equivalent to

```
<initialize>;  
while (<boolean expression>)  
{  
    <block statements>  
    <increment>;  
}
```

We use `for` loops to support this initialize-and-increment programming idiom.

statement	examples	definition
<i>declaration</i>	<code>int i; double c;</code>	create a variable of a specified type, named with a given identifier
<i>assignment</i>	<code>a = b + 3; discriminant = b*b - 4.0*c;</code>	assign a data-type value to a variable
<i>initializing declaration</i>	<code>int i = 1; double c = 3.141592625;</code>	declaration that also assigns an initial value
<i>implicit assignment</i>	<code>i++; i += 1;</code>	<code>i = i + 1;</code>
<i>conditional (if)</i>	<code>if (x < 0) x = -x;</code>	execute a statement, depending on boolean expression
<i>conditional (if-else)</i>	<code>if (x > y) max = x; else max = y;</code>	execute one or the other statement, depending on boolean expression
<i>loop (while)</i>	<code>int v = 1; while (v <= N) v = 2*v; double t = c; while (Math.abs(t - c/t) > 1e-15*t) t = (c/t + t) / 2.0;</code>	execute statement until boolean expression is false
<i>loop (for)</i>	<code>for (int i = 1; i <= N; i++) sum += 1.0/i; for (int i = 0; i <= N; i++) StdOut.println(2*Math.PI*i/N);</code>	compact version of while statement
<i>call</i>	<code>int key = StdIn.readInt();</code>	invoke other methods (see page 22)
<i>return</i>	<code>return false;</code>	return from a method (see page 24)

Java statements

Arrays An array stores a sequence of values that are all of the same type. We want not only to store values but also to access each individual value. The method that we use to refer to individual values in an array is numbering and then indexing them. If we have N values, we think of them as being numbered from 0 to $N-1$. Then, we can unambiguously specify one of them in Java code by using the notation `a[i]` to refer to the i th value for any value of i from 0 to $N-1$. This Java construct is known as a one-dimensional array.

Creating and initializing an array. Making an array in a Java program involves three distinct steps:

- Declare the array name and type.
- Create the array.
- Initialize the array values.

To declare the array, you need to specify a name and the type of data it will contain. To create it, you need to specify its length (the number of values). For example, the “long form” code shown at right makes an array of N numbers of type `double`, all initialized to `0.0`. The first statement is the array declaration. It is just like a declaration of a variable of the corresponding primitive type except for the square brackets following the type name, which specify that we are declaring an array. The keyword `new` in the second statement is a Java directive to create the array. The reason that we need to explicitly create arrays at run time is that the Java compiler cannot know how much space to reserve for the array at compile time (as it can for primitive-type values). The `for` statement initializes the N array values. This code sets all of the array entries to the value `0.0`. When you begin to write code that uses an array, you must be sure that your code declares, creates, and initializes it. Omitting one of these steps is a common programming mistake.

```

long form
double[] a;           ← declaration
a = new double[N];   ← creation
for (int i = 0; i < N; i++)
    a[i] = 0.0;      ← initialization

short form
double[] a = new double[N];

initializing declaration
int[] a = { 1, 1, 2, 3, 5, 8 };

```

Declaring, creating, and initializing an array

Default array initialization. For economy in code, we often take advantage of Java’s default array initialization convention and combine all three steps into a single statement, as in the “short form” code in our example. The code to the left of the equal sign constitutes the declaration; the code to the right constitutes the creation. The `for` loop is unnecessary in this case because the default initial value of variables of type `double`

in a Java array is 0.0, but it would be required if a nonzero value were desired. The default initial value is zero for numeric types and `false` for type `boolean`.

Initializing declaration. The third option shown for our example is to specify the initialization values at compile time, by listing literal values between curly braces, separated by commas.

Using an array. Typical array-processing code is shown on page 21. After declaring and creating an array, you can refer to any individual value anywhere you would use a variable name in a program by enclosing an integer index in square brackets after the array name. Once we create an array, its size is fixed. A program can refer to the length of an array `a[]` with the code `a.length`. The last element of an array `a[]` is always `a[a.length-1]`. Java does *automatic bounds checking*—if you have created an array of size `N` and use an index whose value is less than 0 or greater than `N-1`, your program will terminate with an `ArrayOutOfBoundsException` runtime exception.

Aliasing. Note carefully that *an array name refers to the whole array*—if we assign one array name to another, then both refer to the same array, as illustrated in the following code fragment.

```
int[] a = new int[N];
...
a[i] = 1234;
...
int[] b = a;
...
b[i] = 5678; // a[i] is now 5678.
```

This situation is known as aliasing and can lead to subtle bugs. If your intent is to make a copy of an array, then you need to declare, create, and initialize a new array and then copy all of the entries in the original array to the new array, as in the third example on page 21.

Two-dimensional arrays. A two-dimensional array in Java is an array of one-dimensional arrays. A two-dimensional array may be ragged (its arrays may all be of differing lengths), but we most often work with (for appropriate parameters `M` and `N`) `M`-by-`N` two-dimensional arrays that are arrays of `M` rows, each an array of length `N` (so it also makes sense to refer to the array as having `N` columns). Extending Java array constructs to handle two-dimensional arrays is straightforward. To refer to the entry in row `i` and column `j` of a two-dimensional array `a[][]`, we use the notation `a[i][j]`; to declare a two-dimensional array, we add another pair of square brackets; and to create the array,

we specify the number of rows followed by the number of columns after the type name (both within square brackets), as follows:

```
double[][] a = new double[M][N];
```

We refer to such an array as an *M*-by-*N* array. By convention, the first dimension is the number of rows and the second is the number of columns. As with one-dimensional arrays, Java initializes all entries in arrays of numeric types to zero and in arrays of `boolean` values to `false`. Default initialization of two-dimensional arrays is useful because it masks more code than for one-dimensional arrays. The following code is equivalent to the single-line create-and-initialize idiom that we just considered:

```
double[][] a;  
a = new double[M][N];  
for (int i = 0; i < M; i++)  
    for (int j = 0; j < N; j++)  
        a[i][j] = 0.0;
```

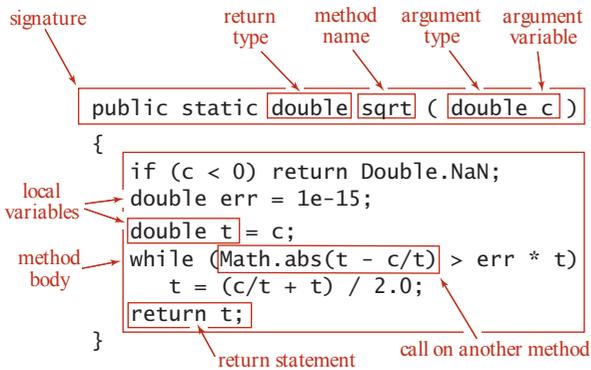
This code is superfluous when initializing to zero, but the nested `for` loops are needed to initialize to other value(s).

task	implementation (code fragment)
<i>find the maximum of the array values</i>	<pre>double max = a[0]; for (int i = 1; i < a.length; i++) if (a[i] > max) max = a[i];</pre>
<i>compute the average of the array values</i>	<pre>int N = a.length; double sum = 0.0; for (int i = 0; i < N; i++) sum += a[i]; double average = sum / N;</pre>
<i>copy to another array</i>	<pre>int N = a.length; double[] b = new double[N]; for (int i = 0; i < N; i++) b[i] = a[i];</pre>
<i>reverse the elements within an array</i>	<pre>int N = a.length; for (int i = 0; i < N/2; i++) { double temp = a[i]; a[i] = a[N-i-1]; a[N-i-1] = temp; }</pre>
matrix-matrix multiplication (square matrices)	<pre>int N = a.length; double[][] c = new double[N][N]; for (int i = 0; i < N; i++) for (int j = 0; j < N; j++) { // Compute dot product of row i and column j. for (int k = 0; k < N; k++) c[i][j] += a[i][k]*b[k][j]; }</pre>
$a[][] * b[][] = c[][]$	

Typical array-processing code

Static methods Every Java program in this book is either a *data-type definition* (which we describe in detail in SECTION 1.2) or a *library of static methods* (which we describe here). Static methods are called functions in many programming languages, since they can behave like mathematical functions, as described next. Each static method is a sequence of statements that are executed, one after the other, when the static method is called, in the manner described below. The modifier `static` distinguishes these methods from *instance methods*, which we discuss in SECTION 1.2. We use the word *method* without a modifier when describing characteristics shared by both kinds of methods.

Defining a static method. A *method* encapsulates a computation that is defined as a sequence of statements. A method takes arguments (values of given data types) and computes a return value of some data type that depends upon the arguments (such as a value defined by a mathematical function) or causes a side effect that depends on the arguments (such as printing a value). The static method `rank()` in `BinarySearch`



Anatomy of a static method

is an example of the first; `main()` is an example of the second. Each static method is composed of a signature (the keywords `public static` followed by a return type, the method name, and a sequence of arguments, each with a declared type) and a body (a statement block: a sequence of statements, enclosed in curly braces). Examples of static methods are shown in the table on the facing page.

Invoking a static method. A call on a static method is its name followed by expressions that specify argument values in parentheses, separated by commas.

When the method call is part of an expression, the method computes a value and that value is used in place of the call in the expression. For example the call on `rank()` in `BinarySearch()` returns an `int` value. A method call followed by a semicolon is a statement that generally causes side effects. For example, the call `Arrays.sort()` in `main()` in `BinarySearch` is a call on the system method `Arrays.sort()` that has the side effect of putting the entries in the array in sorted order. When a method is called, its argument variables are initialized with the values of the corresponding expressions in the call. A return statement terminates a static method, returning control to the caller. If the static method is to compute a value, that value must be specified in a return statement (if such a static method can reach the end of its sequence of statements without a return, the compiler will report the error).

task	implementation
absolute value of an int value	<pre>public static int abs(int x) { if (x < 0) return -x; else return x; }</pre>
absolute value of a double value	<pre>public static double abs(double x) { if (x < 0.0) return -x; else return x; }</pre>
primality test	<pre>public static boolean isPrime(int N) { if (N < 2) return false; for (int i = 2; i*i <= N; i++) if (N % i == 0) return false; return true; }</pre>
square root (Newton's method)	<pre>public static double sqrt(double c) { if (c < 0.0) return Double.NaN; double err = 1e-15; double t = c; while (Math.abs(t - c/t) > err * t) t = (c/t + t) / 2.0; return t; }</pre>
hypotenuse of a right triangle	<pre>public static double hypotenuse(double a, double b) { return Math.sqrt(a*a + b*b); }</pre>
Harmonic number (see page 185)	<pre>public static double H(int N) { double sum = 0.0; for (int i = 1; i <= N; i++) sum += 1.0 / i; return sum; }</pre>

Typical implementations of static methods

Properties of methods. A complete detailed description of the properties of methods is beyond our scope, but the following points are worth noting:

- *Arguments are passed by value.* You can use argument variables anywhere in the code in the body of the method in the same way you use local variables. The only difference between an argument variable and a local variable is that the argument variable is initialized with the argument value provided by the calling code. The method works with the value of its arguments, not the arguments themselves. One consequence of this approach is that changing the value of an argument variable within a static method has no effect on the calling code. Generally, we do not change argument variables in the code in this book. The pass-by-value convention implies that array arguments are aliased (see page 19)—the method uses the argument variable to refer to the caller’s array and can change the contents of the array (though it cannot change the array itself). For example, `Arrays.sort()` certainly changes the contents of the array passed as argument: it puts the entries in order.
- *Method names can be overloaded.* For example, the Java Math library uses this approach to provide implementations of `Math.abs()`, `Math.min()`, and `Math.max()` for all primitive numeric types. Another common use of overloading is to define two different versions of a function, one that takes an argument and another that uses a default value of that argument.
- *A method has a single return value but may have multiple return statements.* A Java method can provide only one return value, of the type declared in the method signature. Control goes back to the calling program as soon as the first return statement in a static method is reached. You can put return statements wherever you need them. Even though there may be multiple return statements, any static method returns a single value each time it is invoked: the value following the first return statement encountered.
- *A method can have side effects.* A method may use the keyword `void` as its return type, to indicate that it has no return value. An explicit return is not necessary in a `void` static method: control returns to the caller after the last statement. A `void` static method is said to produce side effects (consume input, produce output, change entries in an array, or otherwise change the state of the system). For example, the `main()` static method in our programs has a `void` return type because its purpose is to produce output. Technically, `void` methods do not implement mathematical functions (and neither does `Math.random()`, which takes no arguments but does produce a return value).

The instance methods that are the subject of SECTION 2.1 share these properties, though profound differences surround the issue of side effects.

Recursion. A method can call itself (if you are not comfortable with this idea, known as recursion, you are encouraged to work EXERCISES 1.1.16 through 1.1.22). For example, the code at the bottom of this page gives an alternate implementation of the `rank()` method in `BinarySearch`. We often use recursive implementations of methods because they can lead to compact, elegant code that is easier to understand than a corresponding implementation that does not use recursion. For example, the comment in the implementation below provides a succinct description of what the code is supposed to do. We can use this comment to convince ourselves that it operates correctly, by mathematical induction. We will expand on this topic and provide such a proof for binary search in SECTION 3.1. There are three important rules of thumb in developing recursive programs:

- The recursion has a base case—we always include a conditional statement as the first statement in the program that has a `return`.
- Recursive calls must address subproblems that are smaller in some sense, so that recursive calls converge to the base case. In the code below, the difference between the values of the fourth and the third arguments always decreases.
- Recursive calls should not address subproblems that overlap. In the code below, the portions of the array referenced by the two subproblems are disjoint.

Violating any of these guidelines is likely to lead to incorrect results or a spectacularly inefficient program (see EXERCISES 1.1.19 and 1.1.27). Adhering to them is likely to lead to a clear and correct program whose performance is easy to understand. Another reason to use recursive methods is that they lead to mathematical models that we can use to understand performance. We address this issue for binary search in SECTION 3.2 and in several other instances throughout the book.

```
public static int rank(int key, int[] a)
{ return rank(key, a, 0, a.length - 1); }

public static int rank(int key, int[] a, int lo, int hi)
{ // Index of key in a[], if present, is not smaller than lo
  //                                     and not larger than hi.
  if (lo > hi) return -1;
  int mid = lo + (hi - lo) / 2;
  if (key < a[mid]) return rank(key, a, lo, mid - 1);
  else if (key > a[mid]) return rank(key, a, mid + 1, hi);
  else return mid;
}
```

Recursive implementation of binary search

Basic programming model. A *library of static methods* is a set of static methods that are defined in a Java class, by creating a file with the keywords `public class` followed by the class name, followed by the static methods, enclosed in braces, kept in a file with the same name as the class and a `.java` extension. A basic model for Java programming is to develop a program that addresses a specific computational task by creating a library of static methods, one of which is named `main()`. Typing `java` followed by a class name followed by a sequence of strings leads to a call on `main()` in that class, with an array containing those strings as argument. After the last statement in `main()` executes, the program terminates. In this book, when we talk of a Java program for accomplishing a task, we are talking about code developed along these lines (possibly also including a data-type definition, as described in SECTION 1.2). For example, `BinarySearch` is a Java program composed of two static methods, `rank()` and `main()`, that accomplishes the task of printing numbers from an input stream that are not found in a whitelist file given as command-line argument.

Modular programming. Of critical importance in this model is that libraries of static methods enable modular programming where we build libraries of static methods (modules) and a static method in one library can call static methods defined in other libraries. This approach has many important advantages. It allows us to

- Work with modules of reasonable size, even in program involving a large amount of code
- Share and reuse code without having to reimplement it
- Easily substitute improved implementations
- Develop appropriate abstract models for addressing programming problems
- Localize debugging (see the paragraph below on unit testing)

For example, `BinarySearch` makes use of three other independently developed libraries, our `StdIn` and `In` library and Java's `Arrays` library. Each of these libraries, in turn, makes use of several other libraries.

Unit testing. A best practice in Java programming is to include a `main()` in every library of static methods that tests the methods in the library (some other programming languages disallow multiple `main()` methods and thus do not support this approach). Proper unit testing can be a significant programming challenge in itself. At a minimum, every module should contain a `main()` method that exercises the code in the module and provides some assurance that it works. As a module matures, we often refine the `main()` method to be a development client that helps us do more detailed tests as we develop the code, or a test client that tests all the code extensively. As a client becomes more complicated, we might put it in an independent module. In this book, we use `main()` to help illustrate the purpose of each module and leave test clients for exercises.

External libraries. We use static methods from four different kinds of libraries, each requiring (slightly) differing procedures for code reuse. Most of these are libraries of static methods, but a few are data-type definitions that also include some static methods.

- The standard system libraries `java.lang.*`. These include `Math`, which contains methods for commonly used mathematical functions; `Integer` and `Double`, which we use for converting between strings of characters and `int` and `double` values; `String` and `StringBuilder`, which we discuss in detail later in this section and in CHAPTER 5; and dozens of other libraries that we do not use.
- Imported system libraries such as `java.util.Arrays`. There are thousands of such libraries in a standard Java release, but we make scant use of them in this book. An `import` statement at the beginning of the program is needed to use such libraries (and signal that we are doing so).
- Other libraries in this book. For example, another program can use `rank()` in `BinarySearch`. To use such a program, download the source from the booksite into your working directory.
- The standard libraries `Std*` that we have developed for use in this book (and our introductory book *An Introduction to Programming in Java: An Interdisciplinary Approach*). These libraries are summarized in the following several pages. Source code and instructions for downloading them are available on the booksite.

To invoke a method from another library (one in the same directory or a specified directory, a standard system library, or a system library that is named in an `import` statement before the class definition), we prepend the library name to the method name for each call. For example, the `main()` method in `BinarySearch` calls the `sort()` method in the system library `java.util.Arrays`, the `readInts()` method in our library `In`, and the `println()` method in our library `StdOut`.

LIBRARIES OF METHODS IMPLEMENTED BY OURSELVES AND BY OTHERS in a modular programming environment can vastly expand the scope of our programming model. Beyond all of the libraries available in a standard Java release, thousands more are available on the web for applications of all sorts. To limit the scope of our programming model to a manageable size so that we can concentrate on algorithms, we use just the libraries listed in the table at right on this page, with a subset of their methods listed in *APIs*, as described next.

standard system libraries

`Math`
`Integer`[†]
`Double`[†]
`String`[†]
`StringBuilder`
`System`

imported system libraries

`java.util.Arrays`

our standard libraries

`StdIn`
`StdOut`
`StdDraw`
`StdRandom`
`StdStats`
`In`[†]
`Out`[†]

[†] data type definitions that include some static methods

Libraries with static methods used in this book

APIs A critical component of modular programming is documentation that explains the operation of library methods that are intended for use by others. We will consistently describe the library methods that we use in this book in application programming *interfaces (APIs)* that list the library name and the signatures and short descriptions of each of the methods that we use. We use the term *client* to refer to a program that calls a method in another library and the term *implementation* to describe the Java code that implements the methods in an API.

Example. The following example, the API for commonly used static methods from the standard Math library in `java.lang`, illustrates our conventions for APIs:

```
public class Math
```

```
    static double abs(double a)           absolute value of a
    static double max(double a, double b) maximum of a and b
    static double min(double a, double b) minimum of a and b
```

Note 1: `abs()`, `max()`, and `min()` are defined also for `int`, `long`, and `float`.

```
    static double sin(double theta)      sine function
    static double cos(double theta)      cosine function
    static double tan(double theta)      tangent function
```

Note 2: Angles are expressed in radians. Use `toDegrees()` and `toRadians()` to convert.

Note 3: Use `asin()`, `acos()`, and `atan()` for inverse functions.

```
    static double exp(double a)          exponential (ea)
    static double log(double a)          natural log (loge a, or ln a)
    static double pow(double a, double b) raise a to the bth power (ab)

    static double random()               random number in [0, 1)
    static double sqrt(double a)         square root of a

    static double E                       value of e (constant)
    static double PI                       value of π (constant)
```

See *booksite* for other available functions.

API for Java's mathematics library (excerpts)

These methods implement mathematical functions—they use their arguments to compute a value of a specified type (except `random()`, which does not implement a mathematical function because it does not take an argument). Since they all operate on `double` values and compute a `double` result, you can consider them as extending the `double` data type—extensibility of this nature is one of the characteristic features of modern programming languages. Each method is described by a line in the API that specifies the information you need to know in order to use the method. The `Math` library also defines the precise constant values `PI` (for π) and `E` (for e), so that you can use those names to refer to those constants in your programs. For example, the value of `Math.sin(Math.PI/2)` is `1.0` and the value of `Math.log(Math.E)` is `1.0` (because `Math.sin()` takes its argument in radians and `Math.log()` implements the natural logarithm function).

Java libraries. Extensive online descriptions of thousands of libraries are part of every Java release, but we excerpt just a few methods that we use in the book, in order to clearly delineate our programming model. For example, `BinarySearch` uses the `sort()` method from Java's `Arrays` library, which we document as follows:

```
public class Arrays
```

```
    static void sort(int[] a)           put the array in increasing order
```

Note: This method is defined also for other primitive types and `Object`.

Excerpt from Java's `Arrays` library (`java.util.Arrays`)

The `Arrays` library is not in `java.lang`, so an `import` statement is needed to use it, as in `BinarySearch`. Actually, CHAPTER 2 of this book is devoted to implementations of `sort()` for arrays, including the mergesort and quicksort algorithms that are implemented in `Arrays.sort()`. Many of the fundamental algorithms that we consider in this book are implemented in Java and in many other programming environments. For example, `Arrays` also includes an implementation of binary search. To avoid confusion, we generally use our own implementations, although there is nothing wrong with using a finely tuned library implementation of an algorithm that you understand.

Our standard libraries. We have developed a number of libraries that provide useful functionality for introductory Java programming, for scientific applications, and for the development, study, and application of algorithms. Most of these libraries are for input and output; we also make use of the following two libraries to test and analyze our implementations. The first extends `Math.random()` to allow us to draw random values from various distributions; the second supports statistical calculations:

```
public class StdRandom
```

static void setSeed(long seed)	<i>initialize</i>
static double random()	<i>real between 0 and 1</i>
static int uniform(int N)	<i>integer between 0 and N-1</i>
static int uniform(int lo, int hi)	<i>integer between lo and hi-1</i>
static double uniform(double lo, double hi)	<i>real between lo and hi</i>
static boolean bernoulli(double p)	<i>true with probability p</i>
static double gaussian()	<i>normal, mean 0, std dev 1</i>
static double gaussian(double m, double s)	<i>normal, mean m, std dev s</i>
static int discrete(double[] a)	<i>i with probability a[i]</i>
static void shuffle(double[] a)	<i>randomly shuffle the array a[]</i>

Note: overloaded implementations of `shuffle()` are included for other primitive types and for `Object`.

API for our library of static methods for random numbers

```
public class StdStats
```

static double max(double[] a)	<i>largest value</i>
static double min(double[] a)	<i>smallest value</i>
static double mean(double[] a)	<i>average</i>
static double var(double[] a)	<i>sample variance</i>
static double stddev(double[] a)	<i>sample standard deviation</i>
static double median(double[] a)	<i>median</i>

API for our library of static methods for data analysis

The `setSeed()` method in `StdRandom` allows us to seed the random number generator so that we can reproduce experiments involving random numbers. For reference, implementations of many of these methods are given on page 32. Some of these methods are extremely easy to implement; why do we bother including them in a library? Answers to this question are standard for well-designed libraries:

- They implement a level of abstraction that allow us to focus on implementing and testing the algorithms in the book, not generating random objects or calculating statistics. Client code that uses such methods is clearer and easier to understand than homegrown code that does the same calculation.
- Library implementations test for exceptional conditions, cover rarely encountered situations, and submit to extensive testing, so that we can count on them to operate as expected. Such implementations might involve a significant amount of code. For example, we often want implementations for various types of data. For example, Java's `Arrays` library includes multiple overloaded implementations of `sort()`, one for each type of data that you might need to sort.

These are bedrock considerations for modular programming in Java, but perhaps a bit overstated in this case. While the methods in both of these libraries are essentially self-documenting and many of them are not difficult to implement, some of them represent interesting algorithmic exercises. Accordingly, you are well-advised to *both* study the code in `StdRandom.java` and `StdStats.java` on the booksite and to take advantage of these tried-and-true implementations. The easiest way to use these libraries (and to examine the code) is to download the source code from the booksite and put them in your working directory; various system-dependent mechanisms for using them without making multiple copies are also described on the booksite.

Your own libraries. It is worthwhile to consider *every program that you write* as a library implementation, for possible reuse in the future.

- Write code for the client, a top-level implementation that breaks the computation up into manageable parts.
- Articulate an API for a library (or multiple APIs for multiple libraries) of static methods that can address each part.
- Develop an implementation of the API, with a `main()` that tests the methods independent of the client.

Not only does this approach provide you with valuable software that you can later reuse, but also taking advantage of modular programming in this way is a key to successfully addressing a complex programming task.

intended result	implementation
random double value in [a, b)	<pre>public static double uniform(double a, double b) { return a + StdRandom.random() * (b-a); }</pre>
random int value in [0..N)	<pre>public static int uniform(int N) { return (int) (StdRandom.random() * N); }</pre>
random int value in [lo..hi)	<pre>public static int uniform(int lo, int hi) { return lo + StdRandom.uniform(hi - lo); }</pre>
random int value drawn from discrete distribution (i with probability a[i])	<pre>public static int discrete(double[] a) { // Entries in a[] must sum to 1. double r = StdRandom.random(); double sum = 0.0; for (int i = 0; i < a.length; i++) { sum = sum + a[i]; if (sum >= r) return i; } return -1; }</pre>
randomly shuffle the elements in an array of double values (See Exercise 1.1.36)	<pre>public static void shuffle(double[] a) { int N = a.length; for (int i = 0; i < N; i++) { // Exchange a[i] with random element in a[i..N-1] int r = i + StdRandom.uniform(N-i); double temp = a[i]; a[i] = a[r]; a[r] = temp; } }</pre>

Implementations of static methods in StdRandom library

THE PURPOSE OF AN API is to separate the client from the implementation: the client should know nothing about the implementation other than information given in the API, and the implementation should not take properties of any particular client into account. APIs enable us to separately develop code for various purposes, then reuse it widely. No Java library can contain all the methods that we might need for a given computation, so this ability is a crucial step in addressing complex programming applications. Accordingly, programmers normally think of the API as a contract between the client and the implementation that is a clear specification of what each method is to do. Our goal when developing an implementation is to honor the terms of the contract. Often, there are many ways to do so, and separating client code from implementation code gives us the freedom to substitute new and improved implementations. In the study of algorithms, this ability is an important ingredient in our ability to understand the impact of algorithmic improvements that we develop.

Strings A `String` is a sequence of characters (`char` values). A literal `String` is a sequence of characters within double quotes, such as `"Hello, World"`. The data type `String` is a Java data type but it is not a primitive type. We consider `String` now because it is a fundamental data type that almost every Java program uses.

Concatenation. Java has a built-in concatenation operator (+) for `String` like the built-in operators that it has for primitive types, justifying the addition of the row in the table below to the primitive-type table on page 12. The result of concatenating two `String` values is a single `String` value, the first string followed by the second.

type	set of values	typical literals	operators	typical expressions	
				expression	value
<code>String</code>	character sequences	<code>"AB"</code>	+ (concatenate)	<code>"Hi, " + "Bob"</code>	<code>"Hi, Bob"</code>
		<code>"Hello"</code>		<code>"12" + "34"</code>	<code>"1234"</code>
		<code>"2.5"</code>		<code>"1" + "+" + "2"</code>	<code>"1+2"</code>

Java's `String` data type

Conversion. Two primary uses of strings are to convert values that we can enter on a keyboard into data-type values and to convert data-type values to values that we can read on a display. Java has built-in operations for `String` to facilitate these operations. In particular, the language includes libraries `Integer` and `Double` that contain static methods to convert between `String` values and `int` values and between `String` values and `double` values, respectively.

```
public class Integer
```

```
    static int parseInt(String s)           convert s to an int value
    static String toString(int i)         convert i to a String value
```

```
public class Double
```

```
    static double parseDouble(String s)   convert s to a double value
    static String toString(double x)      convert x to a String value
```

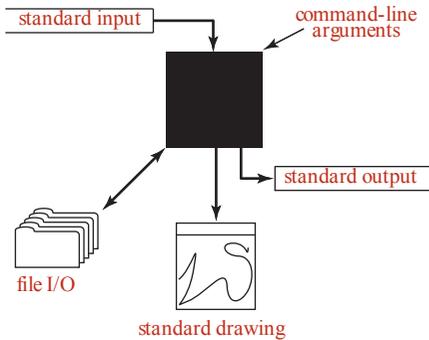
APIs for conversion between numbers and `String` values

Automatic conversion. We rarely explicitly use the static `toString()` methods just described because Java has a built-in mechanism that allows us to convert from any data type value to a `String` value by using concatenation: if one of the arguments of `+` is a `String`, Java automatically converts the other argument to a `String` (if it is not already a `String`). Beyond usage like `"The square root of 2.0 is " + Math.sqrt(2.0)` this mechanism enables conversion of any data-type value to a `String`, by concatenating it with the empty string `""`.

Command-line arguments. One important use of strings in Java programming is to enable a mechanism for passing information from the command line to the program. The mechanism is simple. When you type the `java` command followed by a library name followed by a sequence of strings, the Java system invokes the `main()` method in that library with an array of strings as argument: the strings typed after the library name. For example, the `main()` method in `BinarySearch` takes one command-line argument, so the system creates an array of size one. The program uses that value, `args[0]`, to name the file containing the whitelist, for use as the argument to `In.readInts()`. Another typical paradigm that we often use in our code is when a command-line argument is intended to represent a number, so we use `parseInt()` to convert to an `int` value or `parseDouble()` to convert to a `double` value.

COMPUTING WITH STRINGS is an essential component of modern computing. For the moment, we make use of `String` just to convert between external representation of numbers as sequences of characters and internal representation of numeric data-type values. In SECTION 1.2, we will see that Java supports many, many more operations on `String` values that we use throughout the book; in SECTION 1.4, we will examine the internal representation of `String` values; and in CHAPTER 5, we consider in depth algorithms that process `String` data. These algorithms are among the most interesting, intricate, and impactful methods that we consider in this book.

Input and output The primary purpose of our standard libraries for input, output, and drawing is to support a simple model for Java programs to interact with the outside world. These libraries are built upon extensive capabilities that are available in Java libraries, but are generally much more complicated and much more difficult to learn and use. We begin by briefly reviewing the model.



A bird's-eye view of a Java program

In our model, a Java program takes input values from command-line arguments or from an abstract stream of characters known as the standard input stream and writes to another abstract stream of characters known as the standard output stream.

Necessarily, we need to consider the interface between Java and the operating system, so we need to briefly discuss basic mechanisms that are provided by most modern operating systems and program-development environments. You can find more details about your particular system on the booksite. By default, command-line arguments, standard input, and standard output are associated

with an application supported by either the operating system or the program development environment that takes commands. We use the generic term terminal window to refer to the window maintained by this application, where we type and read text. Since early Unix systems in the 1970s this model has proven to be a convenient and direct way for us to interact with our programs and data. We add to the classical model a standard drawing that allows us to create visual representations for data analysis.

Commands and arguments. In the terminal window, we see a prompt, where we type commands to the operating system that may take arguments. We use only a few commands in this book, shown in the table below. Most often, we use the `java` command, to run our programs. As mentioned on page 35, Java classes have a `main()` static method that takes a `String` array `args[]` as its argument. That array is the sequence of command-line arguments that we type, provided to Java by the operating system.

command	arguments	purpose
<code>javac</code>	<code>. java file name</code>	compile Java program
<code>java</code>	<code>. class file name (no extension) and command-line arguments</code>	run Java program
<code>more</code>	any text file name	print file contents

Typical operating-system commands

By convention, both Java and the operating system process the arguments as strings. If we intend for an argument to be a number, we use a method such as `Integer.parseInt()` to convert it from `String` to the appropriate type.

Standard output. Our StdOut library provides support for standard output. By default, the system connects standard output to the terminal window. The `print()` method puts its argument on standard output; the `println()` method adds a newline; and the `printf()` method supports formatted output, as described next. Java provides a similar method in its `System.out` library; we use `StdOut` to treat standard input and standard output in a uniform manner (and to provide a few technical improvements).

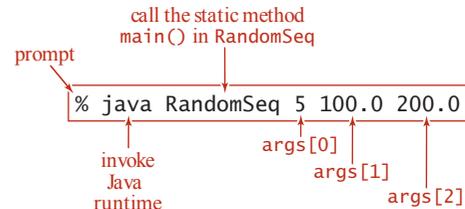
<code>public class StdOut</code>	
<code>static void print(String s)</code>	print s
<code>static void println(String s)</code>	print s, followed by newline
<code>static void println()</code>	print a new line
<code>static void printf(String f, ...)</code>	formatted print

Note: overloaded implementations are included for primitive types and for `Object`.

API for our library of static methods for standard output

To use these methods, download into your working directory `StdOut.java` from the booksite and use code such as `StdOut.println("Hello, World");` to call them. A sample client is shown at right.

Formatted output. In its simplest form, `printf()` takes two arguments. The first argument is a format string that describes how the second argument is to be converted to a string for output. The simplest type of format string begins with `%` and ends with a one-letter conversion code. The conversion codes that we use most frequently are `d` (for decimal values from Java's integer types), `f` (for floating-point values), and `s` (for String values). Between the `%` and the conversion code is an integer value that specifies the *field width* of the



Anatomy of a command

```
public class RandomSeq
{
    public static void main(String[] args)
    { // Print N random values in (lo, hi).
        int N = Integer.parseInt(args[0]);
        double lo = Double.parseDouble(args[1]);
        double hi = Double.parseDouble(args[2]);
        for (int i = 0; i < N; i++)
        {
            double x = StdRandom.uniform(lo, hi);
            StdOut.printf("%.2f\n", x);
        }
    }
}
```

Sample StdOut client

```
% java RandomSeq 5 100.0 200.0
123.43
153.13
144.38
155.18
104.02
```

converted value (the number of characters in the converted output string). By default, blank spaces are added on the left to make the length of the converted output equal to the field width; if we want the spaces on the right, we can insert a minus sign before the field width. (If the converted output string is bigger than the field width, the field width is ignored.) Following the width, we have the option of including a period followed by the number of digits to put after the decimal point (the precision) for a `double` value or the number of characters to take from the beginning of the string for a `String` value. The most important thing to remember about using `printf()` is that *the conversion code in the format and the type of the corresponding argument must match*. That is, Java must be able to convert from the type of the argument to the type required by the conversion code. The first argument of `printf()` is a `String` that may contain characters other than a format string. Any part of the argument that is not part of a format string passes through to the output, with the format string replaced by the argument value (converted to a `String` as specified). For example, the statement

```
StdOut.printf("PI is approximately %.2f\n", Math.PI);
```

prints the line

```
PI is approximately 3.14
```

Note that we need to explicitly include the newline character `\n` in the argument in order to print a new line with `printf()`. The `printf()` function can take more than two arguments. In this case, the format string will have a format specifier for each additional argument, perhaps separated by other characters to pass through to the output. You can also use the static method `String.format()` with arguments exactly as just described for `printf()` to get a formatted string without printing it. Formatted printing is a convenient mechanism that allows us to develop compact code that can produce tabulated experimental data (our primary use in this book).

type	code	typical literal	sample format strings	converted string values for output
<code>int</code>	<code>d</code>	512	<code>"%14d"</code> <code>"%-14d"</code>	" 512" "512"
<code>double</code>	<code>f</code> <code>e</code>	1595.1680010754388	<code>"%14.2f"</code> <code>"%.7f"</code> <code>"%14.4e"</code>	" 1595.17" "1595.1680011" " 1.5952e+03"
<code>String</code>	<code>s</code>	"Hello, World"	<code>"%14s"</code> <code>"%-14s"</code> <code>"%-14.5s"</code>	" Hello, World" "Hello, World " "Hello"

Format conventions for `printf()` (see the booksite for many other options)

Standard input. Our `StdIn` library takes data from the standard input stream that may be empty or may contain a sequence of values separated by whitespace (spaces, tabs, newline characters, and the like). By default, the system connects standard output to the terminal window—what you type is the input stream (terminated by `<ctrl-d>` or `<ctrl-z>`, depending on your terminal window application). Each value is a `String` or a value from one of Java's primitive types. One of the key features of the standard input stream is that your program consumes values when it reads them. Once your program has read a value, it cannot back up and read it again. This assumption is restrictive, but it reflects physical characteristics of some input devices and simplifies implementing the abstraction. Within the input stream model, the static methods in this library are largely self-documenting (described by their signatures).

```
public class Average
{
    public static void main(String[] args)
    { // Average the numbers on StdIn.
        double sum = 0.0;
        int cnt = 0;
        while (!StdIn.isEmpty())
        { // Read a number and cumulate the sum.
            sum += StdIn.readDouble();
            cnt++;
        }
        double avg = sum / cnt;
        StdOut.printf("Average is %.5f\n", avg);
    }
}
```

Sample `StdIn` client

```
% java Average
1.23456
2.34567
3.45678
4.56789
<ctrl-d>
Average is 2.90123
```

<code>public class StdIn</code>	
<code>static boolean isEmpty()</code>	<i>true if no more values, false otherwise</i>
<code>static int readInt()</code>	<i>read a value of type int</i>
<code>static double readDouble()</code>	<i>read a value of type double</i>
<code>static float readFloat()</code>	<i>read a value of type float</i>
<code>static long readLong()</code>	<i>read a value of type long</i>
<code>static boolean readBoolean()</code>	<i>read a value of type boolean</i>
<code>static char readChar()</code>	<i>read a value of type char</i>
<code>static byte readByte()</code>	<i>read a value of type byte</i>
<code>static String readString()</code>	<i>read a value of type String</i>
<code>static boolean hasNextLine()</code>	<i>is there another line in the input stream?</i>
<code>static String readLine()</code>	<i>read the rest of the line</i>
<code>static String readAll()</code>	<i>read the rest of the input stream</i>

API for our library of static methods for standard input

Redirection and piping. Standard input and output enable us to take advantage of command-line extensions supported by many operating-systems. By adding a simple directive to the command that invokes a program, we can redirect its standard output to a file, either for permanent storage or for input to another program at a later time:

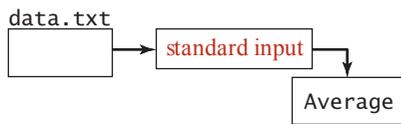
```
% java RandomSeq 1000 100.0 200.0 > data.txt
```

This command specifies that the standard output stream is not to be printed in the terminal window, but instead is to be written to a text file named `data.txt`. Each call to

`StdOut.print()` or `StdOut.println()` appends text at the end of that file. In this example, the end result is a file that contains 1,000 random values. No output appears in the terminal window: it goes directly into the file named after the `>` symbol. Thus, we can save away information for later retrieval. Note that we do not have to change `RandomSeq` in any way—it is using the standard output abstraction and is unaffected by our use of a different implementation of that abstraction. Similarly, we can redirect standard input so that `StdIn` reads data from a file instead of the terminal application:

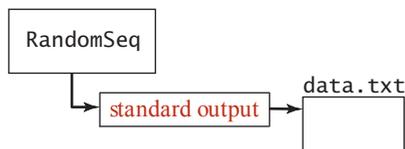
redirecting from a file to standard input

```
% java Average < data.txt
```



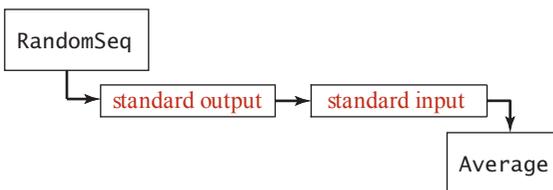
redirecting standard output to a file

```
% java RandomSeq 1000 100.0 200.0 > data.txt
```



piping the output of one program to the input of another

```
% java RandomSeq 1000 100.0 200.0 | java Average
```



Redirection and piping from the command line

terminal window. When the program calls `StdIn.readDouble()`, the operating system reads the value from the file. Combining these to redirect the output of one program to the input of another is known as piping:

```
% java RandomSeq 1000 100.0 200.0 | java Average
```

```
% java Average < data.txt
```

This command reads a sequence of numbers from the file `data.txt` and computes their average value. Specifically, the `<` symbol is a directive that tells the operating system to implement the standard input stream by reading from the text file `data.txt` instead of waiting for the user to type something into the

This command specifies that standard output for `RandomSeq` and standard input for `Average` are the same stream. The effect is as if `RandomSeq` were typing the numbers it generates into the terminal window while `Average` is running. This difference is profound, because it removes the limitation on the size of the input and output streams that we can process. For example, we could replace 1000 in our example with 1000000000, even though we might not have the space to save a billion numbers on our computer (we do need the time to process them). When `RandomSeq` calls `StdOut.printf()`, a string is added to the end of the stream; when `Average` calls `StdIn.readDouble()`, a string is removed from the beginning of the stream. The timing of precisely what happens is up to the operating system: it might run `RandomSeq` until it produces some numbers, and then run `Average` to consume those numbers, or it might run `Average` until it needs to consume a number, and then run `RandomSeq` until it produces the needed number. The end result is the same, but our programs are freed from worrying about such details because they work solely with the standard input and standard output abstractions.

Input and output from a file. Our `In` and `Out` libraries provide static methods that implement the abstraction of reading from and writing to a file the contents of an array of values of a primitive type (or `String`). We use `readInts()`, `readDoubles()`, and `readStrings()` in the `In` library and `writeInts()`, `writeDoubles()`, and `writeStrings()` in the `Out` library. The named argument can be a file or a web page. For example, this ability allows us to use a file and standard input for two different purposes in the same program, as in `BinarySearch`. The `In` and `Out` libraries also implement data types with instance methods that allow us the more general ability to treat multiple files as input and output streams, and web pages as input streams, so we will revisit them in SECTION 1.2.

```
public class In


---


    static int[] readInts(String name)           read int values
    static double[] readDoubles(String name)     read double values
    static String[] readStrings(String name)     read String values

public class Out


---


    static void write(int[] a, String name)      write int values
    static void write(double[] a, String name)  write double values
    static void write(String[] a, String name)  write String values
```

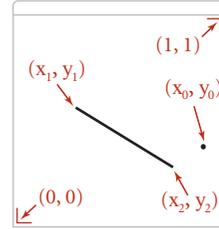
Note 1: Other primitive types are supported.

Note 2: `StdIn` and `StdOut` are supported (omit name argument).

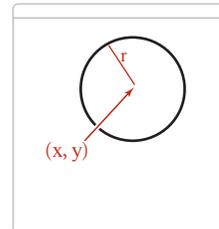
APIs for our static methods for reading and writing arrays

Standard drawing (basic methods). Up to this point, our input/output abstractions have focused exclusively on text strings. Now we introduce an abstraction for producing drawings as output. This library is easy to use and allows us to take advantage of a visual medium to cope with far more information than is possible with just text. As with standard input/output, our standard drawing abstraction is implemented in a library `StdDraw` that you can access by downloading the file `StdDraw.java` from the booksite into your working directory. Standard draw is very simple: we imagine an abstract drawing device capable of drawing lines and points on a two-dimensional canvas. The device is capable of responding to the commands to draw basic geometric shapes that our programs issue in the form of calls to static methods in `StdDraw`, including methods for drawing lines, points, text strings, circles, rectangles, and polygons. Like the methods for standard input and standard output, these methods are nearly self-documenting: `StdDraw.line()` draws a straight line segment connecting the point (x_0, y_0) with the point (x_1, y_1) whose coordinates are given as arguments. `StdDraw.point()` draws a spot centered on the point (x, y) whose coordinates are given as arguments, and so forth, as illustrated in the diagrams at right. Geometric shapes can be filled (in black, by default). The default scale is the unit square (all coordinates are between 0 and 1). The standard implementation displays the canvas in a window on your computer's screen, with black lines and points on a white background.

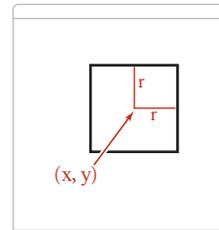
```
StdDraw.point(x0, y0);
StdDraw.line(x1, y1, x2, y2);
```



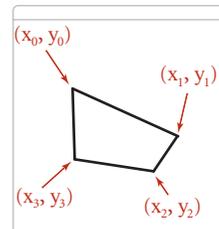
```
StdDraw.circle(x, y, r);
```



```
StdDraw.square(x, y, r);
```



```
double[] x = {x0, x1, x2, x3};
double[] y = {y0, y1, y2, y3};
StdDraw.polygon(x, y);
```



StdDraw examples

```
public class StdDraw
{
    static void line(double x0, double y0, double x1, double y1)
    static void point(double x, double y)
    static void text(double x, double y, String s)
    static void circle(double x, double y, double r)
    static void filledCircle(double x, double y, double r)
    static void ellipse(double x, double y, double rw, double rh)
    static void filledEllipse(double x, double y, double rw, double rh)
    static void square(double x, double y, double r)
    static void filledSquare(double x, double y, double r)
    static void rectangle(double x, double y, double rw, double rh)
    static void filledRectangle(double x, double y, double rw, double rh)
    static void polygon(double[] x, double[] y)
    static void filledPolygon(double[] x, double[] y)
}
```

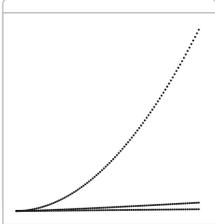
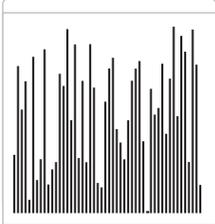
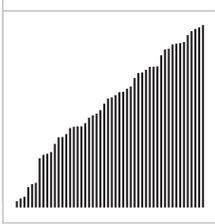
API for our library of static methods for standard drawing (drawing methods)

Standard drawing (control methods). The library also includes methods to change the scale and size of the canvas, the color and width of the lines, the text font, and the timing of drawing (for use in animation). As arguments for `setPenColor()` you can use one of the predefined colors BLACK, BLUE, CYAN, DARK_GRAY, GRAY, GREEN, LIGHT_GRAY, MAGENTA, ORANGE, PINK, RED, BOOK_RED, WHITE, and YELLOW that are defined as constants in `StdDraw` (so we refer to one of them with code like `StdDraw.RED`). The window also includes a menu option to save your drawing to a file, in a format suitable for publishing on the web.

```
public class StdDraw
{
    static void setXscale(double x0, double x1)           reset x range to (x0, x1)
    static void setYscale(double y0, double y1)           reset y range to (y0, y1)
    static void setPenRadius(double r)                   set pen radius to r
    static void setPenColor(Color c)                     set pen color to c
    static void setFont(Font f)                          set text font to f
    static void setCanvasSize(int w, int h)              set canvas to w-by-h window
    static void clear(Color c)                           clear the canvas; color it c
    static void show(int dt)                             show all; pause dt milliseconds
}
```

API for our library of static methods for standard drawing (control methods)

IN THIS BOOK, we use `StdDraw` for data analysis and for creating visual representations of algorithms in operation. The table on the opposite page indicates some possibilities; we will consider many more examples in the text and the exercises throughout the book. The library also supports animation—of course, this topic is treated primarily on the booksite.

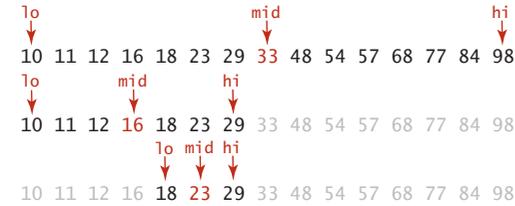
data	plot implementation (code fragment)	result
function values	<pre> int N = 100; StdDraw.setXscale(0, N); StdDraw.setYscale(0, N*N); StdDraw.setPenRadius(.01); for (int i = 1; i <= N; i++) { StdDraw.point(i, i); StdDraw.point(i, i*i); StdDraw.point(i, i*Math.log(i)); } </pre>	
array of random values	<pre> int N = 50; double[] a = new double[N]; for (int i = 0; i < N; i++) a[i] = StdRandom.random(); for (int i = 0; i < N; i++) { double x = 1.0*i/N; double y = a[i]/2.0; double rw = 0.5/N; double rh = a[i]/2.0; StdDraw.filledRectangle(x, y, rw, rh); } </pre>	
sorted array of random values	<pre> int N = 50; double[] a = new double[N]; for (int i = 0; i < N; i++) a[i] = StdRandom.random(); Arrays.sort(a); for (int i = 0; i < N; i++) { double x = 1.0*i/N; double y = a[i]/2.0; double rw = 0.5/N; double rh = a[i]/2.0; StdDraw.filledRectangle(x, y, rw, rh); } </pre>	

StdDraw plotting examples

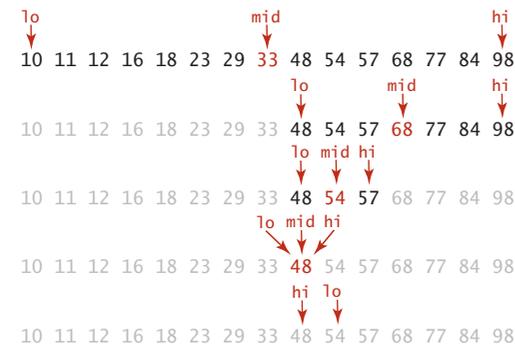
Binary search The sample Java program that we started with, shown on the facing page, is based on the famous, effective, and widely used *binary search* algorithm. This example is a prototype of the way in which we will examine new algorithms throughout the book. As with all of the programs we consider, it is both a precise definition of the method and a complete Java implementation that you can download from the booksite.

Binary search. We will study the binary search algorithm in detail in SECTION 3.2, but a brief description is appropriate here. The algorithm is implemented in the static

successful search for 23



unsuccessful search for 50



Binary search in an ordered array

Development client. For every algorithm implementation, we include a development client `main()` that you can use with sample input files provided in the book and on the booksite to learn about the algorithm and to test its performance. In this example, the client reads integers from the file named on the command line, then prints any integers to standard output that do not appear in the file. We use small test files such as those shown at right to demonstrate this behavior, and as the basis for traces and examples such as those at left above. We use large test files to model real-world applications and to test performance (see page 48).

method `rank()`, which takes an integer `key` and a sorted array of `int` values as arguments and returns the index of the key if it is present in the array, `-1` otherwise. It accomplishes this task by maintaining variables `lo` and `hi` such that the key is in `a[lo..hi]` if it is in the array, then entering into a loop that tests the middle entry in the interval (at index `mid`). If the key is equal to `a[mid]`, the return value is `mid`; otherwise the method cuts the interval size about in half, looking at the left half if the key is less than `a[mid]` and at the right half if the key is greater than `a[mid]`. The process terminates when the key is found or the interval is empty. Binary search is effective because it needs to examine just a few array entries (relative to the size of the array) to find the key (or determine that it is not there).

tinyW.txt	tinyT.txt
84	23
48	50
68	10
10	99
18	18
98	23
12	98
23	84
54	11
57	10
48	48
33	77
16	13
77	54
11	98
29	77
	77
	68

Small test files for BinarySearch test client

Binary Search

```
import java.util.Arrays;
public class BinarySearch
{
    public static int rank(int key, int[] a)
    { // Array must be sorted.
        int lo = 0;
        int hi = a.length - 1;
        while (lo <= hi)
        { // Key is in a[lo..hi] or not present.
            int mid = lo + (hi - lo) / 2;
            if (key < a[mid]) hi = mid - 1;
            else if (key > a[mid]) lo = mid + 1;
            else
                return mid;
        }
        return -1;
    }

    public static void main(String[] args)
    {
        int[] whitelist = In.readInts(args[0]);
        Arrays.sort(whitelist);
        while (!StdIn.isEmpty())
        { // Read key, print if not in whitelist.
            int key = StdIn.readInt();
            if (rank(key, whitelist) == -1)
                StdOut.println(key);
        }
    }
}
```

This program takes the name of a whitelist file (a sequence of integers) as argument and filters any entry that is on the whitelist from standard input, leaving only integers that are not on the whitelist on standard output. It uses the binary search algorithm, implemented in the static method `rank()`, to accomplish the task efficiently. See SECTION 3.1 for a full discussion of the binary search algorithm, its correctness, its performance analysis, and its applications.

```
% java BinarySearch tinyW.txt < tinyT.txt
50
99
13
```

Whitelisting. When possible, our development clients are intended to mirror practical situations and demonstrate the need for the algorithm at hand. In this case, the process is known as *whitelisting*. Specifically, imagine a credit card company that needs to check whether customer transactions are for a valid account. To do so, it can

- Keep customers account numbers in a file, which we refer to as a *whitelist*.
- Produce the account number associated with each transaction in the standard input stream.
- Use the test client to put onto standard output the numbers that are not associated with any customer. Presumably the company would refuse such transactions.

It would not be unusual for a big company with millions of customers to have to process millions of transactions or more. To model this situation, we provide on the book-site the files `largeW.txt` (1 million integers) and `largeT.txt` (10 million integers).

Performance. A working program is often not sufficient. For example, a much simpler implementation of `rank()`, which does not even require the array to be sorted, is to check every entry, as follows:

```
public static int rank(int key, int[] a)
{
    for (int i = 0; i < a.length; i++)
        if (a[i] == key) return i;
    return -1;
}
```

Given this simple and easy-to-understand solution, why do we use mergesort and binary search? If you work EXERCISE 1.1.38, you will see that your computer is too slow to run this brute-force implementation of `rank()` for large numbers of inputs (say, 1 million whitelist entries and 10 million transactions). *Solving the whitelist problem for a large number of inputs is not feasible without efficient algorithms such as binary search and mergesort.* Good performance is often of critical importance, so we lay the groundwork for studying performance in SECTION 1.4 and analyze the performance characteristics of all of our algorithms (including binary search, in SECTION 3.1 and mergesort, in SECTION 2.2).

IN THE PRESENT CONTEXT, our goal in thoroughly outlining our programming model is to ensure that you can run code like `BinarySearch` on your computer, use it on test data like ours, and modify it to adapt to various situations (such as those described in the exercises at the end of this section), in order to best understand its applicability. The programming model that we have sketched is designed to facilitate such activities, which are crucial to our approach to studying algorithms.

largeW.txt	largeT.txt
489910	944443
18940	293674
774392	572153
490636	600579
125544	499569
407391	984875
115771	763178
992663	295754
923282	44696
176914	207807
217904	138910
571222	903531
519039	140925
395667	699418
...	759984
↑	199694
1,000,000	774549
int values	635871
	161828
	805380
	...
	↑
	10,000,000
	int values

not in
largeW.txt

```
% java BinarySearch largeW.txt < largeT.txt
499569
984875
295754
207807
140925
161828
...
↑
367,966
int values
```

Large files for BinarySearch test client

Perspective In this section, we have described a fine and complete programming model that served (and still serves) many programmers for many decades. Modern programming, however, goes one step further. This next level is called data abstraction, sometimes known as object-oriented programming, and is the subject of the next section. Simply put, the idea behind data abstraction is to allow a program to define data types (sets of values and sets of operations on those values), not just static methods that operate on predefined data types.

Object-oriented programming has come into widespread use in recent decades, and data abstraction is central to modern program development. We embrace data abstraction in this book for three primary reasons:

- It enables us to expand our ability to reuse code through modular programming. For example, our sorts in CHAPTER 2 and binary search and other algorithms in CHAPTER 3 allow clients to make use of the same code for any type of data (not just integers), including one defined by the client.
- It provides a convenient mechanism for building so-called linked data structures that provide more flexibility than arrays and are the basis of efficient algorithms in many settings.
- It enables us to precisely define the algorithmic challenges that we face. For example, our union-find algorithms in SECTION 1.5, our priority-queue algorithms in SECTION 2.4, and our symbol-table algorithms in CHAPTER 3 are all oriented toward defining data structures that enable efficient implementations of a set of operations. This challenge aligns perfectly with data abstraction.

Despite all of these considerations, our focus remains on the study of algorithms. In this context, we proceed to consider next the essential features of object-oriented programming that are relevant to our mission.

Q&A

Q. What is Java bytecode?

A. A low-level version of your program that runs on the Java *virtual machine*. This level of abstraction makes it easier for the developers of Java to ensure that our programs run on a broad variety of devices.

Q. It seems wrong that Java should just let `ints` overflow and give bad values. Shouldn't Java automatically check for overflow?

A. This issue is a contentious one among programmers. The short answer is that the lack of such checking is one reason such types are called *primitive data types*. A little knowledge can go a long way in avoiding such problems. We use the `int` type for small numbers (less than ten decimal digits), and the `long` type when values run into the billions or more.

Q. What is the value of `Math.abs(-2147483648)`?

A. `-2147483648`. This strange (but true) result is a typical example of the effects of integer overflow.

Q. How can I initialize a `double` variable to infinity?

A. Java has built-in constants available for this purpose: `Double.POSITIVE_INFINITY` and `Double.NEGATIVE_INFINITY`.

Q. Can you compare a `double` to an `int`?

A. Not without doing a type conversion, but remember that Java usually does the requisite type conversion automatically. For example, if `x` is an `int` with the value 3, then the expression `(x < 3.1)` is `true`—Java converts `x` to `double` (because `3.1` is a `double` literal) before performing the comparison.

Q. What happens if I use a variable before initializing it to a value?

A. Java will report a compile-time error if there is any path through your code that would lead to use of an uninitialized variable.

Q. What are the values of `1/0` and `1.0/0.0` as Java expressions?

A. The first generates a runtime exception for division by zero (which stops your program because the value is undefined); the second has the value `Infinity`.

Q&A *(continued)*

Q. Can you use `<` and `>` to compare `String` variables?

A. No. Those operators are defined only for primitive types. See page 80.

Q. What is the result of division and remainder for negative integers?

A. The quotient `a/b` rounds toward 0; the remainder `a % b` is defined such that `(a / b) * b + a % b` is always equal to `a`. For example, `-14/3` and `14/-3` are both `-4`, but `-14 % 3` is `-2` and `14 % -3` is `2`.

Q. Why do we say `(a && b)` and not `(a & b)`?

A. The operators `&`, `|`, and `^` are bitwise logical operations for integer types that do and, or, and exclusive or (respectively) on each bit position. Thus the value of `10 & 6` is 2, the value of `10 | 6` is 14, and the value of `10 ^ 6` is 12. We use these operators rarely (but occasionally) in this book. The operators `&&` and `||` are valid only in boolean expressions; they differ from the operators `&` and `|` because of *short-circuiting*: an expression is evaluated left-to-right and the evaluation stops when the value is known.

Q. Is ambiguity in nested `if` statements a problem?

A. Yes. In Java, when you write

```
if <expr1> if <expr2> <stmtA> else <stmtB>
```

it is equivalent to

```
if <expr1> { if <expr2> <stmtA> else <stmtB> }
```

even if you might have been thinking

```
if <expr1> { if <expr2> <stmtA> } else <stmtB>
```

Using explicit braces is a good way to avoid this dangling else pitfall.

Q. What is the difference between a `for` loop and its `while` formulation?

A. The code in the `for` loop header is considered to be in the same block as the `for` loop body. In a typical `for` loop, the incrementing variable is not available for use in later statements; in the corresponding `while` loop, it is. This distinction is often a reason to use a `while` instead of a `for` loop.

Q. Some Java programmers use `int a[]` instead of `int[] a` to declare arrays. What's the difference?

A. In Java, both are legal and equivalent. The former is how arrays are declared in C. The latter is the preferred style in Java since the type of the variable `int[]` more clearly indicates that it is an array of integers.

Q. Why do array indices start at 0 instead of 1?

A. This convention originated with machine-language programming, where the address of an array element would be computed by adding the index to the address of the beginning of an array. Starting indices at 1 would entail either a waste of space at the beginning of the array or a waste of time to subtract the 1.

Q. If `a[]` is an array, why does `StdOut.println(a)` print out a hexadecimal integer, such as `@f62373`, instead of the elements of the array?

A. Good question. Typically, it prints the memory address of the array, which, unfortunately, is rarely what you want. Instead, you can first call `Arrays.toString(a)`.

Q. Why are we not using the standard Java libraries for input and graphics?

A. We are using them, but we prefer to work with simpler abstract models. The Java libraries behind `StdIn` and `StdDraw` are built for production programming, and the libraries and their APIs are a bit unwieldy. To get an idea of what they are like, look at the code in `StdIn.java` and `StdDraw.java`.

Q. Can my program reread data from standard input?

A. No. You only get one shot at it, in the same way that you cannot undo `println()`.

Q. What happens if my program attempts to read after standard input is exhausted?

A. You will get an error. `StdIn.isEmpty()` allows you to avoid such an error by checking whether there is more input available.

Q. What does this error message mean?

```
Exception in thread "main" java.lang.NoClassDefFoundError: StdIn
```

A. You probably forgot to put `StdIn.java` in your working directory.

Q. Can a static method take another static method as an argument in Java?

A. No. Good question, since many other languages do support this capability.

EXERCISES

1.1.1 Give the value of each of the following expressions:

- a. $(0 + 15) / 2$
- b. $2.0e-6 * 100000000.1$
- c. `true && false || true && true`

1.1.2 Give the type and value of each of the following expressions:

- a. $(1 + 2.236)/2$
- b. $1 + 2 + 3 + 4.0$
- c. $4.1 >= 4$
- d. $1 + 2 + "3"$

1.1.3 Write a program that takes three integer command-line arguments and prints `equal` if all three are equal, and `not equal` otherwise.

1.1.4 What (if anything) is wrong with each of the following statements?

- a. `if (a > b) then c = 0;`
- b. `if a > b { c = 0; }`
- c. `if (a > b) c = 0;`
- d. `if (a > b) c = 0 else b = 0;`

1.1.5 Write a code fragment that prints `true` if the double variables `x` and `y` are both strictly between 0 and 1 and `false` otherwise.

1.1.6 What does the following program print?

```
int f = 0;
int g = 1;
for (int i = 0; i <= 15; i++)
{
    StdOut.println(f);
    f = f + g;
    g = f - g;
}
```

1.1.7 Give the value printed by each of the following code fragments:

- a.

```
double t = 9.0;
while (Math.abs(t - 9.0/t) > .001)
    t = (9.0/t + t) / 2.0;
StdOut.printf("%.5f\n", t);
```
- b.

```
int sum = 0;
for (int i = 1; i < 1000; i++)
    for (int j = 0; j < i; j++)
        sum++;
StdOut.println(sum);
```
- c.

```
int sum = 0;
for (int i = 1; i < 1000; i *= 2)
    for (int j = 0; j < 1000; j++)
        sum++;
StdOut.println(sum);
```

1.1.8 What do each of the following print?

- a. `System.out.println('b');`
- b. `System.out.println('b' + 'c');`
- c. `System.out.println((char) ('a' + 4));`

Explain each outcome.

1.1.9 Write a code fragment that puts the binary representation of a positive integer *N* into a `String` *s*.

Solution: Java has a built-in method `Integer.toBinaryString(N)` for this job, but the point of the exercise is to see how such a method might be implemented. Here is a particularly concise solution:

```
String s = "";
for (int n = N; n > 0; n /= 2)
    s = (n % 2) + s;
```

EXERCISES *(continued)*

1.1.10 What is wrong with the following code fragment?

```
int[] a;  
for (int i = 0; i < 10; i++)  
    a[i] = i * i;
```

Solution: It does not allocate memory for `a[]` with `new`. This code results in a variable `a` might not have been initialized compile-time error.

1.1.11 Write a code fragment that prints the contents of a two-dimensional boolean array, using `*` to represent `true` and a space to represent `false`. Include row and column numbers.

1.1.12 What does the following code fragment print?

```
int[] a = new int[10];  
for (int i = 0; i < 10; i++)  
    a[i] = 9 - i;  
for (int i = 0; i < 10; i++)  
    a[i] = a[a[i]];  
for (int i = 0; i < 10; i++)  
    System.out.println(a[i]);
```

1.1.13 Write a code fragment to print the transposition (rows and columns changed) of a two-dimensional array with `M` rows and `N` columns.

1.1.14 Write a static method `lg()` that takes an `int` value `N` as argument and returns the largest `int` not larger than the base-2 logarithm of `N`. Do not use `Math`.

1.1.15 Write a static method `histogram()` that takes an array `a[]` of `int` values and an integer `M` as arguments and returns an array of length `M` whose `i`th entry is the number of times the integer `i` appeared in the argument array. If the values in `a[]` are all between 0 and `M-1`, the sum of the values in the returned array should be equal to `a.length`.

1.1.16 Give the value of `exR1(6)`:

```
public static String exR1(int n)  
{  
    if (n <= 0) return "";  
    return exR1(n-3) + n + exR1(n-2) + n;  
}
```

1.1.17 Criticize the following recursive function:

```
public static String exR2(int n)
{
    String s = exR2(n-3) + n + exR2(n-2) + n;
    if (n <= 0) return "";
    return s;
}
```

Answer: The base case will never be reached. A call to `exR2(3)` will result in calls to `exR2(0)`, `exR2(-3)`, `exR2(-6)`, and so forth until a `StackOverflowError` occurs.

1.1.18 Consider the following recursive function:

```
public static int mystery(int a, int b)
{
    if (b == 0) return 0;
    if (b % 2 == 0) return mystery(a+a, b/2);
    return mystery(a+a, b/2) + a;
}
```

What are the values of `mystery(2, 25)` and `mystery(3, 11)`? Given positive integers `a` and `b`, describe what value `mystery(a, b)` computes. Answer the same question, but replace the three `+` operators with `*` and replace `return 0` with `return 1`.

1.1.19 Run the following program on your computer:

```
public class Fibonacci
{
    public static long F(int N)
    {
        if (N == 0) return 0;
        if (N == 1) return 1;
        return F(N-1) + F(N-2);
    }

    public static void main(String[] args)
    {
        for (int N = 0; N < 100; N++)
            StdOut.println(N + " " + F(N));
    }
}
```

EXERCISES *(continued)*

What is the largest value of N for which this program takes less than 1 hour to compute the value of $F(N)$? Develop a better implementation of $F(N)$ that saves computed values in an array.

1.1.20 Write a recursive static method that computes the value of $\ln(N!)$.

1.1.21 Write a program that reads in lines from standard input with each line containing a name and two integers and then uses `printf()` to print a table with a column of the names, the integers, and the result of dividing the first by the second, accurate to three decimal places. You could use a program like this to tabulate batting averages for baseball players or grades for students.

1.1.22 Write a version of `BinarySearch` that uses the recursive `rank()` given on page 25 and traces the method calls. Each time the recursive method is called, print the argument values `lo` and `hi`, indented by the depth of the recursion. *Hint:* Add an argument to the recursive method that keeps track of the depth.

1.1.23 Add to the `BinarySearch` test client the ability to respond to a second argument: `+` to print numbers from standard input that are not in the whitelist, `-` to print numbers that are in the whitelist.

1.1.24 Give the sequence of values of p and q that are computed when Euclid's algorithm is used to compute the greatest common divisor of 105 and 24. Extend the code given on page 4 to develop a program `Euclid` that takes two integers from the command line and computes their greatest common divisor, printing out the two arguments for each call on the recursive method. Use your program to compute the greatest common divisor of 1111111 and 1234567.

1.1.25 Use mathematical induction to prove that Euclid's algorithm computes the greatest common divisor of any pair of nonnegative integers p and q .

CREATIVE PROBLEMS

1.1.26 *Sorting three numbers.* Suppose that the variables `a`, `b`, `c`, and `t` are all of the same numeric primitive type. Show that the following code puts `a`, `b`, and `c` in ascending order:

```
if (a > b) { t = a; a = b; b = t; }
if (a > c) { t = a; a = c; c = t; }
if (b > c) { t = b; b = c; c = t; }
```

1.1.27 *Binomial distribution.* Estimate the number of recursive calls that would be used by the code

```
public static double binomial(int N, int k, double p)
{
    if ((N == 0) && (k == 0)) return 1.0;
    if ((N < 0) || (k < 0)) return 0.0;
    return (1 - p)*binomial(N-1, k, p) + p*binomial(N-1, k-1, p);
}
```

to compute `binomial(100, 50, 0.25)`. Develop a better implementation that is based on saving computed values in an array.

1.1.28 *Remove duplicates.* Modify the test client in `BinarySearch` to remove any duplicate keys in the whitelist after the sort.

1.1.29 *Equal keys.* Add to `BinarySearch` a static method `rank()` that takes a key and a sorted array of `int` values (some of which may be equal) as arguments and returns the number of elements that are smaller than the key and a similar method `count()` that returns the number of elements equal to the key. Note: If `i` and `j` are the values returned by `rank(key, a)` and `count(key, a)` respectively, then `a[i .. i+j-1]` are the values in the array that are equal to key.

1.1.30 *Array exercise.* Write a code fragment that creates an N -by- N boolean array `a[][]` such that `a[i][j]` is `true` if `i` and `j` are relatively prime (have no common factors), and `false` otherwise.

1.1.31 *Random connections.* Write a program that takes as command-line arguments an integer `N` and a `double` value `p` (between 0 and 1), plots N equally spaced dots of size .05 on the circumference of a circle, and then, with probability `p` for each pair of points, draws a gray line connecting them.

CREATIVE PROBLEMS *(continued)*

1.1.32 Histogram. Suppose that the standard input stream is a sequence of `double` values. Write a program that takes an integer N and two `double` values l and r from the command line and uses `StdDraw` to plot a histogram of the count of the numbers in the standard input stream that fall in each of the N intervals defined by dividing (l, r) into N equal-sized intervals.

1.1.33 Matrix library. Write a library `Matrix` that implements the following API:

```
public class Matrix
{
    static double dot(double[] x, double[] y)           vector dot product
    static double[][] mult(double[][] a, double[][] b)  matrix-matrix product
    static double[][] transpose(double[][] a)          transpose
    static double[] mult(double[][] a, double[] x)     matrix-vector product
    static double[] mult(double[] y, double[][] a)     vector-matrix product
}
```

Develop a test client that reads values from standard input and tests all the methods.

1.1.34 Filtering. Which of the following *require* saving all the values from standard input (in an array, say), and which could be implemented as a filter using only a fixed number of variables and arrays of fixed size (not dependent on N)? For each, the input comes from standard input and consists of N real numbers between 0 and 1.

- Print the maximum and minimum numbers.
- Print the median of the numbers.
- Print the k th smallest value, for k less than 100.
- Print the sum of the squares of the numbers.
- Print the average of the N numbers.
- Print the percentage of numbers greater than the average.
- Print the N numbers in increasing order.
- Print the N numbers in random order.

EXPERIMENTS

1.1.35 *Dice simulation.* The following code computes the exact probability distribution for the sum of two dice:

```
int SIDES = 6;
double[] dist = new double[2*SIDES+1];
for (int i = 1; i <= SIDES; i++)
    for (int j = 1; j <= SIDES; j++)
        dist[i+j] += 1.0;

for (int k = 2; k <= 2*SIDES; k++)
    dist[k] /= 36.0;
```

The value `dist[k]` is the probability that the dice sum to `k`. Run experiments to validate this calculation simulating N dice throws, keeping track of the frequencies of occurrence of each value when you compute the sum of two random integers between 1 and 6. How large does N have to be before your empirical results match the exact results to three decimal places?

1.1.36 *Empirical shuffle check.* Run computational experiments to check that our shuffling code on page 32 works as advertised. Write a program `ShuffleTest` that takes command-line arguments M and N , does N shuffles of an array of size M that is initialized with `a[i] = i` before each shuffle, and prints an M -by- M table such that row i gives the number of times i wound up in position j for all j . All entries in the table should be close to N/M .

1.1.37 *Bad shuffling.* Suppose that you choose a random integer between 0 and $N-1$ in our shuffling code instead of one between i and $N-1$. Show that the resulting order is not equally likely to be one of the $N!$ possibilities. Run the test of the previous exercise for this version.

1.1.38 *Binary search versus brute-force search.* Write a program `BruteForceSearch` that uses the brute-force search method given on page 48 and compare its running time on your computer with that of `BinarySearch` for `largeW.txt` and `largeT.txt`.

EXPERIMENTS *(continued)*

1.1.39 *Random matches.* Write a `BinarySearch` client that takes an `int` value `T` as command-line argument and runs `T` trials of the following experiment for $N = 10^3$, 10^4 , 10^5 , and 10^6 : generate two arrays of `N` randomly generated positive six-digit `int` values, and find the number of values that appear in both arrays. Print a table giving the average value of this quantity over the `T` trials for each value of `N`.

This page intentionally left blank



1.2 DATA ABSTRACTION

A DATA TYPE is a set of values and a set of operations on those values. So far, we have discussed in detail Java's primitive data types: for example, the values of the primitive data type `int` are integers between -2^{31} and $2^{31} - 1$; the operations of `int` include `+`, `*`, `-`, `/`, `%`, `<`, and `>`. In principle, we could write all of our programs using only the built-in primitive types, but it is much more convenient to write programs at a higher level of abstraction. In this section, we focus on the process of defining and using data types, which is known as data abstraction (and supplements the function abstraction style that is the basis of SECTION 1.1).

Programming in Java is largely based on building data types known as reference types with the familiar Java `class`. This style of programming is known as object-oriented programming, as it revolves around the concept of an object, an entity that holds a data type value. With Java's primitive types we are largely confined to programs that operate on numbers, but with reference types we can write programs that operate on strings, pictures, sounds, any of hundreds of other abstractions that are available in Java's standard libraries or on our booksite. Even more significant than libraries of predefined data types is that the range of data types available in Java programming is open-ended, because *you can define your own data types* to implement any abstraction whatsoever.

An abstract data type (ADT) is a data type whose representation is hidden from the client. Implementing an ADT as a Java class is not very different from implementing a function library as a set of static methods. The primary difference is that we associate data with the function implementations and we hide the representation of the data from the client. When using an ADT, we focus on the operations specified in the API and pay no attention to the data representation; when implementing an ADT, we focus on the data, then implement operations on that data.

Abstract data types are important because they support encapsulation in program design. In this book, we use them as a means to

- Precisely specify problems in the form of APIs for use by diverse clients
- Describe algorithms and data structures as API implementations

Our primary reason for studying different algorithms for the same task is that performance characteristics differ. Abstract data types are an appropriate framework for the study of algorithms because they allow us to put knowledge of algorithm performance to immediate use: we can substitute one algorithm for another to improve performance for all clients without changing any client code.

Using abstract data types *You do not need to know how a data type is implemented in order to be able to use it, so we begin by describing how to write programs that use a simple data type named Counter whose values are a name and a nonnegative integer and whose operations are *create and initialize to zero*, *increment by one*, and *examine the current value*. This abstraction is useful in many contexts. For example, it would be reasonable to use such a data type in electronic voting software, to ensure that the only thing that a voter can do is increment a chosen candidate's tally by one. Or, we might use a Counter to keep track of fundamental operations when analyzing the performance of algorithms. To use a Counter, you need to learn our mechanism for specifying the operations defined in the data type and the Java language mechanisms for creating and manipulating data-type values. Such mechanisms are critically important in modern programming, and we use them throughout this book, so this first example is worthy of careful attention.*

API for an abstract data type. To specify the behavior of an abstract data type, we use an application programming interface (API), which is a list of constructors and instance *methods* (operations), with an informal description of the effect of each, as in this API for Counter:

```
public class Counter
    Counter(String id)      create a counter named id
    void increment()       increment the counter by one
    int tally()            number of increments since creation
    String toString()      string representation
```

An API for a counter

Even though the basis of a data-type definition is a set of values, the role of the values is not visible from the API, only the operations on those values. Accordingly, an ADT definition has many similarities with a library of static methods (see page 24):

- Both are implemented as a Java class.
- Instance methods may take zero or more arguments of a specified type, separated by commas and enclosed in parentheses.
- They may provide a return value of a specified type or no return value (signified by void).

And there are three significant differences:

- Some entries in the API have the same name as the class and lack a return type. Such entries are known as constructors and play a special role. In this case, Counter has a constructor that takes a String argument.

- Instance methods lack the `static` modifier. They are not static methods—their purpose is to operate on data type values.
- Some instance methods are present so as to adhere to Java conventions—we refer to such methods as *inherited methods* and shade them gray in the API.

As with APIs for libraries of static methods, an API for an abstract data type is a contract with all clients and, therefore, the starting point both for developing any client code and for developing any data-type implementation. In this case, the API tells us that to use `Counter`, we have available the `Counter()` constructor, the `increment()` and `tally()` instance methods, and the inherited `toString()` method.

Inherited methods. Various Java conventions enable a data type to take advantage of built-in language mechanisms by including specific methods in the API. For example, all Java data types *inherit* a `toString()` method that typically returns a `String` representation of the data-type values. Java calls this method when any data-type value is to be concatenated with a `String` value with the `+` operator. The default implementation is not particularly useful (it gives a string representation of the memory address of the data-type value), so we often provide an implementation that overrides the default, and include `toString()` in the API whenever we do so. Other examples of such methods include `equals()`, `compareTo()`, and `hashCode()` (see page 101).

Client code. As with modular programming based on static methods, the API allows us to write client code without knowing details of the implementation (and to write implementation code without knowing details of any particular client). The mechanisms introduced on page 28 for organizing programs as independent modules are useful for all Java classes, and thus are effective for modular programming with ADTs as well as for libraries of static methods. Accordingly, we can use an ADT in any program provided that the source code is in a `.java` file in the same directory, or in the standard Java library, or accessible through an `import` statement, or through one of the classpath mechanisms described on the booksite. All of the benefits of modular programming follow. By encapsulating all the code that implements a data type within a single Java class, we enable the development of client code at a higher level of abstraction. To develop client code, you need to be able to declare variables, create objects to hold data-type values, and provide access to the values for instance methods to operate on them. These processes are different from the corresponding processes for primitive types, though you will notice many similarities.

Objects. Naturally, you can declare that a variable `heads` is to be associated with data of type `Counter` with the code

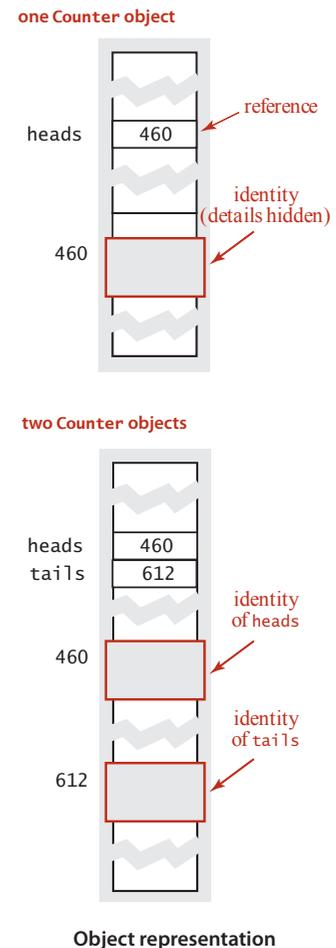
```
Counter heads;
```

but how can you assign values or specify operations? The answer to this question involves a fundamental concept in data abstraction: an object is an entity that can take on a data-type value. Objects are characterized by three essential properties: state, identity, and *behavior*. The state of an object is a value from its data type. The identity of an object distinguishes one object from another. It is useful to think of an object's identity as the place where its value is stored in memory. The *behavior* of an object is the effect of data-type operations. The implementation has the sole responsibility for maintaining an object's identity, so that client code can use a data type without regard to the representation of its state by conforming to an API that describes an object's behavior. An object's state might be used to provide information to a client or cause a side effect or be changed by one of its data type's operations, but the details of the representation of the data-type value are not relevant to client code. A reference is a mechanism for accessing an object. Java nomenclature makes clear the distinction from primitive types (where variables are associated with values) by using the term reference types for nonprimitive types. The details of implementing references vary in Java implementations, but it is useful to think of a reference as a memory address, as shown at right (for brevity, we use three-digit memory addresses in the diagram).

Creating objects. Each data-type value is stored in an object. To create (or instantiate) an individual object, we invoke a constructor by using the keyword `new`, followed by the class name, followed by `()` (or a list of argument values enclosed in parentheses, if the constructor takes arguments). A constructor has no return type because it always returns a reference to an object of its data type. Each time that a client uses `new()`, the system

- Allocates memory space for the object
- Invokes the constructor to initialize its value
- Returns a reference to the object

In client code we typically create objects in an initializing declaration that associates a variable with the object, as we often do with variables of primitive types. Unlike primitive types, variables are associated with references to objects, not the data-type values



themselves. We can create any number of objects from the same class—each object has its own identity and may or may not store the same value as another object of the same type. For example, the code

```
Counter heads = new Counter("heads");
Counter tails = new Counter("tails");
```

creates two different `Counter` objects. In an abstract data type, details of the representation of the value are hidden from client code. You might assume that the value associated with each `Counter` object is a `String` name and an `int` tally, but you cannot write *code that depends on any specific representation* (or even know whether that assumption is true—perhaps the tally is a `long` value).

Invoking instance methods. The purpose of an instance method is to operate on data-type values, so the Java language includes a special mechanism to invoke instance methods that emphasizes a connection to an object. Specifically, we invoke an instance method

declaration to associate variable with object reference
↓
Counter heads = invoke constructor to create an object
↓
new Counter("heads");;
Creating an object

`Counter heads;` ← declaration

with new (constructor)

```
heads = new Counter ("heads");
```

↑
invoke a constructor (create an object)

as a statement (void return value)

```
heads.increment();
```

↑ object name
↑ invoke an instance method that changes the object's value

as an expression

```
heads.tally() - tails.tally()
```

↑ object name
↑ invoke an instance method that accesses the object's value

via automatic type conversion (toString)

```
StdOut.println( heads );
```

↑
invoke heads.toString()

Invoking instance methods

by writing a variable name that refers to an object, followed by a period, followed by an instance method name, followed by 0 or more arguments, enclosed in parentheses and separated by commas. An instance method might *change* the data-type value or just examine the data-type value. Instance methods have all of the properties of static methods that we considered on page 24—arguments are passed by value, method names can be overloaded, they may have a return value, and they may cause side effects—but they have an additional property that characterizes them: *each invocation is associated with an object*. For example, the code

```
heads.increment();
```

invokes the instance method `increment()` to operate on the `Counter` object `heads` (in this case the operation involves incrementing the tally), and the code

```
heads.tally() - tails.tally();
```

invokes the instance method `tally()` twice, first to operate on the `Counter` object `heads` and then to operate on the `Counter` object `tails` (in this case the

operation involves returning the tally as an `int` value). As these examples illustrate, you can use calls on instance methods in client code in the same way as you use calls on static methods—as statements (`void` methods) or values in expressions (methods that return a value). The primary purpose of static methods is to implement functions; the primary purpose of non-static (instance) methods is to implement data-type operations. Either type of method may appear in client code, but you can easily distinguish between them, because a static method call starts with a class name (uppercase, by convention) and a non-static method call always starts with an object name (lowercase, by convention). These differences are summarized in the table at right.

	instance method	static method
<i>sample call</i>	<code>heads.increment()</code>	<code>Math.sqrt(2.0)</code>
<i>invoked with</i>	object name	class name
<i>parameters</i>	reference to object and argument(s)	argument(s)
<i>primary purpose</i>	examine or change object value	compute return value

Instance methods versus static methods

Using objects. Declarations give us variable names for objects that we can use in code not just to create objects and invoke instance methods, but also in the same way as we use variable names for integers, floating-point numbers, and other primitive types. To develop client code for a given data type, we:

- Declare variables of the type, for use in referring to objects
- Use the keyword `new` to invoke a constructor that creates objects of the type
- Use the object name to invoke instance methods, either as statements or within expressions

For example, the class `Flips` shown at the top of the next page is a `Counter` client that takes a command-line argument `T` and simulates `T` coin flips (it is also a `StdRandom` client). Beyond these direct uses, we can use variables associated with objects in the same way as we use variables associated with primitive-type values:

- In assignment statements
- To pass or return objects from methods
- To create and use arrays of object.

Understanding the behavior of each of these types of uses requires thinking in terms of references, not values, as you will see when we consider them, in turn.

Assignment statements. An assignment statement with a reference type creates a copy of the reference. The assignment statement does not create a new object, just another reference to an existing object. This situation is known as aliasing: both variables refer to the same object. The effect of aliasing is a bit unexpected, because it is different for variables holding values of a primitive type. Be sure that you understand the difference.

```

public class Flips
{
    public static void main(String[] args)
    {
        int T = Integer.parseInt(args[0]);
        Counter heads = new Counter("heads");
        Counter tails = new Counter("tails");
        for (int t = 0; t < T; t++)
            if (StdRandom.bernoulli(0.5))
                heads.increment();
            else tails.increment();
        StdOut.println(heads);
        StdOut.println(tails);
        int d = heads.tally() - tails.tally();
        StdOut.println("delta: " + Math.abs(d));
    }
}

```

```

% java Flips 10
5 heads
5 tails
delta: 0

% java Flips 10
8 heads
2 tails
delta: 6

% java Flips 1000000
499710 heads
500290 tails
delta: 580

```

Counter client that simulates T coin flips

If x and y are variables of a primitive type, then the assignment $x = y$ copies the value of y to x . For reference types, the reference is copied (not the value). Aliasing is a common source of bugs in Java programs, as illustrated by the following example:

```

Counter c1 = new Counter("ones");
c1.increment();
Counter c2 = c1;
c2.increment();
StdOut.println(c1);

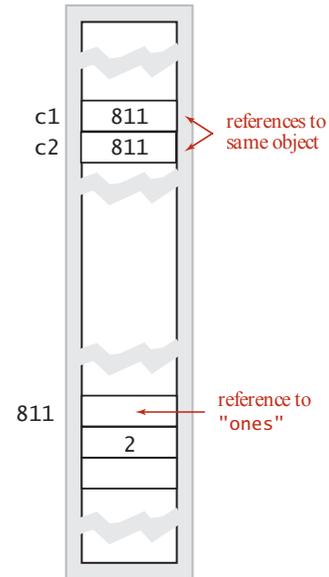
```

With a typical `toString()` implementation this code would print the string "2 ones" which may or may not be what was intended and is counterintuitive at first. Such bugs are common in programs written by people without much experience in using objects (that may be you, so pay attention here!). Changing the state of an object impacts all code involving aliased variables referencing that object. We are used to thinking of two different variables of primitive types as being independent, but that intuition does not carry over to variables of reference types.

```

Counter c1;
c1 = new Counter("ones");
c1.increment();
Counter c2 = c1;
c2.increment();

```



Aliasing

Objects as arguments. You can pass objects as arguments to methods. This ability typically simplifies client code. For example, when we use a `Counter` as an argument, we are essentially passing both a name and a tally, but need only specify one variable. When we call a method with arguments, the effect in Java is as if each argument value were to appear on the right-hand side of an assignment statement with the corresponding argument name on the left. That is, Java passes a copy of the argument value from the calling program to the method. This arrangement is known as pass by value (see page 24). One important consequence is that the method cannot change the value of a caller's variable. For primitive types, this policy is what we expect (the two variables are independent), but each time that we use a reference type as a method argument we create an alias, so we must be cautious. In other words, the convention is to pass the reference by value (make a copy of it) but to pass the object by reference. For example, if we pass a reference to an object of type `Counter`, the method cannot change the original reference (make it point to a different `Counter`), but it can change the value of the object, for example by using the reference to call `increment()`.

Objects as return values. Naturally, you can also use an object as a return value from a method. The method might return an object passed to it as an argument, as in the example below, or it might create an object and return a reference to it. This capability is important because Java methods allow only one return value—using objects enables us to write code that, in effect, returns multiple values.

```
% java FlipsMax 1000000
500281 tails wins
```

```
public class FlipsMax
{
    public static Counter max(Counter x, Counter y)
    {
        if (x.tally() > y.tally()) return x;
        else return y;
    }

    public static void main(String[] args)
    {
        int T = Integer.parseInt(args[0]);
        Counter heads = new Counter("heads");
        Counter tails = new Counter("tails");
        for (int t = 0; t < T; t++)
            if (StdRandom.bernoulli(0.5))
                heads.increment();
            else tails.increment();

        if (heads.tally() == tails.tally())
            StdOut.println("Tie");
        else StdOut.println(max(heads, tails) + " wins");
    }
}
```

Example of a static method with object arguments and return values

Arrays are objects. In Java, every value of any nonprimitive type is an object. In particular, arrays are objects. As with strings, there is special language support for certain operations on arrays: declarations, initialization, and indexing. As with any other object, when we pass an array to a method or use an array variable on the right hand side of an assignment statement, we are making a copy of the array reference, not a copy of the array. This convention is appropriate for the typical case where we expect the method to be able to modify the array, by rearranging its entries, as, for example, in `java.util.Arrays.sort()` or the `shuffle()` method that we considered on page 32.

Arrays of objects. Array entries can be of any type, as we have already seen: `args[]` in our `main()` implementations is an array of `String` objects. When we create an array of objects, we do so in two steps:

- Create the array, using the bracket syntax for array constructors.
- Create each object in the array, using a standard constructor for each.

For example, the code below simulates rolling a die, using an array of `Counter` objects to keep track of the number of occurrences of each possible value. An array of objects in Java is an array of references to objects, not the objects themselves. If the objects are large, then we may gain efficiency by not having to move them around, just their references. If they are small, we may lose efficiency by having to follow a reference each time we need to get to some information.

```
public class Rolls
{
    public static void main(String[] args)
    {
        int T = Integer.parseInt(args[0]);
        int SIDES = 6;
        Counter[] rolls = new Counter[SIDES+1];
        for (int i = 1; i <= SIDES; i++)
            rolls[i] = new Counter(i + "'s");

        for (int t = 0; t < T; t++)
        {
            int result = StdRandom.uniform(1, SIDES+1);
            rolls[result].increment();
        }
        for (int i = 1; i <= SIDES; i++)
            StdOut.println(rolls[i]);
    }
}
```

Counter client that simulates T rolls of a die

```
% java Rolls 1000000
167308 1's
166540 2's
166087 3's
167051 4's
166422 5's
166592 6's
```

WITH THIS FOCUS ON OBJECTS, writing code that embraces data abstraction (defining and using data types, with data-type values held in objects) is widely referred to as object-oriented programming. The basic concepts that we have just covered are the starting point for object-oriented programming, so it is worthwhile to briefly summarize them. A data type is a set of values and a set of operations defined on those values. We implement data types in independent Java `class` modules and write client programs that use them. An object is an entity that can take on a data-type value or an instance of a data type. Objects are characterized by three essential properties: state, identity, and *behavior*. A data-type implementation supports clients of the data type as follows:

- Client code can create objects (establish identity) by using the `new` construct to invoke a constructor that creates an object, initializes its instance variables, and returns a reference to that object.
- Client code can manipulate data-type values (control an object's behavior, possibly changing its state) by using a variable associated with an object to invoke an instance method that operates on that object's instance variables.
- Client code can manipulate objects by creating arrays of objects and passing them and returning them to methods, in the same way as for primitive-type values, except that variables refer to references to values, not the values themselves.

These capabilities are the foundation of a flexible, modern, and widely useful programming style that we will use as the basis for studying algorithms in this book.

Examples of abstract data types The Java language has thousands of built-in ADTs, and we have defined many other ADTs to facilitate the study of algorithms. Indeed, every Java program that we write is a data-type implementation (or a library of static methods). To control complexity, we will specifically cite APIs for any ADT that we use in this book (not many, actually).

In this section, we introduce as examples several data types, with some examples of client code. In some cases, we present excerpts of APIs that may contain dozens of instance methods or more. We articulate these APIs to present real-world examples, to specify the instance methods that we will use in the book, and to emphasize that you do not need to know the details of an ADT implementation in order to be able to use it.

For reference, the data types that we use and develop in this book are shown on the facing page. These fall into several different categories:

- Standard system ADTs in `java.lang.*`, which can be used in any Java program.
- Java ADTs in libraries such as `java.awt`, `java.net`, and `java.io`, which can also be used in any Java program, but need an `import` statement.
- Our I/O ADTs that allow us to work with multiple input/output streams similar to `StdIn` and `StdOut`.
- Data-oriented ADTs whose primary purpose is to facilitate organizing and processing data by encapsulating the representation. We describe several examples for applications in computational geometry and information processing later in this section and use them as examples in client code later on.
- Collection ADTs whose primary purpose is to facilitate manipulating collections of data of the same type. We describe the basic `Bag`, `Stack`, and `Queue` types in SECTION 1.3, PQ types in CHAPTER 2, and the `ST` and `SET` types in CHAPTERS 3 and 5.
- Operations-oriented ADTs that we use to analyze algorithms, as described in SECTION 1.4 and SECTION 1.5.
- ADTs for graph algorithms, including both data-oriented ADTs that focus on encapsulating representations of various kinds of graphs and operations-oriented ADTs that focus on providing specifications for graph-processing algorithms.

This list does not include the dozens of types that we consider in exercises, which may be found in the index. Also, as described on page 90, we often distinguish multiple implementations of various ADTs with a descriptive prefix. As a group, the ADTs that we use demonstrate that organizing and understanding the data types that you use is an important factor in modern programming.

A typical application might use only five to ten of these ADTs. A prime goal in the development and organization of the ADTs in this book is to enable programmers to easily take advantage of a relatively small set of them in developing client code.

standard Java system types in java.lang

Integer	int wrapper
Double	double wrapper
String	indexed chars
StringBuilder	builder for strings

other Java types

java.awt.Color	colors
java.awt.Font	fonts
java.net.URL	URLs
java.io.File	files

our standard I/O types

In	input stream
Out	output stream
Draw	drawing

data-oriented types for client examples

Point2D	point in the plane
Interval1D	1D interval
Interval2D	2D interval
Date	date
Transaction	transaction

types for the analysis of algorithms

Counter	counter
Accumulator	accumulator
VisualAccumulator	visual version
Stopwatch	stopwatch

collection types

Stack	pushdown stack
Queue	FIFO queue
Bag	bag
MinPQ MaxPQ	priority queue
IndexMinPQ IndexMaxPQ	priority queue (indexed)
ST	symbol table
SET	set
StringST	symbol table (string keys)

data-oriented graph types

Graph	graph
Digraph	directed graph
Edge	edge (weighted)
EdgeWeightedGraph	graph (weighted)
DirectedEdge	edge (directed, weighted)
EdgeWeightedDigraph	graph (directed, weighted)

operations-oriented graph types

UF	dynamic connectivity
DepthFirstPaths	DFS path search
CC	connected components
BreadthFirstPaths	BFS path search
DirectedDFS	DFS digraph path search
DirectedBFS	BFS digraph path search
TransitiveClosure	all paths
Topological	topological order
DepthFirstOrder	DFS order
DirectedCycle	cycle search
SCC	strong components
MST	minimum spanning tree
SP	shortest paths

Selected ADTs used in this book

Geometric objects. A natural example of object-oriented programming is designing data types for geometric objects. For example, the APIs on the facing page define abstract

data types for three familiar geometric objects: `Point2D` (points in the plane), `Interval1D` (intervals on the line), and `Interval2D` (two-dimensional intervals in the plane, or axis-aligned rectangles). As usual, the APIs are essentially self-documenting and lead immediately to easily understood client code such as the example at left, which reads the boundaries of an `Interval2D` and an integer T from the command line, generates T random points in the unit square, and counts the number of points that fall in the interval (an estimate of the area of the rectangle). For dramatic effect, the client also draws the interval and the points that fall outside the interval. This computation is a model for a method that reduces the problem of computing the area and volume of geometric shapes to the problem of determining

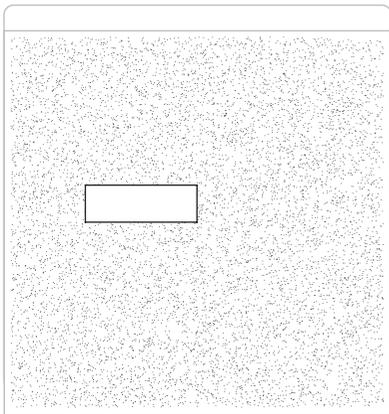
```
public static void main(String[] args)
{
    double xlo = Double.parseDouble(args[0]);
    double xhi = Double.parseDouble(args[1]);
    double ylo = Double.parseDouble(args[2]);
    double yhi = Double.parseDouble(args[3]);
    int T = Integer.parseInt(args[4]);

    Interval1D xint = new Interval1D(xlo, xhi);
    Interval1D yint = new Interval1D(ylo, yhi);
    Interval2D box = new Interval2D(xint, yint);
    box.draw();

    Counter c = new Counter("hits");
    for (int t = 0; t < T; t++)
    {
        double x = StdRandom.random();
        double y = StdRandom.random();
        Point2D p = new Point2D(x, y);
        if (box.contains(p)) c.increment();
        else
            p.draw();
    }

    StdOut.println(c);
    StdOut.printf("area = %.2f\n", box.area());
}
```

Interval2D test client



```
% java Interval2D .2 .5 .5 .6 10000
297 hits
area = .03
```

whether a point falls within the shape or not (a less difficult but not trivial problem). Of course, we can define APIs for other geometric objects such as line segments, triangles, polygons, circles, and so forth, though implementing operations on them can be challenging. Several examples are addressed in the exercises at the end of this section.

PROGRAMS THAT PROCESS GEOMETRIC OBJECTS have wide application in computing with models of the natural world, in scientific computing, video games, movies, and many other applications. The development and study of such programs and applications has blossomed into a far-reaching field of study known as computational geometry, which is a

```
public class Point2D
```

```

    Point2D(double x, double y)  create a point
double  x()                    x coordinate
double  y()                    y coordinate
double  r()                    radius (polar coordinates)
double  theta()                angle (polar coordinates)
double  distanceTo(Point2D that)  Euclidean distance from this point to that
void    draw()                 draw the point on StdDraw

```

An API for points in the plane

```
public class Interval1D
```

```

    Interval1D(double left, double right)  create an interval
double  left()                            left endpoint
double  right()                           right endpoint
double  length()                          length of the interval
boolean contains(double x)                does the interval contain x?
boolean intersects(Interval1D that)       does the interval intersect that?

```

An API for intervals on the line

```
public class Interval2D
```

```

    Interval2D(Interval1D x, Interval1D y)  create a 2D interval
double  area()                             area of the 2D interval
boolean contains(Point2D p)                does the 2D interval contain p?
boolean intersects(Interval2D that)        does the 2D interval intersect that?
void    draw()                             draw the 2D interval on StdDraw

```

An API for two dimensional intervals in the plane

fertile area of examples for the application of the algorithms that we address in this book, as you will see in examples throughout the book. In the present context, our interest is to suggest that abstract data types that directly represent geometric abstractions are not difficult to define and can lead to simple and clear client code. This idea is reinforced in several exercises at the end of this section and on the booksite.

Information processing. Whether it be a bank processing millions of credit card transactions or a web analytics company processing billions of touchpad taps or a scientific research group processing millions of experimental observations, a great many applications are centered around processing and organizing information. Abstract data types provide a natural mechanism for organizing the information. Without getting into details, the two APIs on the facing page suggest a typical approach for a commercial application. The idea is to define data types that allow us to keep information in objects that correspond to things in the real world. A date is a day, a month, and a year and a transaction is a customer, a date, and an amount. These two are just examples: we might also define data types that can hold detailed information for customers, times, locations, goods and services, or whatever. Each data type consists of constructors that create objects containing the data and methods for use by client code to access it. To simplify client code, we provide two constructors for each type, one that presents the data in its appropriate type and another that parses a string to get the data (see EXERCISE 1.2.19 for details). As usual, there is no reason for client code to know the representation of the data. Most often, the reason to organize the data in this way is to treat the data associated with an object as a single entity: we can maintain arrays of `Transaction` values, use `Date` values as a argument or a return value for a method, and so forth. The focus of such data types is on encapsulating the data, while at the same time enabling the development of client code that does not depend on the representation of the data. We do not dwell on organizing information in this way, except to take note that doing so and including the inherited methods `toString()`, `compareTo()`, `equals()`, and `hashCode()` allows us to take advantage of algorithm implementations that can process any type of data. We will discuss inherited methods in more detail on page 100. For example, we have already noted Java's convention that enables clients to print a string representation of every value if we include `toString()` implementation in a data type. We consider conventions corresponding to the other inherited methods in SECTION 1.3, SECTION 2.5, SECTION 3.4, and SECTION 3.5, using `Date` and `Transaction` as examples. SECTION 1.3 gives classic examples of data types and a Java language mechanism known as *parameterized types*, or generics, that takes advantage of these conventions, and CHAPTER 2 and CHAPTER 3 are also devoted to taking advantage of generic types and inherited methods to develop implementations of sorting and searching algorithms that are effective for any type of data.

WHenever you have data of different types that logically belong together, it is worthwhile to contemplate defining an ADT as in these examples. The ability to do so helps to organize the data, can greatly simplify client code in typical applications, and is an important step on the road to data abstraction.

```
public class Date implements Comparable<Date>
```

```
    Date(int month, int day, int year) create a date
    Date(String date) create a date (parse constructor)
    int month() month
    int day() day
    int year() year
    String toString() string representation
    boolean equals(Object that) is this the same date as that?
    int compareTo(Date that) compare this date to that
    int hashCode() hash code
```

```
public class Transaction implements Comparable<Transaction>
```

```
    Transaction(String who, Date when, double amount)
    Transaction(String transaction) create a transaction (parse constructor)
    String who() customer name
    Date when() date
    double amount() amount
    String toString() string representation
    boolean equals(Object that) is this the same transaction as that?
    int compareTo(Transaction that) compare this transaction to that
    int hashCode() hash code
```

Sample APIs for commercial applications (dates and transactions)

Strings. Java's `String` is an important and useful ADT. A `String` is an indexed sequence of `char` values. `String` has dozens of instance methods, including the following:

```
public class String
```

<code>String()</code>	<i>create an empty string</i>
<code>int length()</code>	<i>length of the string</i>
<code>char charAt(int i)</code>	<i>ith character</i>
<code>int indexOf(String p)</code>	<i>first occurrence of p (-1 if none)</i>
<code>int indexOf(String p, int i)</code>	<i>first occurrence of p after i (-1 if none)</i>
<code>String concat(String t)</code>	<i>this string with t appended</i>
<code>String substring(int i, int j)</code>	<i>substring of this string (ith to j-1st chars)</i>
<code>String[] split(String delim)</code>	<i>strings between occurrences of delim</i>
<code>int compareTo(String t)</code>	<i>string comparison</i>
<code>boolean equals(String t)</code>	<i>is this string's value the same as t's?</i>
<code>int hashCode()</code>	<i>hash code</i>

Java `String` API (partial list of methods)

`String` values are similar to arrays of characters, but the two are not the same. Arrays have built-in Java language syntax for accessing a character; `String` has instance methods for indexed access, `length`, and many other operations. On the other hand, `String` has special language support for initialization and concatenation: instead of creating and initializing a string with a constructor, we can use a string literal; instead of invoking the method `concat()` we can use the `+` operator. We do not need to consider the details of the implementation, though understanding performance characteristics of some of the methods is important when developing string-processing algorithms, as you will see in CHAPTER 5. Why not just use arrays of characters instead of `String` values? The answer to this question is the same as for any ADT: to simplify and clarify client code. With `String`, we can write clear and simple client code that uses numerous convenient instance methods without regard to the way in which strings are represented (see facing page). Even this short list contains powerful operations that require advanced algorithms such

```
String a = "now is ";
String b = "the time ";
String c = "to"
```

call	value
<code>a.length()</code>	7
<code>a.charAt(4)</code>	i
<code>a.concat(c)</code>	"now is to"
<code>a.indexOf("is")</code>	4
<code>a.substring(2, 5)</code>	"w i"
<code>a.split(" ")[0]</code>	"now"
<code>a.split(" ")[1]</code>	"is"
<code>b.equals(c)</code>	false

Examples of string operations

task	implementation
<i>is the string a palindrome?</i>	<pre> public static boolean isPalindrome(String s) { int N = s.length(); for (int i = 0; i < N/2; i++) if (s.charAt(i) != s.charAt(N-1-i)) return false; return true; } </pre>
<i>extract file name and extension from a command-line argument</i>	<pre> String s = args[0]; int dot = s.indexOf("."); String base = s.substring(0, dot); String extension = s.substring(dot + 1, s.length()); </pre>
<i>print all lines in standard input that contain a string specified on the command line</i>	<pre> String query = args[0]; while (!StdIn.isEmpty()) { String s = StdIn.readLine(); if (s.contains(query)) StdOut.println(s); } </pre>
<i>create an array of the strings on StdIn delimited by whitespace</i>	<pre> String input = StdIn.readAll(); String[] words = input.split("\\s+"); </pre>
<i>check whether an array of strings is in alphabetical order</i>	<pre> public boolean isSorted(String[] a) { for (int i = 1; i < a.length; i++) { if (a[i-1].compareTo(a[i]) > 0) return false; } return true; } </pre>

Typical string-processing code

as those considered in CHAPTER 5. For example, the argument of `split()` can be a regular expression (see SECTION 5.4)—the `split()` example on page 81 uses the argument `"\\s+"`, which means “one or more tabs, spaces, newlines, or returns.”

Input and output revisited. A disadvantage of the `StdIn`, `StdOut`, and `StdDraw` standard libraries of SECTION 1.1 is that they restrict us to working with just one input file, one output file, and one drawing for any given program. With object-oriented programming, we can define similar mechanisms that allow us to work with multiple input streams, output streams, and drawings within one program. Specifically, our standard library includes the data types `In`, `Out`, and `Draw` with the APIs shown on the facing page. When invoked with a constructor having a `String` argument, `In` and `Out` will first try to find a file in the current directory of your computer that has that name. If it cannot

```
public class Cat
{
    public static void main(String[] args)
    { // Copy input files to out (last argument).
      Out out = new Out(args[args.length-1]);
      for (int i = 0; i < args.length - 1; i++)
      { // Copy input file named on ith arg to out.
        In in = new In(args[i]);
        String s = in.readAll();
        out.println(s);
        in.close();
      }
      out.close();
    }
}
```

A sample `In` and `Out` client

```
% more in1.txt
This is

% more in2.txt
a tiny
test.

% java Cat in1.txt in2.txt out.txt

% more out.txt
This is
a tiny
test.
```

do so, it will assume the argument to be a website name and will try to connect to that website (if no such website exists, it will issue a runtime exception). In either case, the specified file or website becomes the source/target of the input/output for the stream object thus created, and the `read*()` and `print*()` methods will refer to that file or website. (If you use the no-argument constructor, then you obtain the standard streams.) This arrangement makes it possible for a single program to process

multiple files and drawings. You also can assign such objects to variables, pass them as arguments or return values from methods, create arrays of them, and manipulate them just as you manipulate objects of any type. The program `Cat` shown at left is a sample client of `In` and `Out` that uses multiple input streams to concatenate several input files into a single output file. The `In` and `Out` classes also contain static methods for reading files containing values that are all `int`, `double`, or `String` types into an array (see page 126 and EXERCISE 1.2.15).

```
public class In
```

```

    In()                create an input stream from standard input
    In(String name)    create an input stream from a file or website
    boolean isEmpty()  true if no more input, false otherwise
    int readInt()      read a value of type int
    double readDouble() read a value of type double
    ...
    void close()       close the input stream

```

Note: all operations supported by StdIn are also supported for In *objects*.

API for our data type for input streams

```
public class Out
```

```

    Out()                create an output stream to standard output
    Out(String name)    create an output stream to a file
    void print(String s) append s to the output stream
    void println(String s) append s and a newline to the output stream
    void println()      append a newline to the output stream
    void printf(String f, ...) formatted print to the output stream
    void close()        close the output stream

```

Note: all operations supported by StdOut are also supported for Out *objects*.

API for our data type for output streams

```
public class Draw
```

```

    Draw()
    void line(double x0, double y0, double x1, double y1)
    void point(double x, double y)
    ...

```

Note: all operations supported by StdDraw are also supported for Draw *objects*.

API for our data type for drawings

Implementing abstract data types As with libraries of static methods, we implement ADTs with a Java class, putting the code in a file with the same name as the class, followed by the `.java` extension. The first statements in the file declare instance variables that define the data-type values. Following the instance variables are the constructor and the *instance methods* that implement operations on data-type values. Instance methods may be public (specified in the API) or private (used to organize the computation and not available to clients). A data-type definition may have multiple constructors and may also include definitions of static methods. In particular, a unit-test client `main()` is normally useful for testing and debugging. As a first example, we consider an implementation of the Counter ADT that we defined on page 65. A full annotated implementation is shown on the facing page, for reference as we discuss its constituent parts. Every ADT implementation that you will develop has the same basic ingredients as this simple example.

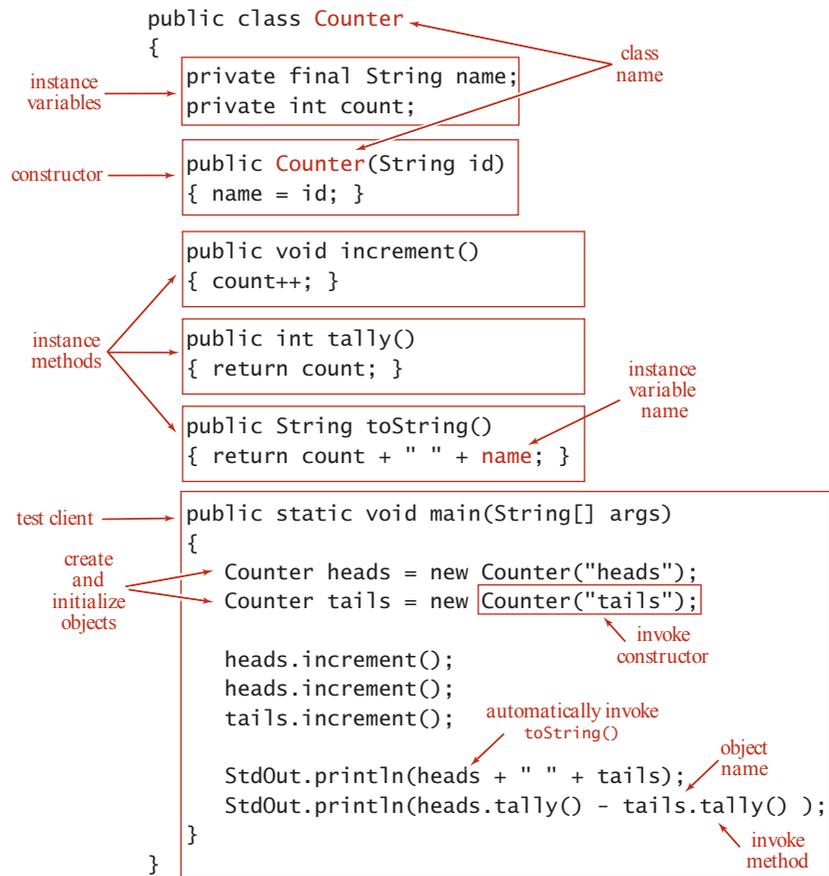
Instance variables. To define data-type values (the state of each object), we declare instance variables in much the same way as we declare local variables. There is a critical distinction between instance variables and the local variables within a static method or a block that you are accustomed to: there is just one value corresponding to

each local variable at a given time, but there are numerous values corresponding to each instance variable (one for each object that is an instance of the data type). There is no ambiguity with this arrangement, because each time that we access an instance variable, we do so with an object name—that object is the one whose value we are accessing. Also, each declaration is qualified by a *visibility modifier*. In ADT implementations, we use `private`, using a Java language mechanism to enforce the idea that the representation of an ADT is to be hidden from the client, and also `final`, if the value is not to be changed once it is initialized. Counter has two instance variables: a `String` value `name` and an `int` value `count`. If we were to use `public` instance variables (allowed in Java) the data type would, by definition, not be abstract, so we do not do so.

Constructors. Every Java class has at least one constructor that establishes an object's identity. A constructor is like a static method, but it can refer directly to instance variables and has no return value. Generally, the purpose of a constructor is to initialize the instance variables. Every constructor creates an object and provides to the client a reference to that object. Constructors always share the same name as the class. We can overload the name and have multiple constructors with different signatures, just as with methods. If no other constructor is defined, a default no-argument constructor is

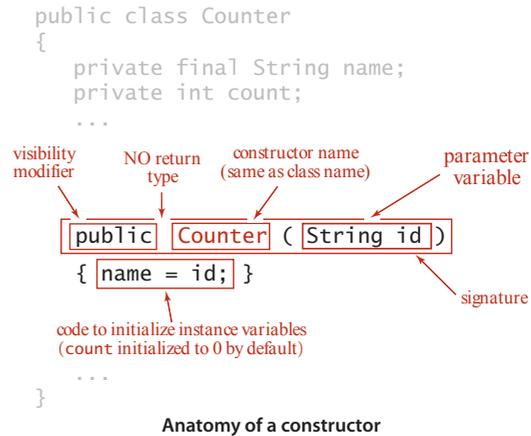
```
public class Counter
{
    private final String name;
    private int count;
    ...
}
```

Instance variables in ADTs are private

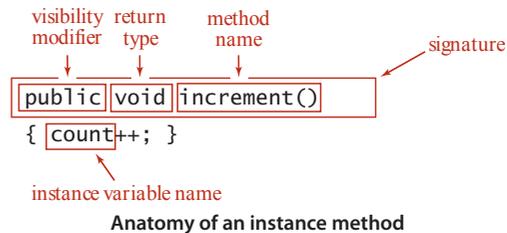


Anatomy of a class that defines a data type

implicit, has no arguments, and initializes instance values to default values. The default values of instance variables are 0 for primitive numeric types, `false` for boolean, and `null` for reference types. These defaults may be changed by using initializing declarations for instance variables. Java automatically invokes a constructor when a client program uses the keyword `new`. Overloaded constructors are typically used to initialize instance variables to client-supplied values other than the defaults. For example, `Counter` has a one-argument constructor that initializes the `name` instance variable to the value given as argument (leaving the `count` instance variable to be initialized to the default value 0).



Instance methods. To implement data-type operations (the *behavior* of each object), we implement *instance methods* with code that is precisely like the code that you learned in SECTION 1.1 to implement static methods (functions). Each instance method has a return type, a signature (which specifies its name and the types and names of its parameter variables), and a body (which consists of a sequence of statements, including a return statement that provides a value of the return type back to the client). When a client invokes a method, the parameter values (if any) are initialized with client values, the statements are executed until a return value is computed, and the value is returned to the client, with the same effect as if the method invocation in the client were replaced with that value. All of this action is the same as for static methods, but there is one critical distinction for instance methods: *they can access and perform operations on instance variables*. How do we specify which object's instance variables we want to use? If you think about this question for a moment, you will see the logical answer: a reference to a variable in an instance method refers to the value for *the object that was used to invoke the method*. When we say `heads.increment()` the code in `increment()` is referring to the instance variables for `heads`. In other words,



object-oriented programming adds one critically important additional way to use variables in a Java program:

- to invoke an instance method that operates on the object's values.

The difference from working solely with static methods is semantic (see the Q&A), but has reoriented the way that modern programmers think about developing code in many situations. As you will see, it also dovetails well with the study of algorithms and data structures.

Scope. In summary, the Java code that we write to implement instance methods uses *three* kinds of variables:

- Parameter variables
- Local variables
- *Instance* variables

The first two of these are the same as for static methods: parameter variables are specified in the method signature and initialized with client values when the method is called, and local variables are declared and initialized within the method body. The scope of parameter variables is the entire method; the scope of local variables is the following statements in the block where they are defined. Instance variables are completely different: they hold data-type values for objects in a class, and their scope is the entire class (whenever there is an ambiguity, you can use the `this` prefix to identify instance variables). Understanding the distinctions among these three kinds of variables in instance methods is a key to success in object-oriented programming.

```

public class Example
{
    private int var;
    ...

    private void method1()
    {
        int var;
        ... var ...
        ... this.var ...
    }

    private void method2()
    {
        ... var ...
    }
    ...
}

```

Annotations in the diagram:

- Red arrow from `private int var;` to `instance variable`
- Red arrow from `int var;` to `local variable`
- Red arrow from `this.var` to `refers to local variable, NOT instance variable`
- Red arrow from `this.var` to `refers to instance variable`
- Red arrow from `var` in `method2()` to `refers to instance variable`

Scope of instance and local variables in an instance method

API, clients, and implementations. These are the basic components that you need to understand to be able to build and use abstract data types in Java. Every ADT implementation that we will consider will be a Java class with private instance variables, constructors, instance methods, and a client. To fully understand a data type, we need the API, typical client code, and an implementation, summarized for Counter on the facing page. To emphasize the separation of client and implementation, we normally present each client as a separate class containing a static method `main()` and reserve test client's `main()` in the data-type definition for minimal unit testing and development (calling each instance method at least once). In each data type that we develop, we go through the same steps. Rather than thinking about what action we need to take next to accomplish a computational goal (as we did when first learning to program), we think about the needs of a client, then accommodate them in an ADT, following these three steps:

- Specify an API. The purpose of the API is to separate clients from implementations, to enable modular programming. We have two goals when specifying an API. First, we want to enable clear and correct client code. Indeed, it is a good idea to write some client code before finalizing the API to gain confidence that the specified data-type operations are the ones that clients need. Second, we want to be able to implement the operations. There is no point specifying operations that we have no idea how to implement.
- Implement a Java class that meets the API specifications. First we choose the instance variables, then we write constructors and the instance methods.
- Develop multiple test clients, to validate the design decisions made in the first two steps.

What operations do clients need to perform, and what data-type values can best support those operations? These basic decisions are at the heart of every implementation that we develop.

API `public class Counter`

<code>Counter(String id)</code>	<i>create a counter named id</i>
<code>void increment()</code>	<i>increment the counter</i>
<code>int tally()</code>	<i>number of increments since creation</i>
<code>String toString()</code>	<i>string representation</i>

typical client

```
public class Flips
{
    public static void main(String[] args)
    {
        int T = Integer.parseInt(args[0]);

        Counter heads = new Counter("heads");
        Counter tails = new Counter("tails");

        for (int t = 0; t < T; t++)
            if (StdRandom.bernoulli(0.5))
                heads.increment();
            else tails.increment();

        StdOut.println(heads);
        StdOut.println(tails);
        int d = heads.tally() - tails.tally();
        StdOut.println("delta: " + Math.abs(d));
    }
}
```

implementation

```
public class Counter
{
    private final String name;
    private int count;

    public Counter(String id)
    { name = id; }

    public void increment()
    { count++; }

    public int tally()
    { return count; }

    public String toString()
    { return count + " " + name; }
}
```

application

```
% java Flips 1000000
500172 heads
499828 tails
delta: 344
```

An abstract data type for a simple counter

More implementations of abstract data types As with any programming concept, the best way to understand the power and utility of ADTs is to consider carefully more examples and more implementations. There will be ample opportunity for you to do so, as much of this book is devoted to ADT implementations, but a few more simple examples will help us lay the groundwork for addressing them.

Date. Shown on the facing page are two implementations of the Date ADT that we considered on page 79. To reduce clutter, we omit the parsing constructor (which is described in EXERCISE 1.2.19) and the inherited methods `equals()` (see page 103), `compareTo()` (see page 247), and `hashCode()` (see EXERCISE 3.4.22). The straightforward implementation on the left maintains the day, month, and year as instance variables, so that the instance methods can just return the appropriate value; the more space-efficient implementation on the right uses only a single `int` value to represent a date, using a mixed-radix number that represents the date with day d , month m , and year y as $512y + 32m + d$. One way that a client might notice the difference between these implementations is by violating implicit assumptions: the second implementation depends for its correctness on the day being between 0 and 31, the month being between 0 and 15, and the year being positive (in practice, both implementations should check that months are between 1 and 12, days are between 1 and 31, and that dates such as June 31 and February 29, 2009, are illegal, though that requires a bit more work). This example highlights the idea that we rarely fully specify implementation requirements in an API (we normally do the best we can, and could do better here). Another way that a client might notice the difference between the two implementations is performance: the implementation on the right uses less space to hold data-type values at the cost of more time to provide them to the client in the agreed form (one or two arithmetic operations are needed). Such tradeoffs are common: one client may prefer one of the implementations and another client might prefer the other, so we need to accommodate both. Indeed, one of the recurring themes of this book is that we need to understand the space and time requirements of various implementations and their suitability for use by various clients. One of the key advantages of using data abstraction in our implementations is that we can normally change from one implementation to another *without changing any client code*.

Maintaining multiple implementations. Multiple implementations of the same API can present maintenance and nomenclature issues. In some cases, we simply want to replace an old implementation with an improved one. In others, we may need to maintain two implementations, one suitable for some clients, the other suitable for others. Indeed, a prime goal of this book is to consider in depth several implementations of each of a number of fundamental ADTs, generally with different performance characteristics. In this book, we often compare the performance of a single client using two

API `public class Date`

<code>Date(int month, int day, int year)</code>	<i>create a date</i>
<code>int month()</code>	<i>month</i>
<code>int day()</code>	<i>day</i>
<code>int year()</code>	<i>year</i>
<code>String toString()</code>	<i>string representation</i>

test client

```
public static void main(String[] args)
{
    int m = Integer.parseInt(args[0]);
    int d = Integer.parseInt(args[1]);
    int y = Integer.parseInt(args[2]);
    Date date = new Date(m, d, y);
    StdOut.println(date);
}
```

application

```
% java Date 12 31 1999
12/31/1999
```

implementation

```
public class Date
{
    private final int month;
    private final int day;
    private final int year;

    public Date(int m, int d, int y)
    { month = m; day = d; year = y; }

    public int month()
    { return month; }

    public int day()
    { return day; }

    public int year()
    { return year; }

    public String toString()
    { return month() + "/" + day()
      + "/" + year(); }
}
```

alternate implementation

```
public class Date
{
    private final int value;

    public Date(int m, int d, int y)
    { value = y*512 + m*32 + d; }

    public int month()
    { return (value / 32) % 16; }

    public int day()
    { return value % 32; }

    public int year()
    { return value / 512; }

    public String toString()
    { return month() + "/" + day()
      + "/" + year(); }
}
```

An abstract data type to encapsulate dates, with two implementations

different implementations of the same API. For this reason, we generally adopt an informal naming convention where we:

- Identify different implementations of the same API by prepending a descriptive modifier. For example, we might name our `Date` implementations on the previous page `BasicDate` and `SmallDate`, and we might wish to develop a `SmartDate` implementation that can validate that dates are legal.
- Maintain a reference implementation with no prefix that makes a choice that should be suitable for most clients. That is, most clients should just use `Date`.

In a large system, this solution is not ideal, as it might involve changing client code. For example, if we were to develop a new implementation `ExtraSmallDate`, then our only options are to change client code or to make it the reference implementation for use by all clients. Java has various advanced language mechanisms for maintaining multiple implementations without needing to change client code, but we use them sparingly because their use is challenging (and even controversial) even for experts, especially in conjunction with other advanced language features that we do value (generics and iterators). These issues are important (for example, ignoring them led to the celebrated Y2K problem at the turn of the millennium, because many programs used their own implementations of the date abstraction that did not take into account the first two digits of the year), but detailed consideration of these issues would take us rather far afield from the study of algorithms.

Accumulator. The accumulator API shown on the facing page defines an abstract data type that provides to clients the ability to maintain a running average of data values. For example, we use this data type frequently in this book to process experimental results (see SECTION 1.4). The implementation is straightforward: it maintains an `int` instance variable `N` that counts the number of data values seen so far and a `double` instance variable `total` that keeps track of the sum of the values seen so far; to compute the average it divides the sum by the count. Note that the implementation does not save the data values—it could be used for a huge number of them (even on a device that is not capable of holding that many), or a huge number of accumulators could be used on a big system. This performance characteristic is subtle and might be specified in the API, because an implementation that does save the values might cause an application to run out of memory.

API	<pre>public class Accumulator</pre>	
	<pre> Accumulator() void addDataValue(double val) double mean() String toString()</pre>	<p><i>create an accumulator</i></p> <p><i>add a new data value</i></p> <p><i>mean of all data values</i></p> <p><i>string representation</i></p>

typical client

```
public class TestAccumulator
{
    public static void main(String[] args)
    {
        int T = Integer.parseInt(args[0]);
        Accumulator a = new Accumulator();
        for (int t = 0; t < T; t++)
            a.addDataValue(StdRandom.random());
        StdOut.println(a);
    }
}
```

application

```
% java TestAccumulator 1000
Mean (1000 values): 0.51829

% java TestAccumulator 1000000
Mean (1000000 values): 0.49948

% java TestAccumulator 1000000
Mean (1000000 values): 0.50014
```

implementation

```
public class Accumulator
{
    private double total;
    private int N;

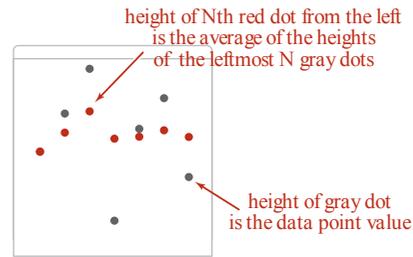
    public void addDataValue(double val)
    {
        N++;
        total += val;
    }

    public double mean()
    { return total/N; }

    public String toString()
    { return "Mean (" + N + " values): "
        + String.format("%7.5f", mean()); }
}
```

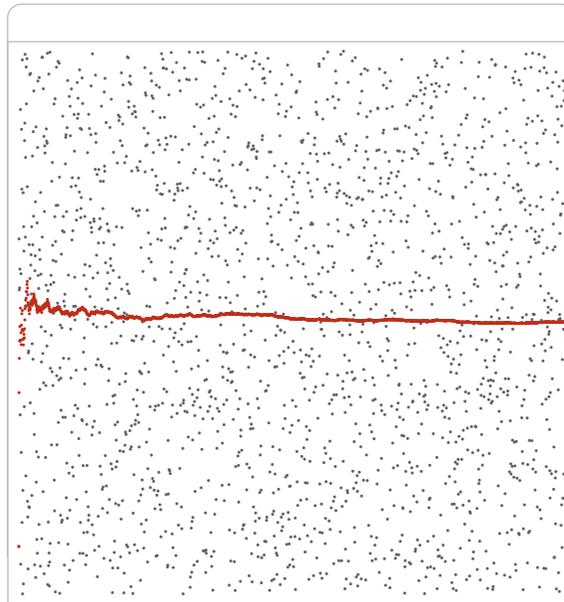
An abstract data type for accumulating data values

Visual accumulator. The visual accumulator implementation shown on the facing page extends `Accumulator` to present a useful side effect: it draws on `StdDraw` all the data (in gray) and the running average (in red). The easiest way to do so is to add a constructor that provides the number of points to be plotted and the maximum value, for rescaling the plot. `VisualAccumulator` is not technically an implementation of the `Accumulator` API (its constructor has a different signature and it causes a different prescribed side effect). Generally, we are careful to fully specify APIs and are loath to make any changes in an API once articulated, as it might involve changing an unknown amount of client (and implementation) code, but adding a constructor to gain functionality can sometimes be defended because it involves changing the same line in client code that we change when changing a class name. In this example, if we have developed a client that uses an `Accumulator` and perhaps has many calls to `addDataValue()` and `mean()`, we can enjoy the benefits of `VisualAccumulator` by just changing one line of client code.



Visual accumulator plot

application



```
% java TestVisualAccumulator 2000
Mean (2000 values): 0.509789
```

API `public class VisualAccumulator`

```

    VisualAccumulator(int trials, double max)
    void addDataValue(double val) add a new data value
    double mean() mean of all data values
    String toString() string representation

```

typical client

```

public class TestVisualAccumulator
{
    public static void main(String[] args)
    {
        int T = Integer.parseInt(args[0]);
        VisualAccumulator a = new VisualAccumulator(T, 1.0);
        for (int t = 0; t < T; t++)
            a.addDataValue(StdRandom.random());
        StdOut.println(a);
    }
}

```

implementation

```

public class VisualAccumulator
{
    private double total;
    private int N;

    public VisualAccumulator(int trials, double max)
    {
        StdDraw.setXscale(0, trials);
        StdDraw.setYscale(0, max);
        StdDraw.setPenRadius(.005);
    }

    public void addDataValue(double val)
    {
        N++;
        total += val;
        StdDraw.setPenColor(StdDraw.DARK_GRAY);
        StdDraw.point(N, val);
        StdDraw.setPenColor(StdDraw.RED);
        StdDraw.point(N, mean());
    }

    public double mean()
    public String toString()
    // Same as Accumulator.
}

```

An abstract data type for accumulating data values (visual version)

Designing abstract data types *An abstract data type is a data type whose representation is hidden from the client.* This idea has had a powerful effect on modern programming. The various examples that we have considered give us the vocabulary to address advanced characteristics of ADTs and their implementation as Java classes. Many of these topics are, on the surface, tangential to the study of algorithms, so it is safe for you to skim this section and refer to it later in the context of specific implementation problems. Our goal is to put important information related to designing data types in one place for reference and to set the stage for implementations throughout this book.

Encapsulation. A hallmark of object-oriented programming is that it enables us to encapsulate data types within their implementations, to facilitate separate development of clients and data type implementations. Encapsulation enables modular programming, allowing us to

- Independently develop client and implementation code
- Substitute improved implementations without affecting clients
- Support programs not yet written (the API is a guide for any future client)

Encapsulation also isolates data-type operations, which leads to the possibility of

- Limiting the potential for error
- Adding consistency checks and other debugging tools in implementations
- Clarifying client code

An encapsulated data type can be used by any client, so it extends the Java language. The programming style that we are advocating is predicated on the idea of breaking large programs into small modules that can be developed and debugged independently. This approach improves the resiliency of our software by limiting and localizing the effects of making changes, and it promotes code reuse by making it possible to substitute new implementations of a data type to improve performance, accuracy, or memory footprint. The same idea works in many settings. We often reap the benefits of encapsulation when we use system libraries. New versions of the Java system often include new implementations of various data types or static method libraries, but *the APIs do not change*. In the context of the study of algorithms and data structures, there is strong and constant motivation to develop better algorithms because we can improve performance for all clients by substituting an improved ADT implementation without changing the code of any client. The key to success in modular programming is to maintain independence among modules. We do so by insisting on the API being the only point of dependence between client and implementation. *You do not need to know how a data type is implemented in order to use it and you can assume that a client knows nothing but the API* when implementing a data type. Encapsulation is the key to attaining both of these advantages.

Designing APIs. One of the most important and most challenging steps in building modern software is designing APIs. This task takes practice, careful deliberation, and many iterations, but any time spent designing a good API is certain to be repaid in time saved debugging or code reuse. Articulating an API might seem to be overkill when writing a small program, but you should consider writing every program as though you will need to reuse the code someday. Ideally, an API would clearly articulate behavior for all possible inputs, including side effects, and then we would have software to check that implementations meet the specification. Unfortunately, a fundamental result from theoretical computer science known as the *specification problem* implies that this goal is actually impossible to achieve. Briefly, such a specification would have to be written in a formal language like a programming language, and the problem of determining whether two programs perform the same computation is known, mathematically, to be undecidable. Therefore, our APIs are brief English-language descriptions of the set of values in the associated abstract data type along with a list of constructors and instance methods, again with brief English-language descriptions of their purpose, including side effects. To validate the design, we always include examples of client code in the text surrounding our APIs. Within this broad outline, there are numerous pitfalls that every API design is susceptible to:

- An API may be *too hard to implement*, implying implementations that are difficult or impossible to develop.
- An API may be *too hard to use*, leading to client code that is more complicated than it would be without the API.
- An API may be too narrow, omitting methods that clients need.
- An API may be too wide, including a large number of methods not needed by any client. This pitfall is perhaps the most common, and one of the most difficult to avoid. The size of an API tends to grow over time because it is not difficult to add methods to an existing API, but it is difficult to remove methods without breaking existing clients.
- An API may be too general, providing no useful abstractions.
- An API may be *too specific*, providing abstractions so detailed or so diffuse as to be useless.
- An API may be too dependent on a particular representation, therefore not serving the purpose of freeing client code from the details of using that representation. This pitfall is also difficult to avoid, because the representation is certainly central to the development of the implementation.

These considerations are sometimes summarized in yet another motto: provide to clients the methods they need and no others.

Algorithms and abstract data types. Data abstraction is naturally suited to the study of algorithms, because it helps us provide a framework within which we can precisely specify both what an algorithm needs to accomplish and how a client can make use of an algorithm. Typically, in this book, an algorithm is an implementation of an instance method in an abstract data type. For example, our whitelisting example at the beginning of the chapter is naturally cast as an ADT client, based on the following operations:

- Construct a SET from an array of given values.
- Determine whether a given value is in the set.

These operations are encapsulated in the `StaticSETofInts` ADT, shown on the facing page along with `Whitelist`, a typical client. `StaticSETofInts` is a special case of the more general and more useful symbol table ADT that is the focus of CHAPTER 3. Binary search is one of several algorithms that we study that is suitable for implementing these ADTs. By comparison with the `BinarySearch` implementation on page 47, this implementation leads to clearer and more useful client code. For example, `StaticSETofInts` enforces the idea that the array must be sorted before `rank()` is called. With the abstract data type, we separate the client from the implementation making it easier for any client to benefit from the ingenuity of the binary search algorithm, just by following the API (clients of `rank()` in `BinarySearch` have to know to sort the array first). Whitelisting is one of many clients that can take advantage of binary search.

EVERY JAVA PROGRAM is a set of static methods and/or a data type implementation. In this book, we focus primarily on abstract data type implementations such as `StaticSETofInts`, where the focus is on operations and the representation of the data is hidden from the client. As this example illustrates, data abstraction enables us to

- Precisely specify what algorithms can provide for clients
- Separate algorithm implementations from the client code
- Develop layers of abstraction, where we make use of well-understood algorithms to develop other algorithms

These are desirable properties of any approach to describing algorithms, whether it be an English-language description or pseudo-code. By embracing the Java `class` mechanism in support of data abstraction, we have little to lose and much to gain: working code that we can test and use to compare performance for diverse clients.

application

```
% java Whitelist largeW.txt < largeT.txt
499569
984875
295754
207807
140925
161828
...
```

API `public class StaticSETofInts`

`StaticSETofInts(int[] a)` *create a set from the values in a[]*
`boolean contains(int key)` *is key in the set?*

typical client

```
public class Whitelist
{
    public static void main(String[] args)
    {
        int[] w = In.readInts(args[0]);
        StaticSETofInts set = new StaticSETofInts(w);
        while (!StdIn.isEmpty())
        { // Read key, print if not in whitelist.
            int key = StdIn.readInt();
            if (!set.contains(key))
                StdOut.println(key);
        }
    }
}
```

implementation

```
import java.util.Arrays;

public class StaticSETofInts
{
    private int[] a;

    public StaticSETofInts(int[] keys)
    {
        a = new int[keys.length];
        for (int i = 0; i < keys.length; i++)
            a[i] = keys[i]; // defensive copy
        Arrays.sort(a);
    }

    public boolean contains(int key)
    { return rank(key) != -1; }

    private int rank(int key)
    { // Binary search.
        int lo = 0;
        int hi = a.length - 1;
        while (lo <= hi)
        { // Key is in a[lo..hi] or not present.
            int mid = lo + (hi - lo) / 2;
            if (key < a[mid]) hi = mid - 1;
            else if (key > a[mid]) lo = mid + 1;
            else
                return mid;
        }
        return -1;
    }
}
```

Binary search recast as an object-oriented program (an ADT for search in a set of integers)

Interface inheritance. Java provides language support for defining relationships among objects, known as *inheritance*. These mechanisms are widely used by software developers, so you will study them in detail if you take a course in software engineering. The first inheritance mechanism that we consider is known as subtyping, which allows us to specify a relationship between otherwise unrelated classes by specifying in an interface a set of common methods that each implementing class must contain. An interface is nothing more than a list of instance methods. For example, instead of using our informal API, we might have articulated an interface for `Date`:

```
public interface Datable
{
    int month();
    int day();
    int year();
}
```

and then referred to the interface in our implementation code

```
public class Date implements Datable
{
    // implementation code (same as before)
}
```

so that the Java compiler will check that it matches the interface. Adding the code `implements Datable` to any class that implements `month()`, `day()`, and `year()` provides a guarantee to any client that an object of that class can invoke those methods. This arrangement is known as *interface inheritance*—an implementing class *inherits* the interface. Interface inheritance allows us to write client programs that can manipulate

objects of any type that implements the interface (even a type to be created in the future), by invoking methods in the interface. We might have used interface inheritance in place of our more informal APIs, but chose not to do so to avoid dependence on specific high-level language mechanisms that are not critical to the understanding of algorithms and to avoid the extra baggage of interface files. But there are a few situations where Java conventions make

	interface	methods	section
comparison	<code>java.lang.Comparable</code>	<code>compareTo()</code>	2.1
	<code>java.util.Comparator</code>	<code>compare()</code>	2.5
iteration	<code>java.lang.Iterable</code>	<code>iterator()</code>	1.3
	<code>java.util.Iterator</code>	<code>hasNext()</code> <code>next()</code> <code>remove()</code>	1.3

Java interfaces used in this book

it worthwhile for us to take advantage of interfaces: we use them for comparison and for iteration, as detailed in the table at the bottom of the previous page, and will consider them in more detail when we cover those concepts.

Implementation inheritance. Java also supports another inheritance mechanism known as subclassing, which is a powerful technique that enables a programmer to change behavior and add functionality without rewriting an entire class from scratch. The idea is to define a new class (subclass, or derived class) that inherits instance methods and instance variables from another class (superclass, or base class). The subclass contains more methods than the superclass. Moreover, the subclass can redefine or override methods in the superclass. Subclassing is widely used by systems programmers to build so-called extensible libraries—one programmer (even you) can add methods to a library built by another programmer (or, perhaps, a team of systems programmers), effectively reusing the code in a potentially huge library. For example, this approach is widely used in the development of graphical user interfaces, so that the large amount of code required to provide all the facilities that users expect (drop-down menus, cut-and-paste, access to files, and so forth) can be reused. The use of subclassing is controversial among systems and applications programmers (its advantages over interface inheritance are debatable), and we avoid it in this book because it generally works against encapsulation. Certain vestiges of the approach are built into Java and therefore unavoidable: specifically, every class is a subtype of Java’s `Object` class. This structure enables the “convention” that every class includes an implementation of `getClass()`, `toString()`, `equals()`, `hashCode()`, and several other methods that we do not use in this book. Actually, every class *inherits* these methods from `Object` through subclassing, so any client can use them for any object. We usually override `toString()`, `equals()`, `hashCode()` in new classes because the default `Object` implementation generally does not lead to the desired behavior. We now will consider `toString()` and `equals()`; we discuss `hashCode()` in SECTION 3.4.

	method	purpose	section
Class	<code>getClass()</code>	<i>what class is this object?</i>	1.2
String	<code>toString()</code>	<i>string representation of this object</i>	1.1
boolean	<code>equals(Object that)</code>	<i>is this object equal to that?</i>	1.2
int	<code>hashCode()</code>	<i>hash code for this object</i>	3.4

Inherited methods from `Object` used in this book

String conversion. By convention, every Java type inherits `toString()` from `Object`, so any client can invoke `toString()` for any object. This convention is the basis for Java's automatic conversion of one operand of the concatenation operator `+` to a `String` whenever the other operand is a `String`. If an object's data type does not include an implementation of `toString()`, then the default implementation in `Object` is invoked, which is normally not helpful, since it typically returns a string representation of the memory address of the object. Accordingly, we generally include implementations of `toString()` that override the default in every class that we develop, as highlighted for `Date` on the facing page. As illustrated in this code, `toString()` implementations are often quite simple, implicitly (through `+`) using `toString()` for each instance variable.

Wrapper types. Java supplies built-in reference types known as wrapper types, one for each of the primitive types: `Boolean`, `Byte`, `Character`, `Double`, `Float`, `Integer`, `Long`, and `Short` correspond to `boolean`, `byte`, `char`, `double`, `float`, `int`, `long`, and `short`, respectively. These classes consist primarily of static methods such as `parseInt()` but they also include the inherited instance methods `toString()`, `compareTo()`, `equals()`, and `hashCode()`. Java automatically converts from primitive types to wrapper types when warranted, as described on page 122. For example, when an `int` value is concatenated with a `String`, it is converted to an `Integer` that can invoke `toString()`.

Equality. What does it mean for two objects to be equal? If we test equality with `(a == b)` where `a` and `b` are reference variables of the same type, we are testing whether they have the same identity: whether the references are equal. Typical clients would rather be able to test whether the data-type values (object state) are the same, or to implement some type-specific rule. Java gives us a head start by providing implementations both for standard types such as `Integer`, `Double`, and `String` and for more complicated types such as `File` and `URL`. When using these types of data, you can just use the built-in implementation. For example, if `x` and `y` are `String` values, then `x.equals(y)` is `true` if and only if `x` and `y` have the same length and are identical in each character position. When we define our own data types, such as `Date` or `Transaction`, we need to override `equals()`. Java's convention is that `equals()` must be an *equivalence relation*. It must be

- *Reflexive*: `x.equals(x)` is `true`.
- *Symmetric*: `x.equals(y)` is `true` if and only if `y.equals(x)` is `true`.
- *Transitive*: if `x.equals(y)` and `y.equals(z)` are `true`, then so is `x.equals(z)`.

In addition, it must take an `Object` as argument and satisfy the following properties.

- *Consistent*: multiple invocations of `x.equals(y)` consistently return the same value, provided neither object is modified.
- *Not null*: `x.equals(null)` returns `false`.

These are natural definitions, but ensuring that these properties hold, adhering to Java conventions, and avoiding unnecessary work in an implementation can be tricky, as illustrated for `Date` below. It takes the following step-by-step approach:

- If the reference to this object is the same as the reference to the argument object, return `true`. This test saves the work of doing all the other checks in this case.
- If the argument is null, return `false`, to adhere to the convention (and to avoid following a null reference in code to follow).
- If the objects are not from the same class, return `false`. To determine an object's class, we use `getClass()`. Note that we can use `==` to tell us whether two objects of type `Class` are equal because `getClass()` is guaranteed to return the same reference for all objects in any given class.
- Cast the argument from `Object` to `Date` (this cast must succeed because of the previous test).
- Return `false` if any instance variables do not match. For other classes, some other definition of equality might be appropriate. For example, we might regard two `Counter` objects as equal if their count instance variables are equal.

This implementation is a model that you can use to implement `equals()` for any type that you implement. Once you have implemented one `equals()`, you will not find it difficult to implement another.

```
public class Date
{
    private final int month;
    private final int day;
    private final int year;

    public Date(int m, int d, int y)
    { month = m; day = d; year = y; }

    public int month()
    { return month; }

    public int day()
    { return day; }

    public int year()
    { return year; }

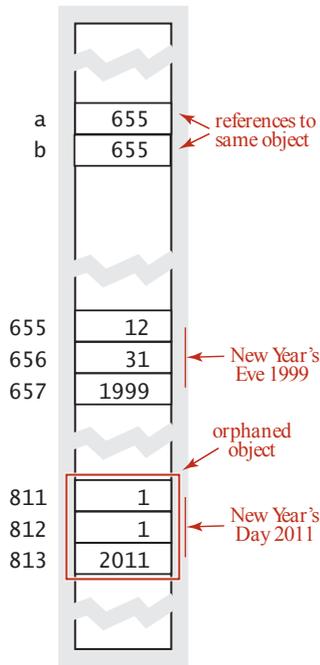
    public String toString()
    { return month() + "/" + day() + "/" + year(); }

    public boolean equals(Object x)
    {
        if (this == x) return true;
        if (x == null) return false;
        if (this.getClass() != x.getClass()) return false;
        Date that = (Date) x;
        if (this.day != that.day) return false;
        if (this.month != that.month) return false;
        if (this.year != that.year) return false;
        return true;
    }
}
```

Overriding `toString()` and `equals()` in a data-type definition

Memory management. The ability to assign a new value to a reference variable creates the possibility that a program may have created an object that can no longer be referenced. For example, consider the three assignment statements in the figure at left. After the third assignment statement, not only do a and b refer to the same Date object (12/31/1999), but also there is no longer a reference to the Date object that was created

```
Date a = new Date(12, 31, 1999);
Date b = new Date( 1,  1, 2011);
b = a;
```



An orphaned object

and used to initialize b. The only reference to that object was in the variable b, and this reference was overwritten by the assignment, so there is no way to refer to the object again. Such an object is said to be *orphaned*. Objects are also orphaned when they go out of scope. Java programs tend to create huge numbers of objects (and variables that hold primitive data-type values), but only have a need for a small number of them at any given point in time. Accordingly, programming languages and systems need mechanisms to allocate memory for data-type values during the time they are needed and to free the memory when they are no longer needed (for an object, sometime after it is orphaned). Memory management turns out to be easier for primitive types because all of the information needed for memory allocation is known at compile time. Java (and most other systems) takes care of reserving space for variables when they are declared and freeing that space when they go out of scope. Memory management for objects is more complicated: the system can allocate memory for an object when it is created, but cannot know precisely when to free the memory associated with each object because the dynamics of a program in execution determines when objects are orphaned. In many languages (such as C and C++) the programmer is responsible for both allocating and freeing memory. Doing so is tedious and notoriously

error-prone. One of Java's most significant features is its ability to automatically manage memory. The idea is to free the programmers from the responsibility of managing memory by keeping track of orphaned objects and returning the memory they use to a pool of free memory. Reclaiming memory in this way is known as *garbage collection*. One of Java's characteristic features is its policy that references cannot be modified. This policy enables Java to do efficient automatic garbage collection. Programmers still debate whether the overhead of automatic garbage collection justifies the convenience of not having to worry about memory management.

Immutability. An immutable data type, such as `Date`, has the property that the value of an object never changes once constructed. By contrast, a mutable data type, such as `Counter` or `Accumulator`, manipulates object values that are intended to change. Java’s language support for helping to enforce immutability is the `final` modifier. When you declare a variable to be `final`, you are promising to assign it a value only once, either in an initializer or in the constructor. Code that could modify the value of a `final` variable leads to a compile-time error. In our code, we use the modifier `final` with instance variables whose values never change. This policy serves as documentation that the value does not change, prevents accidental changes, and makes programs easier to debug. For example, you do not have to include a `final` value in a trace, since you know that its value never changes. A data type such as `Date` whose instance variables are all primitive and `final` is immutable (in code that does not use implementation inheritance, our convention). Whether to make a data type immutable is an important design decision and depends on the application at hand. For data types such as `Date`, the purpose of the abstraction is to encapsulate values that do not change so that we can use them in assignment statements and as arguments and return values from functions in the same way as we use primitive types (without having to worry about their values changing). A programmer implementing a `Date` client might reasonably expect to write the code `d = d0` for two `Date` variables, in the same way as for `double` or `int` values. But if `Date` were mutable and the value of `d` were to change after the assignment `d = d0`, then the value of `d0` would also change (they are both references to the same object)! On the other hand, for data types such as `Counter` and `Accumulator`, the very purpose of the abstraction is to encapsulate values as they change. You have already encountered this distinction as a client programmer, when using Java arrays (mutable) and Java’s `String` data type (immutable). When you pass a `String` to a method, you do not worry about that method changing the sequence of characters in the `String`, but when you pass an array to a method, the method is free to change the contents of the array. `String` objects are immutable because we generally do not want `String` values to change, and Java arrays are mutable because we generally do want array values to change. There are also situations where we want to have mutable strings (that is the purpose of Java’s `StringBuilder` class) and where we want to have immutable arrays (that is the purpose of the `Vector` class that we consider later in this section). Generally, immutable types are easier to use and harder to misuse than mutable types because the scope of code that can change their values is far smaller. It is easier to debug code that uses immutable types because it is easier to guarantee that variables in client code that uses them remain in a consistent state. When using mutable types,

mutable	immutable
Counter	Date
Java arrays	String

Mutable/immutable examples

you must always be concerned about where and when their values change. The downside of immutability is that a new object must be created for every value. This expense is normally manageable because Java garbage collectors are typically optimized for such situations. Another downside of immutability stems from the fact that, unfortunately, `final` guarantees immutability only when instance variables are primitive types, not reference types. If an instance variable of a reference type has the `final` modifier, the value of that instance variable (the reference to an object) will never change—it will always refer to the same object—but the value of the object itself can change. For example, this code does not implement an immutable type:

```
public class Vector
{
    private final double[] coords;

    public Vector(double[] a)
    { coords = a; }
    ...
}
```

A client program could create a `Vector` by specifying the entries in an array, and then (bypassing the API) change the elements of the `Vector` after construction:

```
double[] a = { 3.0, 4.0 };
Vector vector = new Vector(a);
a[0] = 0.0; // Bypasses the public API.
```

The instance variable `coords[]` is `private` and `final`, but `Vector` is mutable because the client holds a reference to the data. Immutability needs to be taken into account in any data-type design, and whether a data type is immutable should be specified in the API, so that clients know that object values will not change. In this book, our primary interest in immutability is for use in certifying the correctness of our algorithms. For example, if the type of data used for a binary search algorithm were mutable, then clients could invalidate our assumption that the array is sorted for binary search.

Design by contract. To conclude, we briefly discuss Java language mechanisms that enables you to verify assumptions about your program as it is running. We use two Java language mechanisms for this purpose:

- Exceptions and errors, which generally handle unforeseen errors outside our control
- Assertions, which verify assumptions that we make *within* code we develop

Liberal use of both exceptions and assertions is good programming practice. We use them sparingly in the book for economy, but you will find them throughout the code on the booksite. This code aligns with a substantial amount of the surrounding commentary about each algorithm in the text that has to do with exceptional conditions and with asserted invariants.

Exceptions and errors. *Exceptions* and errors are disruptive events that occur while a program is running, often to signal an error. The action taken is known as *throwing an exception or throwing an error*. We have already encountered exceptions thrown by Java system methods in the course of learning basic features of Java: `StackOverflowError`, `ArithmeticException`, `ArrayIndexOutOfBoundsException`, `OutOfMemoryError`, and `NullPointerException` are typical examples. You can also create your own exceptions. The simplest kind is a `RuntimeException` that terminates execution of the program and prints an error message

```
throw new RuntimeException("Error message here.");
```

A general practice known as fail fast programming suggests that an error is more easily pinpointed if an exception is thrown as soon as an error is discovered (as opposed to ignoring the error and deferring the exception to sometime in the future).

Assertions. An assertion is a boolean expression that you are affirming is true at that point in the program. If the expression is false, the program will terminate and report an error message. We use assertions both to gain confidence in the correctness of programs and to document intent. For example, suppose that you have a computed value that you might use to index into an array. If this value were negative, it would cause an `ArrayIndexOutOfBoundsException` sometime later. But if you write the code `assert index >= 0;` you can pinpoint the place where the error occurred. You can also add an optional detail message such as

```
assert index >= 0 : "Negative index in method X";
```

to help you locate the bug. By default, assertions are disabled. You can enable them from the command line by using the `-enableassertions` flag (`-ea` for short). Assertions are for debugging: your program should not rely on assertions for normal operation since they may be disabled. When you take a course in systems programming, you will learn

to use assertions to ensure that your code never terminates in a system error or goes into an infinite loop. One model, known as the design-by-contract model of programming expresses the idea. The designer of a data type expresses a precondition (the condition that the client promises to satisfy when calling a method), a postcondition (the condition that the implementation promises to achieve when returning from a method), and side effects (any other change in state that the method could cause). During development, these conditions can be tested with assertions.

Summary. The language mechanisms discussed throughout this section illustrate that effective data-type design leads to nontrivial issues that are not easy to resolve. Experts are still debating the best ways to support some of the design ideas that we are discussing. Why does Java not allow functions as arguments? Why does Matlab copy arrays passed as arguments to functions? As mentioned early in CHAPTER 1, it is a slippery slope from complaining about features in a programming language to becoming a programming-language designer. If you do not plan to do so, your best strategy is to use widely available languages. Most systems have extensive libraries that you certainly should use when appropriate, but you often can simplify your client code and protect yourself by building abstractions that can easily transport to other languages. Your main goal is to develop data types so that most of your work is done at a level of abstraction that is appropriate to the problem at hand.

The table on the facing page summarizes the various kinds of Java classes that we have considered.

kind of class	examples	characteristics
<i>static methods</i>	Math StdIn StdOut	no instance variables
<i>immutable abstract data type</i>	Date Transaction String Integer	instance variables all private instance variables all final defensive copy for reference types <i>Note: these are necessary but not sufficient.</i>
<i>mutable abstract data type</i>	Counter Accumulator	instance variables all private not all instance variables final
<i>abstract data type with I/O side effects</i>	VisualAccumulator In Out Draw	instance variables all private instance methods do I/O
Java classes (data-type implementations)		

Q & A

Q. Why bother with data abstraction?

A. It helps us produce reliable and correct code. For example, in the 2000 presidential election, Al Gore received –16,022 votes on an electronic voting machine in Volusia County, Florida—the tally was clearly not properly encapsulated in the voting machine software!

Q. Why the distinction between primitive and reference types? Why not just have reference types?

A. Performance. Java provides the reference types `Integer`, `Double`, and so forth that correspond to primitive types that can be used by programmers who prefer to ignore the distinction. Primitive types are closer to the types of data that are supported by computer hardware, so programs that use them usually run faster than programs that use corresponding reference types.

Q. Do data types *have* to be abstract?

A. No. Java also allows `public` and `protected` to allow some clients to refer directly to instance variables. As described in the text, the advantages of allowing client code to directly refer to data are greatly outweighed by the disadvantages of dependence on a particular representation, so all instance variables are `private` in our code. We also occasionally use `private` instance methods to share code among public methods.

Q. What happens if I forget to use `new` when creating an object?

A. To Java, it looks as though you want to call a static method with a return value of the object type. Since you have not defined such a method, the error message is the same as anytime you refer to an undefined symbol. If you compile the code

```
Counter c = Counter("test");
```

you get this error message:

```
cannot find symbol
symbol  : method Counter(String)
```

You get the same kind of error message if you provide the wrong number of arguments to a constructor.

Q. What happens if I forget to use `new` when creating an array of objects?

A. You need to use `new` for each object that you create, so when you create an array of N objects, you need to use `new` $N+1$ times: once for the array and once for each of the objects. If you forget to create the array:

```
Counter[] a;  
a[0] = new Counter("test");
```

you get the same error message that you would get when trying to assign a value to any uninitialized variable:

```
variable a might not have been initialized  
a[0] = new Counter("test");  
  ^
```

but if you forget to use `new` when creating an object within the array and then try to use it to invoke a method:

```
Counter[] a = new Counter[2];  
a[0].increment();
```

you get a `NullPointerException`.

Q. Why not write `StdOut.println(x.toString())` to print objects?

A. That code works fine, but Java saves us the trouble of writing it by automatically invoking the `toString()` method for any object, since `println()` has a method that takes an `Object` as argument.

Q. What is a pointer?

A. Good question. Perhaps that should be `NullReferenceException`. Like a Java reference, you can think of a pointer as a machine address. In many programming languages, the pointer is a primitive data type that programmers can manipulate in many ways. But programming with pointers is notoriously error-prone, so operations provided for pointers need to be carefully designed to help programmers avoid errors. Java takes this point of view to an extreme (that is favored by many modern programming-language designers). In Java, there is only one way to create a reference (`new`) and only one way to change a reference (with an assignment statement). That is, the only things that a programmer can do with references are to create them and copy them. In

Q & A *(continued)*

programming-language jargon, Java references are known as safe pointers, because Java can guarantee that each reference points to an object of the specified type (and it can determine which objects are not in use, for garbage collection). Programmers used to writing code that directly manipulates pointers think of Java as having no pointers at all, but people still debate whether it is really desirable to have unsafe pointers.

Q. Where can I find more details on how Java implements references and does garbage collection?

A. One Java system might differ completely from another. For example, one natural scheme is to use a pointer (machine address); another is to use a *handle* (a pointer to a pointer). The former gives faster access to data; the latter provides for better garbage collection.

Q. What exactly does it mean to `import` a name?

A. Not much: it just saves some typing. You could type `java.util.Arrays` instead of `Arrays` everywhere in your code instead of using the `import` statement.

Q. What is the problem with implementation inheritance?

A. Subtyping makes modular programming more difficult for two reasons. First, any change in the superclass affects all subclasses. The subclass cannot be developed independently of the superclass; indeed, it is completely dependent on the superclass. This problem is known as the fragile base class problem. Second, the subclass code, having access to instance variables, can subvert the intention of the superclass code. For example, the designer of a class like `Counter` for a voting system may take great care to make it so that `Counter` can only increment the tally by one (remember Al Gore's problem). But a subclass, with full access to the instance variable, can change it to any value whatever.

Q. How do I make a class immutable?

A. To ensure immutability of a data type that includes an instance variable of a mutable type, we need to make a local copy, known as a defensive copy. And that may not be enough. Making the copy is one challenge; ensuring that none of the instance methods change values is another.

Q. What is `null`?

A. It is a literal value that refers to no object. Invoking a method using the `null` reference is meaningless and results in a `NullPointerException`. If you get this error message, check to make sure that your constructor properly initializes all of its instance variables.

Q. Can I have a static method in a class that implements a data type?

A. Of course. For example, all of our classes have `main()`. Also, it is natural to consider adding static methods for operations that involve multiple objects where none of them naturally suggests itself as the one that should invoke the method. For example, we might define a static method like the following within `Point`:

```
public static double distance(Point a, Point b)
{
    return a.distTo(b);
}
```

Often, including such methods can serve to clarify client code.

Q. Are there other kinds of variables besides parameter, local, and instance variables?

A. If you include the keyword `static` in a class declaration (outside of any type) it creates a completely different type of variable, known as a static variable. Like instance variables, static variables are accessible to every method in the class; however, they are not associated with any object. In older programming languages, such variables are known as `global variables`, because of their global scope. In modern programming, we focus on limiting scope and therefore rarely use such variables. When we do, we will call attention to them.

Q. What is a deprecated method?

A. A method that is no longer fully supported, but kept in an API to maintain compatibility. For example, Java once included a method `Character.isSpace()`, and programmers wrote programs that relied on using that method's behavior. When the designers of Java later wanted to support additional Unicode whitespace characters, they could not change the behavior of `isSpace()` without breaking client programs, so, instead, they added a new method, `Character.isWhiteSpace()`, and deprecated the old method. As time wears on, this practice certainly complicates APIs. Sometimes, entire classes are deprecated. For example, Java deprecated its `java.util.Date` in order to better support internationalization.

EXERCISES

1.2.1 Write a `Point2D` client that takes an integer value N from the command line, generates N random points in the unit square, and computes the distance separating the closest pair of points.

1.2.2 Write an `Interval1D` client that takes an `int` value N as command-line argument, reads N intervals (each defined by a pair of `double` values) from standard input, and prints all pairs that intersect.

1.2.3 Write an `Interval2D` client that takes command-line arguments N , min , and max and generates N random 2D intervals whose width and height are uniformly distributed between min and max in the unit square. Draw them on `StdDraw` and print the number of pairs of intervals that intersect and the number of intervals that are contained in one another.

1.2.4 What does the following code fragment print?

```
String string1 = "hello";
String string2 = string1;
string1 = "world";
StdOut.println(string1);
StdOut.println(string2);
```

1.2.5 What does the following code fragment print?

```
String s = "Hello World";
s.toUpperCase();
s.substring(6, 11);
StdOut.println(s);
```

Answer: "Hello World". `String` objects are immutable—string methods return a new `String` object with the appropriate value (but they do not change the value of the object that was used to invoke them). This code ignores the objects returned and just prints the original string. To print "WORLD", use `s = s.toUpperCase()` and `s = s.substring(6, 11)`.

1.2.6 A string s is a circular rotation of a string t if it matches when the characters are circularly shifted by any number of positions; e.g., ACTGACG is a circular shift of TGACGAC, and vice versa. Detecting this condition is important in the study of genomic sequences. Write a program that checks whether two given strings s and t are circular

shifts of one another. *Hint*: The solution is a one-liner with `indexOf()`, `length()`, and string concatenation.

1.2.7 What does the following recursive function return?

```
public static String mystery(String s)
{
    int N = s.length();
    if (N <= 1) return s;
    String a = s.substring(0, N/2);
    String b = s.substring(N/2, N);
    return mystery(b) + mystery(a);
}
```

1.2.8 Suppose that `a[]` and `b[]` are each integer arrays consisting of millions of integers. What does the follow code do? Is it reasonably efficient?

```
int[] t = a; a = b; b = t;
```

Answer. It swaps them. It could hardly be more efficient because it does so by copying references, so that it is not necessary to copy millions of elements.

1.2.9 Instrument `BinarySearch` (page 47) to use a `Counter` to count the total number of keys examined during all searches and then print the total after all searches are complete. *Hint*: Create a `Counter` in `main()` and pass it as an argument to `rank()`.

1.2.10 Develop a class `VisualCounter` that allows both increment and decrement operations. Take two arguments `N` and `max` in the constructor, where `N` specifies the maximum number of operations and `max` specifies the maximum absolute value for the counter. As a side effect, create a plot showing the value of the counter each time its tally changes.

1.2.11 Develop an implementation `SmartDate` of our `Date` API that raises an exception if the date is not legal.

1.2.12 Add a method `dayOfTheWeek()` to `SmartDate` that returns a `String` value `Monday`, `Tuesday`, `Wednesday`, `Thursday`, `Friday`, `Saturday`, or `Sunday`, giving the appropriate day of the week for the date. You may assume that the date is in the 21st century.

EXERCISES *(continued)*

1.2.13 Using our implementation of `Date` as a model (page 91), develop an implementation of `Transaction`.

1.2.14 Using our implementation of `equals()` in `Date` as a model (page 103), develop an implementation of `equals()` for `Transaction`.

CREATIVE PROBLEMS

1.2.15 *File input.* Develop a possible implementation of the static `readInts()` method from `In` (which we use for various test clients, such as binary search on page 47) that is based on the `split()` method in `String`.

Solution:

```
public static int[] readInts(String name)
{
    In in = new In(name);
    String input = in.readAll();
    String[] words = input.split("\\s+");
    int[] ints = new int[words.length];
    for int i = 0; i < words.length; i++)
        ints[i] = Integer.parseInt(words[i]);
    return ints;
}
```

We will consider a different implementation in SECTION 1.3 (see page 126).

1.2.16 *Rational numbers.* Implement an immutable data type `Rational` for rational numbers that supports addition, subtraction, multiplication, and division.

```
public class Rational
{
    Rational(int numerator, int denominator)
    Rational plus(Rational b)           sum of this number and b
    Rational minus(Rational b)         difference of this number and b
    Rational times(Rational b)         product of this number and b
    Rational dividedBy(Rational b)     quotient of this number and b
    boolean equals(Object that)        is this number equal to that?
    String toString()                  string representation
}
```

You do not have to worry about testing for overflow (see EXERCISE 1.2.17), but use as instance variables two `long` values that represent the numerator and denominator to limit the possibility of overflow. Use Euclid's algorithm (see page 4) to ensure that the numerator and denominator never have any common factors. Include a test client that exercises all of your methods.

CREATIVE PROBLEMS *(continued)*

1.2.17 *Robust implementation of rational numbers.* Use assertions to develop an implementation of `Rational` (see EXERCISE 1.2.16) that is immune to overflow.

1.2.18 *Variance for accumulator.* Validate that the following code, which adds the methods `var()` and `stddev()` to `Accumulator`, computes both the sample mean, sample variance, and sample standard deviation of the numbers presented as arguments to `addDataValue()`:

```
public class Accumulator
{
    private double m;
    private double s;
    private int N;

    public void addDataValue(double x)
    {
        N++;
        s = s + 1.0 * (N-1) / N * (x - m) * (x - m);
        m = m + (x - m) / N;
    }

    public double mean()
    { return m; }

    public double var()
    { return s/(N - 1); }

    public double stddev()
    { return Math.sqrt(this.var()); }
}
```

This implementation is less susceptible to roundoff error than the straightforward implementation based on saving the sum of the squares of the numbers.

1.2.19 Parsing. Develop the parse constructors for your `Date` and `Transaction` implementations of EXERCISE 1.2.13 that take a single `String` argument to specify the initialization values, using the formats given in the table below.

Partial solution:

```
public Date(String date)
{
    String[] fields = date.split("/");
    month = Integer.parseInt(fields[0]);
    day   = Integer.parseInt(fields[1]);
    year  = Integer.parseInt(fields[2]);
}
```

type	format	example
Date	integers separated by slashes	5/22/1939
Transaction	customer, date, and amount, separated by whitespace	Turing 5/22/1939 11.99

Formats for parsing



1.3 BAGS, QUEUES, AND STACKS

SEVERAL FUNDAMENTAL DATA TYPES involve collections of objects. Specifically, the set of values is a collection of objects, and the operations revolve around adding, removing, or examining objects in the collection. In this section, we consider three such data types, known as the bag, the *queue*, and the stack. They differ in the specification of which object is to be removed or examined next.

Bags, queues, and stacks are fundamental and broadly useful. We use them in implementations throughout the book. Beyond this direct applicability, the client and implementation code in this section serves as an introduction to our general approach to the development of data structures and algorithms.

One goal of this section is to emphasize the idea that the way in which we represent the objects in the collection directly impacts the efficiency of the various operations. For collections, we design data structures for representing the collection of objects that can support efficient implementation of the requisite operations.

A second goal of this section is to introduce generics and iteration, basic Java constructs that substantially simplify client code. These are advanced programming-language mechanisms that are not necessarily essential to the understanding of algorithms, but their use allows us to develop client code (and implementations of algorithms) that is more clear, compact, and elegant than would otherwise be possible.

A third goal of this section is to introduce and show the importance of linked data structures. In particular, a classic data structure known as the linked list enables implementation of bags, queues, and stacks that achieve efficiencies not otherwise possible. Understanding linked lists is a key first step to the study of algorithms and data structures.

For each of the three types, we consider APIs and sample client programs, then look at possible representations of the data type values and implementations of the data-type operations. This scenario repeats (with more complicated data structures) throughout this book. The implementations here are models of implementations later in the book and worthy of careful study.

APIs As usual, we begin our discussion of abstract data types for collections by defining their APIs, shown below. Each contains a no-argument constructor, a method to add an item to the collection, a method to test whether the collection is empty, and a method that returns the size of the collection. Stack and Queue each have a method to remove a particular item from the collection. Beyond these basics, these APIs reflect two Java features that we will describe on the next few pages: generics and iterable collections.

Bag

```
public class Bag<Item> implements Iterable<Item>
    Bag()                create an empty bag
    void add(Item item)  add an item
    boolean isEmpty()    is the bag empty?
    int size()           number of items in the bag
```

FIFO queue

```
public class Queue<Item> implements Iterable<Item>
    Queue()              create an empty queue
    void enqueue(Item item) add an item
    Item dequeue()       remove the least recently added item
    boolean isEmpty()    is the queue empty?
    int size()           number of items in the queue
```

Pushdown (LIFO) stack

```
public class Stack<Item> implements Iterable<Item>
    Stack()              create an empty stack
    void push(Item item) add an item
    Item pop()           remove the most recently added item
    boolean isEmpty()    is the stack empty?
    int size()           number of items in the stack
```

APIs for fundamental generic iterable collections

Generics. An essential characteristic of collection ADTs is that we should be able to use them for any type of data. A specific Java mechanism known as *generics*, also known as *parameterized types*, enables this capability. The impact of generics on the programming language is sufficiently deep that they are not found in many languages (including early versions of Java), but our use of them in the present context involves just a small bit of extra Java syntax and is easy to understand. The notation `<Item>` after the class name in each of our APIs defines the name `Item` as a type parameter, a symbolic placeholder for some concrete type to be used by the client. You can read `Stack<Item>` as “stack of items.” When implementing `Stack`, we do not know the concrete type of `Item`, but a client can use our stack for any type of data, including one defined long after we develop our implementation. The client code provides a concrete type when the stack is created: we can replace `Item` with the name of any reference data type (consistently, everywhere it appears). This provides exactly the capability that we need. For example, you can write code such as

```
Stack<String> stack = new Stack<String>();
stack.push("Test");
...
String next = stack.pop();
```

to use a stack for `String` objects and code such as

```
Queue<Date> queue = new Queue<Date>();
queue.enqueue(new Date(12, 31, 1999));
...
Date next = queue.dequeue();
```

to use a queue for `Date` objects. If you try to add a `Date` (or data of any other type than `String`) to `stack` or a `String` (or data of any other type than `Date`) to `queue`, you will get a compile-time error. Without generics, we would have to define (and implement) different APIs for each type of data we might need to collect; with generics, we can use one API (and one implementation) for all types of data, even types that are implemented in the future. As you will soon see, generic types lead to clear client code that is easy to understand and debug, so we use them throughout this book.

Autoboxing. Type parameters have to be instantiated as reference types, so Java has special mechanisms to allow generic code to be used with primitive types. Recall that Java’s wrapper types are reference types that correspond to primitive types: `Boolean`, `Byte`, `Character`, `Double`, `Float`, `Integer`, `Long`, and `Short` correspond to `boolean`, `byte`, `char`, `double`, `float`, `int`, `long`, and `short`, respectively. Java automatically converts between these reference types and the corresponding primitive types—in assignments, method arguments, and arithmetic/logic expressions. In the present context,

this conversion is helpful because it enables us to use generics with primitive types, as in the following code:

```
Stack<Integer> stack = new Stack<Integer>();
stack.push(17);      // auto-boxing (int -> Integer)
int i = stack.pop(); // auto-unboxing (Integer -> int)
```

Automatically casting a primitive type to a wrapper type is known as autoboxing, and automatically casting a wrapper type to a primitive type is known as auto-unboxing. In this example, Java automatically casts (autoboxes) the primitive value 17 to be of type `Integer` when we pass it to the `push()` method. The `pop()` method returns an `Integer`, which Java casts (auto-unboxes) to an `int` before assigning it to the variable `i`.

Iterable collections. For many applications, the client's requirement is just to process each of the items in some way, or to iterate through the items in the collection. This paradigm is so important that it has achieved first-class status in Java and many other modern languages (the programming language itself has specific mechanisms to support it, not just the libraries). With it, we can write clear and compact code that is free from dependence on the details of a collection's implementation. For example, suppose that a client maintains a collection of transactions in a `Queue`, as follows:

```
Queue<Transaction> collection = new Queue<Transaction>();
```

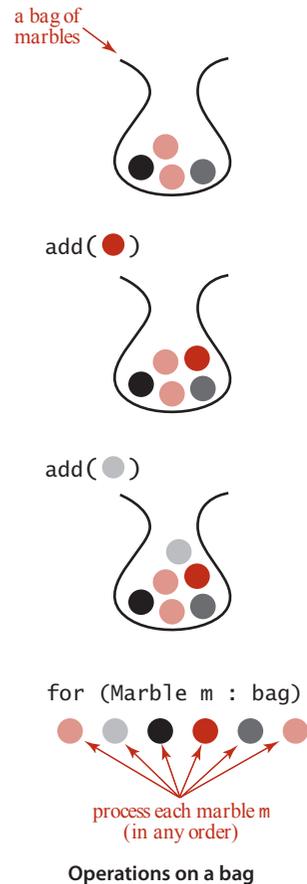
If the collection is iterable, the client can print a transaction list with a single statement:

```
for (Transaction t : collection)
{   StdOut.println(t);   }
```

This construct is known as the *foreach* statement: you can read the `for` statement as *for each transaction t in the collection, execute the following block of code*. This client code does not need to know anything about the representation or the implementation of the collection; it just wants to process each of the items in the collection. The same `for` loop would work with a `Bag` of transactions or any other iterable collection. We could hardly imagine client code that is more clear and compact. As you will see, supporting this capability requires extra effort in the implementation, but this effort is well worthwhile.

IT IS INTERESTING TO NOTE that the only differences between the APIs for `Stack` and `Queue` are their names and the names of the methods. This observation highlights the idea that we cannot easily specify all of the characteristics of a data type in a list of method signatures. In this case, the true specification has to do with the English-language descriptions that specify the rules by which an item is chosen to be removed (or to be processed next in the *foreach* statement). Differences in these rules are profound, *part of the API*, and certainly of critical importance in developing client code.

Bags. A bag is a collection where removing items is not supported—its purpose is to provide clients with the ability to collect items and then to iterate through the collected items (the client can also test if a bag is empty and find its number of items). The order of iteration is unspecified and should be immaterial to the client. To appreciate the concept, consider the idea of an avid marble collector, who might put marbles in a bag, one at a time, and periodically process all the marbles to look for one having some particular characteristic. With our Bag API, a client can add items to a bag and process them all with a *foreach* statement whenever needed. Such a client could use a stack or a queue, but one way to emphasize that the order in which items are processed is immaterial is to use a Bag. The class `Stats` at right illustrates a typical Bag client. The task is simply to compute the average and the sample standard deviation of the `double` values on standard input. If there are N numbers on standard input, their average is computed by adding the numbers and dividing by N ; their sample standard deviation is computed by adding the squares of the difference between each number and the average, dividing by $N-1$, and taking the square root. The order in which the numbers are considered is not relevant for either of these calculations, so we save them in a Bag and use the *foreach* construct to compute each sum. Note: It is possible to compute the standard deviation without saving all the numbers (as we did for the average in `Accumulator`—see EXERCISE 1.2.18). Keeping the numbers in a Bag is required for more complicated statistics.



typical Bag client

```
public class Stats
{
    public static void main(String[] args)
    {
        Bag<Double> numbers = new Bag<Double>();
        while (!StdIn.isEmpty())
            numbers.add(StdIn.readDouble());
        int N = numbers.size();

        double sum = 0.0;
        for (double x : numbers)
            sum += x;
        double mean = sum/N;

        sum = 0.0;
        for (double x : numbers)
            sum += (x - mean)*(x - mean);
        double std = Math.sqrt(sum/(N-1));

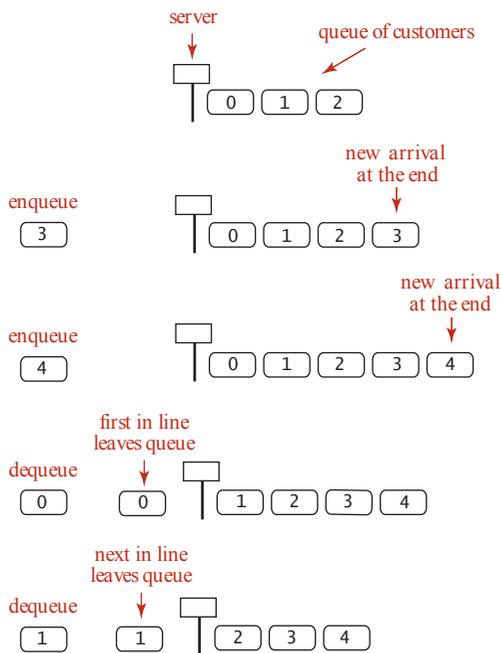
        StdOut.printf("Mean: %.2f\n", mean);
        StdOut.printf("Std dev: %.2f\n", std);
    }
}
```

application

```
% java Stats
100
99
101
120
98
107
109
81
101
90

Mean: 100.60
Std dev: 10.51
```

FIFO queues. A *FIFO queue* (or just a *queue*) is a collection that is based on the *first-in-first-out* (FIFO) policy. The policy of doing tasks in the same order that they arrive



A typical FIFO queue

is one that we encounter frequently in everyday life: from people waiting in line at a theater, to cars waiting in line at a toll booth, to tasks waiting to be serviced by an application on your computer. One bedrock principle of any service policy is the perception of fairness. The first idea that comes to mind when most people think about fairness is that whoever has been waiting the longest should be served first. That is precisely the FIFO discipline. Queues are a natural model for many everyday phenomena, and they play a central role in numerous applications. When a client iterates through the items in a queue with the *foreach* construct, the items are processed in the order they were added to the queue. A typical reason to use a queue in an application is to save items in a collection while at the same time *preserving their* relative order: they come out in the same order in which they were put in. For example, the client below is a possible implementation of the `readInts()` static method from our `In` class. The problem that this method solves for the client is that the client can get numbers from a file into an array *without knowing the file size ahead of time*. We *enqueue* the numbers from the file, use the `size()` method from `Queue` to find the size needed for the array, create the array, and then *dequeue* the numbers to move them to the array. A queue is appropriate because it puts the numbers into the array in the order in which they appear in the file (we might use a `Bag` if that order is immaterial). This code uses autoboxing and auto-unboxing to convert between the client's `int` primitive type and the queue's `Integer` wrapper type.

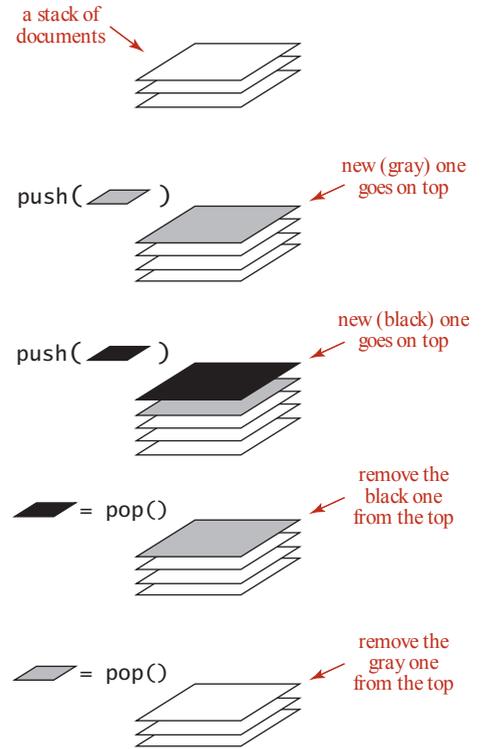
from people waiting in line at a theater, to cars waiting in line at a toll booth, to tasks waiting to be serviced by an application on your computer. One bedrock principle of any service policy is the perception of fairness. The first idea that comes to mind when most people think about fairness is that whoever has been waiting the longest should be served first. That is precisely the FIFO discipline. Queues are a natural model for many everyday phenomena, and they play a central role in numerous applications. When a client iterates through the items in a queue with the *foreach* construct, the items are processed in the order they were added to the queue. A typical reason to use a queue in an application is to save items in a collection while at the same time *preserving their* relative order: they come out in the same order in which they were put in. For example, the client below is a possible implementation of the `readInts()` static method from our `In` class. The problem that this method solves for the client is that the client can get numbers from a file into an array *without knowing the file size ahead of time*. We *enqueue* the numbers from the file, use the `size()` method from `Queue` to find the size needed for the array, create the array, and then *dequeue* the numbers to move them to the array. A queue is appropriate because it puts the numbers into the array in the order in which they appear in the file (we might use a `Bag` if that order is immaterial). This code uses autoboxing and auto-unboxing to convert between the client's `int` primitive type and the queue's `Integer` wrapper type.

```
public static int[] readInts(String name)
{
    In in = new In(name);
    Queue<Integer> q = new Queue<Integer>();
    while (!in.isEmpty())
        q.enqueue(in.readInt());

    int N = q.size();
    int[] a = new int[N];
    for (int i = 0; i < N; i++)
        a[i] = q.dequeue();
    return a;
}
```

Sample Queue client

Pushdown stacks. A *pushdown stack* (or just a stack) is a collection that is based on the *last-in-first-out* (LIFO) policy. When you keep your mail in a pile on your desk, you are using a stack. You pile pieces of new mail on the top when they arrive and take each piece of mail from the top when you are ready to read it. People do not process as many papers as they did in the past, but the same organizing principle underlies several of the applications that you use regularly on your computer. For example, many people organize their email as a stack—they *push* messages on the top when they are received and *pop* them from the top when they read them, with most recently received first (last in, first out). The advantage of this strategy is that we see interesting email as soon as possible; the disadvantage is that some old email might never get read if we never empty the stack. You have likely encountered another common example of a stack when surfing the web. When you click a hyperlink, your browser displays the new page (and pushes onto a stack). You can keep clicking on hyperlinks to visit new pages, but you can always revisit the previous page by clicking the back button (popping it from the stack). The LIFO policy offered by a stack provides just the behavior that you expect. When a client iterates through the items in a stack with the *foreach* construct, the items are processed in the reverse of the order in which they were added. A typical reason to use a stack iterator in an application is to save items in a collection while at the same time reversing their relative order. For example, the client `Reverse` at right reverses the order of the integers on standard input, again without having to know ahead of time how many there are. The importance of stacks in computing is fundamental and profound, as indicated in the detailed example that we consider next.



Operations on a pushdown stack

```
public class Reverse
{
    public static void main(String[] args)
    {
        Stack<Integer> stack;
        stack = new Stack<Integer>();
        while (!StdIn.isEmpty())
            stack.push(StdIn.readInt());

        for (int i : stack)
            StdOut.println(i);
    }
}
```

Sample Stack client

Arithmetic expression evaluation. As another example of a stack client, we consider a classic example that also demonstrates the utility of generics. Some of the first programs that we considered in SECTION 1.1 involved computing the value of arithmetic expressions like this one:

$$(1 + ((2 + 3) * (4 * 5)))$$

If you multiply 4 by 5, add 3 to 2, multiply the result, and then add 1, you get the value 101. But how does the Java system do this calculation? Without going into the details of how the Java system is built, we can address the essential ideas by writing a Java program that can take a string as input (the expression) and produce the number represented by the expression as output. For simplicity, we begin with the following explicit recursive definition: an *arithmetic expression* is either a number, or a left parenthesis followed by an arithmetic expression followed by an operator followed by another arithmetic expression followed by a right parenthesis. For simplicity, this definition is for fully parenthesized arithmetic expressions, which specify precisely which operators apply to which operands—you are a bit more familiar with expressions such as $1 + 2 * 3$, where we often rely on precedence rules instead of parentheses. The same basic mechanisms that we consider can handle precedence rules, but we avoid that complication. For specificity, we support the familiar binary operators $*$, $+$, $-$, and $/$, as well as a square-root operator `sqrt` that takes just one argument. We could easily allow more operators and more kinds of operators to embrace a large class of familiar mathematical expressions, involving trigonometric, exponential, and logarithmic functions. Our focus is on understanding how to interpret the string of parentheses, operators, and numbers to enable performing in the proper order the low-level arithmetic operations that are available on any computer. Precisely how can we convert an arithmetic expression—a string of characters—to the value that it represents? A remarkably simple algorithm that was developed by E. W. Dijkstra in the 1960s uses two stacks (one for operands and one for operators) to do this job. An expression consists of parentheses, operators, and operands (numbers). Proceeding from left to right and taking these entities one at a time, we manipulate the stacks according to four possible cases, as follows:

- Push operands onto the operand stack.
- Push operators onto the operator stack.
- Ignore left parentheses.
- On encountering a *right* parenthesis, pop an operator, pop the requisite number of operands, and push onto the operand stack the result of applying that operator to those operands.

After the final right parenthesis has been processed, there is one value on the stack, which is the value of the expression. This method may seem mysterious at first, but it

Dijkstra's Two-Stack Algorithm for Expression Evaluation

```

public class Evaluate
{
    public static void main(String[] args)
    {
        Stack<String> ops = new Stack<String>();
        Stack<Double> vals = new Stack<Double>();
        while (!StdIn.isEmpty())
        { // Read token, push if operator.
            String s = StdIn.readString();
            if (s.equals("(")) ;
            else if (s.equals("+")) ops.push(s);
            else if (s.equals("-")) ops.push(s);
            else if (s.equals("*")) ops.push(s);
            else if (s.equals("/")) ops.push(s);
            else if (s.equals("sqrt")) ops.push(s);
            else if (s.equals(")"))
            { // Pop, evaluate, and push result if token is ")".
                String op = ops.pop();
                double v = vals.pop();
                if (op.equals("+")) v = vals.pop() + v;
                else if (op.equals("-")) v = vals.pop() - v;
                else if (op.equals("*")) v = vals.pop() * v;
                else if (op.equals("/")) v = vals.pop() / v;
                else if (op.equals("sqrt")) v = Math.sqrt(v);
                vals.push(v);
            } // Token not operator or paren: push double value.
            else vals.push(Double.parseDouble(s));
        }
        StdOut.println(vals.pop());
    }
}

```

This Stack client uses two stacks to evaluate arithmetic expressions, illustrating an essential computational process: interpreting a string as a program and executing that program to compute the desired result. With generics, we can use the code in a single Stack implementation to implement one stack of String values and another stack of Double values. For simplicity, this code assumes that the expression is fully parenthesized, with numbers and characters separated by whitespace.

```

% java Evaluate
( 1 + ( ( 2 + 3 ) * ( 4 * 5 ) ) )
101.0

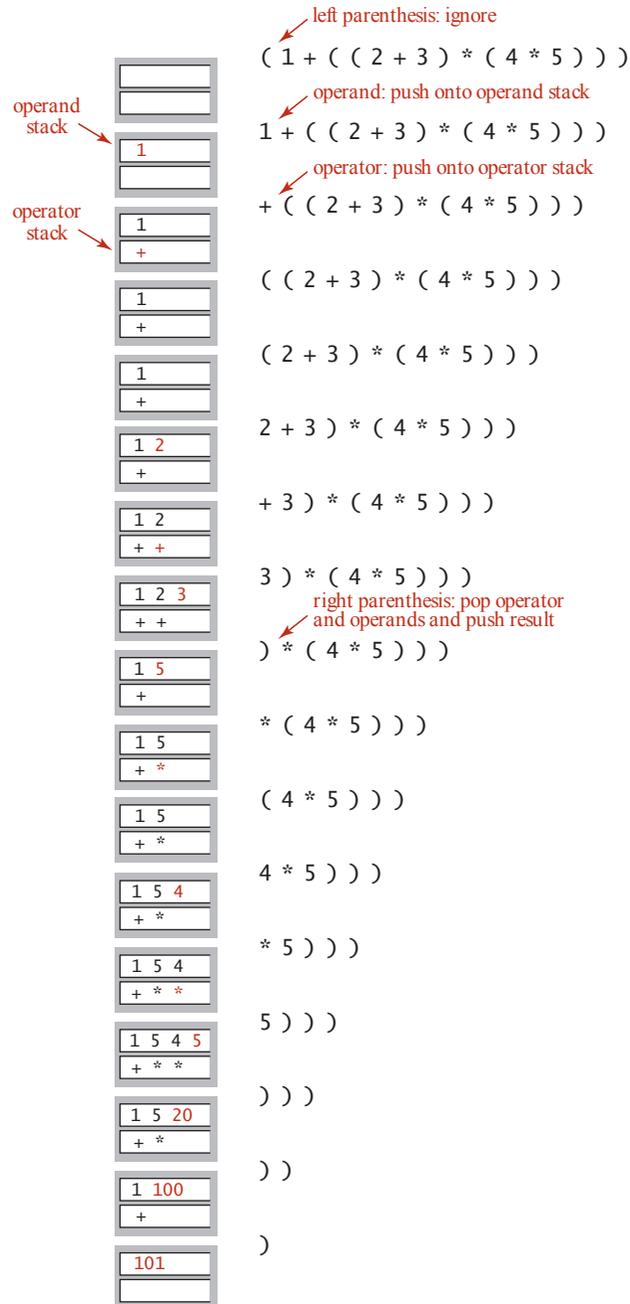
% java Evaluate
( ( 1 + sqrt ( 5.0 ) ) ) / 2.0 )
1.618033988749895

```

is easy to convince yourself that it computes the proper value: any time the algorithm encounters a subexpression consisting of two operands separated by an operator, all surrounded by parentheses, it leaves the result of performing that operation on those operands on the operand stack. The result is the same as if that value had appeared in the input instead of the subexpression, so we can think of replacing the subexpression by the value to get an expression that would yield the same result. We can apply this argument again and again until we get a single value. For example, the algorithm computes the same value for all of these expressions:

```
( 1 + ( ( 2 + 3 ) * ( 4 * 5 ) ) )  
( 1 + ( 5 * ( 4 * 5 ) ) )  
( 1 + ( 5 * 20 ) )  
( 1 + 100 )  
101
```

Evaluate on the previous page is an implementation of this algorithm. This code is a simple example of an interpreter: a program that interprets the computation specified by a given string and performs the computation to arrive at the result.



Trace of Dijkstra's two-stack arithmetic expression-evaluation algorithm

Implementing collections To address the issue of implementing Bag, Stack and Queue, we begin with a simple classic implementation, then address improvements that lead us to implementations of the APIs articulated on page 121.

Fixed-capacity stack. As a strawman, we consider an abstract data type for a fixed-capacity stack of strings, shown on the opposite page. The API differs from our Stack API: it works only for String values, it requires the client to specify a capacity, and it does not support iteration. The primary choice in developing an API implementation is to *choose a representation for the data*. For FixedCapacityStackOfStrings, an obvious choice is to use an array of String values. Pursuing this choice leads to the implementation shown at the bottom on the opposite page, which could hardly be simpler (each method is a one-liner). The instance variables are an array a[] that holds the items in the stack and an integer N that counts the number of items in the stack. To remove an item, we decrement N and then return a[N]; to insert a new item, we set a[N] equal to the new item and then increment N. These operations preserve the following properties:

- The items in the array are in their insertion order.
- The stack is empty when N is 0.
- The top of the stack (if it is nonempty) is at a[N-1].

As usual, thinking in terms of invariants of this sort is the easiest way to verify that an implementation operates as intended. *Be sure that you fully understand this implementation.* The best way to do so is to examine a trace of the stack contents for a sequence of

StdIn (push)	StdOut (pop)	N	a[]					
			0	1	2	3	4	
		0						
to		1	to					
be		2	to	be				
or		3	to	be	or			
not		4	to	be	or	not		
to		5	to	be	or	not	to	
-	to	4	to	be	or	not	to	
be		5	to	be	or	not	be	
-	be	4	to	be	or	not	be	
-	not	3	to	be	or	not	be	
that		4	to	be	or	that	be	
-	that	3	to	be	or	that	be	
-	or	2	to	be	or	that	be	
-	be	1	to	be	or	that	be	
is		2	to	is	or	not	to	

Trace of FixedCapacityStackOfStrings test client

operations, as illustrated at left for the test client, which reads strings from standard input and pushes each string onto a stack, unless it is "-", when it pops the stack and prints the result. The primary performance characteristic of this implementation is that the *push and pop operations take time independent of the stack size*. For many applications, it is the method of choice because of its simplicity. But it has several drawbacks that limit its potential applicability as a general-purpose tool, which we now address. With a moderate amount of effort (and some help from Java language mechanisms), we can develop an implementation that is broadly useful. This effort is worthwhile because the implementations that we develop serve as a model for implementations of other, more powerful, abstract data types throughout the book.

API public class FixedCapacityStackOfStrings

```

    FixedCapacityStackOfStrings(int cap) create an empty stack of capacity cap
    void push(String item) add a string
    String pop() remove the most recently added string
    boolean isEmpty() is the stack empty?
    int size() number of strings on the stack

```

test client

```

public static void main(String[] args)
{
    FixedCapacityStackOfStrings s;
    s = new FixedCapacityStackOfStrings(100);
    while (!StdIn.isEmpty())
    {
        String item = StdIn.readString();
        if (!item.equals("-"))
            s.push(item);
        else if (!s.isEmpty()) StdOut.print(s.pop() + " ");
    }
    StdOut.println("(" + s.size() + " left on stack)");
}

```

application

```

% more tobe.txt
to be or not to - be - - that - - - is
% java FixedCapacityStackOfStrings < tobe.txt
to be not that or be (2 left on stack)

```

implementation

```

public class FixedCapacityStackOfStrings
{
    private String[] a; // stack entries
    private int N;     // size

    public FixedCapacityStackOfStrings(int cap)
    { a = new String[cap]; }

    public boolean isEmpty() { return N == 0; }
    public int size()       { return N; }

    public void push(String item)
    { a[N++] = item; }

    public String pop()
    { return a[--N]; }
}

```

An abstract data type for a fixed-capacity stack of strings

Generics. The first drawback of `FixedCapacityStackOfStrings` is that it works only for `String` objects. If we want a stack of `double` values, we would need to develop another class with similar code, essentially replacing `String` with `double` everywhere. This is easy enough but becomes burdensome when we consider building a stack of `Transaction` values or a queue of `Date` values, and so forth. As discussed on page 122, Java’s parameterized types (generics) are specifically designed to address this situation, and we saw several examples of client code (on pages 125, 126, 127, and 129). But how do we implement a generic stack? The code on the facing page shows the details. It implements a class `FixedCapacityStack` that differs from `FixedCapacityStackOfStrings` only in the code highlighted in red—we replace every occurrence of `String` with `Item` (with one exception, discussed below) and declare the class with the following first line of code:

```
public class FixedCapacityStack<Item>
```

The name `Item` is a type parameter, a symbolic placeholder for some concrete type to be used by the client. You can read `FixedCapacityStack<Item>` as stack of items, which is precisely what we want. When implementing `FixedCapacityStack`, we do not know the actual type of `Item`, but a client can use our stack for any type of data by providing a concrete type when the stack is created. Concrete types must be reference types, but clients can depend on autoboxing to convert primitive types to their corresponding wrapper types. Java uses the type parameter `Item` to check for type mismatch errors—even though no concrete type is yet known, variables of type `Item` must be assigned values of type `Item`, and so forth. But there is one significant hitch in this story: We would like to implement the constructor in `FixedCapacityStack` with the code

```
a = new Item[cap];
```

which calls for creation of a generic array. For historical and technical reasons beyond our scope, generic array creation is disallowed in Java. Instead, we need to use a cast:

```
a = (Item[]) new Object[cap];
```

This code produces the desired effect (though the Java compiler gives a warning, which we can safely ignore), and we use this idiom throughout the book (the Java system library implementations of similar abstract data types use the same idiom).

API	<pre>public class FixedCapacityStack<Item> FixedCapacityStack(int cap) void push(Item item) Item pop() boolean isEmpty() int size()</pre>	<hr/> <p><i>create an empty stack of capacity cap</i></p> <p><i>add an item</i></p> <p><i>remove the most recently added item</i></p> <p><i>is the stack empty?</i></p> <p><i>number of items on the stack</i></p>
------------	---	--

test client

```
public static void main(String[] args)
{
    FixedCapacityStack<String> s;
    s = new FixedCapacityStack<String>(100);
    while (!StdIn.isEmpty())
    {
        String item = StdIn.readString();
        if (!item.equals("-"))
            s.push(item);
        else if (!s.isEmpty()) StdOut.print(s.pop() + " ");
    }
    StdOut.println("(" + s.size() + " left on stack)");
}
```

application

```
% more tobe.txt
to be or not to - be - - that - - - is
% java FixedCapacityStack < tobe.txt
to be not that or be (2 left on stack)
```

implementation

```
public class FixedCapacityStack<Item>
{
    private Item[] a; // stack entries
    private int N; // size

    public FixedCapacityStack(int cap)
    { a = (Item[]) new Object[cap]; }

    public boolean isEmpty() { return N == 0; }
    public int size() { return N; }

    public void push(Item item)
    { a[N++] = item; }

    public Item pop()
    { return a[--N]; }
}
```

An abstract data type for a fixed-capacity generic stack

Array resizing. Choosing an array to represent the stack contents implies that clients must estimate the maximum size of the stack ahead of time. In Java, we cannot change the size of an array once created, so the stack always uses space proportional to that maximum. A client that chooses a large capacity risks wasting a large amount of memory at times when the collection is empty or nearly empty. For example, a transaction system might involve billions of items and thousands of collections of them. Such a client would have to allow for the possibility that each of those collections could hold all of those items, even though a typical constraint in such systems is that each item can appear in only one collection. Moreover, every client risks *overflow* if the collection grows larger than the array. For this reason, `push()` needs code to test for a full stack, and we should have an `isFull()` method in the API to allow clients to test for that condition. We omit that code, because our desire is to relieve the client from having to deal with the concept of a full stack, as articulated in our original Stack API. Instead, we modify the array implementation to dynamically adjust the size of the array `a[]` so that it is both sufficiently large to hold all of the items and not so large as to waste an excessive amount of space. Achieving these goals turns out to be remarkably easy. First, we implement a method that moves a stack into an array of a different size:

```
private void resize(int max)
{ // Move stack of size N <= max to a new array of size max.
  Item[] temp = (Item[]) new Object[max];
  for (int i = 0; i < N; i++)
    temp[i] = a[i];
  a = temp;
}
```

Now, in `push()`, we check whether the array is too small. In particular, we check whether there is room for the new item in the array by checking whether the stack size `N` is equal to the array size `a.length`. If there is no room, we double the size of the array. Then we simply insert the new item with the code `a[N++] = item`, as before:

```
public void push(Item item)
{ // Add item to top of stack.
  if (N == a.length) resize(2*a.length);
  a[N++] = item;
}
```

Similarly, in `pop()`, we begin by deleting the item, then we *halve* the array size if it is too large. If you think a bit about the situation, you will see that the appropriate test is whether the stack size is less than *one-fourth* the array size. After the array is halved, it will be about half full and can accommodate a substantial number of `push()` and `pop()` operations before having to change the size of the array again.

```

public Item pop()
{ // Remove item from top of stack.
  String item = a[--N];
  a[N] = null; // Avoid loitering (see text).
  if (N > 0 && N == a.length/4) resize(a.length/2);
  return item;
}

```

With this implementation, the stack never overflows and never becomes less than one-quarter full (unless the stack is empty, when the array size is 2). We will address the performance analysis of this approach in more detail in SECTION 1.4.

Loitering. Java’s garbage collection policy is to reclaim the memory associated with any objects that can no longer be accessed. In our `pop()` implementations, the reference to the popped item remains in the array. The item is effectively an *orphan*—it will never be accessed again—but the Java garbage collector has no way to know this until it is overwritten. Even when the client is done with the item, the reference in the array may keep it alive. This condition (holding a reference to an item that is no longer needed) is known as loitering. In this case, loitering is easy to avoid, by setting the array entry corresponding to the popped item to `null`, thus overwriting the unused reference and making it possible for the system to reclaim the memory associated with the popped item when the client is finished with it.

push()	pop()	N	a.length	a[]								
				0	1	2	3	4	5	6	7	
		0	1	null								
to		1	1	to								
be		2	2	to	be							
or		3	4	to	be	or	null					
not		4	4	to	be	or	not					
to		5	8	to	be	or	not	to	null	null	null	
-	to	4	8	to	be	or	not	null	null	null	null	
be		5	8	to	be	or	not	be	null	null	null	
-	be	4	8	to	be	or	not	null	null	null	null	
-	not	3	8	to	be	or	null	null	null	null	null	
that		4	8	to	be	or	that	null	null	null	null	
-	that	3	8	to	be	or	null	null	null	null	null	
-	or	2	4	to	be	null	null					
-	be	1	2	to	null							
is		2	2	to	is							

Trace of array resizing during a sequence of `push()` and `pop()` operations

Iteration. As mentioned earlier in this section, one of the fundamental operations on collections is to process each item by iterating through the collection using Java's *foreach* statement. This paradigm leads to clear and compact code that is free from dependence on the details of a collection's implementation. To consider the task of implementing iteration, we start with a snippet of client code that prints all of the items in a collection of strings, one per line:

```
Stack<String> collection = new Stack<String>();
...
for (String s : collection)
    StdOut.println(s);
...
```

Now, this *foreach* statement is shorthand for a *while* construct (just like the *for* statement itself). It is essentially equivalent to the following *while* statement:

```
Iterator<String> i = collection.iterator();
while (i.hasNext())
{
    String s = i.next();
    StdOut.println(s);
}
```

This code exposes the ingredients that we need to implement in any iterable collection:

- The collection must implement an `iterator()` method that returns an `Iterator` object.
- The `Iterator` class must include two methods: `hasNext()` (which returns a `boolean` value) and `next()` (which returns a generic item from the collection).

In Java, we use the interface mechanism to express the idea that a class implements a specific method (see page 100). For iterable collections, the necessary interfaces are already defined for us in Java. To make a class iterable, the first step is to add the phrase `implements Iterable<Item>` to its declaration, matching the interface

```
public interface Iterable<Item>
{
    Iterator<Item> iterator();
}
```

(which is in `java.lang.Iterable`), and to add a method `iterator()` to the class that returns an `Iterator<Item>`. Iterators are generic, so we can use our parameterized type `Item` to allow clients to iterate through objects of whatever type is provided by our client. For the array representation that we have been using, we need to iterate through

an array in reverse order, so we name the iterator `ReverseArrayIterator` and add this method:

```
public Iterator<Item> iterator()
{ return new ReverseArrayIterator(); }
```

What is an iterator? An object from a class that implements the methods `hasNext()` and `next()`, as defined in the following interface (which is in `java.util.Iterator`):

```
public interface Iterator<Item>
{
    boolean hasNext();
    Item next();
    void remove();
}
```

Although the interface specifies a `remove()` method, we always use an empty method for `remove()` in this book, because interleaving iteration with operations that modify the data structure is best avoided. For `ReverseArrayIterator`, these methods are all one-liners, implemented in a nested class within our stack class:

```
private class ReverseArrayIterator implements Iterator<Item>
{
    private int i = N;

    public boolean hasNext() { return i > 0; }
    public Item next()      { return a[--i]; }
    public void remove()   { }
}
```

Note that this nested class can access the instance variables of the enclosing class, in this case `a[]` and `N` (this ability is the main reason we use nested classes for iterators). Technically, to conform to the `Iterator` specification, we should throw exceptions in two cases: an `UnsupportedOperationException` if a client calls `remove()` and a `NoSuchElementException` if a client calls `next()` when `i` is 0. Since we only use iterators in the *foreach* construction where these conditions do not arise, we omit this code. One crucial detail remains: we have to include

```
import java.util.Iterator;
```

at the beginning of the program because (for historical reasons) `Iterator` is not part of `java.lang` (even though `Iterable` is part of `java.lang`). Now a client using the *foreach* statement for this class will get behavior equivalent to the common `for` loop for arrays, but does not need to be aware of the array representation (an implementation

detail). This arrangement is of critical importance for implementations of fundamental data types like the collections that we consider in this book and those included in Java libraries. For example, it frees us to switch to a totally different representation *without having to change any client code*. More important, taking the client's point of view, it allows clients to use iteration *without having to know any details of the class implementation*.

ALGORITHM 1.1 is an implementation of our Stack API that resizes the array, allows clients to make stacks for any type of data, and supports client use of *foreach* to iterate through the stack items in LIFO order. This implementation is based on Java language nuances involving `Iterator` and `Iterable`, but there is no need to study those nuances in detail, as the code itself is not complicated and can be used as a template for other collection implementations.

For example, we can implement the Queue API by maintaining two indices as instance variables, a variable `head` for the beginning of the queue and a variable `tail` for the end of the queue. To remove an item, use `head` to access it and then increment `head`; to insert an item, use `tail` to store it, and then increment `tail`. If incrementing an index brings it past the end of the array, reset it to 0. Developing the details of checking when the queue is empty and when the array is full and needs resizing is an interesting and worthwhile programming exercise (see EXERCISE 1.3.14).

StdIn (<i>enqueue</i>)	StdOut (<i>dequeue</i>)	N	head	tail	a[]							
					0	1	2	3	4	5	6	7
		5	0	5	to	be	or	not	to			
-	to	4	1	5	to	be	or	not	to			
be		5	1	6	to	be	or	not	to	be		
-	be	4	2	6	to	be	or	not	to	be		
-	or	3	3	6	to	be	or	not	to	be		

Trace of `ResizingArrayQueue` test client

In the context of the study of algorithms, ALGORITHM 1.1 is significant because it almost (but not quite) achieves optimum performance goals for any collection implementation:

- Each operation should require time independent of the collection size.
- The space used should always be within a constant factor of the collection size.

The flaw in `ResizingArrayStack` is that some *push* and *pop* operations require resizing: this takes time proportional to the size of the stack. Next, we consider a way to correct this flaw, using a fundamentally different way to structure data.

ALGORITHM 1.1 Pushdown (LIFO) stack (resizing array implementation)

```

import java.util.Iterator;
public class ResizingArrayStack<Item> implements Iterable<Item>
{
    private Item[] a = (Item[]) new Object[1]; // stack items
    private int N = 0; // number of items

    public boolean isEmpty() { return N == 0; }
    public int size() { return N; }

    private void resize(int max)
    { // Move stack to a new array of size max.
      Item[] temp = (Item[]) new Object[max];
      for (int i = 0; i < N; i++)
          temp[i] = a[i];
      a = temp;
    }

    public void push(Item item)
    { // Add item to top of stack.
      if (N == a.length) resize(2*a.length);
      a[N++] = item;
    }

    public Item pop()
    { // Remove item from top of stack.
      Item item = a[--N];
      a[N] = null; // Avoid loitering (see text).
      if (N > 0 && N == a.length/4) resize(a.length/2);
      return item;
    }

    public Iterator<Item> iterator()
    { return new ReverseArrayIterator(); }

    private class ReverseArrayIterator implements Iterator<Item>
    { // Support LIFO iteration.
      private int i = N;
      public boolean hasNext() { return i > 0; }
      public Item next() { return a[--i]; }
      public void remove() { }
    }
}

```

This generic, iterable implementation of our Stack API is a model for collection ADTs that keep items in an array. It resizes the array to keep the array size within a constant factor of the stack size.

Linked lists Now we consider the use of a fundamental data structure that is an appropriate choice for representing the data in a collection ADT implementation. This is our first example of building a data structure that is not directly supported by the Java language. Our implementation serves as a model for the code that we use for building more complex data structures throughout the book, so you should read this section carefully, even if you have experience working with linked lists.

Definition. A linked list is a recursive data structure that is either empty (null) or a reference to a node having a generic item and a reference to a linked list.

The node in this definition is an abstract entity that might hold any kind of data, in addition to the node reference that characterizes its role in building linked lists. As with a recursive program, the concept of a recursive data structure can be a bit mindbending at first, but is of great value because of its simplicity.

Node record. With object-oriented programming, implementing linked lists is not difficult. We start with a nested class that defines the node abstraction:

```
private class Node
{
    Item item;
    Node next;
}
```

A `Node` has two instance variables: an `Item` (a parameterized type) and a `Node`. We define `Node` within the class where we want to use it, and make it `private` because it is not for use by clients. As with any data type, we create an object of type `Node` by invoking the (no-argument) constructor with `new Node()`. The result is a reference to a `Node` object whose instance variables are both initialized to the value `null`. The `Item` is a placeholder for any data that we might want to structure with a linked list (we will use Java's generic mechanism so that it can represent any reference type); the instance variable of type `Node` characterizes the linked nature of the data structure. To emphasize that we are just using the `Node` class to structure the data, we define no methods and we refer directly to the instance variables in code: if `first` is a variable associated with an object of type `Node`, we can refer to the instance variables with the code `first.item` and `first.next`. Classes of this kind are sometimes called *records*. They do not implement abstract data types because we refer directly to instance variables. However, `Node` and its client code are in the same class in all of our implementations and not accessible by clients of that class, so we still enjoy the benefits of data abstraction.

Building a linked list. Now, from the recursive definition, we can represent a linked list with a variable of type `Node` simply by ensuring that its value is either null or a reference to a `Node` whose `next` field is a reference to a linked list. For example, to build a linked list that contains the items `to`, `be`, and `or`, we create a `Node` for each item:

```
Node first = new Node();
Node second = new Node();
Node third = new Node();
```

and set the `item` field in each of the nodes to the desired value (for simplicity, these examples assume that `Item` is `String`):

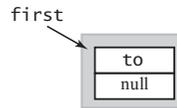
```
first.item = "to";
second.item = "be";
third.item = "or";
```

and set the `next` fields to build the linked list:

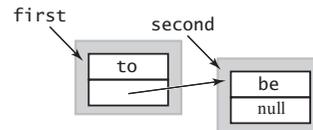
```
first.next = second;
second.next = third;
```

(Note that `third.next` remains `null`, the value it was initialized to at the time of creation.) As a result, `third` is a linked list (it is a reference to a node that has a reference to `null`, which is the null reference to an empty linked list), and `second` is a linked list (it is a reference to a node that has a reference to `third`, which is a linked list), and `first` is a linked list (it is a reference to a node that has a reference to `second`, which is a linked list). The code that we will examine does these assignment statements in a different order, depicted in the diagram on this page.

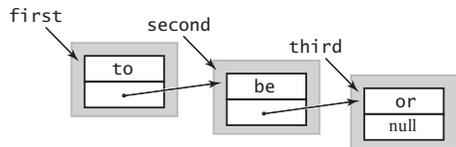
```
Node first = new Node();
first.item = "to";
```



```
Node second = new Node();
second.item = "be";
first.next = second;
```



```
Node third = new Node();
third.item = "or";
second.next = third;
```



Linking together a list

A LINKED LIST REPRESENTS A SEQUENCE of items. In the example just considered, `first` represents the sequence `to be or`. We can also use an array to represent a sequence of items. For example, we could use

```
String[] s = { "to", "be", "or" };
```

to represent the same sequence of strings. The difference is that it is easier to insert items into the sequence and to remove items from the sequence with linked lists. Next, we consider code to accomplish these tasks.

When tracing code that uses linked lists and other linked structures, we use a visual representation where

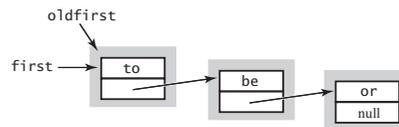
- We draw a rectangle to represent each object
- We put the values of instance variables within the rectangle
- We use arrows that point to the referenced objects to depict references

This visual representation captures the essential characteristic of linked lists. For economy, we use the term links to refer to node references. For simplicity, when item values are strings (as in our examples), we put the string within the object rectangle rather than the more accurate rendition depicting the string object and the character array that we discussed in SECTION 1.2. This visual representation allows us to focus on the links.

Insert at the beginning. First, suppose that you want to insert a new node into a linked list. The easiest place to do so is at the beginning of the list. For example, to insert the string `not` at the beginning of a given linked list whose first node is `first`, we save `first` in `oldfirst`, assign to `first` a new `Node`, and assign its `item` field to `not` and its `next` field to `oldfirst`. This code for inserting a node at the beginning of a linked list involves just a few assignment statements, so the amount of time that it takes is independent of the length of the list.

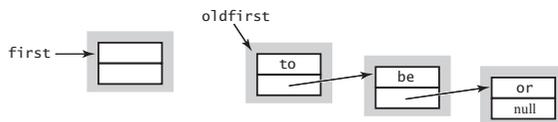
save a link to the list

```
Node oldfirst = first;
```



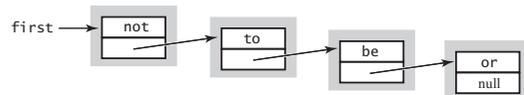
create a new node for the beginning

```
first = new Node();
```



set the instance variables in the new node

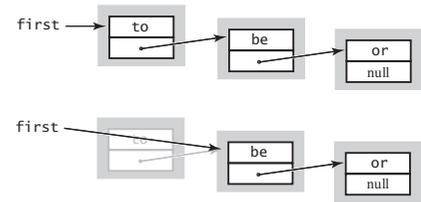
```
first.item = "not";
first.next = oldfirst;
```



Inserting a new node at the beginning of a linked list

Remove from the beginning. Next, suppose that you want to remove the first node from a list. This operation is even easier: simply assign to `first` the value `first.next`. Normally, you would retrieve the value of the item (by assigning it to some variable of type `Item`) before doing this assignment, because once you change the value of `first`, you may not have any access to the node to which it was referring. Typically, the node object becomes an orphan, and the Java memory management system eventually reclaims the memory it occupies. Again, this operation just involves one assignment statement, so its running time is independent of the length of the list.

```
first = first.next;
```



Removing the first node in a linked list

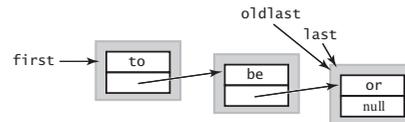
Insert at the end. How do we add a node to the end of a linked list? To do so, we need a link to the last node in the list, because that node's link has to be changed to reference a new node containing the item to be inserted. Maintaining an extra link is not something that should be taken lightly in linked-list code, because every method that modifies the list needs code to check whether that variable needs to be modified (and to make the necessary modifications). For example, the code that we just examined for removing the first node in the list might involve changing the reference to the last node in the list, since when there is only one node in the list, it is both the first one and the last one! Also, this code does not work (it follows a null link) in the case that the list is empty. Details like these make linked-list code notoriously difficult to debug.

Insert/remove at other positions. In summary, we have shown that we can implement the following operations on linked lists with just a few instructions, provided that we have access to both a link `first` to the first element in the list and a link `last` to the last element in the list:

- Insert at the beginning.
- Remove from the beginning.
- Insert at the end.

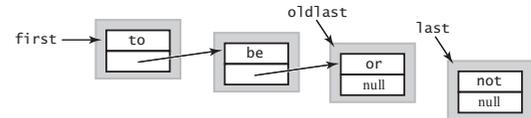
save a link to the last node

```
Node oldlast = last;
```



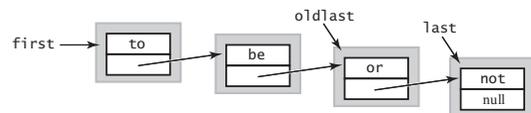
create a new node for the end

```
last = new Node();
last.item = "not";
```



link the new node to the end of the list

```
oldlast.next = last;
```



Inserting a new node at the end of a linked list

Other operations, such as the following, are not so easily handled:

- Remove a given node.
- Insert a new node before a given node.

For example, how can we remove the last node from a list? The link `last` is no help, because we need to set the link in the previous node in the list (the one with the same value as `last`) to `null`. In the absence of any other information, the only solution is to traverse the entire list looking for the node that links to `last` (see below and EXERCISE 1.3.19). Such a solution is undesirable because it takes time proportional to the length of the list. The standard solution to enable arbitrary insertions and deletions is to use a doubly-linked list, where each node has two links, one in each direction. We leave the code for these operations as an exercise (see EXERCISE 1.3.31). We do not need doubly linked lists for any of our implementations.

Traversal. To examine every item in an array, we use familiar code like the following loop for processing the items in an array `a[]`:

```
for (int i = 0; i < N; i++)
{
    // Process a[i].
}
```

There is a corresponding idiom for examining the items in a linked list: We initialize a loop index variable `x` to reference the first `Node` of the linked list. Then we find the item associated with `x` by accessing `x.item`, and then update `x` to refer to the next `Node` in the linked list, assigning to it the value of `x.next` and repeating this process until `x` is `null` (which indicates that we have reached the end of the linked list). This process is known as traversing the list and is succinctly expressed in code like the following loop for processing the items in a linked list whose first item is associated with the variable `first`:

```
for (Node x = first; x != null; x = x.next)
{
    // Process x.item.
}
```

This idiom is as natural as the standard idiom for iterating through the items in an array. In our implementations, we use it as the basis for iterators for providing client code the capability of iterating through the items, without having to know the details of the linked-list implementation.

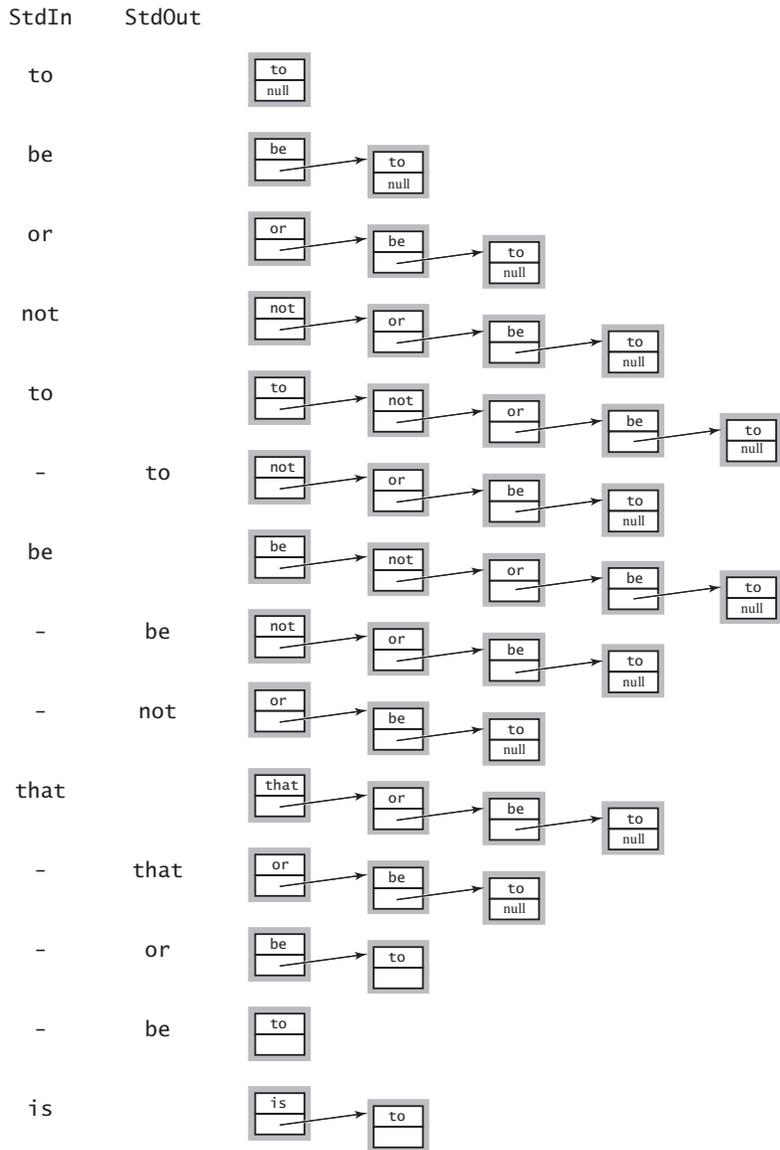
Stack implementation. Given these preliminaries, developing an implementation for our Stack API is straightforward, as shown in ALGORITHM 1.2 on page 149. It maintains the stack as a linked list, with the top of the stack at the beginning, referenced by an instance variable `first`. Thus, to `push()` an item, we add it to the beginning of the list, using the code discussed on page 144 and to `pop()` an item, we remove it from the beginning of the list, using the code discussed on page 145. To implement `size()`, we keep track of the number of items in an instance variable `N`, incrementing `N` when we push and decrementing `N` when we pop. To implement `isEmpty()` we check whether `first` is `null` (alternatively, we could check whether `N` is 0). The implementation uses the generic type `Item`—you can think of the code `<Item>` after the class name as meaning that any occurrence of `Item` in the implementation will be replaced by a client-supplied data-type name (see page 134). For now, we omit the code to support iteration, which we consider on page 155. A trace for the test client that we have been using is shown on the next page. This use of linked lists achieves our optimum design goals:

- It can be used for any type of data.
- The space required is always proportional to the size of the collection.
- The time per operation is always independent of the size of the collection.

This implementation is a prototype for many *algorithm* implementations that we consider. It defines the linked-list data structure and implements the client methods `push()` and `pop()` that achieve the specified effect with just a few lines of code. The algorithms and data structure go hand in hand. In this case, the code for the algorithm implementations is quite simple, but the properties of the data structure are not at all elementary, requiring explanations on the past several pages. This interaction between data structure definition and algorithm implementation is typical and is our focus in ADT implementations throughout this book.

```
public static void main(String[] args)
{ // Create a stack and push/pop strings as directed on StdIn.
    Stack<String> s = new Stack<String>();
    while (!StdIn.isEmpty())
    {
        String item = StdIn.readString();
        if (!item.equals("-"))
            s.push(item);
        else if (!s.isEmpty()) StdOut.print(s.pop() + " ");
    }
    StdOut.println("(" + s.size() + " left on stack)");
}
```

Test client for Stack



Trace of Stack development client

ALGORITHM 1.2 Pushdown stack (linked-list implementation)

```

public class Stack<Item> implements Iterable<Item>
{
    private Node first; // top of stack (most recently added node)
    private int N;      // number of items

    private class Node
    { // nested class to define nodes
        Item item;
        Node next;
    }

    public boolean isEmpty() { return first == null; }
    public int size()       { return N; }

    public void push(Item item)
    { // Add item to top of stack.
        Node oldfirst = first;
        first = new Node();
        first.item = item;
        first.next = oldfirst;
        N++;
    }

    public Item pop()
    { // Remove item from top of stack.
        Item item = first.item;
        first = first.next;
        N--;
        return item;
    }

    // See page 155 for iterator() implementation.
    // See page 147 for test client main().
}

```

This generic Stack implementation is based on a linked-list data structure. It can be used to create stacks containing any type of data. To support iteration, add the highlighted code described for Bag on page 155.

```

% more tobe.txt
to be or not to - be - - that - - - is

% java Stack < tobe.txt
to be not that or be (2 left on stack)

```

Queue implementation. An implementation of our Queue API based on the linked-list data structure is also straightforward, as shown in ALGORITHM 1.3 on the facing page. It maintains the queue as a linked list in order from least recently to most recently added items, with the beginning of the queue referenced by an instance variable `first` and the end of the queue referenced by an instance variable `last`. Thus, to `enqueue()` an item, we add it to the end of the list (using the code discussed on page 145, augmented to set both `first` and `last` to refer to the new node when the list is empty) and to `dequeue()` an item, we remove it from the beginning of the list (using the same code as for `pop()` in `Stack`, augmented to update `last` when the list becomes empty). The implementations of `size()` and `isEmpty()` are the same as for `Stack`. As with `Stack` the implementation uses the generic type parameter `Item`, and we omit the code to support iteration, which we consider in our `Bag` implementation on page 155. A development client similar to the one we used for `Stack` is shown below, and the trace for this client is shown on the following page. This implementation uses the same data structure as does `Stack`—a linked list—but it implements different *algorithms* for adding and removing items, which make the difference between LIFO and FIFO for the client. Again, the use of linked lists achieves our optimum design goals: it can be used for any type of data, the space required is proportional to the number of items in the collection, and the time required per operation is always independent of the size of the collection.

```
public static void main(String[] args)
{ // Create a queue and enqueue/dequeue strings.
    Queue<String> q = new Queue<String>();
    while (!StdIn.isEmpty())
    {
        String item = StdIn.readString();
        if (!item.equals("-"))
            q.enqueue(item);
        else if (!q.isEmpty()) StdOut.print(q.dequeue() + " ");
    }
    StdOut.println("(" + q.size() + " left on queue");
}
```

Test client for Queue

```
% more tobe.txt
to be or not to - be - - that - - - is
% java Queue < tobe.txt
to be or not to be (2 left on queue)
```

ALGORITHM 1.3 FIFO queue

```
public class Queue<Item> implements Iterable<Item>
{
    private Node first; // link to least recently added node
    private Node last;  // link to most recently added node
    private int N;      // number of items on the queue

    private class Node
    { // nested class to define nodes
        Item item;
        Node next;
    }

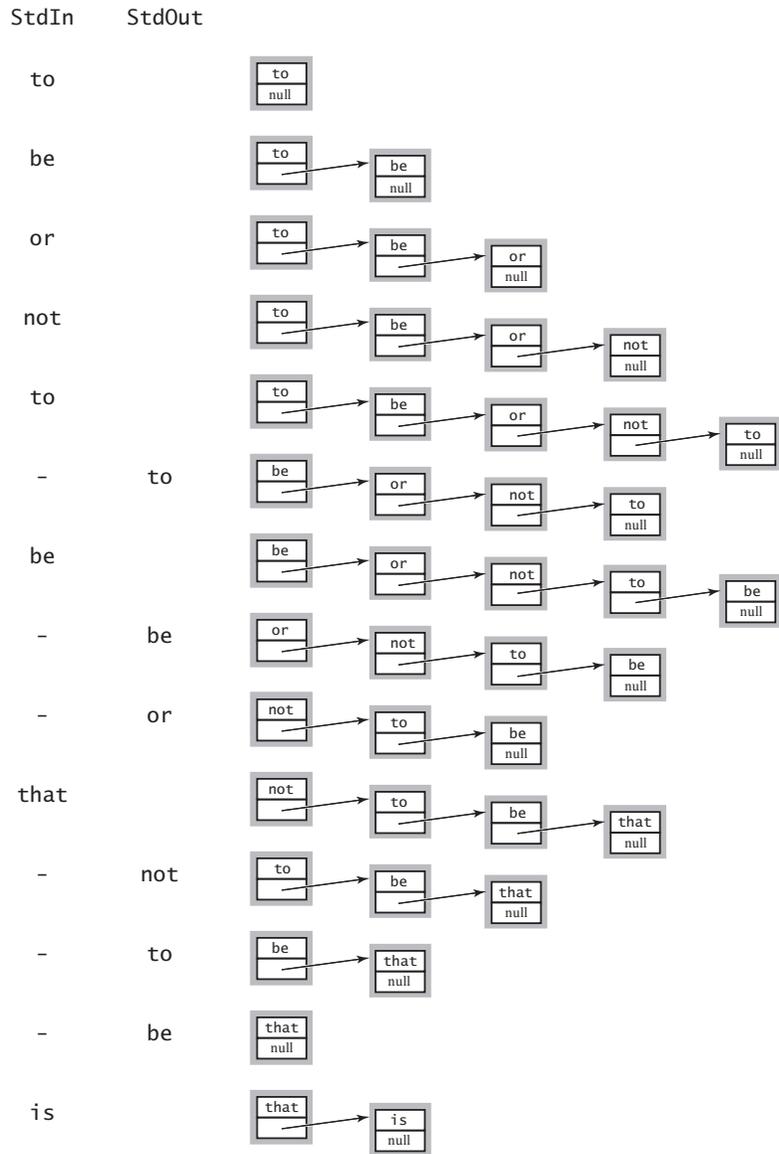
    public boolean isEmpty() { return first == null; }
    public int size()       { return N; }

    public void enqueue(Item item)
    { // Add item to the end of the list.
        Node oldlast = last;
        last = new Node();
        last.item = item;
        last.next = null;
        if (isEmpty()) first = last;
        else          oldlast.next = last;
        N++;
    }

    public Item dequeue()
    { // Remove item from the beginning of the list.
        Item item = first.item;
        first = first.next;
        N--;
        if (isEmpty()) last = null;
        return item;
    }

    // See page 155 for iterator() implementation.
    // See page 150 for test client main().
}
```

This generic Queue implementation is based on a linked-list data structure. It can be used to create queues containing any type of data. To support iteration, add the highlighted code described for Bag on page 155.



Trace of Queue development client

LINKED LISTS ARE A FUNDAMENTAL ALTERNATIVE to arrays for structuring a collection of data. From a historical perspective, this alternative has been available to programmers for many decades. Indeed, a landmark in the history of programming languages was the development of LISP by John McCarthy in the 1950s, where linked lists are the primary structure for programs and data. Programming with linked lists presents all sorts of challenges and is notoriously difficult to debug, as you can see in the exercises. In modern code, the use of safe pointers, automatic garbage collection (page 111), and ADTs allows us to encapsulate list-processing code in just a few classes such as the ones presented here.

Bag implementation. Implementing our Bag API using a linked-list data structure is simply a matter of changing the name of `push()` in `Stack` to `add()` and removing the implementation of `pop()`, as shown in ALGORITHM 1.4 on the facing page (doing the same for `Queue` would also be effective but requires a bit more code). This implementation also highlights the code needed to make `Stack`, `Queue`, and `Bag` all iterable, by traversing the list. For `Stack` the list is in LIFO order; for `Queue` it is in FIFO order; and for `Bag` it happens to be in LIFO order, but the order is not relevant. As detailed in the highlighted code in ALGORITHM 1.4, to implement iteration in a collection, the first step is to include

```
import java.util.Iterator;
```

so that our code can refer to Java's `Iterator` interface. The second step is to add

```
implements Iterable<Item>
```

to the class declaration, a promise to provide an `iterator()` method. The `iterator()` method itself simply returns an object from a class that implements the `Iterator` interface:

```
public Iterator<Item> iterator()  
{ return new ListIterator(); }
```

This code is a promise to implement a class that implements the `hasNext()`, `next()`, and `remove()` methods that are called when a client uses the *foreach* construct. To implement these methods, the nested class `ListIterator` in ALGORITHM 1.4 maintains an instance variable `current` that keeps track of the current node on the list. Then the `hasNext()` method tests if `current` is `null`, and the `next()` method saves a reference to the current item, updates `current` to refer to the next node on the list, and returns the saved reference.

ALGORITHM 1.4 Bag

```
import java.util.Iterator;
public class Bag<Item> implements Iterable<Item>
{
    private Node first; // first node in list
    private class Node
    {
        Item item;
        Node next;
    }
    public void add(Item item)
    { // same as push() in Stack
        Node oldfirst = first;
        first = new Node();
        first.item = item;
        first.next = oldfirst;
    }
    public Iterator<Item> iterator()
    { return new ListIterator(); }
    private class ListIterator implements Iterator<Item>
    {
        private Node current = first;
        public boolean hasNext()
        { return current != null; }
        public void remove() { }
        public Item next()
        {
            Item item = current.item;
            current = current.next;
            return item;
        }
    }
}
```

This Bag implementation maintains a linked list of the items provided in calls to `add()`. Code for `isEmpty()` and `size()` is the same as in Stack and is omitted. The iterator traverses the list, maintaining the current node in `current`. We can make Stack and Queue iterable by adding the code highlighted in red to ALGORITHMS 1.2 and ALGORITHM 1.3, because they use the same underlying data structure and Stack and Queue maintain the list in LIFO and FIFO order, respectively.

Overview The implementations of bags, queues, and stacks that support generics and iteration that we have considered in this section provide a level of abstraction that allows us to write compact client programs that manipulate collections of objects. Detailed understanding of these ADTs is important as an introduction to the study of algorithms and data structures for three reasons. First, we use these data types as building blocks in higher-level data structures throughout this book. Second, they illustrate the interplay between data structures and algorithms and the challenge of simultaneously achieving natural performance goals that may conflict. Third, the focus of several of our implementations is on ADTs that support more powerful operations on collections of objects, and we use the implementations here as starting points.

Data structures. We now have two ways to represent collections of objects, arrays and linked lists. Arrays are built into Java; linked lists are easy to build with standard Java records. These two alternatives, often referred to as *sequential allocation* and linked allocation, are fundamental. Later in the book, we develop ADT implementations that

data structure	advantage	disadvantage
<i>array</i>	index provides immediate access to any item	need to know size on initialization
<i>linked list</i>	uses space proportional to size	need reference to access an item

Fundamental data structures

combine and extend these basic structures in numerous ways. One important extension is to data structures with multiple links. For example, our focus in SECTIONS 3.2 and 3.3 is on data structures known as binary trees that are built from nodes that each have two links. Another important extension is to compose data structures: we can have a bag of stacks, a queue of arrays, and so forth. For example, our focus in CHAPTER 4 is on graphs, which we represent as arrays of bags. It is very easy to define data structures of arbitrary complexity in this way: one important reason for our focus on abstract data types is an attempt to control such complexity.

combine and extend these basic structures in numerous ways. One important extension is to data structures with multiple links. For example, our focus in SECTIONS 3.2 and 3.3 is on data structures known as binary trees that are built from nodes that each have two links. Another important extension is to compose data structures: we can have a bag of stacks, a queue of arrays, and so forth. For example, our focus in CHAPTER 4 is on graphs, which we represent as arrays of bags. It is very easy to define data structures of arbitrary complexity in this way: one important reason for our focus on abstract data types is an attempt to control such complexity.

OUR TREATMENT OF BAGS, QUEUES, AND STACKS in this section is a prototypical example of the approach that we use throughout this book to describe data structures and algorithms. In approaching a new applications domain, we identify computational challenges and use data abstraction to address them, proceeding as follows:

- Specify an API.
- Develop client code with reference to specific applications.
- Describe a data structure (representation of the set of values) that can serve as the basis for the instance variables in a class that will implement an ADT that meets the specification in the API.
- Describe algorithms (approaches to implementing the set of operations) that can serve as the basis for implementing the *instance methods* in the class.
- Analyze the performance characteristics of the algorithms.

In the next section, we consider this last step in detail, as it often dictates which algorithms and implementations can be most useful in addressing real-world applications.

data structure	section	ADT	representation
<i>parent-link tree</i>	1.5	UnionFind	array of integers
<i>binary search tree</i>	3.2, 3.3	BST	two links per node
<i>string</i>	5.1	String	array, offset, and length
<i>binary heap</i>	2.4	PQ	array of objects
<i>hash table (separate chaining)</i>	3.4	SeparateChainingHashST	arrays of linked lists
<i>hash table (linear probing)</i>	3.4	LinearProbingHashST	two arrays of objects
<i>graph adjacency lists</i>	4.1, 4.2	Graph	array of Bag objects
<i>trie</i>	5.2	TrieST	node with array of links
<i>ternary search trie</i>	5.3	TST	three links per node

Examples of data structures developed in this book

Q&A

Q. Not all programming languages have generics, even early versions of Java. What are the alternatives?

A. One alternative is to maintain a different implementation for each type of data, as mentioned in the text. Another is to build a stack of `Object` values, then cast to the desired type in client code for `pop()`. The problem with this approach is that type mismatch errors cannot be detected until run time. But with generics, if you write code to push an object of the wrong type on the stack, like this:

```
Stack<Apple> stack = new Stack<Apple>();
Apple a = new Apple();
...
Orange b = new Orange();
...
stack.push(a);
...
stack.push(b);    // compile-time error
```

you will get a compile-time error:

```
push(Apple) in Stack<Apple> cannot be applied to (Orange)
```

This ability to discover such errors at compile time is reason enough to use generics.

Q. Why does Java disallow generic arrays?

A. Experts still debate this point. You might need to become one to understand it! For starters, learn about covariant arrays and type erasure.

Q. How do I create an array of stacks of strings?

A. Use a cast, such as the following:

```
Stack<String>[] a = (Stack<String>[]) new Stack[N];
```

Warning: This cast, in client code, is different from the one described on page 134. You might have expected to use `Object` instead of `Stack`. When using generics, Java checks for type safety at compile time, but throws away that information at run time, so it is left with `Stack<Object>[]` or just `Stack[]`, for short, which we must cast to `Stack<String>[]`.

Q. What happens if my program calls `pop()` for an empty stack?

A. It depends on the implementation. For our implementation on page 149, you will get a `NullPointerException`. In our implementations on the booksite, we throw a runtime exception to help users pinpoint the error. Generally, including as many such checks as possible is wise in code that is likely to be used by many people.

Q. Why do we care about resizing arrays, when we have linked lists?

A. We will see several examples of ADT implementations that need to use arrays to perform other operations that are not easily supported with linked lists. `ResizingArrayStack` is a model for keeping their memory usage under control.

Q. Why declare `Node` as a nested class? Why `private`?

A. By declaring the nested class `Node` to be `private`, we restrict its access only to methods and instance variables within the enclosing class. One characteristic of a `private` nested class is that its instance variables can be directly accessed from within the enclosing class but nowhere else, so there is no need to declare the instance variables `public` or `private`. Note for experts: A nested class that is not static is known as an inner class, so technically our `Node` classes are inner classes.

Q. When I type `javac Stack.java` to compile ALGORITHM 1.2 and similar programs, I find `Stack.class` and a file `Stack$Node.class`. What is the purpose of that second one?

A. That file is for the inner class `Node`. Java's naming convention is to use `$` to separate the name of the outer class from the inner class.

Q. Are there Java libraries for stacks and queues?

A. Yes and no. Java has a built-in library called `java.util.Stack`, but you should avoid using it when you want a stack. It has several additional operations that are not normally associated with a stack, e.g., getting the `i`th element. It also allows adding an element to the bottom of the stack (instead of the top), so it can implement a queue! Although having such extra operations may appear to be a bonus, it is actually a curse. We use data types not just as libraries of all the operations we can imagine, but also as a mechanism to precisely specify the operations we need. The prime benefit of doing so is that the system can prevent us from performing operations that we do not actually want. The `java.util.Stack` API is an example of a wide interface, which we generally strive to avoid.

Q & A *(continued)*

Q. Should a client be allowed to insert `null` items onto a stack or queue?

A. This question arises frequently when implementing collections in Java. Our implementation (and Java's stack and queue libraries) do permit the insertion of `null` values.

Q. What should the `Stack` iterator do if the client calls `push()` or `pop()` during iteration?

A. Throw a `java.util.ConcurrentModificationException` to make it a fail-fast iterator. See EXERCISE 1.3.50.

Q. Can I use a *foreach* loop with arrays?

A. Yes (even though arrays do not implement the `Iterable` interface). The following one-liner prints out the command-line arguments:

```
public static void main(String[] args)
{ for (String s : args) StdOut.println(s); }
```

Q. Can I use a *foreach* loop with strings?

A. No. `String` does not implement `Iterable`.

Q. Why not have a single `Collection` data type that implements methods to add items, remove the most recently inserted, remove the least recently inserted, remove random, iterate, return the number of items in the collection, and whatever other operations we might desire? Then we could get them all implemented in a single class that could be used by many clients.

A. Again, this is an example of a wide interface. Java has such implementations in its `java.util.ArrayList` and `java.util.LinkedList` classes. One reason to avoid them is that there is no assurance that all operations are implemented efficiently. Throughout this book, we use APIs as starting points for designing efficient algorithms and data structures, which is certainly easier to do for interfaces with just a few operations as opposed to an interface with many operations. Another reason to insist on narrow interfaces is that they enforce a certain discipline on client programs, which makes client code much easier to understand. If one client uses `Stack<String>` and another uses `Queue<Transaction>`, we have a good idea that the LIFO discipline is important to the first and the FIFO discipline is important to the second.

EXERCISES

1.3.1 Add a method `isFull()` to `FixedCapacityStackOfStrings`.

1.3.2 Give the output printed by `java Stack` for the input

```
it was - the best - of times - - - it was - the - -
```

1.3.3 Suppose that a client performs an intermixed sequence of (stack) *push* and *pop* operations. The push operations put the integers 0 through 9 in order onto the stack; the pop operations print out the return values. Which of the following sequence(s) could not occur?

a. 4 3 2 1 0 9 8 7 6 5

b. 4 6 8 7 5 3 2 9 0 1

c. 2 5 6 7 4 8 9 3 1 0

d. 4 3 2 1 0 5 6 7 8 9

e. 1 2 3 4 5 6 9 8 7 0

f. 0 4 6 5 3 8 1 7 2 9

g. 1 4 7 9 8 6 5 3 0 2

h. 2 1 4 3 6 5 8 7 9 0

1.3.4 Write a stack client `Parentheses` that reads in a text stream from standard input and uses a stack to determine whether its parentheses are properly balanced. For example, your program should print `true` for `[()]{ }{[()] }()` and `false` for `[()]`.

1.3.5 What does the following code fragment print when `N` is 50? Give a high-level description of what it does when presented with a positive integer `N`.

```
Stack<Integer> stack = new Stack<Integer>();
while (N > 0)
{
    stack.push(N % 2);
    N = N / 2;
}
for (int d : stack) StdOut.print(d);
StdOut.println();
```

Answer: Prints the binary representation of `N` (110010 when `N` is 50).

EXERCISES *(continued)*

1.3.6 What does the following code fragment do to the queue `q`?

```
Stack<String> stack = new Stack<String>();
while (!q.isEmpty())
    stack.push(q.dequeue());
while (!stack.isEmpty())
    q.enqueue(stack.pop());
```

1.3.7 Add a method `peek()` to `Stack` that returns the most recently inserted item on the stack (without popping it).

1.3.8 Give the contents and size of the array for `ResizingArrayStackOfStrings` with the input

it was - the best - of times - - - it was - the - -

1.3.9 Write a program that takes from standard input an expression without left parentheses and prints the equivalent infix expression with the parentheses inserted. For example, given the input:

1 + 2) * 3 - 4) * 5 - 6)))

your program should print

((1 + 2) * ((3 - 4) * (5 - 6))))

1.3.10 Write a filter `InfixToPostfix` that converts an arithmetic expression from infix to postfix.

1.3.11 Write a program `EvaluatePostfix` that takes a postfix expression from standard input, evaluates it, and prints the value. (Piping the output of your program from the previous exercise to this program gives equivalent behavior to `Evaluate`.)

1.3.12 Write an iterable `Stack` client that has a static method `copy()` that takes a stack of strings as argument and returns a copy of the stack. Note: This ability is a prime example of the value of having an iterator, because it allows development of such functionality without changing the basic API.

1.3.13 Suppose that a client performs an intermixed sequence of (queue) *enqueue* and *dequeue* operations. The enqueue operations put the integers 0 through 9 in order onto

the queue; the dequeue operations print out the return value. Which of the following sequence(s) could not occur?

- a. 0 1 2 3 4 5 6 7 8 9
- b. 4 6 8 7 5 3 2 9 0 1
- c. 2 5 6 7 4 8 9 3 1 0
- d. 4 3 2 1 0 5 6 7 8 9

1.3.14 Develop a class `ResizingArrayQueueOfStrings` that implements the queue abstraction with a fixed-size array, and then extend your implementation to use array resizing to remove the size restriction.

1.3.15 Write a `Queue` client that takes a command-line argument `k` and prints the `k`th from the last string found on standard input (assuming that standard input has `k` or more strings).

1.3.16 Using `readInts()` on page 126 as a model, write a static method `readDates()` for `Date` that reads dates from standard input in the format specified in the table on page 119 and returns an array containing them.

1.3.17 DO EXERCISE 1.3.16 for `Transaction`.

LINKED-LIST EXERCISES

This list of exercises is intended to give you experience in working with linked lists. Suggestion: make drawings using the visual representation described in the text.

1.3.18 Suppose `x` is a linked-list node and not the last node on the list. What is the effect of the following code fragment?

```
x.next = x.next.next;
```

Answer: Deletes from the list the node immediately following `x`.

1.3.19 Give a code fragment that removes the last node in a linked list whose first node is `first`.

1.3.20 Write a method `delete()` that takes an `int` argument `k` and deletes the `k`th element in a linked list, if it exists.

1.3.21 Write a method `find()` that takes a linked list and a string `key` as arguments and returns `true` if some node in the list has `key` as its `item` field, `false` otherwise.

1.3.22 Suppose that `x` is a linked list `Node`. What does the following code fragment do?

```
t.next = x.next;  
x.next = t;
```

Answer: Inserts node `t` immediately after node `x`.

1.3.23 Why does the following code fragment not do the same thing as in the previous question?

```
x.next = t;  
t.next = x.next;
```

Answer: When it comes time to update `t.next`, `x.next` is no longer the original node following `x`, but is instead `t` itself!

1.3.24 Write a method `removeAfter()` that takes a linked-list `Node` as argument and removes the node following the given one (and does nothing if the argument or the next field in the argument node is null).

1.3.25 Write a method `insertAfter()` that takes two linked-list `Node` arguments and inserts the second after the first on its list (and does nothing if either argument is null).

1.3.26 Write a method `remove()` that takes a linked list and a string key as arguments and removes all of the nodes in the list that have key as its item field.

1.3.27 Write a method `max()` that takes a reference to the first node in a linked list as argument and returns the value of the maximum key in the list. Assume that all keys are positive integers, and return 0 if the list is empty.

1.3.28 Develop a recursive solution to the previous question.

1.3.29 Write a Queue implementation that uses a circular linked list, which is the same as a linked list except that no links are null and the value of `last.next` is `first` whenever the list is not empty. Keep only one Node instance variable (`last`).

1.3.30 Write a function that takes the first Node in a linked list as argument and (destructively) reverses the list, returning the first Node in the result.

Iterative solution: To accomplish this task, we maintain references to three consecutive nodes in the linked list, `reverse`, `first`, and `second`. At each iteration, we extract the node `first` from the original linked list and insert it at the beginning of the reversed list. We maintain the invariant that `first` is the first node of what's left of the original list, `second` is the second node of what's left of the original list, and `reverse` is the first node of the resulting reversed list.

```
public Node reverse(Node x)
{
    Node first = x;
    Node reverse = null;
    while (first != null)
    {
        Node second = first.next;
        first.next = reverse;
        reverse = first;
        first = second;
    }
    return reverse;
}
```

When writing code involving linked lists, we must always be careful to properly handle the exceptional cases (when the linked list is empty, when the list has only one or two

LINKED-LIST EXERCISES *(continued)*

nodes) and the boundary cases (dealing with the first or last items). This is usually much trickier than handling the normal cases.

Recursive solution: Assuming the linked list has N nodes, we recursively reverse the last $N-1$ nodes, and then carefully append the first node to the end.

```
public Node reverse(Node first)
{
    if (first == null) return null;
    if (first.next == null) return first;
    Node second = first.next;
    Node rest = reverse(second);
    second.next = first;
    first.next = null;
    return rest;
}
```

1.3.31 Implement a nested class `DoubleNode` for building doubly-linked lists, where each node contains a reference to the item preceding it and the item following it in the list (`null` if there is no such item). Then implement static methods for the following tasks: insert at the beginning, insert at the end, remove from the beginning, remove from the end, insert before a given node, insert after a given node, and remove a given node.

CREATIVE PROBLEMS

1.3.32 *Steque*. A stack-ended queue or *steque* is a data type that supports *push*, *pop*, and *enqueue*. Articulate an API for this ADT. Develop a linked-list-based implementation.

1.3.33 *Deque*. A double-ended queue or *deque* (pronounced “deck”) is like a stack or a queue but supports adding and removing items at both ends. A deque stores a collection of items and supports the following API:

```
public class Deque<Item> implements Iterable<Item>
    Deque()                create an empty deque
    boolean isEmpty()      is the deque empty?
    int size()             number of items in the deque
    void pushLeft(Item item) add an item to the left end
    void pushRight(Item item) add an item to the right end
    Item popLeft()         remove an item from the left end
    Item popRight()        remove an item from the right end
```

API for a generic double-ended queue

Write a class `Deque` that uses a doubly-linked list to implement this API and a class `ResizingArrayDeque` that uses a resizing array.

1.3.34 *Random bag*. A random bag stores a collection of items and supports the following API:

```
public class RandomBag<Item> implements Iterable<Item>
    RandomBag()            create an empty random bag
    boolean isEmpty()      is the bag empty?
    int size()             number of items in the bag
    void add(Item item)    add an item
```

API for a generic random bag

Write a class `RandomBag` that implements this API. Note that this API is the same as for `Bag`, except for the adjective `random`, which indicates that the iteration should provide

CREATIVE PROBLEMS *(continued)*

the items in random order (all $N!$ permutations equally likely, for each iterator). *Hint:* Put the items in an array and randomize their order in the iterator's constructor.

1.3.35 *Random queue.* A *random queue* stores a collection of items and supports the following API:

```
public class RandomQueue<Item>
    RandomQueue()           create an empty random queue
    boolean isEmpty()       is the queue empty?
    void enqueue(Item item) add an item
    Item dequeue()          remove and return a random item
                           (sample without replacement)
    Item sample()           return a random item, but do not remove
                           (sample with replacement)
```

API for a generic random queue

Write a class `RandomQueue` that implements this API. *Hint:* Use an array representation (with resizing). To remove an item, swap one at a random position (indexed 0 through $N-1$) with the one at the last position (index $N-1$). Then delete and return the last object, as in `ResizingArrayStack`. Write a client that deals bridge hands (13 cards each) using `RandomQueue<Card>`.

1.3.36 *Random iterator.* Write an iterator for `RandomQueue<Item>` from the previous exercise that returns the items in random order.

1.3.37 *Josephus problem.* In the Josephus problem from antiquity, N people are in dire straits and agree to the following strategy to reduce the population. They arrange themselves in a circle (at positions numbered from 0 to $N-1$) and proceed around the circle, eliminating every M th person until only one person is left. Legend has it that Josephus figured out where to sit to avoid being eliminated. Write a `Queue` client `Josephus` that takes M and N from the command line and prints out the order in which people are eliminated (and thus would show Josephus where to sit in the circle).

```
% java Josephus 2 7
1 3 5 0 4 2 6
```

1.3.38 Delete *k*th element. Implement a class that supports the following API:

```
public class GeneralizedQueue<Item>
{
    GeneralizedQueue()           create an empty queue
    boolean isEmpty()           is the queue empty?
    void insert(Item x)         add an item
    Item delete(int k)          delete and return the kth least recently inserted item
}
```

API for a generic generalized queue

First, develop an implementation that uses an array implementation, and then develop one that uses a linked-list implementation. Note: the algorithms and data structures that we introduce in CHAPTER 3 make it possible to develop an implementation that can guarantee that both `insert()` and `delete()` take time proportional to the logarithm of the number of items in the queue—see EXERCISE 3.5.27.

1.3.39 *Ring buffer.* A ring buffer, or circular queue, is a FIFO data structure of a fixed size N . It is useful for transferring data between asynchronous processes or for storing log files. When the buffer is empty, the consumer waits until data is deposited; when the buffer is full, the producer waits to deposit data. Develop an API for a `RingBuffer` and an implementation that uses an array representation (with circular wrap-around).

1.3.40 *Move-to-front.* Read in a sequence of characters from standard input and maintain the characters in a linked list with no duplicates. When you read in a previously unseen character, insert it at the front of the list. When you read in a duplicate character, delete it from the list and reinsert it at the beginning. Name your program `MoveToFront`: it implements the well-known move-to-front strategy, which is useful for caching, data compression, and many other applications where items that have been recently accessed are more likely to be reaccessed.

1.3.41 *Copy a queue.* Create a new constructor so that

```
Queue<Item> r = new Queue<Item>(q);
```

makes `r` a reference to a new and independent copy of the queue `q`. You should be able to enqueue and dequeue from either `q` or `r` without influencing the other. *Hint*: Delete all of the elements from `q` and add these elements to both `q` and `r`.

CREATIVE PROBLEMS *(continued)*

1.3.42 *Copy a stack.* Create a new constructor for the linked-list implementation of Stack so that

```
Stack<Item> t = new Stack<Item>(s);
```

makes `t` a reference to a new and independent copy of the stack `s`.

1.3.43 *Listing files.* A folder is a list of files and folders. Write a program that takes the name of a folder as a command-line argument and prints out all of the files contained in that folder, with the contents of each folder recursively listed (indented) under that folder's name. *Hint:* Use a queue, and see `java.io.File`.

1.3.44 *Text editor buffer.* Develop a data type for a buffer in a text editor that implements the following API:

```
public class Buffer
```

Buffer()	<i>create an empty buffer</i>
void insert(char c)	<i>insert c at the cursor position</i>
char get()	<i>character at the cursor position</i>
char delete()	<i>delete and return the character at the cursor</i>
void left(int k)	<i>move the cursor k positions to the left</i>
void right(int k)	<i>move the cursor k positions to the right</i>
int size()	<i>number of characters in the buffer</i>

API for a text buffer

Hint: Use two stacks.

1.3.45 *Stack generability.* Suppose that we have a sequence of intermixed *push* and *pop* operations as with our test stack client, where the integers 0, 1, ..., N-1 in that order (*push* directives) are intermixed with N minus signs (*pop* directives). Devise an algorithm that determines whether the intermixed sequence causes the stack to underflow. (You may use only an amount of space independent of N—you cannot store the integers in a data structure.) Devise a linear-time algorithm that determines whether a given permutation can be generated as output by our test client (depending on where the *pop* directives occur).

1.3.46 *Forbidden triple for stack generability.* Prove that a permutation can be generated by a stack (as in the previous question) if and only if it has no forbidden triple (a, b, c) such that $a < b < c$ with c first, a second, and b third (possibly with other intervening integers between c and a and between a and b).

Partial solution: Suppose that there is a forbidden triple (a, b, c). Item c is popped before a and b, but a and b are pushed before c. Thus, when c is pushed, both a and b are on the stack. Therefore, a cannot be popped before b.

1.3.47 *Catenable queues, stacks, or steques.* Add an extra operation catenation that (destructively) concatenates two queues, stacks, or steques (see EXERCISE 1.3.32). *Hint:* Use a circular linked list, maintaining a pointer to the last item.

1.3.48 *Two stacks with a deque.* Implement two stacks with a single deque so that each operation takes a constant number of deque operations (see EXERCISE 1.3.33).

1.3.49 *Queue with a constant number of stacks.* Implement a queue with a constant number of stacks so that each queue operation takes a constant (worst-case) number of stack operations. *Warning:* high degree of difficulty.

1.3.50 *Fail-fast iterator.* Modify the iterator code in Stack to immediately throw a `java.util.ConcurrentModificationException` if the client modifies the collection (via `push()` or `pop()`) during iteration?

Solution: Maintain a counter that counts the number of `push()` and `pop()` operations. When creating an iterator, store this value as an `Iterator` instance variable. Before each call to `hasNext()` and `next()`, check that this value has not changed since construction of the iterator; if it has, throw the exception.



1.4 ANALYSIS OF ALGORITHMS

AS PEOPLE GAIN EXPERIENCE USING COMPUTERS, they use them to solve difficult problems or to process large amounts of data and are invariably led to questions like these:

How long will my program take?

Why does my program run out of memory?

You certainly have asked yourself these questions, perhaps when rebuilding a music or photo library, installing a new application, working with a large document, or working with a large amount of experimental data. The questions are much too vague to be answered precisely—the answers depend on many factors such as properties of the particular computer being used, the particular data being processed, and the particular program that is doing the job (which implements some algorithm). All of these factors leave us with a daunting amount of information to analyze.

Despite these challenges, the path to developing useful answers to these basic questions is often remarkably straightforward, as you will see in this section. This process is based on the *scientific method*, the commonly accepted body of techniques used by scientists to develop knowledge about the natural world. We apply *mathematical analysis* to develop concise models of costs and do experimental studies to validate these models.

Scientific method The very same approach that scientists use to understand the natural world is effective for studying the running time of programs:

- Observe some feature of the natural world, generally with precise measurements.
- *Hypothesize* a model that is consistent with the observations.
- Predict events using the hypothesis.
- Verify the predictions by making further observations.
- Validate by repeating until the hypothesis and observations agree.

One of the key tenets of the scientific method is that the experiments we design must be reproducible, so that others can convince themselves of the validity of the hypothesis. Hypotheses must also be *falsifiable*, so that we can know for sure when a given hypothesis is wrong (and thus needs revision). As Einstein famously is reported to have said (“*No amount of experimentation can ever prove me right; a single experiment can prove me wrong*”), we can never know for sure that any hypothesis is absolutely correct; we can only validate that it is consistent with our observations.

Observations Our first challenge is to determine how to make quantitative measurements of the running time of our programs. This task is far easier than in the natural sciences. We do not have to send a rocket to Mars or kill laboratory animals or split an atom—we can simply run the program. Indeed, every time you run a program, you are performing a scientific experiment that relates the program to the natural world and answers one of our core questions: *How long will my program take?*

Our first qualitative observation about most programs is that there is a *problem size* that characterizes the difficulty of the computational task. Normally, the problem size is either the size of the input or the value of a command-line argument. Intuitively, the running time should increase with problem size, but the question of *by how much* it increases naturally comes up every time we develop and run a program.

Another qualitative observation for many programs is that the running time is relatively insensitive to the input itself; it depends primarily on the problem size. If this relationship does not hold, we need to take steps to better understand and perhaps better control the running time’s sensitivity to the input. But it does often hold, so we now focus on the goal of better quantifying the relationship between problem size and running time.

Example. As a running example, we will work with the program `ThreeSum` shown here, which counts the number of triples in a file of N integers that sum to 0 (assuming that overflow plays no role). This computation may seem contrived to you, but it is deeply related to numerous fundamental computational tasks (for example, see EXERCISE 1.4.26). As a test input, consider the file `1Mints.txt` from the booksite, which contains 1 million randomly generated `int` values. The second, eighth, and tenth entries in `1Mints.txt` sum to 0. How many more such triples are there in the file? `ThreeSum` can tell us, but can it do so in a reasonable amount of time? What is the relationship between the problem size N and running time for `ThreeSum`? As a first experiment, try running `ThreeSum` on your computer for the files `1Kints.txt`, `2Kints.txt`, `4Kints.txt`, and `8Kints.txt` on the

```
public class ThreeSum
{
    public static int count(int[] a)
    { // Count triples that sum to 0.
        int N = a.length;
        int cnt = 0;
        for (int i = 0; i < N; i++)
            for (int j = i+1; j < N; j++)
                for (int k = j+1; k < N; k++)
                    if (a[i] + a[j] + a[k] == 0)
                        cnt++;
        return cnt;
    }

    public static void main(String[] args)
    {
        int[] a = In.readInts(args[0]);
        StdOut.println(count(a));
    }
}
```

Given N , how long will this program take?

API `public class Stopwatch`

<code>Stopwatch()</code>	<i>create a stopwatch</i>
<code>double elapsedTime()</code>	<i>return elapsed time since creation</i>

typical client

```
public static void main(String[] args)
{
    int N = Integer.parseInt(args[0]);
    int[] a = new int[N];
    for (int i = 0; i < N; i++)
        a[i] = StdRandom.uniform(-1000000, 1000000);
    Stopwatch timer = new Stopwatch();
    int cnt = ThreeSum.count(a);
    double time = timer.elapsedTime();
    StdOut.println(cnt + " triples " + time + "seconds");
}
```

application

```
% java Stopwatch 1000
51 triples 0.488 seconds

% java Stopwatch 2000
516 triples 3.855 seconds
```

implementation

```
public class Stopwatch
{
    private final long start;
    public Stopwatch()
    { start = System.currentTimeMillis(); }
    public double elapsedTime()
    {
        long now = System.currentTimeMillis();
        return (now - start) / 1000.0;
    }
}
```

An abstract data type for a stopwatch

Analysis of experimental data. The program `DoublingTest` on the facing page is a more sophisticated `Stopwatch` client that produces experimental data for `ThreeSum`. It generates a sequence of random input arrays, doubling the array size at each step, and prints the running times of `ThreeSum.count()` for each input size. These experiments are certainly reproducible—you can also run them on your own computer, as many times as you like. When you run `DoublingTest`, you will find yourself in a prediction-verification cycle: it prints several lines very quickly, but then slows down considerably. Each time it prints a line, you find yourself wondering how long it will be until it prints the next line. Of course, since you have a different computer from ours, the actual running times that you get are likely to be different from those shown for our computer. Indeed, if your computer is twice as fast as ours, your running times will be about half ours, which leads immediately to the well-founded hypothesis that running times on different computers are likely to differ by a constant factor. Still, you will find yourself asking the more detailed question *How long will my program take, as a function of the input size?* To help answer this question, we plot the data. The diagrams at the bottom of the facing page show the result of plotting the data, both on a normal and on a log-log scale, with the problem size N on the x-axis and the running time $T(N)$ on the y-axis. The log-log plot immediately leads to a hypothesis about the running time—the data fits a straight line of slope 3 on the log-log plot. The equation of such a line is

$$\lg(T(N)) = 3 \lg N + \lg a$$

(where a is a constant) which is equivalent to

$$T(N) = a N^3$$

the running time, as a function of the input size, as desired. We can use one of our data points to solve for a —for example, $T(8000) = 51.1 = a 8000^3$, so $a = 9.98 \times 10^{-11}$ —and then use the equation

$$T(N) = 9.98 \times 10^{-11} N^3$$

to predict running times for large N . Informally, we are checking the hypothesis that the data points on the log-log plot fall close to this line. Statistical methods are available for doing a more careful analysis to find estimates of a and the exponent b , but our quick calculations suffice to estimate running time for most purposes. For example, we can estimate the running time on our computer for $N = 16,000$ to be about $9.98 \times 10^{-11} 16000^3 = 408.8$ seconds, or about 6.8 minutes (the actual time was 409.3 seconds). While waiting for your computer to print the line for $N = 16,000$ in `DoublingTest`, you might use this method to predict when it will finish, then check the result by waiting to see if your prediction is true.

program to perform experiments

```

public class DoublingTest
{
    public static double timeTrial(int N)
    { // Time ThreeSum.count() for N random 6-digit ints.
        int MAX = 1000000;
        int[] a = new int[N];
        for (int i = 0; i < N; i++)
            a[i] = StdRandom.uniform(-MAX, MAX);
        Stopwatch timer = new Stopwatch();
        int cnt = ThreeSum.count(a);
        return timer.elapsedTime();
    }

    public static void main(String[] args)
    { // Print table of running times.
        for (int N = 250; true; N += N)
        { // Print time for problem size N.
            double time = timeTrial(N);
            StdOut.printf("%7d %5.1f\n", N, time);
        }
    }
}

```

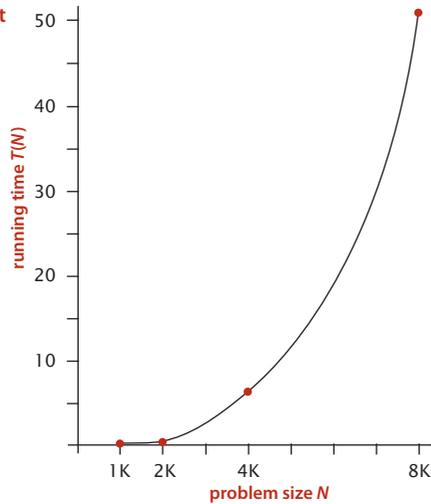
results of experiments

```

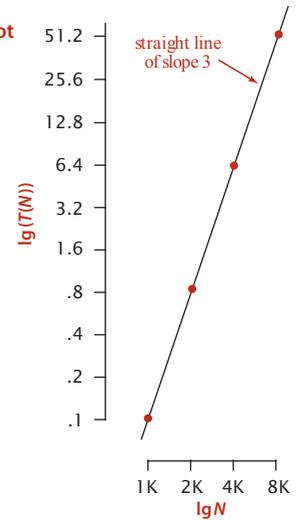
% java DoublingTest
250 0.0
500 0.0
1000 0.1
2000 0.8
4000 6.4
8000 51.1
...

```

standard plot



log-log plot



Analysis of experimental data (the running time of ThreeSum.count())

So far, this process mirrors the process scientists use when trying to understand properties of the real world. A straight line in a log-log plot is equivalent to the hypothesis that the data fits the equation $T(N) = aN^b$. Such a fit is known as a power law. A great many natural and synthetic phenomena are described by power laws, and it is reasonable to hypothesize that the running time of a program does, as well. Indeed, for the analysis of algorithms, we have mathematical models that strongly support this and similar hypotheses, to which we now turn.

Mathematical models In the early days of computer science, D. E. Knuth postulated that, despite all of the complicating factors in understanding the running times of our programs, it is possible, in principle, to build a mathematical model to describe the running time of any program. Knuth's basic insight is simple: the total running time of a program is determined by two primary factors:

- The cost of executing each statement
- The frequency of execution of each statement

The former is a property of the computer, the Java compiler and the operating system; the latter is a property of the program and the input. If we know both for all instructions in the program, we can multiply them together and sum for all instructions in the program to get the running time.

The primary challenge is to determine the frequency of execution of the statements. Some statements are easy to analyze: for example, the statement that sets `cnt` to 0 in `ThreeSum.count()` is executed exactly once. Others require higher-level reasoning: for example, the `if` statement in `ThreeSum.count()` is executed precisely

$$N(N-1)(N-2)/6$$

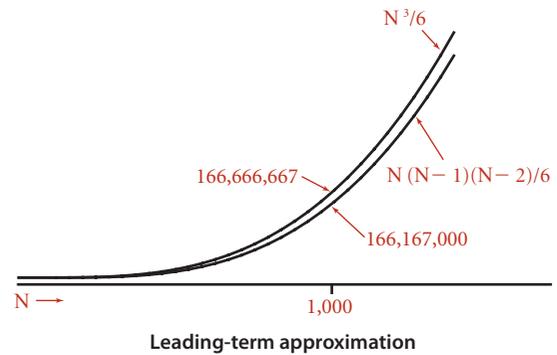
times (the number of ways to pick three different numbers from the input array—see EXERCISE 1.4.1). Others depend on the input data: for example the number of times the instruction `cnt++` in `ThreeSum.count()` is executed is precisely the number of triples that sum to 0 in the input, which could range from 0 of them to all of them. In the case of `DoubleTest`, where we generate the numbers randomly, it is possible to do a probabilistic analysis to determine the expected value of this quantity (see EXERCISE 1.4.40).

Tilde approximations. Frequency analyses of this sort can lead to complicated and lengthy mathematical expressions. For example, consider the count just considered of the number of times the `if` statement in `ThreeSum` is executed:

$$N(N-1)(N-2)/6 = N^3/6 - N^2/2 + N/3$$

As is typical in such expressions, the terms after the leading term are relatively small (for example, when $N = 1,000$ the value of $-N^2/2 + N/3 \approx -499,667$ is certainly insignificant by comparison with $N^3/6 \approx 166,666,667$). To allow us to ignore insignificant terms and therefore substantially simplify the mathematical formulas that we work with, we often use a mathematical device known as the tilde notation (\sim). This notation allows us to work with tilde approximations, where we throw away low-order terms that complicate formulas and represent a negligible contribution to values of interest:

Definition. We write $\sim f(N)$ to represent any function that, when divided by $f(N)$, approaches 1 as N grows, and we write $g(N) \sim f(N)$ to indicate that $g(N)/f(N)$ approaches 1 as N grows.



function	tilde approximation	order of growth
$N^3/6 - N^2/2 + N/3$	$\sim N^3/6$	N^3
$N^2/2 - N/2$	$\sim N^2/2$	N^2
$\lg N + 1$	$\sim \lg N$	$\lg N$
3	~ 3	1

Typical tilde approximations

description	function	order of growth
constant	1	
logarithmic	$\log N$	
linear	N	
linearithmic	$N \log N$	
quadratic	N^2	
cubic	N^3	
exponential	2^N	

Commonly encountered order-of-growth functions

For example, we use the approximation $\sim N^3/6$ to describe the number of times the `if` statement in `ThreeSum` is executed, since $N^3/6 - N^2/2 + N/3$ divided by $N^3/6$ approaches 1 as N grows. Most often, we work with tilde approximations of the form $g(N) \sim a f(N)$ where $f(N) = N^b (\log N)^c$ with a , b , and c constants and refer to $f(N)$ as the *order of growth* of $g(N)$. When using the logarithm in the order of growth, we generally do not specify the base, since the constant a can absorb that detail. This usage covers the relatively few functions that are commonly encountered in studying the order of growth of a program's running time shown in the table at left (with the exception of the exponential, which we defer to `CONTEXT`). We will describe these functions in more detail and briefly discuss why they appear in the analysis of algorithms after we complete our treatment of `ThreeSum`.

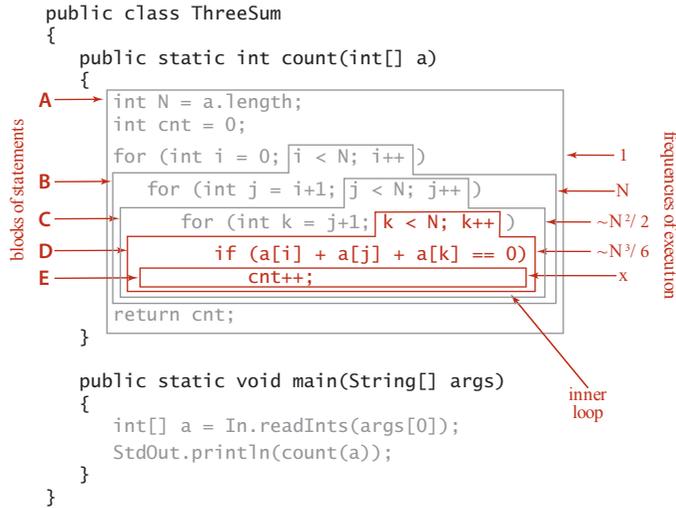
Approximate running time. To follow through on Knuth's approach to develop a mathematical expression for the total running time of a Java program, we can (in principle) study our Java compiler to find the number of machine instructions corresponding to each Java instruction and study our machine specifications to find the time of execution of each of the machine instructions, to produce a grand total. This process, for `ThreeSum`, is briefly summarized on the facing page. We classify blocks of Java statements by their frequency of execution, develop leading-term approximations for the frequencies, determine the cost of each statement, and then compute a total. Note that some frequencies may depend on the input. In this case, the number of times `cnt++` is executed certainly depends on the input—it is the number of triples that sum to 0, and could range from 0 to $\sim N^3/6$. We stop short of exhibiting the details (values of the constants) for any particular system, except to highlight that by using constant values t_0, t_1, t_2, \dots for the time taken by the blocks of statements, we are assuming that each block of Java statements corresponds to machine instructions that require a specified fixed amount of time. A key observation from this exercise is to note that only the instructions that are executed the most frequently play a role in the final total—we refer to these instructions as the inner loop of the program. For `ThreeSum`, the inner loop is the statements that increment `k` and test that it is less than `N` and the statements that test whether the sum of three given numbers is 0 (and possibly the statement that implements the count, depending on the input). This behavior is typical: the running times of a great many programs depend only on a small subset of their instructions.

Order-of-growth hypothesis. In summary, the experiments on page 177 and the mathematical model on page 181 both support the following hypothesis:

Property A. The order of growth of the running time of `ThreeSum` (to compute the number of triples that sum to 0 among N numbers) is N^3 .

Evidence: Let $T(N)$ be the running time of `ThreeSum` for N numbers. The mathematical model just described suggests that $T(N) \sim aN^3$ for some machine-dependent constant a ; experiments on many computers (including yours and ours) validate that approximation.

Throughout this book, we use the term *property* to refer to a hypothesis that needs to be validated through experimentation. The end result of our mathematical analysis is precisely the same as the end result of our experimental analysis—the running time of `ThreeSum` is $\sim aN^3$ for a machine-dependent constant a . This match validates both the experiments and the mathematical model and also exhibits more insight about the



Anatomy of a program's statement execution frequencies

statement block	time in seconds	frequency	total time
E	t_0	x (depends on input)	$t_0 x$
D	t_1	$N^3/6 - N^2/2 + N/3$	$t_1 (N^3/6 - N^2/2 + N/3)$
C	t_2	$N^2/2 - N/2$	$t_2 (N^2/2 - N/2)$
B	t_3	N	$t_3 N$
A	t_4	1	t_4
		grand total	$(t_1/6) N^3$ $+ (t_2/2 - t_1/2) N^2$ $+ (t_1/3 - t_2/2 + t_3) N$ $+ t_4 + t_0 x$
		tilde approximation	$\sim (t_1 / 6) N^3$ (assuming x is small)
		order of growth	N^3

Analyzing the running time of a program (example)

program because it does not require experimentation to determine the exponent. With some effort, we could validate the value of a on a particular system as well, though that activity is generally reserved for experts in situations where performance is critical.

Analysis of algorithms. Hypotheses such as PROPERTY A are significant because they relate the abstract world of a Java program to the real world of a computer running it. Working with the order of growth allows us to take one further step: to separate a program from the algorithm it implements. The idea that the order of growth of the running time of `ThreeSum` is N^3 does not depend on the fact that it is implemented in Java or that it is running on your laptop or someone else's cellphone or a supercomputer; it depends primarily on the fact that it examines all the different triples of numbers in the input. The *algorithm* that you are using (and sometimes the input model) determines the order of growth. Separating the algorithm from the implementation on a particular computer is a powerful concept because it allows us to develop knowledge about the performance of algorithms and then apply that knowledge to any computer. For example, we might say that `ThreeSum` is an implementation of the brute-force algorithm “*compute the sum of all different triples, counting those that sum to 0*”—we expect that an implementation of this algorithm in any programming language on any computer will lead to a running time that is proportional to N^3 . In fact, much of the knowledge about the performance of classic algorithms was developed decades ago, but that knowledge is still relevant to today's computers.

Cost model. We focus attention on properties of algorithms by articulating a cost model that defines the basic operations used by the algorithms we are studying to solve the problem at hand. For example, an appropriate cost model for the 3-sum problem, shown at right, is the number of times we access an array entry. With this cost model, we can make precise mathematical statements about properties of an algorithm, not just a particular implementation, as follows:

3-sum cost model. When studying algorithms to solve the 3-sum problem, we count array accesses (the number of times an array entry is accessed, for read or write).

Proposition B. The brute-force 3-sum algorithm uses $\sim N^3/2$ array accesses to compute the number of triples that sum to 0 among N numbers.

Proof: The algorithm accesses each of the 3 numbers for each of the $\sim N^3/6$ triples.

We use the term *proposition* to refer to mathematical truths about algorithms in terms of a cost model. Throughout this book, we study the algorithms that we consider within

the framework of a specific cost model. Our intent is to articulate cost models such that the order of growth of the running time for a given implementation is the same as the order of growth of the cost of the underlying algorithm (in other words, the cost model should include operations that fall within the inner loop). We seek precise mathematical results about algorithms (propositions) and also hypotheses about performance of implementations (properties) that you can check through experimentation. In this case, **PROPOSITION B** is a mathematical truth that supports the hypothesis stated in **PROPERTY A**, which we have validated with experiments, in accordance with the scientific method.

Summary. For many programs, developing a mathematical model of running time reduces to the following steps:

- Develop an input model, including a definition of the problem size.
- Identify the inner loop.
- Define a cost model that includes operations in the inner loop.
- Determine the frequency of execution of those operations for the given input. Doing so might require mathematical analysis—we will consider some examples in the context of specific fundamental algorithms later in the book.

If a program is defined in terms of multiple methods, we normally consider the methods separately. As an example, consider our example program of SECTION 1.1, `BinarySearch`.

Binary search. The input model is the array `a[]` of size N ; the inner loop is the statements in the single `while` loop; the cost model is the compare operation (compare the values of two array entries); and the analysis, discussed in SECTION 1.1 and given in full detail in PROPOSITION B in SECTION 3.1, shows that the number of compares is at most $\lg N + 1$.

Whitelist. The input model is the N numbers in the whitelist and the M numbers on standard input where we assume $M \gg N$; the inner loop is the statements in the single `while` loop; the cost model is the compare operation (inherited from binary search); and the analysis is immediate given the analysis of binary search—the number of compares is at most $M(\lg N + 1)$.

Thus, we draw the conclusion that the order of growth of the running time of the whitelist computation is at most $M \lg N$, subject to the following considerations:

- If N is small, the input-output cost might dominate.
- The number of compares depends on the input—it lies between $\sim M$ and $\sim M \lg N$, depending on how many of the numbers on standard input are in the whitelist and on how long the binary search takes to find the ones that are (typically it is $\sim M \lg N$).
- We are assuming that the cost of `Arrays.sort()` is small compared to $M \lg N$. `Arrays.sort()` implements the mergesort algorithm, and in SECTION 2.2, we will see that the order of growth of the running time of mergesort is $N \log N$ (see PROPOSITION G in CHAPTER 2), so this assumption is justified.

Thus, the model supports our hypothesis from SECTION 1.1 that the *binary search algorithm* makes the computation feasible when M and N are large. If we double the length of the standard input stream, then we can expect the running time to double; if we double the size of the whitelist, then we can expect the running time to increase only slightly.

DEVELOPING MATHEMATICAL MODELS for the analysis of algorithms is a fruitful area of research that is somewhat beyond the scope of this book. Still, as you will see with binary search, mergesort, and many other algorithms, understanding certain mathematical models is critical to understanding the efficiency of fundamental algorithms, so we often present details and/or quote the results of classic studies. When doing so, we encounter various functions and approximations that are widely used in mathematical analysis. For reference, we summarize some of this information in the tables below.

description	notation	definition
<i>floor</i>	$\lfloor x \rfloor$	largest integer not greater than x
<i>ceiling</i>	$\lceil x \rceil$	smallest integer not smaller than x
<i>natural logarithm</i>	$\ln N$	$\log_e N$ (x such that $e^x = N$)
<i>binary logarithm</i>	$\lg N$	$\log_2 N$ (x such that $2^x = N$)
<i>integer binary logarithm</i>	$\lfloor \lg N \rfloor$	largest integer not greater than $\lg N$ (# bits in binary representation of N) - 1
<i>harmonic numbers</i>	H_N	$1 + 1/2 + 1/3 + 1/4 + \dots + 1/N$
<i>factorial</i>	$N!$	$1 \times 2 \times 3 \times 4 \times \dots \times N$

Commonly encountered functions in the analysis of algorithms

description	approximation
<i>harmonic sum</i>	$H_N = 1 + 1/2 + 1/3 + 1/4 + \dots + 1/N \sim \ln N$
<i>triangular sum</i>	$1 + 2 + 3 + 4 + \dots + N \sim N^2/2$
<i>geometric sum</i>	$1 + 2 + 4 + 8 + \dots + N = 2N - 1 \sim 2N$ when $N = 2^n$
<i>Stirling's approximation</i>	$\lg N! = \lg 1 + \lg 2 + \lg 3 + \lg 4 + \dots + \lg N \sim N \lg N$
<i>binomial coefficients</i>	$\binom{N}{k} \sim N^k/k!$ when k is a small constant
<i>exponential</i>	$(1 - 1/x)^x \sim 1/e$

Useful approximations for the analysis of algorithms

Order-of-growth classifications We use just a few structural primitives (statements, conditionals, loops, nesting, and method calls) to implement algorithms, so very often the order of growth of the cost is one of just a few functions of the problem size N . These functions are summarized in the table on the facing page, along with the names that we use to refer to them, typical code that leads to each function, and examples.

Constant. A program whose running time's order of growth is constant executes a fixed number of operations to finish its job; consequently its running time does not depend on N . Most Java operations take constant time.

Logarithmic. A program whose running time's order of growth is *logarithmic* is barely slower than a constant-time program. The classic example of a program whose running time is logarithmic in the problem size is *binary search* (see `BinarySearch` on page 47). The base of the logarithm is not relevant with respect to the order of growth (since all logarithms with a constant base are related by a constant factor), so we use $\log N$ when referring to order of growth.

Linear. Programs that spend a constant amount of time processing each piece of input data, or that are based on a single for loop, are quite common. The order of growth of such a program is said to be linear—its running time is proportional to N .

Linearithmic. We use the term *linearithmic* to describe programs whose running time for a problem of size N has order of growth $N \log N$. Again, the base of the logarithm is not relevant with respect to the order of growth. The prototypical examples of linearithmic algorithms are `Merge.sort()` (see ALGORITHM 2.4) and `Quick.sort()` (see ALGORITHM 2.5).

Quadratic. A typical program whose running time has order of growth N^2 has two nested for loops, used for some calculation involving all pairs of N elements. The elementary sorting algorithms `Selection.sort()` (see ALGORITHM 2.1) and `Insertion.sort()` (see ALGORITHM 2.2) are prototypes of the programs in this classification.

Cubic. A typical program whose running time has order of growth N^3 has three nested for loops, used for some calculation involving all triples of N elements. Our example for this section, `ThreeSum`, is a prototype.

Exponential. In CHAPTER 6 (but not until then!) we will consider programs whose running times are proportional to 2^N or higher. Generally, we use the term exponential to refer to algorithms whose order of growth is b^N for any constant $b > 1$, even though different values of b lead to vastly different running times. Exponential algorithms are extremely slow—you will never run one of them to completion for a large problem. Still, exponential algorithms play a critical role in the theory of algorithms because

description	order of growth	typical code framework	description	example
constant	1	<code>a = b + c;</code>	statement	add two numbers
logarithmic	$\log N$	[see page 47]	divide in half	binary search
linear	N	<pre>double max = a[0]; for (int i = 1; i < N; i++) if (a[i] > max) max = a[i];</pre>	loop	find the maximum
linearithmic	$N \log N$	[see ALGORITHM 2.4]	divide and conquer	mergesort
quadratic	N^2	<pre>for (int i = 0; i < N; i++) for (int j = i+1; j < N; j++) if (a[i] + a[j] == 0) cnt++;</pre>	double loop	check all pairs
cubic	N^3	<pre>for (int i = 0; i < N; i++) for (int j = i+1; j < N; j++) for (int k = j+1; k < N; k++) if (a[i] + a[j] + a[k] == 0) cnt++;</pre>	triple loop	check all triples
exponential	2^N	[see CHAPTER 6]	exhasutive search	check all subsets

Summary of common order-of-growth hypotheses

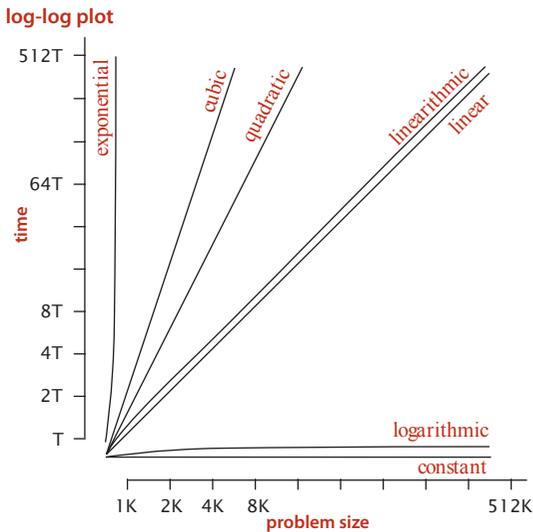
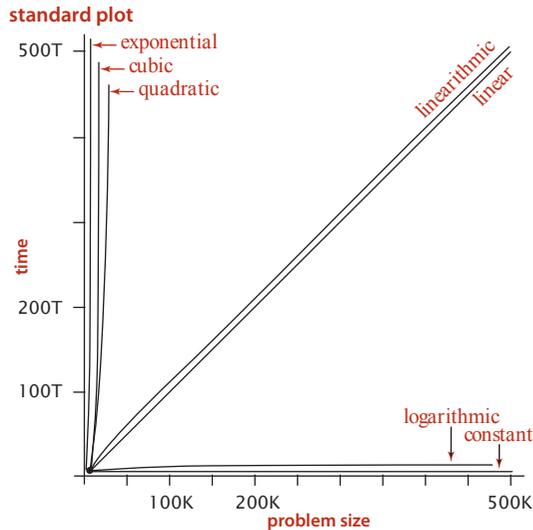
there exists a large class of problems for which it seems that an exponential algorithm is the best possible choice.

THESE CLASSIFICATIONS ARE THE MOST COMMON, but certainly not a complete set. The order of growth of an algorithm's cost might be $N^2 \log N$ or $N^{3/2}$ or some similar function.

Indeed, the detailed analysis of algorithms can require the full gamut of mathematical tools that have been developed over the centuries.

A great many of the algorithms that we consider have straightforward performance characteristics that can be accurately described by one of the orders of growth that we have considered. Accordingly, we can usually work with specific propositions with a cost model, such as mergesort uses between $\frac{1}{2}N \lg N$ and $N \lg N$ compares that immediately imply hypotheses (properties) such as *the order of growth of mergesort's running time is linearithmic*. For economy, we abbreviate such a statement to just say *mergesort is linearithmic*.

The plots at left indicate the importance of the order of growth in practice. The x-axis is the problem size; the y-axis is the running time. These charts make plain that quadratic and cubic algorithms are not feasible for use on large problems. As it turns out, several important problems have natural solutions that are quadratic but clever algorithms that are linearithmic. Such algorithms (including mergesort) are critically important in practice because they enable us to address problem sizes far larger than could be addressed with quadratic solutions. Naturally, we therefore focus in this book on developing logarithmic, linear, and linearithmic algorithms for fundamental problems.



Typical orders of growth

Designing faster algorithms One of the primary reasons to study the order of growth of a program is to help design a faster algorithm to solve the same problem. To illustrate this point, we consider next a faster algorithm for the 3-sum problem. How can we devise a faster algorithm, before even embarking on the study of algorithms? The answer to this question is that we *have* discussed and used two classic algorithms, mergesort and *binary search*, have introduced the facts that the mergesort is linearithmic and binary search is logarithmic. How can we take advantage of these algorithms to solve the 3-sum problem?

Warmup: 2-sum. Consider the easier problem of determining the number of pairs of integers in an input file that sum to 0. To simplify the discussion, assume also that the integers are distinct. This problem is easily solved in quadratic time by deleting the k loop and $a[k]$ from `ThreeSum.count()`, leaving a double loop that examines all pairs, as shown in the *quadratic* entry in the table on page 187 (we refer to such an implementation as `TwoSum`). The implementation below shows how mergesort and binary search (see page 47) can serve as a basis for a *linearithmic* solution to the 2-sum problem. The improved algorithm is based on the fact that an entry $a[i]$ is one of a pair that sums to 0 if and only if the value $-a[i]$ is in the array (and $a[i]$ is not zero). To solve the problem, we sort the array (to enable binary search) and then, for every entry $a[i]$ in the array, do a binary search for $-a[i]$ with `rank()` in `BinarySearch`. If the result is an index j with $j > i$, we increment the count.

This succinct test covers three cases:

- An unsuccessful binary search returns -1 , so we do not increment the count.
- If the binary search returns $j > i$, we have $a[i] + a[j] = 0$, so we increment the count.
- If the binary search returns j between 0 and i , we also have $a[i] + a[j] = 0$ but do not increment the count, to avoid double counting.

The result of the computation is precisely the same as the result of the quadratic algorithm, but it takes much less time. The running time of the mergesort is

```
import java.util.Arrays;

public class TwoSumFast
{
    public static int count(int[] a)
    { // Count pairs that sum to 0.
      Arrays.sort(a);
      int N = a.length;
      int cnt = 0;
      for (int i = 0; i < N; i++)
          if (BinarySearch.rank(-a[i], a) > i)
              cnt++;
      return cnt;
    }

    public static void main(String[] args)
    {
        int[] a = In.readInts(args[0]);
        StdOut.println(count(a));
    }
}
```

Linearithmic solution to the 2-sum problem

proportional to $N \log N$, and the N binary searches each take time proportional to $\log N$, so the running time of the whole algorithm is proportional to $N \log N$. Developing a faster algorithm like this is not merely an academic exercise—the faster algorithm enables us to address much larger problems. For example, you are likely to be able to solve the 2-sum problem for 1 million integers (`1Mints.txt`) in a reasonable amount of time on your computer, but you would have to wait quite a long time to do it with the quadratic algorithm (see EXERCISE 1.4.41).

Fast algorithm for 3-sum. The very same idea is effective for the 3-sum problem. Again, assume also that the integers are distinct. A pair $a[i]$ and $a[j]$ is part of a triple that sums to 0 if and only if the value $-(a[i] + a[j])$ is in the array (and not $a[i]$ or $a[j]$). The code below sorts the array, then does $N(N-1)/2$ binary searches that each take time proportional to $\log N$, for a total running time proportional to $N^2 \log N$. Note that in this case the cost of the sort is insignificant. Again, this solution enables us to address much larger problems (see EXERCISE 1.4.42). The plots in the figure at the bottom of the next page show the disparity in costs among these four algorithms for problem sizes in the range we have considered. Such differences certainly motivate the search for faster algorithms.

Lower bounds. The table on page 191 summarizes the discussion of this section. An interesting question immediately arises: Can we find algorithms for the 2-sum and 3-sum

```
import java.util.Arrays;

public class ThreeSumFast
{
    public static int count(int[] a)
    { // Count triples that sum to 0.
        Arrays.sort(a);
        int N = a.length;
        int cnt = 0;
        for (int i = 0; i < N; i++)
            for (int j = i+1; j < N; j++)
                if (BinarySearch.rank(-a[i]-a[j], a) > j)
                    cnt++;
        return cnt;
    }

    public static void main(String[] args)
    {
        int[] a = In.readInts(args[0]);
        StdOut.println(count(a));
    }
}
```

$N^2 \lg N$ solution to the 3-sum problem

problems that are substantially faster than `TwoSumFast` and `ThreeSumFast`? Is there a linear algorithm for 2-sum or a linearithmic algorithm for 3-sum? The answer to this question is no for 2-sum (under a model that counts and allows only comparisons of linear or quadratic functions of the numbers) and no one knows for 3-sum, though experts believe that the best possible algorithm for 3-sum is quadratic. The idea of a lower bound on the order of growth of the worst-case running time for all possible algorithms to solve a problem is a very powerful one, which we will

revisit in detail in SECTION 2.2 in the context of sorting. Non-trivial lower bounds are difficult to establish, but very helpful in guiding our search for efficient algorithms.

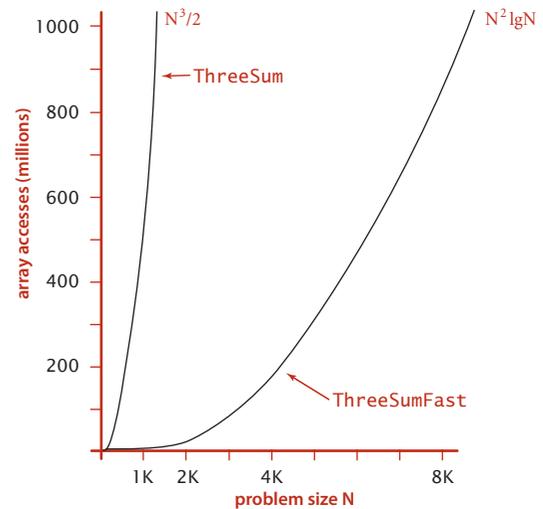
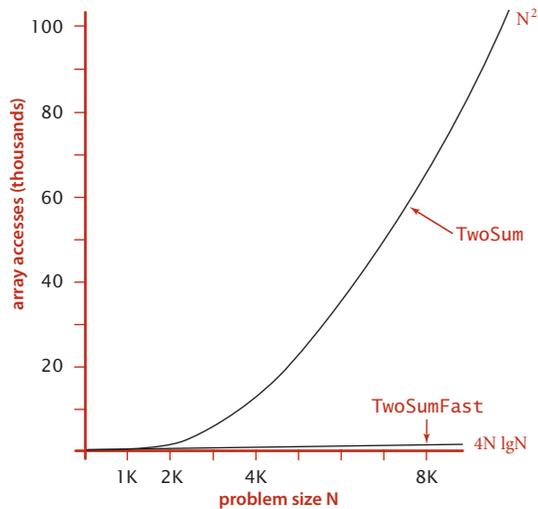
THE EXAMPLES IN THIS SECTION SET THE STAGE for our treatment of algorithms in this book. Throughout the book, our strategy for addressing new problems is the following:

- Implement and analyze a straightforward solution to the problem. We usually refer to such solutions, like ThreeSum and TwoSum, as the brute-force solution.
- Examine algorithmic improvements, usually designed to reduce the order of growth of the running time, such as TwoSumFast and ThreeSumFast.
- Run experiments to validate the hypotheses that the new algorithms are faster.

In many cases, we examine several algorithms for the same problem, because running time is only one consideration when choosing an algorithm for a practical problem. We will develop this idea in detail in the context of fundamental problems throughout the book.

algorithm	order of growth of running time
TwoSum	N^2
TwoSumFast	$N \log N$
ThreeSum	N^3
ThreeSumFast	$N^2 \log N$

Summary of running times



Costs of algorithms to solve the 2-sum and 3-sum problems

Doubling ratio experiments The following is a simple and effective shortcut for predicting performance and for determining the approximate order of growth of the running time of any program:

- Develop an input generator that produces inputs that model the inputs expected in practice (such as the random integers in `timeTrial()` in `DoublingTest`).
- Run the program `DoublingRatio` given below, a modification of `DoublingTest` that calculates the ratio of each running time with the previous.
- Run until the ratios approach a limit 2^b .

This test is not effective if the ratios do not approach a limiting value, but they do for many, many programs, implying the following conclusions:

- The order of growth of the running time is approximately N^b .
- To predict running times, multiply the last observed running time by 2^b and double N , continuing as long as desired. If you want to predict for an input size that is not a power of 2 times N , you can adjust ratios accordingly (see EXERCISE 1.4.9).

As illustrated below, the ratio for `ThreeSum` is about 8 and we can predict the running times for $N = 16,000, 32,000, 64,000$ to be 408.8, 3270.4, 26163.2 seconds, respectively, just by successively multiplying the last time for 8,000 (51.1) by 8.

program to perform experiments

```
public class DoublingRatio
{
    public static double timeTrial(int N)
    // same as for DoublingTest (page 177)
    public static void main(String[] args)
    {
        double prev = timeTrial(125);
        for (int N = 250; true; N += N)
        {
            double time = timeTrial(N);
            StdOut.printf("%6d %7.1f ", N, time);
            StdOut.printf("%5.1f\n", time/prev);
            prev = time;
        }
    }
}
```

results of experiments

```
% java DoublingRatio
 250      0.0   2.7
  500      0.0   4.8
 1000      0.1   6.9
 2000      0.8   7.7
 4000      6.4   8.0
 8000     51.1  8.0
```

predictions

```
16000    408.8   8.0
32000   3270.4   8.0
64000  26163.2   8.0
```

This test is roughly equivalent to the process described on page 176 (run experiments, plot values on a log-log plot to develop the hypothesis that the running time is aN^b , determine the value of b from the slope of the line, then solve for a), but it is simpler to apply. Indeed, you can accurately predict performance by hand when you run `DoublingRatio`. As the ratio approaches a limit, just multiply by that ratio to fill in later values in the table. Your approximate model of the order of growth is a power law with the binary logarithm of that ratio as the power.

Why does the ratio approach a constant? A simple mathematical calculation shows that to be the case for all of the common orders of growth just discussed (except exponential):

Proposition C. (Doubling ratio) If $T(N) \sim aN^b \lg N$ then $T(2N)/T(N) \sim 2^b$.

Proof: Immediate from the following calculation:

$$\begin{aligned} T(2N)/T(N) &= a(2N)^b \lg(2N) / aN^b \lg N \\ &= 2^b (1 + \lg 2 / \lg N) \\ &\sim 2^b \end{aligned}$$

Generally, the logarithmic factor cannot be ignored when developing a mathematical model, but it plays a less important role in predicting performance with a doubling hypothesis.

YOU SHOULD CONSIDER running doubling ratio experiments for every program that you write where performance matters—doing so is a very simple way to estimate the order of growth of the running time, perhaps revealing a performance bug where a program may turn out to be not as efficient as you might think. More generally, we can use hypotheses about the order of growth of the running time of programs to predict performance in one of the following ways:

Estimating the feasibility of solving large problems. You need to be able to answer this basic question for every program that you write: *Will the program be able to process this given input data in a reasonable amount of time?* To address such questions for a large amount of data, we extrapolate by a much larger factor than for doubling, say 10, as shown in the fourth column in the table at the bottom of the next page. Whether it is an investment banker running daily financial models or a scientist running a program to analyze experimental data or an engineer running simulations to test a design, it is not unusual for people to regularly run programs that take several hours to complete,

so the table focuses on that situation. Knowing the order of growth of the running time of an algorithm provides precisely the information that you need to understand limitations on the size of the problems that you can solve. *Developing such understanding is the most important reason to study performance.* Without it, you are likely have no idea how much time a program will consume; with it, you can make a back-of-the-envelope calculation to estimate costs and proceed accordingly.

Estimating the value of using a faster computer. You also may be faced with this basic question, periodically: *How much faster can I solve the problem if I get a faster computer?* Generally, if the new computer is x times faster than the old one, you can improve your running time by a factor of x . But it is usually the case that you can address larger problems with your new computer. How will that change affect the running time? Again, the order of growth is precisely the information needed to answer that question.

A FAMOUS RULE OF THUMB known as *Moore's Law* implies that you can expect to have a computer with about double the speed and double the memory 18 months from now, or a computer with about 10 times the speed and 10 times the memory in about 5 years. The table below demonstrates that an algorithm cannot keep pace with Moore's Law (solving a problem that is twice as big with a computer that is twice as fast) if its running time is quadratic or cubic. You can quickly determine whether that is the case by doing a doubling ratio test and checking that the ratio of running times as the input size doubles approaches 2 (linear or linearithmic), not 4 (quadratic) or 8 (cubic).

order of growth of time		for a program that takes a few hours for input of size N			
description	function	2x factor	10x factor	predicted time for $10N$	predicted time for $10N$ on a 10x faster computer
linear	N	2	10	a day	a few hours
linearithmic	$N \log N$	2	10	a day	a few hours
quadratic	N^2	4	100	a few weeks	a day
cubic	N^3	8	1,000	several months	a few weeks
exponential	2^N	2^N	2^{9N}	never	never

Predictions on the basis of order-of-growth function

Caveats There are many reasons that you might get inconsistent or misleading results when trying to analyze program performance in detail. All of them have to do with the idea that one or more of the basic assumptions underlying our hypotheses might be not quite correct. We can develop new hypotheses based on new assumptions, but the more details that we need to take into account, the more care is required in the analysis.

Large constants. With leading-term approximations, we ignore constant coefficients in lower-order terms, which may not be justified. For example, when we approximate the function $2N^2 + cN$ by $\sim 2N^2$, we are assuming that c is small. If that is not the case (suppose that c is 10^3 or 10^6) the approximation is misleading. Thus, we have to be sensitive to the possibility of large constants.

Nondominant inner loop. The assumption that the inner loop dominates may not always be correct. The cost model might miss the true inner loop, or the problem size N might not be sufficiently large to make the leading term in the mathematical description of the frequency of execution of instructions in the inner loop so much larger than lower-order terms that we can ignore them. Some programs have a significant amount of code outside the inner loop that needs to be taken into consideration. In other words, the cost model may need to be refined.

Instruction time. The assumption that each instruction always takes the same amount of time is not always correct. For example, most modern computer systems use a technique known as *caching* to organize memory, in which case accessing elements in huge arrays can take much longer if they are not close together in the array. You might observe the effect of caching for `ThreeSum` by letting `DoublingRatio` run for a while. After seeming to converge to 8, the ratio of running times may jump to a larger value for large arrays because of caching.

System considerations. Typically, there are many, many things going on in your computer. Java is one application of many competing for resources, and Java itself has many options and controls that significantly affect performance. A garbage collector or a just-in-time compiler or a download from the internet might drastically affect the results of experiments. Such considerations can interfere with the bedrock principle of the scientific method that experiments should be reproducible, since what is happening at this moment in your computer will never be reproduced again. Whatever else is going on in your system should in principle be negligible or possible to control.

Too close to call. Often, when we compare two different programs for the same task, one might be faster in some situations, and slower in others. One or more of the considerations just mentioned could make the difference. There is a natural tendency among

some programmers (and some students) to devote an extreme amount of energy running races to find the “best” implementation, but such work is best left for experts.

Strong dependence on inputs. One of the first assumptions that we made in order to determine the order of growth of the program’s running time of a program was that the running time should be relatively insensitive to the inputs. When that is not the case, we may get inconsistent results or be unable to validate our hypotheses. For example, suppose that we modify `ThreeSum` to answer the question *Does the input have a triple that sums to 0?* by changing it to return a boolean value, replacing `cnt++` by `return true` and adding `return false` as the last statement. The order of growth of the running time of this program is constant if the first three integers sum to 0 and cubic if there are no such triples in the input.

Multiple problem parameters. We have been focusing on measuring performance as a function of a single parameter, generally the value of a command-line argument or the size of the input. However, it is not unusual to have several parameters. A typical example arises when an algorithm involves building a data structure and then performing a sequence of operations that use that data structure. Both the size of the data structure and the number of operations are parameters for such applications. We have already seen an example of this in our analysis of the problem of whitelisting using binary search, where we have N numbers in the whitelist and M numbers on standard input and a typical running time proportional to $M \log N$.

Despite all these caveats, understanding the order of growth of the running time of each program is valuable knowledge for any programmer, and the methods that we have described are powerful and broadly applicable. Knuth’s insight was that we can carry these methods through to the last detail in principle to make detailed, accurate predictions. Typical computer systems are extremely complex and close analysis is best left for experts, but the same methods are effective for developing approximate estimates of the running time of any program. A rocket scientist needs to have some idea of whether a test flight will land in the ocean or in a city; a medical researcher needs to know whether a drug trial will kill or cure all the subjects; and any scientist or engineer using a computer program needs to have some idea of whether it will run for a second or for a year.

Coping with dependence on inputs For many problems, one of the most significant of the caveats just mentioned is the dependence on inputs, because running times can vary widely. The running time of the modification of `ThreeSum` mentioned on the facing page ranges from constant to cubic, depending on the input, so a closer analysis is required if we want to predict performance. We briefly consider here some of the approaches that are effective and that we will consider for specific algorithms later in the book.

Input models. One approach is to more carefully model the kind of input to be processed in the problems that we need to solve. For example, we might assume that the numbers in the input to `ThreeSum` are random `int` values. This approach is challenging for two reasons:

- The model may be unrealistic.
- The analysis may be extremely difficult, requiring mathematical skills quite beyond those of the typical student or programmer.

The first of these is the more significant, often because the goal of a computation is to discover characteristics of the input. For example, if we are writing a program to process a genome, how can we estimate its performance on a different genome? A good model describing the genomes found in nature is precisely what scientists seek, so estimating the running time of our programs on data found in nature actually amounts to contributing to that model! The second challenge leads to a focus on mathematical results only for our most important algorithms. We will see several examples where a simple and tractable input model, in conjunction with classical mathematical analysis, helps us predict performance.

Worst-case performance guarantees. Some applications demand that the running time of a program be less than a certain bound, no matter what the input. To provide such performance guarantees, theoreticians take an extremely pessimistic view of the performance of algorithms: what would the running time be in the worst case? For example, such a conservative approach might be appropriate for the software that runs a nuclear reactor or a pacemaker or the brakes in your car. We want to guarantee that such software completes its job within the bounds that we set because the result could be catastrophic if it does not. Scientists normally do not contemplate the worst case when studying the natural world: in biology, the worst case might be the extinction of the human race; in physics, the worst case might be the end of the universe. But the worst case can be a very real concern in computer systems, where the input may be generated by another (potentially malicious) user, rather than by nature. For example, websites that do not use algorithms with performance guarantees are subject to denial-of-service attacks, where hackers flood them with pathological requests that make them

run much more slowly than planned. Accordingly, many of our algorithms are designed to provide performance guarantees, such as the following:

Proposition D. In the linked-list implementations of Bag (ALGORITHM 1.4), Stack (ALGORITHM 1.2), and Queue (ALGORITHM 1.3), all operations take constant time in the worst case.

Proof: Immediate from the code. The number of instructions executed for each operation is bounded by a small constant. Caveat: This argument depends upon the (reasonable) assumption that the Java system creates a new Node in constant time.

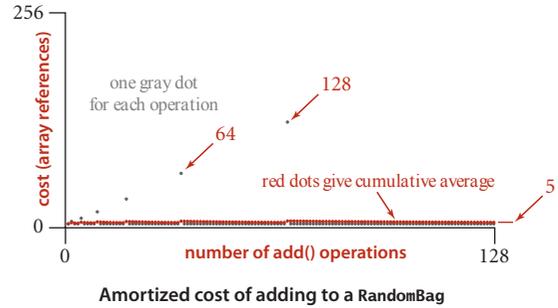
Randomized algorithms. One important way to provide a performance guarantee is to introduce randomness. For example, the quicksort algorithm for sorting that we study in SECTION 2.3 (perhaps the most widely used sorting algorithm) is quadratic in the worst case, but randomly ordering the input gives a probabilistic guarantee that its running time is linearithmic. Every time you run the algorithm, it will take a different amount of time, but the chance that the time will not be linearithmic is so small as to be negligible. Similarly, the hashing algorithms for symbol tables that we study in SECTION 3.4 (again, perhaps the most widely used approach) are linear-time in the worst case, but constant-time under a probabilistic guarantee. These guarantees are not absolute, but the chance that they are invalid is less than the chance your computer will be struck by lightning. Thus, such guarantees are as useful in practice as worst-case guarantees.

Sequences of operations. For many applications, the algorithm “input” might be not just data, but the sequence of operations performed by the client. For example, a pushdown stack where the client pushes N values, then pops them all, may have quite different performance characteristics from one where the client issues an alternating sequence N of push and pop operations. Our analysis has to take both situations into account (or to include a reasonable model of the sequence of operations).

Amortized analysis. Accordingly, another way to provide a performance guarantee is to *amortize* the cost, by keeping track of the total cost of all operations, divided by the number of operations. In this setting, we can allow some expensive operations, while keeping the average cost of operations low. The prototypical example of this type of analysis is the study of the resizing array data structure for Stack that we considered in SECTION 1.3 (ALGORITHM 1.1 on page 141). For simplicity, suppose that N is a power of 2. Starting with an empty structure, how many array entries are accessed for N consecutive calls to `push()`? This quantity is easy to calculate: the number of array accesses is

$$N + 4 + 8 + 16 + \dots + 2N = 5N - 4$$

The first term accounts for the array access within each of the N calls to `push()`; the subsequent terms account for the array accesses to initialize the data structure each time it doubles in size. Thus the average number of array accesses per operation is constant, even though the last operation takes linear time. This is known as an “amortized” analysis because we spread the cost of the few expensive operations, by assigning a portion of it to each of a large number of inexpensive operations. Amortized analysis provides a worst case guarantee on any sequence of operations, starting from an empty data structure. `VisualAccumulator` illustrates the process, shown above.



Proposition E. In the resizing array implementation of `Stack` (ALGORITHM 1.1), the average number of array accesses for any sequence of `push` and `pop` operations starting from an empty data structure is constant in the worst case.

Proof sketch: For each `push` operation that causes the array to grow (say from size N to size $2N$), consider the $N/2 - 1$ `push` operations that most recently caused the stack size to grow to k , for k from $N/2 + 2$ to N . Averaging the $4N$ array accesses to grow the array to size $2N$ ($2N$ array accesses to copy the N items and $2N$ array accesses to initialize an array) with $N/2 - 1$ array accesses (one for each `push`), we get an average cost of 9 array accesses for each of these $N/2 - 1$ `push` operations. Establishing this proposition for any sequence of `push` and `pop` operations is more intricate (see EXERCISE 1.4.32)

This kind of analysis is widely applicable. In particular, we use resizing arrays as the underlying data structure for several algorithms that we consider later in this book.

IT IS THE TASK OF THE ALGORITHM ANALYST to discover as much relevant information about an algorithm as possible, and it is the task of the applications programmer to apply that knowledge to develop programs that effectively solve the problems at hand. Ideally, we want algorithms that lead to clear and compact code that provides both a good guarantee and good performance on input values of interest. Many of the classic algorithms that we consider in this chapter are important for a broad variety of applications precisely because they have these properties. Using them as models, you can develop good solutions yourself for typical problems that you face while programming.

Memory As with running time, a program's memory usage connects directly to the physical world: a substantial amount of your computer's circuitry enables your program to store values and later retrieve them. The more values you need to have stored at any given instant, the more circuitry you need. You probably are aware of limits on memory usage on your computer (even more so than for time) because you probably have paid extra money to get more memory.

Memory usage is well-defined for Java on your computer (every value requires precisely the same amount of memory each time that you run your program), but Java is implemented on a very wide range of computational devices, and memory consumption is implementation-dependent. For economy, we use the word *typical* to signal that values are subject to machine dependencies.

type	bytes
<code>boolean</code>	1
<code>byte</code>	1
<code>char</code>	2
<code>int</code>	4
<code>float</code>	4
<code>long</code>	8
<code>double</code>	8

Typical memory requirements for primitive types

One of Java's most significant features is its memory allocation system, which is supposed to relieve you from having to worry about memory. Certainly, you are well-advised to take advantage of this feature when appropriate. Still, it is your responsibility to know, at least approximately, when a program's memory requirements will prevent you from solving a given problem.

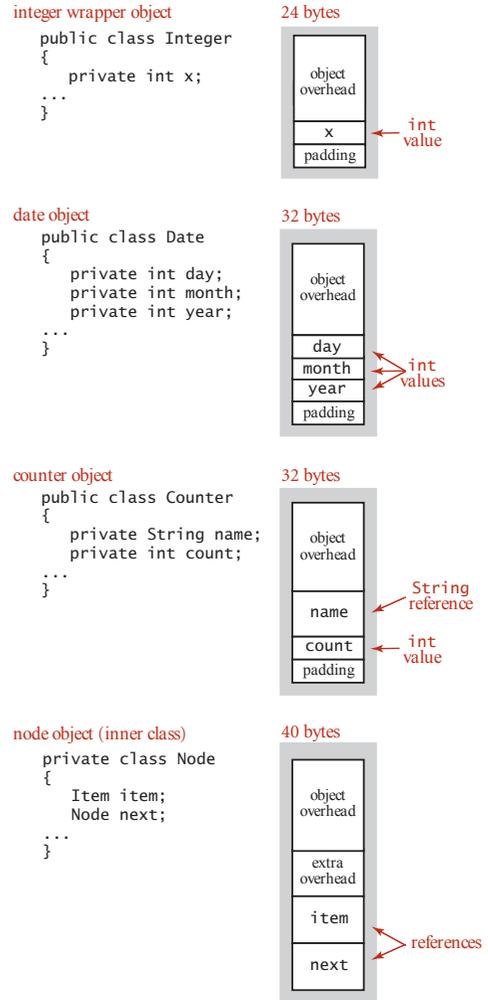
Analyzing memory usage is much easier than analyzing running time, primarily because not as many program statements are involved (just declarations) and because the analysis reduces complex objects to the primitive types, whose memory usage is well-defined and simple to understand: we can count up the number of variables and weight them by the number of bytes according to their type. For example, since the Java `int` data type is the set of integer values between $-2,147,483,648$ and $2,147,483,647$, a grand total of 2^{32} different values, typical Java implementations use 32 bits to represent `int` values. Similar considerations hold for other primitive types: typical Java implementations use 8-bit bytes, representing each `char` value with 2 bytes (16 bits), each `int` value with 4 bytes (32 bits), each `double` and each `long` value with 8 bytes (64 bits), and each `boolean` value with 1 byte (since computers typically access memory one byte at a time). Combined with knowledge of the amount of memory available, you can calculate limitations from these values. For example, if you have 1GB of memory on your computer (1 billion or 2^{30} bytes), you cannot fit more than about 256 million `int` values or 128 million `double` values in memory at any one time.

On the other hand, analyzing memory usage is subject to various differences in machine hardware and in Java implementations, so you should consider the specific examples that we give as indicative of how you might go about determining memory usage when warranted, not the final word for your computer. For example, many data structures involve representation of machine addresses, and the amount of memory

needed for a machine address varies from machine to machine. For consistency, we assume that 8 bytes are needed to represent addresses, as is typical for 64-bit architectures that are now widely used, recognizing that many older machines use a 32-bit architecture that would involve just 4 bytes per machine address.

Objects. To determine the memory usage of an object, we add the amount of memory used by each instance variable to the overhead associated with each object, typically 16 bytes. The overhead includes a reference to the object's class, garbage collection information, and synchronization information. Moreover, the memory usage is typically padded to be a multiple of 8 bytes (machine words, on a 64-bit machine). For example, an `Integer` object uses 24 bytes (16 bytes of overhead, 4 bytes for its `int` instance variable, and 4 bytes of padding). Similarly, a `Date` (page 91) object also uses 32 bytes: 16 bytes of overhead, 4 bytes for each of its three `int` instance variables, and 4 bytes of padding. A reference to an object typically is a memory address and thus uses 8 bytes of memory. For example, a `Counter` (page 89) object uses 32 bytes: 16 bytes of overhead, 8 bytes for its `String` instance variable (a reference), 4 bytes for its `int` instance variable, and 4 bytes of padding. When we account for the memory for a reference, we account separately for the memory for the object itself, so this total does not count the memory for the `String` value.

Linked lists. A nested non-static (inner) class such as our `Node` class (page 142) requires an extra 8 bytes of overhead (for a reference to the enclosing instance). Thus, a `Node` object uses 40 bytes (16 bytes of object overhead, 8 bytes each for the references to the `Item` and `Node` objects, and 8 bytes for the extra overhead). Thus, since an `Integer` object uses 24 bytes, a stack with N integers built with a linked-list representation (ALGORITHM 1.2) uses $32 + 64N$ bytes, the usual 16 for object overhead for `Stack`, 8 for its reference instance variable, 4 for its `int` instance variable, 4 for padding, and 64 for each entry, 40 for a `Node` and 24 for an `Integer`.



Typical object memory requirements

Arrays. Typical memory requirements for various types of arrays in Java are summarized in the diagrams on the facing page. Arrays in Java are implemented as objects, typically with extra overhead for the length. An array of primitive-type values typically requires 24 bytes of header information (16 bytes of object overhead, 4 bytes for the length, and 4 bytes of padding) plus the memory needed to store the values. For example, an array of `N int` values uses $24 + 4N$ bytes (rounded up to be a multiple of 8), and an array of `N double` values uses $24 + 8N$ bytes. An array of objects is an array of references to the objects, so we need to add the space for the references to the space required for the objects. For example, an array of `N Date` objects (page 91) uses 24 bytes (array overhead) plus $8N$ bytes (references) plus 32 bytes for each object, for a grand total of $24 + 40N$ bytes. A two-dimensional array is an array of arrays (each array is an object). For example, a two-dimensional M -by- N array of `double` values uses 24 bytes (overhead for the array of arrays) plus $8M$ bytes (references to the row arrays) plus M times 24 bytes (overhead from the row arrays) plus M times N times 8 bytes (for the N `double` values in each of the M rows) for a grand total of $8NM + 32M + 24 \sim 8NM$ bytes. When array entries are objects, a similar accounting leads to a total of $8NM + 32M + 24 \sim 8NM$ bytes for the array of arrays filled with references to objects, plus the memory for the objects themselves.

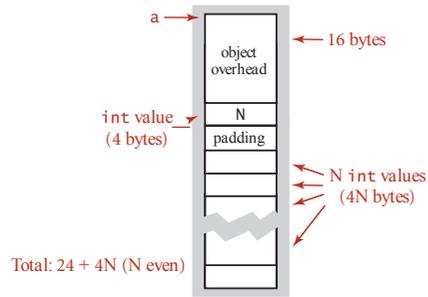
String objects (Java 7 and later). The standard `String` representation (used in typical Java 7 implementations) has two instance variables: a reference to a character array `value[]` that stores the sequence of characters and an `int` value `hash` (that stores a hash code that saves recomputation in certain circumstances that need not concern us now). Therefore, a `String` of length N typically uses 40 bytes for the `String` object (16 bytes for object overhead plus 8 bytes for the array reference plus 4 bytes for the `int` instance variables plus 4 bytes of padding) plus $24 + 2N$ bytes for the character array for a total of $56 + 2N$ bytes.

String objects (Java 6 and earlier). An alternate `String` representation (used in typical Java 6 implementations) maintains two extra `int` instance variables (`offset` and `count`) and represents the sequence of characters `value[offset]` through `value[offset + count - 1]`. Now, a `String` of length N typically uses 40 bytes (for the `String` object) plus $24 + 2N$ bytes (for the character array) for a total of $64 + 2N$ bytes. This representation saves memory when extracting substrings because two `String` objects can share the same underlying character array.

Substring extraction. When using the Java 7 representation to implement the `substring()` method, we must create a new character array, so substring extraction takes linear time and linear space (in the length of the resulting substring). When

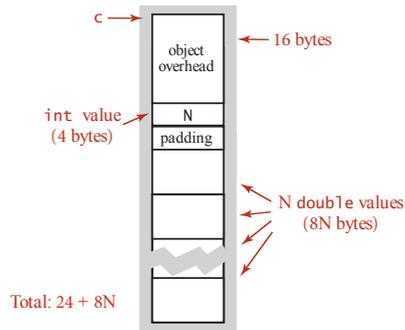
array of int values

```
int[] a = new int[N];
```



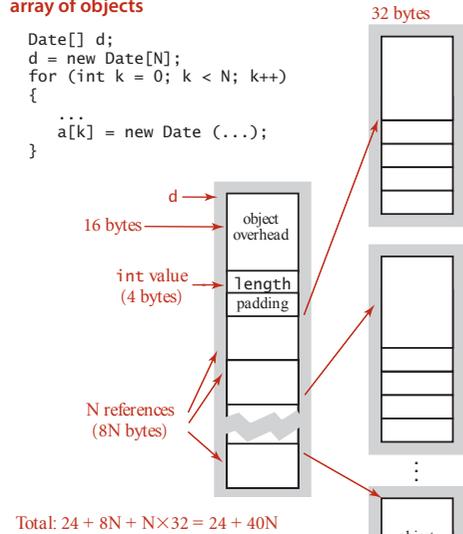
array of double values

```
double[] c = new double[N];
```



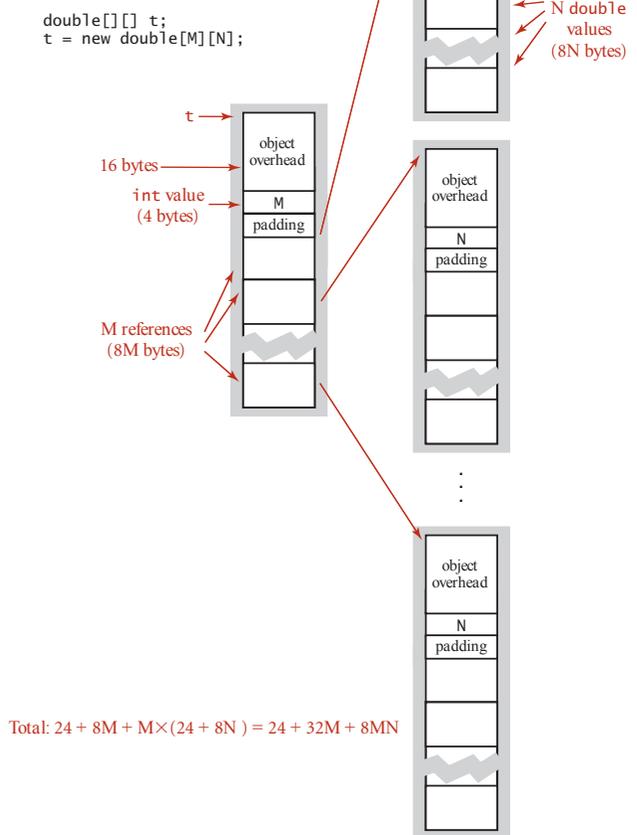
array of objects

```
Date[] d;
d = new Date[N];
for (int k = 0; k < N; k++)
{
    ...
    a[k] = new Date (...);
}
```



array of arrays (two-dimensional array)

```
double[][] t;
t = new double[M][N];
```



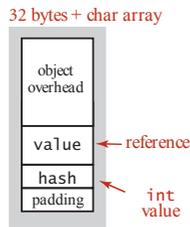
summary

type	bytes
<code>int[]</code>	$\sim 4N$
<code>double[]</code>	$\sim 8N$
<code>Date[]</code>	$\sim 40N$
<code>double[][]</code>	$\sim 8NM$

Typical memory requirements for arrays of `int` values, `double` values, objects, and arrays

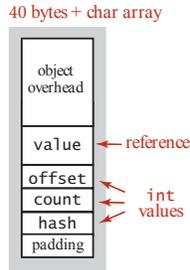
String object (Java 7 library)

```
public class String
{
    private char[] value;
    private int hash;
    ...
}
```



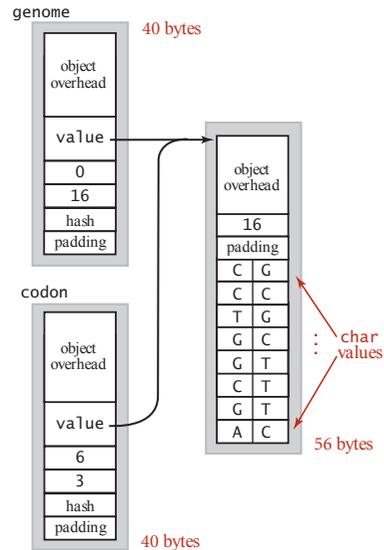
String object (Java 6 library)

```
public class String
{
    private char[] value;
    private int offset;
    private int count;
    private int hash;
    ...
}
```



substring example (Java 6 aliasing)

```
String genome = "CGCCTGGCGTGTGAC";
String codon = genome.substring(6, 9);
```



A String and a substring

using the Java 6 representation, we can implement the `substring()` method without having to make copies of the string's characters: the character array containing the original string is aliased in the object for the substring; the offset and count fields identify the substring. Thus, a substring of an existing string takes just 40 bytes. In summary, *extracting a substring takes either constant extra memory or linear extra memory depending on the underlying implementation*. We will assume the Java 7 representation in this book.

THESE BASIC MECHANISMS ARE EFFECTIVE for estimating the memory usage of a great many programs, but there are numerous complicating factors that can make the task significantly more difficult. We have already noted the potential effect of aliasing. Moreover, memory consumption is a complicated dynamic process when function calls are involved because the system memory allocation mechanism plays a more important role, with more system dependencies. For example, when your program calls a method, the system allocates the memory needed for the method (for its local variables) from a special area of memory called the stack (a system pushdown stack), and when the method returns to the caller, the memory is returned to the stack. For this reason, creating arrays or other large objects in recursive programs is dangerous, since each recursive call implies significant memory usage. When you create an object with `new`, the system allocates the memory needed for the object from another special area of memory known as the *heap* (not the same as the binary heap data structure we consider in SECTION 2.4), and you must remember that every object lives until no references to it remain, at which point a system process known as *garbage collection* reclaims its memory for the heap. Such dynamics can make the task of precisely estimating memory usage of a program challenging.

Perspective Good performance is important. An impossibly slow program is almost as useless as an incorrect one, so it is certainly worthwhile to pay attention to the cost at the outset, to have some idea of which kinds of problems you might feasibly address. In particular, it is always wise to have some idea of which code constitutes the inner loop of your programs.

Perhaps the most common mistake made in programming is to pay too much attention to performance characteristics. Your first priority is to make your code clear and correct. Modifying a program for the sole purpose of speeding it up is best left for experts. Indeed, doing so is often counterproductive, as it tends to create code that is complicated and difficult to understand. C. A. R. Hoare (the inventor of quicksort and a leading proponent of writing clear and correct code) once summarized this idea by saying that “*premature optimization is the root of all evil*,” to which Knuth added the qualifier “(or at least most of it) *in programming*.” Beyond that, improving the running time is not worthwhile if the available cost benefits are insignificant. For example, improving the running time of a program by a factor of 10 is inconsequential if the running time is only an instant. Even when a program takes a few minutes to run, the total time required to implement and debug an improved algorithm might be substantially more than the time required simply to run a slightly slower one—you may as well let the computer do the work. Worse, you might spend a considerable amount of time and effort implementing ideas that should in theory improve a program but do not do so in practice.

Perhaps the second most common mistake made in programming is to ignore performance characteristics. Faster algorithms are often more complicated than brute-force ones, so you might be tempted to accept a slower algorithm to avoid having to deal with more complicated code. However, you can sometimes reap huge savings with just a few lines of good code. Users of a surprising number of computer systems lose substantial time unknowingly waiting for brute-force quadratic algorithms to finish solving a problem, when linear or linearithmic algorithms are available that could solve the problem in a fraction of the time. When we are dealing with huge problem sizes, we often have no choice but to seek better algorithms.

We generally take as implicit the methodology described in this section to estimate memory usage and to develop an order-of-growth hypothesis of the running time from a tilde approximation resulting from a mathematical analysis within a cost model, and to check those hypotheses with experiments. Improving a program to make it more clear, efficient, and elegant should be your goal every time that you work on it. If you pay attention to the cost all the way through the development of a program, you will reap the benefits every time you use it.

Q&A

Q. Why not use `StdRandom` to generate random values instead of maintaining the file `1Mints.txt`?

A. It is easier to debug code in development and to reproduce experiments. `StdRandom` produces different values each time it is called, so running a program after fixing a bug may not test the fix! You could use `StdRandom.setSeed()` to address this problem, but a reference file such as `1Mints.txt` makes it easier to add test cases while debugging. Also, different programmers can compare performance on different systems, without worrying about the input model. Once you have debugged a program and have a good idea of how it performs, it is certainly worthwhile to test it on random data. For example, `DoublingTest` and `DoublingRatio` take this approach.

Q. I ran `DoublingRatio` on my computer, but the results were not as consistent as in the book. Some of the ratios were not close to 8. Why?

A. That is why we discussed “caveats” on page 195. Most likely, your computer’s operating system decided to do something else during the experiment. One way to mitigate such problems is to invest more time in more experiments. For example, you could change `DoublingTest` to run the experiments 1,000 times for each N , giving a much more accurate estimate for the running time for each size (see EXERCISE 1.4.39).

Q. What, exactly, does “as N grows” mean in the definition of the tilde notation?

A. The formal definition of $f(N) \sim g(N)$ is $\lim_{N \rightarrow \infty} f(N)/g(N) = 1$.

Q. I’ve seen other notations for describing order of growth. What’s the story?

A. The “big-Oh” notation is widely used: we say that $f(N)$ is $O(g(N))$ if there exist constants c and N_0 such that $|f(N)| \leq c |g(N)|$ for all $N > N_0$. This notation is very useful in providing asymptotic upper bounds on the performance of algorithms, which is important in the theory of algorithms. But it is not useful for predicting performance or for comparing algorithms.

Q. Why not?

A. The primary reason is that it describes only an upper bound on the running time. Actual performance might be much better. The running time of an algorithm might be both $O(N^2)$ and $\sim a N \log N$. As a result, it cannot be used to justify tests like our doubling ratio test (see PROPOSITION C on page 193).

Q. So why is the big-Oh notation so widely used?

A. It facilitates development of bounds on the order of growth, even for complicated algorithms for which more precise analysis might not be feasible. Moreover, it is compatible with the “big-Omega” and “big-Theta” notations that theoretical computer scientists use to classify algorithms by bounding their worst-case performance. We say that $f(N)$ is $\Omega(g(N))$ if there exist constants c and N_0 such that $|f(N)| \geq c |g(N)|$ for $N > N_0$; and if $f(N)$ is $O(g(N))$ and $\Omega(g(N))$, we say that $f(N)$ is $\Theta(g(N))$. The “big-Omega” notation is typically used to describe a lower bound on the worst case, and the “big-Theta” notation is typically used to describe the performance of algorithms that are optimal in the sense that no algorithm can have better asymptotic worst-case order of growth. Optimal algorithms are certainly worth considering in practical applications, but there are many other considerations, as you will see.

Q. Aren't upper bounds on asymptotic performance important?

A. Yes, but we prefer to discuss precise results in terms of frequency of statement execution with respect to cost models, because they provide more information about algorithm performance and because deriving such results is feasible for the algorithms that we discuss. For example, we say “ThreeSum uses $\sim N^3/2$ array accesses” and “the number of times `cnt++` is executed in ThreeSum is $\sim N^3/6$ in the worst case,” which is a bit more verbose but much more informative than the statement “the running time of ThreeSum is $O(N^3)$.”

Q. When the order of growth of the running time of an algorithm is $N \log N$, the doubling test will lead to the hypothesis that the running time is $\sim a N$ for a constant a . Isn't that a problem?

A. We have to be careful not to try to infer that the experimental data implies a particular mathematical model, but when we are just predicting performance, this is not really a problem. For example, when N is between 16,000 and 32,000, the plots of $14N$ and $N \lg N$ are very close to one another. The data fits both curves. As N increases, the curves become closer together. It actually requires some care to experimentally check the hypothesis that an algorithm's running time is linearithmic but not linear.

Q. Does `int[] a = new int[N]` count as N array accesses (to initialize entries to 0)?

A. Most likely yes, so we make that assumption in this book, though a sophisticated compiler implementation might try to avoid this cost for huge sparse arrays.

EXERCISES

1.4.1 Show that the number of different triples that can be chosen from N items is precisely $N(N-1)(N-2)/6$. *Hint:* Use mathematical induction or a counting argument.

1.4.2 Modify `ThreeSum` to work properly even when the `int` values are so large that adding two of them might cause overflow.

1.4.3 Modify `DoublingTest` to use `StdDraw` to produce plots like the standard and log-log plots in the text, rescaling as necessary so that the plot always fills a substantial portion of the window.

1.4.4 Develop a table like the one on page 181 for `TwoSum`.

1.4.5 Give tilde approximations for the following quantities:

- a. $N + 1$
- b. $1 + 1/N$
- c. $(1 + 1/N)(1 + 2/N)$
- d. $2N^3 - 15N^2 + N$
- e. $\lg(2N) / \lg N$
- f. $\lg(N^2 + 1) / \lg N$
- g. $N^{100} / 2^N$

1.4.6 Give the order of growth (as a function of N) of the running times of each of the following code fragments:

- a.

```
int sum = 0;
for (int n = N; n > 0; n /= 2)
    for(int i = 0; i < n; i++)
        sum++;
```
- b.

```
int sum = 0;
for (int i = 1; i < N; i *= 2)
    for (int j = 0; j < i; j++)
        sum++;
```

```
c.  int sum = 0;
    for (int i = 1; i < N; i *= 2)
        for (int j = 0; j < N; j++)
            sum++;
```

1.4.7 Analyze ThreeSum under a cost model that counts arithmetic operations (and comparisons) involving the input numbers.

1.4.8 Write a program to determine the number pairs of values in an input file that are equal. If your first try is quadratic, think again and use `Arrays.sort()` to develop a linearithmic solution.

1.4.9 Give a formula to predict the running time of a program for a problem of size N when doubling experiments have shown that the doubling factor is 2^b and the running time for problems of size N_0 is T .

1.4.10 Modify binary search so that it always returns the element with the smallest index that matches the search element (and still guarantees logarithmic running time).

1.4.11 Add an instance method `howMany()` to `StaticSetOfInts` (page 99) that finds the number of occurrences of a given key in time proportional to $\log N$ in the worst case.

1.4.12 Write a program that, given two sorted arrays of N int values, prints all elements that appear in both arrays, in sorted order. The running time of your program should be proportional to N in the worst case.

1.4.13 Using the assumptions developed in the text, give the amount of memory needed to represent an object of each of the following types:

- a. Accumulator
- b. Transaction
- c. FixedCapacityStackOfStrings with capacity C and N entries
- d. Point2D
- e. Interval1D
- f. Interval2D
- g. Double

CREATIVE PROBLEMS

1.4.14 *4-sum*. Develop an algorithm for the 4-sum problem.

1.4.15 *Faster 3-sum*. As a warmup, develop an implementation `TwoSumFaster` that uses a *linear* algorithm to count the pairs that sum to zero after the array is sorted (instead of the binary-search-based linearithmic algorithm). Then apply a similar idea to develop a quadratic algorithm for the 3-sum problem.

1.4.16 *Closest pair (in one dimension)*. Write a program that, given an array `a[]` of N `double` values, finds a *closest pair*: two values whose difference is no greater than the the difference of any other pair (in absolute value). The running time of your program should be linearithmic in the worst case.

1.4.17 *Farthest pair (in one dimension)*. Write a program that, given an array `a[]` of N `double` values, finds a *farthest pair*: two values whose difference is no smaller than the the difference of any other pair (in absolute value). The running time of your program should be linear in the worst case.

1.4.18 *Local minimum of an array*. Write a program that, given an array `a[]` of N distinct integers, finds a *local minimum*: an entry `a[i]` that is strictly less than its neighbors. Each internal entry (other than `a[0]` and `a[N-1]`) has 2 neighbors. Your program should use $\sim 2 \lg N$ compares in the worst case.

1.4.19 *Local minimum of a matrix*. Given an N -by- N array `a[]` of N^2 distinct integers, design an algorithm that finds a *local minimum*: an entry `a[i][j]` that is strictly less than its neighbors. Internal entries have 4 neighbors; entries on an edge have 3 neighbors; entries on a corner have 2 neighbors. The running time of your program should be proportional to N in the worst case, which means that you cannot afford to examine all N^2 entries.

1.4.20 *Bitonic search*. An array is bitonic if it is comprised of an increasing sequence of integers followed immediately by a decreasing sequence of integers. Write a program that, given a bitonic array of N distinct `int` values, determines whether a given integer is in the array. Your program should use $\sim 3 \lg N$ compares in the worst case. *Extra credit*: use only $\sim 2 \lg N$ compares in the worst case.

1.4.21 *Binary search on distinct values*. Develop an implementation of binary search for `StaticSETofInts` (see page 99) where the running time of `contains()` is guaranteed to be $\sim \lg R$, where R is the number of different integers in the array given as

argument to the constructor.

1.4.22 *Binary search with only addition and subtraction.* [Mihai Patrascu] Write a program that, given an array of N distinct `int` values in ascending order, determines whether a given integer is in the array. You may use only additions and subtractions and a constant amount of extra memory. The running time of your program should be proportional to $\log N$ in the worst case.

1.4.23 *Binary search for a fraction.* Devise a method that uses a logarithmic number of compares of the form *Is the number less than x ?* to find a rational number p/q such that $0 < p < q < N$. *Hint:* Two different fractions with denominators less than N must differ by at least $1/N^2$.

1.4.24 *Throwing eggs from a building.* Suppose that you have an N -story building and plenty of eggs. Suppose also that an egg is broken if it is thrown off floor F or higher, and intact otherwise. First, devise a strategy to determine the value of F such that the number of broken eggs is $\sim \lg N$ when using $\sim \lg N$ throws, then find a way to reduce the cost to $\sim 2 \lg F$.

1.4.25 *Throwing two eggs from a building.* Consider the previous question, but now suppose you only have two eggs, and your cost model is the number of throws. Devise a strategy to determine F such that the number of throws is at most $2\sqrt{N}$, then find a way to reduce the cost to $\sim c\sqrt{F}$ for some constant c . This is analogous to a situation where search hits (egg intact) are much cheaper than misses (egg broken).

1.4.26 *3-collinearity.* Suppose that you have an algorithm that takes as input N distinct points in the plane and can return the number of triples that fall on the same line. Show that you can use this algorithm to solve the 3-sum problem. *Strong hint:* Use algebra to show that (a, a^3) , (b, b^3) , and (c, c^3) are collinear if and only if $a + b + c = 0$.

CREATIVE PROBLEMS *(continued)*

1.4.27 *Queue with two stacks.* Implement a queue with two stacks so that each queue operation takes a constant amortized number of stack operations. *Hint:* If you push elements onto a stack and then pop them all, they appear in reverse order. If you repeat this process, they're now back in order.

1.4.28 *Stack with a queue.* Implement a stack with a single queue so that each stack operations takes a linear number of queue operations. *Hint:* To delete an item, get all of the elements on the queue one at a time, and put them at the end, except for the last one which you should delete and return. (This solution is admittedly very inefficient.)

1.4.29 *Steque with two stacks.* Implement a steque with two stacks so that each steque operation (see EXERCISE 1.3.32) takes a constant amortized number of stack operations.

1.4.30 *Deque with a stack and a steque.* Implement a deque with a stack and a steque (see EXERCISE 1.3.32) so that each deque operation takes a constant amortized number of stack and steque operations.

1.4.31 *Deque with three stacks.* Implement a deque with three stacks so that each deque operation takes a constant amortized number of stack operations.

1.4.32 *Amortized analysis.* Prove that, starting from an empty stack, the number of array accesses used by any sequence of M operations in the resizing array implementation of Stack is proportional to M .

1.4.33 *Memory requirements on a 32-bit machine.* Give the memory requirements for Integer, Date, Counter, `int[]`, `double[]`, `double[][]`, `String`, `Node`, and `Stack` (linked-list representation) for a 32-bit machine. Assume that references are 4 bytes, object overhead is 8 bytes, and padding is to a multiple of 4 bytes.

1.4.34 *Hot or cold.* Your goal is to guess a secret integer between 1 and N . You repeatedly guess integers between 1 and N . After each guess you learn if your guess equals the secret integer (and the game stops). Otherwise, you learn if the guess is hotter (closer to) or colder (farther from) the secret number than your previous guess. Design an algorithm that finds the secret number in at most $\sim 2 \lg N$ guesses. Then design an algorithm that finds the secret number in at most $\sim 1 \lg N$ guesses.

1.4.35 *Time costs for pushdown stacks.* Justify the entries in the table below, which shows typical time costs for various pushdown stack implementations, using a cost model that counts both data references (references to data pushed onto the stack, either an array reference or a reference to an object's instance variable) and objects created. Assume that the Integer objects are not cached (so they must be created for each push).

data structure	item type	cost to push N int values	
		data references	objects created
<i>linked list</i>	int	$2N$	N
	Integer	$3N$	$2N$
<i>resizing array</i>	int	$\sim 5N$	$\lg N$
	Integer	$\sim 5N$	$\sim N$

Time costs for pushdown stacks (various implementations)

1.4.36 *Space usage for pushdown stacks.* Justify the entries in the table below, which shows typical space usage for various pushdown stack implementations. Use a static nested class for linked-list nodes to avoid the non-static nested class overhead. Assume that the Integer objects are not cached (so they must be created for each push).

data structure	item type	space usage for N int values (bytes)
<i>linked list</i>	int	$\sim 32N$
	Integer	$\sim 56N$
<i>resizing array</i>	int	between $\sim 4N$ and $\sim 16N$
	Integer	between $\sim 32N$ and $\sim 56N$

Space usage in pushdown stacks (various implementations)

EXPERIMENTS

1.4.37 *Autoboxing performance penalty.* Run experiments to determine the performance penalty on your machine for using autoboxing and auto-unboxing. Develop an implementation `FixedCapacityStackOfInts` and use a client such as `DoublingRatio` to compare its performance with the generic `FixedCapacityStack<Integer>`, for a large number of `push()` and `pop()` operations.

1.4.38 *Naive 3-sum implementation.* Run experiments to evaluate the following implementation of the inner loop of `ThreeSum`:

```
for (int i = 0; i < N; i++)
    for (int j = 0; j < N; j++)
        for (int k = 0; k < N; k++)
            if (i < j && j < k)
                if (a[i] + a[j] + a[k] == 0)
                    cnt++;
```

Do so by developing a version of `DoublingTest` that computes the ratio of the running times of this program and `ThreeSum`.

1.4.39 *Improved accuracy for doubling test.* Modify `DoublingRatio` to take a second command-line argument that specifies the number of calls to `timeTrial()` for each value of `N`. Run your program for 10, 100, and 1,000 trials and comment on the precision of the results.

1.4.40 *3-sum for random values.* Formulate and validate a hypothesis describing the number of triples of `N` random `int` values that sum to 0. If you are skilled in mathematical analysis, develop an appropriate mathematical model for this problem, where the values are uniformly distributed between $-M$ and M , where M is not small.

1.4.41 *Running times.* Estimate the amount of time it would take to run `TwoSumFast`, `TwoSum`, `ThreeSumFast` and `ThreeSum` on your computer to solve the problems for a file of 1 million numbers. Use `DoublingRatio` to do so.

1.4.42 *Problem sizes.* Estimate the size of the largest value of P for which you can run `TwoSumFast`, `TwoSum`, `ThreeSumFast`, and `ThreeSum` on your computer to solve the problems for a file of 2^P thousand numbers. Use `DoublingRatio` to do so.

1.4.43 *Resizing arrays versus linked lists.* Run experiments to validate the hypothesis that resizing arrays are faster than linked lists for stacks (see EXERCISE 1.4.35 and EXERCISE 1.4.36). Do so by developing a version of `DoublingRatio` that computes the ratio

of the running times of the two programs.

1.4.44 *Birthday problem.* Write a program that takes an integer N from the command line and uses `StdRandom.uniform()` to generate a random sequence of integers between 0 and $N - 1$. Run experiments to validate the hypothesis that the number of integers generated before the first repeated value is found is $\sim\sqrt{\pi N/2}$.

1.4.45 *Coupon collector problem.* Generating random integers as in the previous exercise, run experiments to validate the hypothesis that the number of integers generated before all possible values are generated is $\sim N H_N$.



1.5 CASE STUDY: UNION-FIND

TO ILLUSTRATE our basic approach to developing and analyzing algorithms, we now consider a detailed example. Our purpose is to emphasize the following themes.

- Good algorithms can make the difference between being able to solve a practical problem and not being able to address it at all.
- An efficient algorithm can be as simple to code as an inefficient one.
- Understanding the performance characteristics of an implementation can be an interesting and satisfying intellectual challenge.
- The scientific method is an important tool in helping us choose among different methods for solving the same problem.
- An iterative refinement process can lead to increasingly efficient algorithms.

These themes are reinforced throughout the book. This prototypical example sets the stage for our use of the same general methodology for many other problems.

The problem that we consider is not a toy problem; it is a fundamental computational task, and the solution that we develop is of use in a variety of applications, from percolation in physical chemistry to connectivity in communications networks. We start with a simple solution, then seek to understand that solution's performance characteristics, which help us to see how to improve the algorithm.

Dynamic connectivity We start with the following problem specification: The input is a sequence of pairs of integers, where each integer represents an object of some type and we are to interpret the pair $p\ q$ as meaning “ p is connected to q .” We assume that “is connected to” is an *equivalence* relation, which means that it is

- *Reflexive*: p is connected to p .
- *Symmetric*: If p is connected to q , then q is connected to p .
- *Transitive*: If p is connected to q and q is connected to r , then p is connected to r .

An equivalence relation partitions the objects into *equivalence classes*. In this case, two objects are in the same equivalence class if and only if they are connected. Our goal is to write a program to filter out extraneous pairs (pairs where both objects are in the same equivalence class) from the sequence. In other words, when the program reads a pair $p\ q$ from the input, it should write the pair to the output only if the pairs it has seen to that point do not imply that p is connected to q . If the previous pairs do imply that p is connected to q , then the program should ignore the pair $p\ q$ and proceed to read in the next pair. The figure on the facing page gives an example of this process. To achieve the desired goal, we need to devise a data structure that can remember sufficient

information about the pairs it has seen to be able to decide whether or not a new pair of objects is connected. Informally, we refer to the task of designing such a method as the dynamic connectivity problem. This problem arises applications such as the following:

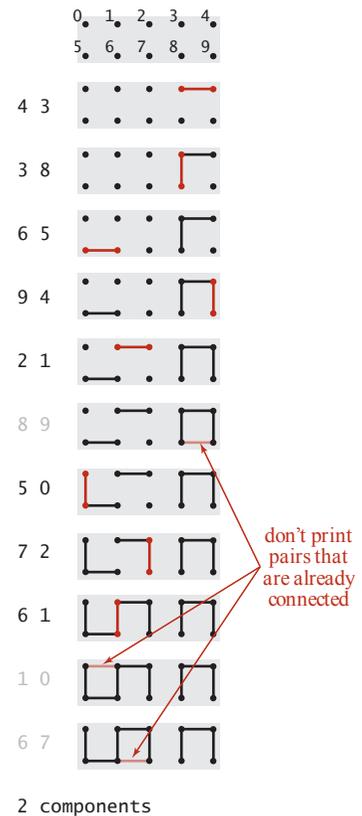
Networks. The integers might represent computers in a large network, and the pairs might represent connections in the network. Then, our program determines whether we need to establish a new direct connection for p and q to be able to communicate or whether we can use existing connections to set up a communications path. Or, the integers might represent contact sites in an electrical circuit, and the pairs might represent wires connecting the sites. Or, the integers might represent people in a social network, and the pairs might represent friendships. In such applications, we might need to process millions of objects and billions of connections.

Variable-name equivalence. In certain programming environments, it is possible to declare two variable names as being equivalent (references to the same object). After a sequence of such declarations, the system needs to be able to determine whether two given names are equivalent. This application is an early one (for the FORTRAN programming language) that motivated the development of the algorithms that we are about to consider.

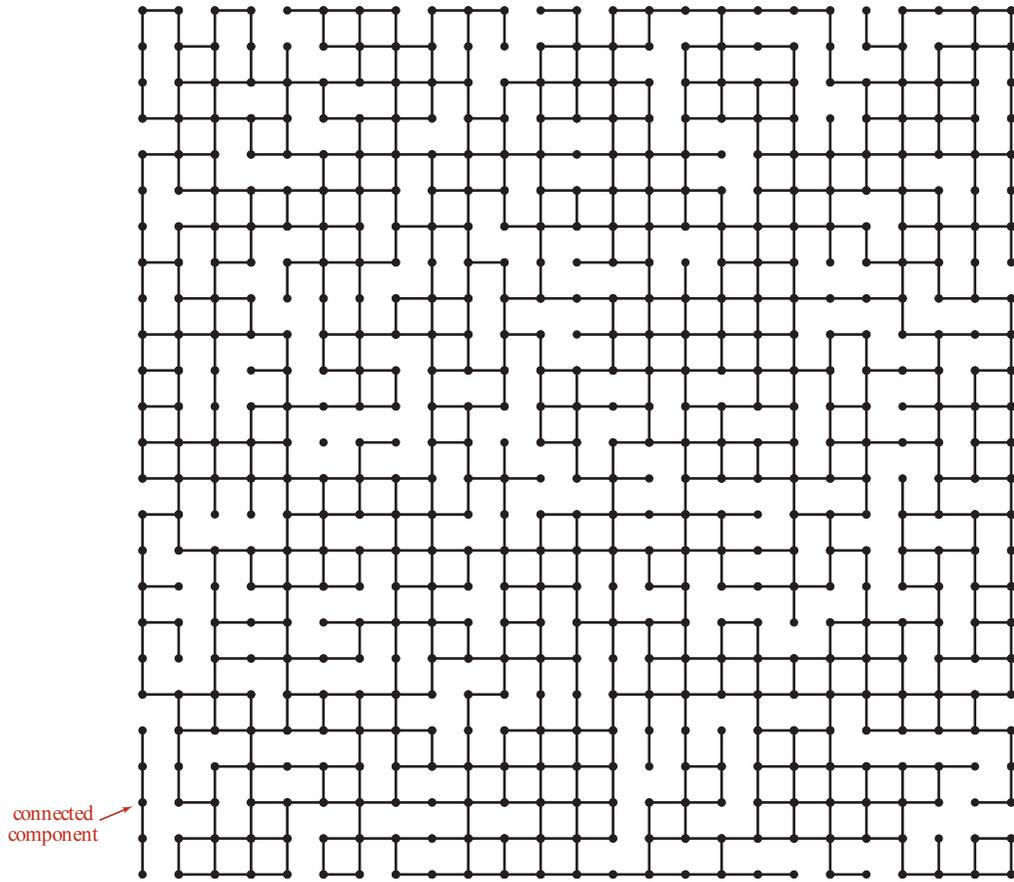
Mathematical sets. On a more abstract level, you can think of the integers as belonging to mathematical sets. When we process a pair p q , we are asking whether they belong to the same set. If not, we unite p 's set and q 's set, putting them in the same set.

TO FIX IDEAS, we will use networking terminology for the rest of this section and refer to the objects as sites, the pairs as connections, and the equivalence classes as connected components, or just components for short. For simplicity, we assume that we have N sites with integer names, from 0 to $N-1$. We do so without loss of generality because we shall be considering a host of algorithms in CHAPTER 3 that can associate arbitrary names with such integer identifiers in an efficient manner.

A larger example that gives some indication of the difficulty of the connectivity problem is depicted in the figure at the top of the next page. You can quickly identify the component consisting of a single site in the left middle of the diagram and the



Dynamic connectivity example



Medium connectivity example (625 sites, 900 edges, 3 connected components)

component consisting of five sites at the bottom left, but you might have difficulty verifying that all of the other sites are connected to one another. For a program, the task is even more difficult, because it has to work just with site names and connections and has no access to the geometric placement of sites in the diagram. How can we tell quickly whether or not any given two sites in such a network are connected?

The first task that we face in developing an algorithm is to specify the problem in a precise manner. The more we require of an algorithm, the more time and space we may expect it to need to finish the job. It is impossible to quantify this relationship a priori, and we often modify a problem specification on finding that it is difficult or expensive to solve or, in happy circumstances, on finding that an algorithm can provide information more useful than what was called for in the original specification. For example, our

connectivity problem specification requires only that our program be able to determine whether or not any given pair p q is connected, and not that it be able to demonstrate a set of connections that connect that pair. Such a requirement makes the problem more difficult and leads us to a different family of algorithms, which we consider in SECTION 4.1.

To specify the problem, we develop an API that encapsulates the basic operations that we need: initialize, add a connection between two sites, identify the component containing a site, determine whether two sites are in the same component, and count the number of components. Thus, we articulate the following API:

```
public class UF
```

```

    UF(int N)                initialize N sites with integer names (0 to N-1)
    void union(int p, int q) add connection between p and q
    int find(int p)          component identifier for p (0 to N-1)
    boolean connected(int p, int q) return true if p and q are in the same component
    int count()              number of components
```

Union-find API

The `union()` operation merges two components if the two sites are in different components, the `find()` operation returns an integer component identifier for a given site, the `connected()` operation determines whether two sites are in the same component, and the `count()` method returns the number of components. We start with N components, and each `union()` that merges two different components decrements the number of components by 1.

As we shall soon see, the development of an algorithmic solution for dynamic connectivity thus reduces to the task of developing an implementation of this API. Every implementation has to

- Define a data structure to represent the known connections
- Develop efficient `union()`, `find()`, `connected()`, and `count()` implementations that are based on that data structure

As usual, the nature of the data structure has a direct impact on the efficiency of the algorithms, so data structure and algorithm design go hand in hand. The API already specifies the convention that both sites and components will be identified by `int` values between 0 and $N-1$, so it makes sense to use a site-indexed array `id[]` as our basic

data structure to represent the components. We always use the name of one of the sites in a component as the component identifier, so you can think of each component as being represented by one of its sites. Initially, we start with N components, each site in its own component, so we initialize $\text{id}[i]$ to i for all i from 0 to $N-1$. For each site i , we keep the information needed by $\text{find}()$ to determine the component containing i in $\text{id}[i]$, using various algorithm-dependent strategies. All of our implementations use a one-line implementation of $\text{connected}()$ that returns the boolean value $\text{find}(p) == \text{find}(q)$.

```
% more tinyUF.txt
10
4 3
3 8
6 5
9 4
2 1
8 9
5 0
7 2
6 1
1 0
6 7

% more mediumUF.txt
625
528 503
548 523
...
[900 connections]

% more largeUF.txt
1000000
786321 134521
696834 98245
...
[2000000 connections]
```

IN SUMMARY, our starting point is ALGORITHM 1.5 on the facing page. We maintain two instance variables, the count of components and the array $\text{id}[]$. Implementations of $\text{find}()$ and $\text{union}()$ are the topic of the remainder of this section.

To test the utility of the API and to provide a basis for development, we include a client in $\text{main}()$ that uses it to solve the dynamic connectivity problem. It reads the value of N followed by a sequence of pairs of integers (each in the range 0 to $N-1$), calling $\text{connected}()$ for each pair: If the two sites in the pair are already connected, it moves on to the next pair; if they are not, it calls $\text{union}()$ and prints the pair. Before considering implementations, we also prepare test data: the file `tinyUF.txt` contains the 11 connections among 10 sites used in the small example illustrated on page 217, the file `mediumUF.txt` contains the 900 connections among 625 sites illustrated on page 218, and the file `largeUF.txt` is an example with 2 million connections among 1 million sites. Our goal is to be able to handle inputs such as `largeUF.txt` in a reasonable amount of time.

To analyze the algorithms, we focus on the number of times each algorithm accesses an array entry. By doing so, we are implicitly formulating the hypothesis that the running times of the algorithms on a particular machine are within a constant factor of

this quantity. This hypothesis is immediate from the code, is not difficult to validate through experimentation, and provides a useful starting point for comparing algorithms, as we will see.

Union-find cost model. When studying algorithms to implement the union-find API, we count array accesses (the number of times an array entry is accessed, for read or write).

ALGORITHM 1.5 Union-find implementation

```

public class UF
{
    private int[] id;    // access to component id (site indexed)
    private int count;  // number of components

    public UF(int N)
    { // Initialize component id array.
        count = N;
        id = new int[N];
        for (int i = 0; i < N; i++)
            id[i] = i;
    }

    public int count()
    { return count; }

    public boolean connected(int p, int q)
    { return find(p) == find(q); }

    public int find(int p)
    public void union(int p, int q)
    // See page 222 (quick-find), page 224 (quick-union) and page 228 (weighted).

    public static void main(String[] args)
    { // Solve dynamic connectivity problem on StdIn.
        int N = StdIn.readInt();           // Read number of sites.
        UF uf = new UF(N);                 // Initialize N components.
        while (!StdIn.isEmpty())
        {
            int p = StdIn.readInt();
            int q = StdIn.readInt();       // Read pair to connect.
            if (uf.connected(p, q)) continue; // Ignore if connected.
            uf.union(p, q);                // Combine components
            StdOut.println(p + " " + q);    // and print connection.
        }
        StdOut.println(uf.count() + " components");
    }
}

```

```

% java UF < tinyUF.txt
4 3
3 8
6 5
9 4
2 1
5 0
7 2
6 1
2 components

```

Our UF implementations are based on this code, which maintains an array of integers `id[]` such that the `find()` method returns the same integer for every site in each connected component. The `union()` method must maintain this invariant.

Implementations We shall consider three different implementations, all based on using the site-indexed `id[]` array, to determine whether two sites are in the same connected component.

Quick-find. One approach is to maintain the invariant that `p` and `q` are connected if and only if `id[p]` is equal to `id[q]`. In other words, all sites in a component must have the same value in `id[]`. This method is called *quick-find* because `find(p)` just returns `id[p]`, which immediately implies that `connected(p, q)` reduces to just the test `id[p] == id[q]` and returns `true` if and only if `p` and `q` are in the same component. To maintain the invariant for the call `union(p, q)`, we first check whether they are already in the same component, in which case there is nothing to do. Otherwise, we are faced with the situation that all of the `id[]` entries corresponding to sites in the same component as `p` have one value and all of the `id[]` entries corresponding to sites in the same component as `q` have another value. To combine the two components into one, we have to make all of the `id[]` entries corresponding to both sets of sites the same value, as shown in the example at right. To do so, we go through the array, changing all the entries with values equal to `id[p]` to the value `id[q]`. We could have decided to change all the entries equal to `id[q]` to the value `id[p]`—the choice between these two alternatives is arbitrary. The

find examines `id[5]` and `id[9]`

p	q	0	1	2	3	4	5	6	7	8	9
5	9	1	1	1	8	8	1	1	1	8	8

union has to change all 1s to 8s

p	q	0	1	2	3	4	5	6	7	8	9
5	9	1	1	1	8	8	1	1	1	8	8
		8	8	8	8	8	8	8	8	8	8

Quick-find overview

```
public int find(int p)
{ return id[p]; }

public void union(int p, int q)
{ // Put p and q into the same component.
  int pID = find(p);
  int qID = find(q);

  // Nothing to do if p and q are already
    in the same component.
  if (pID == qID) return;

  // Change values from id[p] to id[q].
  for (int i = 0; i < id.length; i++)
    if (id[i] == pID) id[i] = qID;
  count--;
}
```

Quick-find

code for `find()` and `union()` based on these descriptions, given at left, is straightforward. A full trace for our development client with our sample test data `tinyUF.txt` is shown on the next page.

Quick-find analysis. The `find()` operation is certainly quick, as it only accesses the `id[]` array once in order to complete the operation. But quick-find is typically not useful for large problems because `union()` needs to scan through the whole `id[]` array for each input pair.

Proposition F. The quick-find algorithm uses one array access for each call to `find()`, two array accesses for each call to `connected()`, and between $N + 3$ and $2N + 1$ array accesses for each call to `union()` that combines two components.

Proof: Immediate from the code. Each call to `connected()` tests two entries in the `id[]` array, one for each of the two calls to `find()`. Each call to `union()` that combines two components does so by making two calls to `find()`, testing each of the N entries in the `id[]` array, and changing between 1 and $N - 1$ of them.

		id[]									
p	q	0	1	2	3	4	5	6	7	8	9
4	3	0	1	2	3	4	5	6	7	8	9
		0	1	2	3	3	5	6	7	8	9
3	8	0	1	2	3	3	5	6	7	8	9
		0	1	2	8	8	5	6	7	8	9
6	5	0	1	2	8	8	5	6	7	8	9
		0	1	2	8	8	5	5	7	8	9
9	4	0	1	2	8	8	5	5	7	8	9
		0	1	2	8	8	5	5	7	8	8
2	1	0	1	2	8	8	5	5	7	8	8
		0	1	1	8	8	5	5	7	8	8
8	9	0	1	1	8	8	5	5	7	8	8
5	0	0	1	1	8	8	5	5	7	8	8
		0	1	1	8	8	0	0	7	8	8
7	2	0	1	1	8	8	0	0	7	8	8
		0	1	1	8	8	0	0	1	8	8
6	1	0	1	1	8	8	0	0	1	8	8
		1	1	1	8	8	1	1	1	8	8
1	0	1	1	1	8	8	1	1	1	8	8
6	7	1	1	1	8	8	1	1	1	8	8

id[p] and id[q] differ, so union() changes entries equal to id[p] to id[q] (in red)
 id[p] and id[q] match, so no change

Quick-find trace

In particular, suppose that we use quick-find for the dynamic connectivity problem and wind up with a single component. This requires at least $N - 1$ calls to `union()`, and, consequently, at least $(N + 3)(N - 1) \sim N^2$ array accesses—we are led immediately to the hypothesis that dynamic connectivity with quick-find can be a *quadratic*-time process. This analysis generalizes to say that quick-find is quadratic for typical applications where we end up with a small number of components. You can easily validate this hypothesis on your computer with a doubling test (see EXERCISE 1.5.23 for an instructive example). Modern computers can execute hundreds of millions or billions of instructions per second, so this cost is not noticeable if N is small, but we also might find ourselves with millions or billions of sites and connections to process in a modern application, as represented by our test file `largeUF.txt`. If you are still not convinced and feel that you have a particularly fast computer, try using quick-find to determine the number of components implied by the pairs in `largeUF.txt`. The inescapable conclusion is that we cannot feasibly solve such a problem using the quick-find algorithm, so we seek better algorithms.

Quick-union. The next algorithm that we consider is a complementary method that concentrates on speeding up the `union()` operation. It is based on the same data structure—the site-indexed `id[]` array—but we interpret the values differently, to define more complicated structures. Specifically, the `id[]` entry for each site is the name of another site in the same component (possibly itself)—we refer to this connection as a link. To implement `find()`, we start at the given site, follow its link to another site, follow that site’s link to yet another site, and so forth, following links until reaching a root, a site that has a link to itself (which is guaranteed to happen, as you will see). Two sites are in the same component if and only if this process leads them to the same root. To validate this process, we need `union(p, q)` to maintain this invariant, which is easily arranged: we follow links to find the roots associated with `p` and `q`, then rename one of the components by linking one of these roots to the other; hence the name *quick-union*. Again, we have an arbitrary choice of whether to rename the component containing `p` or the component containing `q`; the implementation above renames the one containing `p`. The figure on the next page shows a trace of the quick-union algorithm for `tinyUF.txt`. This trace is best understood in terms of the graphical representation depicted at left, which we consider next.

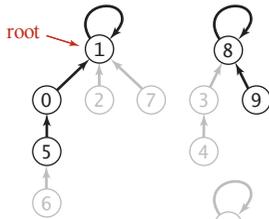
```
public int find(int p)
{ // Find component name.
  while (p != id[p]) p = id[p];
  return p;
}

public void union(int p, int q)
{ // Give p and q the same root.
  int i = find(p);
  int j = find(q);
  if (i == j) return;

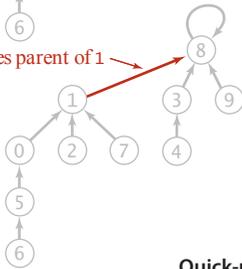
  id[i] = j;
  count--;
}
```

Quick-union

`id[]` is parent-link representation of a forest of trees



8 becomes parent of 1



Quick-union overview

`find` has to follow links to the root

p	q	0	1	2	3	4	5	6	7	8	9
5	9	1	1	1	8	3	0	5	1	8	8

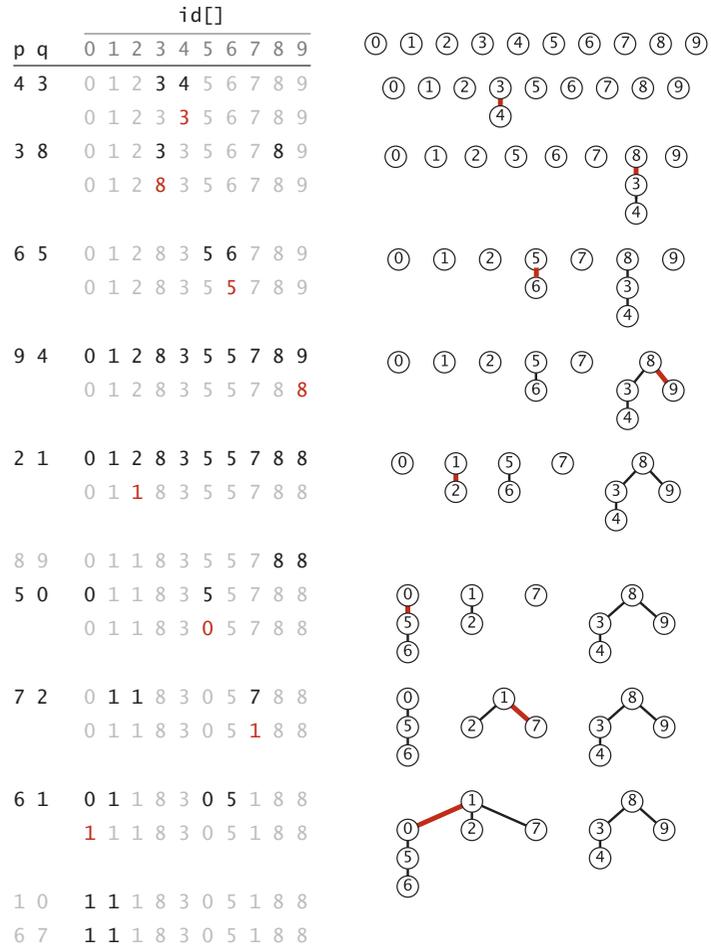
↑ find(5) is id[id[id[5]]] ↑ find(9) is id[id[9]]

union changes just one link

p	q	0	1	2	3	4	5	6	7	8	9
5	9	1	1	1	8	3	0	5	1	8	8
			1	8	1	8	3	0	5	1	8

Forest-of-trees representation. The code for quick-union is compact, but a bit opaque. Representing sites as nodes (labeled circles) and links as arrows from one node to another gives a graphical representation of the data structure that makes it relatively easy to understand the operation of the algorithm. The resulting structures are trees—in technical terms, our `id[]` array is a parent-link representation of a forest (set) of trees. To simplify the diagrams, we often omit both the arrowheads in the links (because they all point upwards) and the self-links in the roots of the trees. The forests corresponding to the `id[]` array for `tinyUF.txt` are shown at right. When we start at the node corresponding to any site and follow links, we eventually end up at the root of the tree containing that node. We can prove this property to be true by induction: It is true after the array is initialized to have every node link to itself, and if it is true before a given `union()` operation, it is certainly true afterward. Thus, the `find()` method on page 224 returns the name of the site at the root (so that `connected()` checks whether two sites are in the same tree). This representation is useful for this problem because the nodes corresponding to two sites are in the same tree if and only if the sites are in the same component.

Moreover, the trees are not difficult to build: the `union()` implementation on page 224 combines two trees into one in a single statement, by making the root of one the parent of the other.

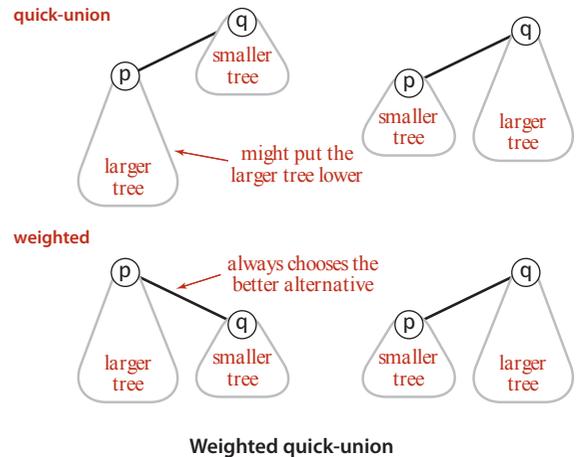


Quick-union trace (with corresponding forests of trees)

Again, suppose that we use quick-union for the dynamic connectivity problem and wind up with a single component. An immediate implication of PROPOSITION 6 is that the running time is quadratic, in the worst case. Suppose that the input pairs come in the order 0-1, then 0-2, then 0-3, and so forth. After $N - 1$ such pairs, we have N sites all in the same set, and the tree that is formed by the quick-union algorithm has height $N - 1$, with 0 linking to 1, which links to 2, which links to 3, and so forth (see the diagram on the facing page). By PROPOSITION 6, the number of array accesses for the `union()` operation for the pair 0- i is exactly $2i + 3$ (site 0 is at depth i and site i at depth 0). Thus, the total number of array accesses for the `find()` operations for these N pairs is $(3 + 5 + 7 + \dots + 2N + 1) \sim N^2$.

Weighted quick-union. Fortunately, there is an easy modification to quick-union that allows us to guarantee that bad cases such as this one do not occur. Rather than arbitrarily connecting the second tree to the first for `union()`, we keep track of the *size* of each tree and always connect the smaller tree to the larger. This change requires slightly more code and another array to hold the node counts, as shown on page 228, but it leads to substantial improvements in efficiency. We refer to this algorithm as the *weighted quick-union* algorithm. The forest of trees constructed by this algorithm for `tinyUF.txt` is shown in the figure at left on the top of page 229. Even for this small example, the tree height is substantially smaller than the height for the unweighted version.

Weighted quick-union analysis. The figure at right on the top of page 229 illustrates the worst case for weighted quick union, when the sizes of the trees to be merged by `union()` are always equal (and a power of 2). These tree structures look complex, but they have the simple property that the height of a tree of 2^n nodes is n . Furthermore, when we merge two trees of 2^n nodes, we get a tree of 2^{n+1} nodes, and we increase the height of the tree to $n + 1$. This observation generalizes to provide a proof that the weighted algorithm can guarantee *logarithmic* performance.



```
% java WeightedQuickUnionUF < mediumUF.txt
528 503
548 523
...
3 components

% java WeightedQuickUnionUF < largeUF.txt
786321 134521
696834 98245
...
6 components
```

ALGORITHM 1.5 (continued) Union-find implementation (weighted quick-union)

```
public class WeightedQuickUnionUF
{
    private int[] id;    // parent link (site indexed)
    private int[] sz;    // size of component for roots (site indexed)
    private int count;  // number of components

    public WeightedQuickUnionUF(int N)
    {
        count = N;
        id = new int[N];
        for (int i = 0; i < N; i++) id[i] = i;
        sz = new int[N];
        for (int i = 0; i < N; i++) sz[i] = 1;
    }

    public int count()
    { return count; }

    public boolean connected(int p, int q)
    { return find(p) == find(q); }

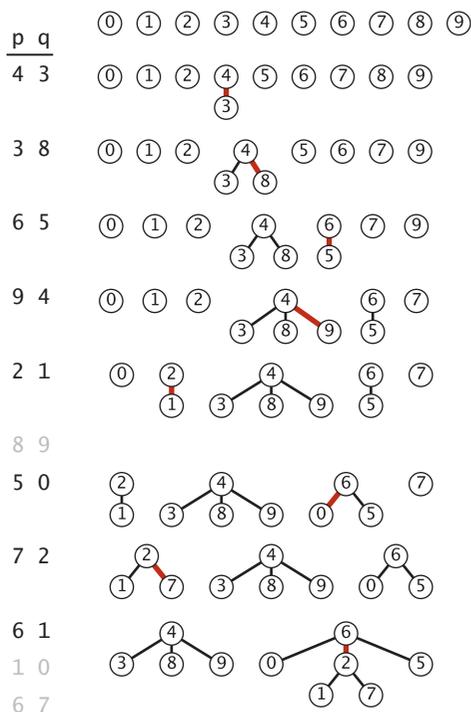
    public int find(int p)
    { // Follow links to find a root.
      while (p != id[p]) p = id[p];
      return p;
    }

    public void union(int p, int q)
    {
        int i = find(p);
        int j = find(q);
        if (i == j) return;

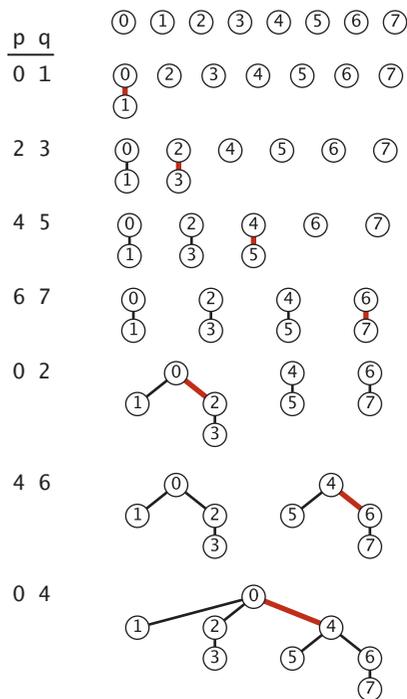
        // Make smaller root point to larger one.
        if (sz[i] < sz[j]) { id[i] = j; sz[j] += sz[i]; }
        else                { id[j] = i; sz[i] += sz[j]; }
        count--;
    }
}
```

This code is best understood in terms of the forest-of-trees representation described in the text. We add a site-indexed array `sz[]` as an instance variable so that `union()` can link the root of the smaller tree to the root of the larger tree. This addition makes it feasible to address large problems.

reference input



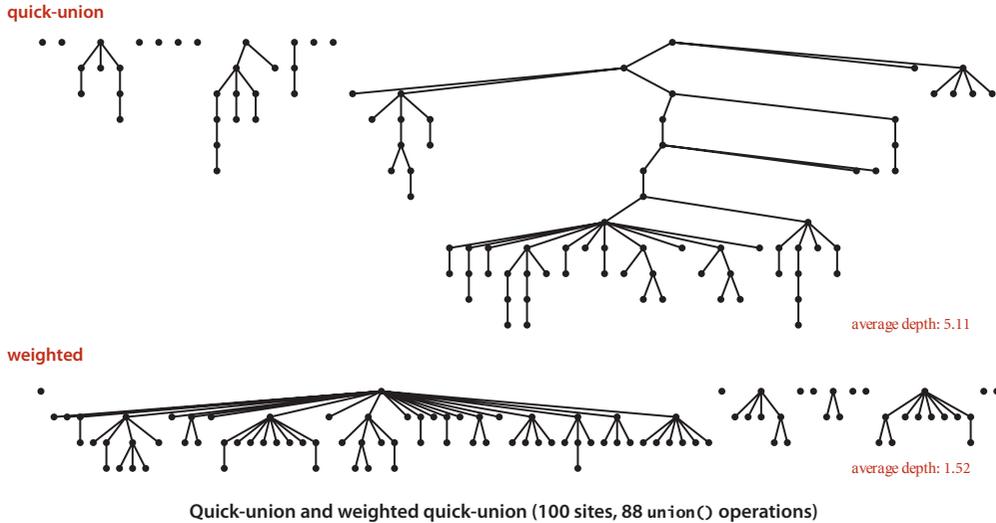
worst-case input



Weighted quick-union traces (forests of trees)

Proposition H. The depth of any node in a forest built by weighted quick-union for N sites is at most $\lg N$.

Proof: We prove a stronger fact by (strong) induction: The height of every tree of size k in the forest is at most $\lg k$. The base case follows from the fact that the tree height is 0 when k is 1. By the inductive hypothesis, assume that the tree height of a tree of size i is at most $\lg i$ for all $i < k$. When we combine a tree of size i with a tree of size j with $i \leq j$ and $i + j = k$, we increase the depth of each node in the smaller set by 1, but they are now in a tree of size $i + j = k$, so the property is preserved because $1 + \lg i = \lg(i + i) \leq \lg(i + j) = \lg k$.



Corollary. For weighted quick-union with N sites, the worst-case order of growth of the cost of `find()`, `connected()`, and `union()` is $\log N$.

Proof. Each operation does at most a constant number of array accesses for each node on the path from a node to a root in the forest.

For dynamic connectivity, the practical implication of **PROPOSITION H** and its corollary is that weighted quick-union is the only one of the three algorithms that can feasibly be used for huge practical problems. The weighted quick-union algorithm uses at most $c M \lg N$ array accesses to process M connections among N sites for a small constant c . This result is in stark contrast to our finding that quick-find always (and quick-union sometimes) uses at least MN array accesses. Thus, with weighted quick-union, we can guarantee that we can solve huge practical dynamic connectivity problems in a reasonable amount of time. For the price of a few extra lines of code, we get a program that can be millions of times faster than the simpler algorithms for the huge dynamic connectivity problems that we might encounter in practical applications.

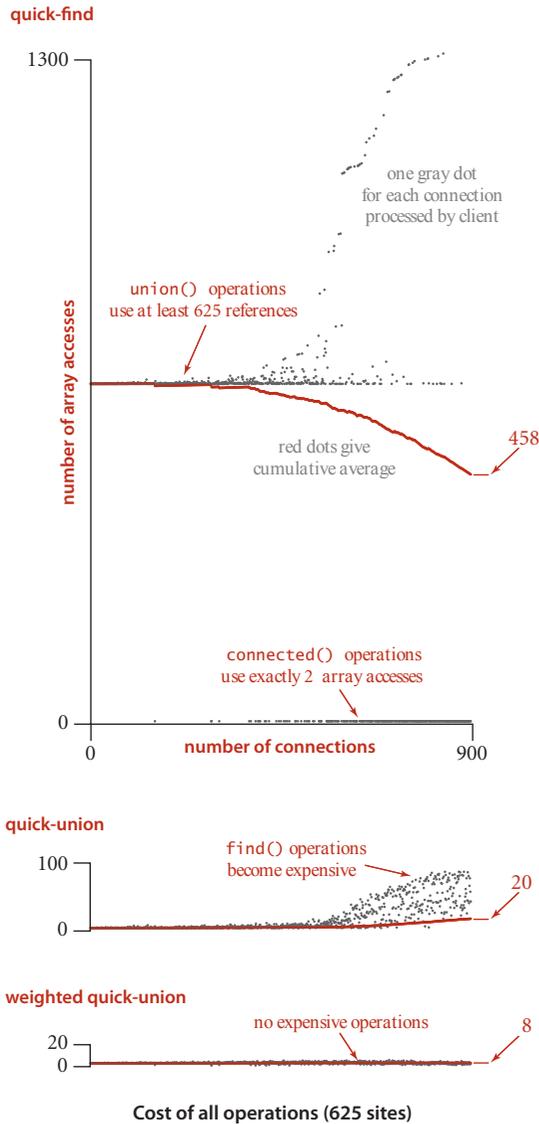
A 100-site example is shown on the top of this page. It is evident from this diagram that relatively few nodes fall far from the root with weighted quick-union. Indeed it is frequently the case that a 1-node tree is merged with a larger tree, which puts the node just one link from the root. Empirical studies on huge problems tell us that weighted quick-union typically solves practical problems in constant time per operation. We could hardly expect to find a more efficient algorithm.

algorithm	order of growth for N sites (worst case)		
	constructor	union	find
<i>quick-find</i>	N	N	1
<i>quick-union</i>	N	tree height	tree height
<i>weighted quick-union</i>	N	$\lg N$	$\lg N$
<i>weighted quick-union with path compression</i>	N	very, very nearly, but not quite 1 (amortized) (see EXERCISE 1.5.13)	
<i>impossible</i>	N	1	1

Performance characteristics of union-find algorithms

Optimal algorithms. Can we find an algorithm that has guaranteed constant-time-per-operation performance? This question is an extremely difficult one that plagued researchers for many years. In pursuit of an answer, a number of variations of quick-union and weighted quick-union have been studied. For example, the following method, known as *path compression*, is easy to implement. Ideally, we would like every node to link directly to the root of its tree, but we do not want to pay the price of changing a large number of links, as we did in the quick-find algorithm. We can approach the ideal simply by making all the nodes that we do examine directly link to the root. This step seems drastic at first blush, but it is easy to implement, and there is nothing sacrosanct about the structure of these trees: if we can modify them to make the algorithm more efficient, we should do so. To implement path compression, we just add another loop to `find()` that sets the `id[]` entry corresponding to each node encountered along the way to link directly to the root. The net result is to flatten the trees almost completely, approximating the ideal achieved by the quick-find algorithm. The method is simple and effective, but you are not likely to be able to discern any improvement over weighted quick-union in a practical situation (see EXERCISE 1.5.24). Theoretical results about the situation are extremely complicated and quite remarkable. *Weighted quick union with path compression is optimal but not quite constant-time per operation.* That is, not only is weighted quick-union with path compression not constant-time per operation in the worst case (amortized), but also there exists no algorithm that can guarantee to perform each union-find operation in amortized constant time (under the very general “cell probe” model of computation). Weighted quick-union with path compression is very close to the best that we can do for this problem.

Amortized cost plots. As with any data type implementation, it is worthwhile to run experiments to test the validity of our performance hypotheses for typical clients, as discussion in SECTION 1.4.



The figure at left shows details of the performance of the algorithms for our dynamic connectivity development client when solving our 625-site connectivity example (`mediumUF.txt`). Such diagrams are easy to produce (see EXERCISE 1.5.16): For the i th connection processed, we maintain a variable `cost` that counts the number of array accesses (to `id[]` or `sz[]`) and a variable `total` that is the sum of the total number of array accesses so far. Then we plot a gray dot at (i, cost) and a red dot at $(i, \text{total}/i)$. The red dots are the average cost per operation, or amortized cost. These plots provide good insights into algorithm behavior. For *quick-find*, every `union()` operation uses at least 625 accesses (plus 1 for each component merged, up to another 625) and every `connected()` operation uses 2 accesses. Initially, most of the connections lead to a call on `union()`, so the cumulative average hovers around 625; later, most connections are calls to `connected()` that cause the call to `union()` to be skipped, so the cumulative average decreases, but still remains relatively high. (Inputs that lead to a large number of `connected()` calls that cause `union()` to be skipped will exhibit significantly better performance—see EXERCISE 1.5.23 for an example). For *quick-union*, all operations initially require only a few array accesses; eventually, the height of the trees becomes a significant factor and the amortized cost grows noticeably. For *weighted quick-union*, the tree height stays small, none of the operations are expensive, and the amortized cost is low. These experiments

validate our conclusion that weighted quick-union is certainly worth implementing and that there is not much further room for improvement for practical problems.

Perspective Each of the UF implementations that we considered is an improvement over the previous in some intuitive sense, but the process is artificially smooth because we have the benefit of hindsight in looking over the development of the algorithms as they were studied by researchers over the years. The implementations are simple and the problem is well specified, so we can evaluate the various algorithms directly by running empirical studies. Furthermore, we can use these studies to validate mathematical results that quantify the performance of these algorithms. When possible, we follow the same basic steps for fundamental problems throughout the book that we have taken for union–find algorithms in this section, some of which are highlighted in this list:

- Decide on a complete and specific problem statement, including identifying fundamental abstract operations that are intrinsic to the problem and an API.
- Carefully develop a succinct implementation for a straightforward algorithm, using a well-thought-out development client and realistic input data.
- Know when an implementation could not possibly be used to solve problems on the scale contemplated and must be improved or abandoned.
- Develop improved implementations through a process of stepwise refinement, validating the efficacy of ideas for improvement through empirical analysis, mathematical analysis, or both.
- Find high-level abstract representations of data structures or algorithms in operation that enable effective high-level design of improved versions.
- Strive for worst-case performance guarantees when possible, but accept good performance on typical data when available.
- Know when to leave further improvements for detailed in-depth study to skilled researchers and move on to the next problem.

The potential for spectacular performance improvements for practical problems such as those that we saw for union–find makes algorithm design a compelling field of study. What other design activities hold the potential to reap savings factors of millions or billions, or more?

Developing an efficient algorithm is an intellectually satisfying activity that can have direct practical payoff. As the dynamic connectivity problem indicates, a simply stated problem can lead us to study numerous algorithms that are not only both useful and interesting, but also intricate and challenging to understand. We shall encounter many ingenious algorithms that have been developed over the years for a host of practical problems. As the scope of applicability of computational solutions to scientific and commercial problems widens, so also grows the importance of being able to use efficient algorithms to solve known problems and of being able to develop efficient solutions to new problems.

Q&A

Q. I'd like to add a `delete()` method to the API that allows clients to delete connections. Any advice on how to proceed?

A. No one has devised an algorithm as simple and efficient as the ones in this section that can handle deletions. This theme recurs throughout this book. Several of the data structures that we consider have the property that deleting something is much more difficult than adding something.

Q. What is the cell-probe model?

A. A model of computation where we only count accesses to a random-access memory large enough to hold the input and consider all other operations to be free.

EXERCISES

1.5.1 Show the contents of the `id[]` array and the number of times the array is accessed for each input pair when you use quick-find for the sequence 9-0 3-4 5-8 7-2 2-1 5-7 0-3 4-2.

1.5.2 Do EXERCISE 1.5.1, but use quick-union (page 224). In addition, draw the forest of trees represented by the `id[]` array after each input pair is processed.

1.5.3 Do EXERCISE 1.5.1, but use weighted quick-union (page 228).

1.5.4 Show the contents of the `sz[]` and `id[]` arrays and the number of array accesses for each input pair corresponding to the weighted quick-union examples in the text (both the reference input and the worst-case input).

1.5.5 Estimate the minimum amount of time (in days) that would be required for quick-find to solve a dynamic connectivity problem with 10^9 sites and 10^6 input pairs, on a computer capable of executing 10^9 instructions per second. Assume that each iteration of the inner for loop requires 10 machine instructions.

1.5.6 Repeat EXERCISE 1.5.5 for weighted quick-union.

1.5.7 Develop classes `QuickUnionUF` and `QuickFindUF` that implement quick-union and quick-find, respectively.

1.5.8 Give a counterexample that shows why this intuitive implementation of `union()` for quick-find is not correct:

```
public void union(int p, int q)
{
    if (connected(p, q)) return;

    // Rename p's component to q's name.
    for (int i = 0; i < id.length; i++)
        if (id[i] == id[p]) id[i] = id[q];
    count--;
}
```

1.5.9 Draw the tree corresponding to the `id[]` array depicted at right. Can this be the result of running weighted quick-union? Explain why this is impossible or give a sequence of operations that results in this array.

<i>i</i>	0	1	2	3	4	5	6	7	8	9
<i>id[i]</i>	1	1	3	1	5	6	1	3	4	5

EXERCISES *(continued)*

1.5.10 In the weighted quick-union algorithm, suppose that we set $\text{id}[\text{find}(p)]$ to q instead of to $\text{id}[\text{find}(q)]$. Would the resulting algorithm be correct?

Answer: Yes, but it would increase the tree height, so the performance guarantee would be invalid.

1.5.11 Implement *weighted quick-find*, where you always change the $\text{id}[]$ entries of the smaller component to the identifier of the larger component. How does this change affect performance?

CREATIVE PROBLEMS

1.5.12 *Quick-union with path compression.* Modify quick-union (page 224) to include *path compression*, by adding a loop to `find()` that links every site on the path from `p` to the root. Give a sequence of input pairs that causes this method to produce a path of length 4. *Note:* The amortized cost per operation for this algorithm is known to be logarithmic.

1.5.13 *Weighted quick-union with path compression.* Modify weighted quick-union (ALGORITHM 1.5) to implement path compression, as described in EXERCISE 1.5.12. Give a sequence of input pairs that causes this method to produce a tree of height 4. *Note:* The amortized cost per operation for this algorithm is known to be bounded by a function known as the inverse Ackermann function and is less than 5 for any conceivable practical value of N .

1.5.14 *Weighted quick-union by height.* Develop a UF implementation that uses the same basic strategy as weighted quick-union but keeps track of tree height and always links the shorter tree to the taller one. Prove a logarithmic upper bound on the height of the trees for N sites with your algorithm.

1.5.15 *Binomial trees.* Show that the number of nodes at each level in the worst-case trees for weighted quick-union are *binomial coefficients*. Compute the average depth of a node in a worst-case tree with $N = 2^n$ nodes.

1.5.16 *Amortized costs plots.* Instrument your implementations from EXERCISE 1.5.7 to make amortized costs plots like those in the text.

1.5.17 *Random connections.* Develop a UF client `ErdosRenyi` that takes an integer value N from the command line, generates random pairs of integers between 0 and $N-1$, calling `connected()` to determine if they are connected and then `union()` if not (as in our development client), looping until all sites are connected, and printing the number of connections generated. Package your program as a static method `count()` that takes N as argument and returns the number of connections and a `main()` that takes N from the command line, calls `count()`, and prints the returned value.

1.5.18 *Random grid generator.* Write a program `RandomGrid` that takes an `int` value N from the command line, generates all the connections in an N -by- N grid, puts them in random order, randomly orients them (so that `p q` and `q p` are equally likely to occur), and prints the result to standard output. To randomly order the connections, use a `RandomBag` (see EXERCISE 1.3.34 on page 167). To encapsulate `p` and `q` in a single object,

CREATIVE PROBLEMS *(continued)*

use the `Connection` nested class shown below. Package your program as two static methods: `generate()`, which takes `N` as argument and returns an array of connections, and `main()`, which takes `N` from the command line, calls `generate()`, and iterates through the returned array to print the connections.

1.5.19 *Animation.* Write a `RandomGrid` client (see EXERCISE 1.5.18) that uses `UnionFind` as in our development client to check connectivity and uses `StdDraw` to draw the connections as they are processed.

1.5.20 *Dynamic growth.* Using linked lists or a resizing array, develop a weighted quick-union implementation that removes the restriction on needing the number of objects ahead of time. Add a method `newSite()` to the API, which returns an `int` identifier.

```
private class Connection
{
    int p;
    int q;

    public Connection(int p, int q)
    { this.p = p; this.q = q; }
}
```

Record to encapsulate connections

EXPERIMENTS

1.5.21 *Erdős-Renyi model.* Use your client from EXERCISE 1.5.17 to test the hypothesis that the number of pairs generated to get one component is $\sim \frac{1}{2}N \ln N$.

1.5.22 *Doubling test for Erdős-Renyi model.* Develop a performance-testing client that takes an `int` value `T` from the command line and performs `T` trials of the following experiment: Use your client from EXERCISE 1.5.17 to generate random connections, using `UnionFind` to determine connectivity as in our development client, looping until all sites are connected. For each `N`, print the value of `N`, the average number of connections processed, and the ratio of the running time to the previous. Use your program to validate the hypotheses in the text that the running times for quick-find and quick-union are quadratic and weighted quick-union is near-linear.

1.5.23 *Compare quick-find with quick-union for Erdős-Renyi model.* Develop a performance-testing client that takes an `int` value `T` from the command line and performs `T` trials of the following experiment: Use your client from EXERCISE 1.5.17 to generate random connections. Save the connections, so that you can use both quick-union and quick-find to determine connectivity as in our development client, looping until all sites are connected. For each `N`, print the value of `N` and the ratio of the two running times.

1.5.24 *Fast algorithms for Erdős-Renyi model.* Add weighted quick-union and weighted quick-union with path compression to your tests from EXERCISE 1.5.23. Can you discern a difference between these two algorithms?

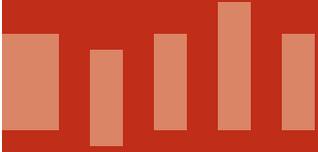
1.5.25 *Doubling test for random grids.* Develop a performance-testing client that takes an `int` value `T` from the command line and performs `T` trials of the following experiment: Use your client from EXERCISE 1.5.18 to generate the connections in an `N`-by-`N` square grid, randomly oriented and in random order, then use `UnionFind` to determine connectivity as in our development client, looping until all sites are connected. For each `N`, print the value of `N`, the average number of connections processed, and the ratio of the running time to the previous. Use your program to validate the hypotheses in the text that the running times for quick-find and quick-union are quadratic and weighted quick-union is near-linear. Note: As `N` doubles, the number of sites in the grid increases by a factor of 4, so expect a doubling factor of 16 for quadratic and 4 for linear.

EXPERIMENTS *(continued)*

1.5.26 *Amortized plot for Erdős-Renyi.* Develop a client that takes an `int` value `N` from the command line and does an amortized plot of the cost of all operations in the style of the plots in the text for the process of generating random pairs of integers between 0 and `N-1`, calling `connected()` to determine if they are connected and then `union()` if not (as in our development client), looping until all sites are connected.

This page intentionally left blank

TWO



Sorting

2.1	Elementary Sorts.	244
2.2	Mergesort	270
2.3	Quicksort	288
2.4	Priority Queues.	308
2.5	Applications.	336

Sorting is the process of rearranging a sequence of objects so as to put them in some logical order. For example, your credit card bill presents transactions in order by date—they were likely put into that order by a sorting algorithm. In the early days of computing, the common wisdom was that up to 30 percent of all computing cycles was spent sorting. If that fraction is lower today, one likely reason is that sorting algorithms are relatively efficient, not that sorting has diminished in relative importance. Indeed, the ubiquity of computer usage has put us awash in data, and the first step to organizing data is often to sort it. All computer systems have implementations of sorting algorithms, for use by the system and by users.

There are three practical reasons for you to study sorting algorithms, even though you might just use a system sort:

- Analyzing sorting algorithms is a thorough introduction to the approach that we use to compare algorithm performance throughout the book.
- Similar techniques are effective in addressing other problems.
- We often use sorting algorithms as a starting point to solve other problems.

More important than these practical reasons is that the algorithms are elegant, classic, and effective.

Sorting plays a major role in commercial data processing and in modern scientific computing. Applications abound in transaction processing, combinatorial optimization, astrophysics, molecular dynamics, linguistics, genomics, weather prediction, and many other fields. Indeed, a sorting algorithm (quicksort, in SECTION 2.3) was named as one of the top ten algorithms for science and engineering of the 20th century.

In this chapter, we consider several classical sorting methods and an efficient implementation of a fundamental data type known as the priority queue. We discuss the theoretical basis for comparing sorting algorithms and conclude the chapter with a survey of applications of sorting and priority queues.



2.1 ELEMENTARY SORTS

FOR OUR FIRST EXCURSION into the area of sorting algorithms, we shall study two elementary sorting methods and a variation of one of them. Among the reasons for studying these relatively simple algorithms in detail are the following: First, they provide context in which we can learn terminology and basic mechanisms. Second, these simple algorithms are more effective in some applications than the sophisticated algorithms that we shall discuss later. Third, they are useful in improving the efficiency of more sophisticated algorithms, as we will see.

Rules of the game Our primary concern is algorithms for rearranging arrays of items where each item contains a key. The objective of the sorting algorithm is to rearrange the items such that their keys are ordered according to some well-defined ordering rule (usually numerical or alphabetical order). We want to rearrange the array so that each entry's key is no smaller than the key in each entry with a lower index and no larger than the key in each entry with a larger index. Specific characteristics of the keys and the items can vary widely across applications. In Java, items are just objects, and the abstract notion of a key is captured in a built-in mechanism—the `Comparable` interface—that is described on page 247.

The class `Example` on the facing page illustrates the conventions that we shall use: we put our sort code in a `sort()` method within a single class along with private helper functions `less()` and `exch()` (and perhaps some others) and a sample client `main()`. `Example` also illustrates code that might be useful for initial debugging: its test client `main()` sorts strings from standard input using the private method `show()` to print the contents of the array. Later in this chapter, we will examine various test clients for comparing algorithms and for studying their performance. To differentiate sorting methods, we give our various sort classes different names. Clients can call different implementations by name: `Insertion.sort()`, `Merge.sort()`, `Quick.sort()`, and so forth.

With but a few exceptions, our sort code refers to the data only through two operations: the method `less()` that compares items and the method `exch()` that exchanges them. The `exch()` method is easy to implement, and the `Comparable` interface makes it easy to implement `less()`. Restricting data access to these two operations makes our code readable and portable, and makes it easier for us to certify that algorithms are correct, to study performance and to compare algorithms. Before proceeding to consider sort implementations, we discuss a number of important issues that need to be carefully considered for every sort.

Template for sort classes

```

public class Example
{
    public static void sort(Comparable[] a)
    { /* See Algorithms 2.1, 2.2, 2.3, 2.4, 2.5, or 2.7. */ }

    private static boolean less(Comparable v, Comparable w)
    { return v.compareTo(w) < 0; }

    private static void exch(Comparable[] a, int i, int j)
    { Comparable t = a[i]; a[i] = a[j]; a[j] = t; }

    private static void show(Comparable[] a)
    { // Print the array, on a single line.
      for (int i = 0; i < a.length; i++)
        StdOut.print(a[i] + " ");
      StdOut.println();
    }

    public static boolean isSorted(Comparable[] a)
    { // Test whether the array entries are in order.
      for (int i = 1; i < a.length; i++)
        if (less(a[i], a[i-1])) return false;
      return true;
    }

    public static void main(String[] args)
    { // Read strings from standard input, sort them, and print.
      String[] a = In.readString();
      sort(a);
      assert isSorted(a);
      show(a);
    }
}

```

This class illustrates our conventions for implementing array sorts. For each sorting algorithm that we consider, we present a `sort()` method for a class like this with `Example` changed to a name that corresponds to the algorithm. The test client sorts strings taken from standard input, but, with this code, our sort methods are effective for any type of data that implements `Comparable`.

```

% more tiny.txt
S O R T E X A M P L E

```

```

% java Example < tiny.txt
A E E L M O P R S T X

```

```

% more words3.txt
bed bug dad yes zoo ... all bad yet

```

```

% java Example < words3.txt
all bad bed bug dad ... yes yet zoo

```

Certification. Does the sort implementation always put the array in order, no matter what the initial order? As a conservative practice, we include the statement `assert isSorted(a);` in our test client to certify that array entries are in order after the sort. It is reasonable to include this statement in every sort implementation, even though we normally test our code and develop mathematical arguments that our algorithms are correct. Note that this test is sufficient only if we use `exch()` exclusively to change array entries. When we use code that stores values into the array directly, we do not have full assurance (for example, code that destroys the original input array by setting all values to be the same would pass this test).

Sorting cost model.

When studying sorting algorithms, we count compares and *exchanges*. For algorithms that do not use exchanges, we count *array accesses*.

Running time. We also test algorithm performance. We start by proving facts about the number of basic operations (compares and exchanges, or perhaps the number of times the array is accessed, for read or write) that the various sorting algorithms perform for various natural input models. Then we use these facts to develop hypotheses about the comparative performance of the algorithms and present tools that you can use to experimentally check the validity of such hypotheses. We use a consistent coding style to facilitate the development of valid hypotheses about performance that will hold true for typical implementations.

Extra memory. The amount of extra memory used by a sorting algorithm is often as important a factor as running time. The sorting algorithms divide into two basic types: those that sort in place and use no extra memory except perhaps for a small function-call stack or a constant number of instance variables, and those that need enough extra memory to hold another copy of the array to be sorted.

Types of data. Our sort code is effective for any item type that implements the `Comparable` interface. Adhering to Java's convention in this way is convenient because many of the types of data that you might want to sort implement `Comparable`. For example, Java's numeric wrapper types such as `Integer` and `Double` implement `Comparable`, as do `String` and various advanced types such as `File` or `URL`. Thus, you can just call one of our sort methods with an array of any of these types as argument. For example, the code at right uses quicksort (see SECTION 2.3) to sort *N* random `Double` values. When we create types of our own, we can enable client code to sort that type of data by implementing the `Comparable` interface. To do so, we just need to implement a `compareTo()` method that defines an ordering on objects of that type known as the natural

```
Double a[] = new Double[N];
for (int i = 0; i < N; i++)
    a[i] = StdRandom.uniform();
Quick.sort(a);
```

Sorting an array of random values

order for that type, as shown here for our `Date` data type (see page 91). Java's convention is that the call `v.compareTo(w)` returns an integer that is negative, zero, or positive (usually `-1`, `0`, or `+1`) when $v < w$, $v = w$, or $v > w$, respectively. For economy, we use standard notation like $v > w$ as shorthand for code like `v.compareTo(w) > 0` for the remainder of this paragraph. By convention, `v.compareTo(w)` throws an exception if `v` and `w` are incompatible types or either is `null`. Furthermore, `compareTo()` must implement a total order: it must be

- *Reflexive* (for all `v`, $v = v$)
- *Antisymmetric* (for all `v` and `w`, if $v < w$ then $w > v$ and if $v = w$ then $w = v$)
- *Transitive* (for all `v`, `w`, and `x`, if $v <= w$ and $w <= x$ then $v <= x$)

These rules are intuitive and standard in mathematics—you will have little difficulty adhering to them. In short, `compareTo()` implements our key abstraction—it defines the ordering of the items (objects) to be sorted, which can be any type of data that implements `Comparable`. Note that `compareTo()` need not use all of the instance variables. Indeed, the key might be a small part of each item.

```
public class Date implements Comparable<Date>
{
    private final int day;
    private final int month;
    private final int year;

    public Date(int d, int m, int y)
    { day = d; month = m; year = y; }

    public int day() { return day; }
    public int month() { return month; }
    public int year() { return year; }

    public int compareTo(Date that)
    {
        if (this.year > that.year ) return +1;
        if (this.year < that.year ) return -1;
        if (this.month > that.month) return +1;
        if (this.month < that.month) return -1;
        if (this.day > that.day ) return +1;
        if (this.day < that.day ) return -1;
        return 0;
    }

    public String toString()
    { return month + "/" + day + "/" + year; }
}
```

Defining a comparable type

FOR THE REMAINDER OF THIS CHAPTER, we shall address numerous algorithms for sorting arrays of objects having a natural order. To compare and contrast the algorithms, we shall examine a number of their properties, including the number of compares and exchanges that they use for various types of inputs and the amount of extra memory that they use. These properties lead to the development of hypotheses about performance properties, many of which have been validated on countless computers over the past several decades. Specific implementations always need to be checked, so we also consider tools for doing so. After considering the classic selection sort, insertion sort, shellsort, mergesort, quicksort, and heapsort algorithms, we will consider practical issues and applications, in SECTION 2.5.

Selection sort One of the simplest sorting algorithms works as follows: First, find the smallest item in the array and exchange it with the first entry (itself if the first entry is already the smallest). Then, find the next smallest item and exchange it with the second entry. Continue in this way until the entire array is sorted. This method is called selection sort because it works by repeatedly selecting the smallest remaining item.

As you can see from the implementation in ALGORITHM 2.1, the inner loop of selection sort is just a compare to test a current item against the smallest item found so far (plus the code necessary to increment the current index and to check that it does not exceed the array bounds); it could hardly be simpler. The work of moving the items around falls outside the inner loop: each exchange puts an item into its final position, so the number of exchanges is N . Thus, the running time is dominated by the number of compares.

Proposition A. Selection sort uses $\sim N^2/2$ compares and N exchanges to sort an array of length N .

Proof: You can prove this fact by examining the trace, which is an N -by- N table in which unshaded letters correspond to compares. About one-half of the entries in the table are unshaded—those on and above the diagonal. The entries on the diagonal each correspond to an exchange. More precisely, examination of the code reveals that, for each i from 0 to $N - 1$, there is one exchange and $N - 1 - i$ compares, so the totals are N exchanges and $(N - 1) + (N - 2) + \dots + 2 + 1 + 0 = N(N - 1) / 2 \sim N^2 / 2$ compares.

IN SUMMARY, selection sort is a simple sorting method that is easy to understand and to implement and is characterized by the following two signature properties:

Running time is insensitive to input. The process of finding the smallest item on one pass through the array does not give much information about where the smallest item might be on the next pass. This property can be disadvantageous in some situations. For example, the person using the sort client might be surprised to realize that it takes about as long to run selection sort for an array that is already in order or for an array with all keys equal as it does for a randomly-ordered array! As we shall see, other algorithms are better able to take advantage of initial order in the input.

Data movement is minimal. Each of the N exchanges changes the value of two array entries, so selection sort uses N exchanges—the number of exchanges is a linear function of the array size. None of the other sorting algorithms that we consider have this property (most involve linearithmic or quadratic growth).

ALGORITHM 2.1 Selection sort

```

public class Selection
{
    public static void sort(Comparable[] a)
    { // Sort a[] into increasing order.
        int N = a.length;           // array length
        for (int i = 0; i < N; i++)
        { // Exchange a[i] with smallest entry in a[i+1..N].
            int min = i;           // index of a minimal entry.
            for (int j = i+1; j < N; j++)
                if (less(a[j], a[min])) min = j;
            exch(a, i, min);
        }
    }
    // See page 245 for less(), exch(), isSorted(), and main().
}

```

For each i , this implementation puts the i th smallest item in $a[i]$. The entries to the left of position i are the i smallest items in the array and are not examined again.

		a[]										
i	min	0	1	2	3	4	5	6	7	8	9	10
		S	O	R	T	E	X	A	M	P	L	E
0	6	S	O	R	T	E	X	A	M	P	L	E
1	4	A	O	R	T	E	X	S	M	P	L	E
2	10	A	E	R	T	O	X	S	M	P	L	E
3	9	A	E	E	T	O	X	S	M	P	L	R
4	7	A	E	E	L	O	X	S	M	P	T	R
5	7	A	E	E	L	M	X	S	O	P	T	R
6	8	A	E	E	L	M	O	S	X	P	T	R
7	10	A	E	E	L	M	O	P	X	S	T	R
8	8	A	E	E	L	M	O	P	R	S	T	X
9	9	A	E	E	L	M	O	P	R	S	T	X
10	10	A	E	E	L	M	O	P	R	S	T	X
		A	E	E	L	M	O	P	R	S	T	X

entries in black are examined to find the minimum
 entries in red are a[min]
 entries in gray are in final position

Trace of selection sort (array contents just after each exchange)

Insertion sort The algorithm that people often use to sort bridge hands is to consider the cards one at a time, inserting each into its proper place among those already considered (keeping them sorted). In a computer implementation, we need to make space to insert the current item by moving larger items one position to the right, before inserting the current item into the vacated position. ALGORITHM 2.2 is an implementation of this method, which is called insertion sort.

As in selection sort, the items to the left of the current index are in sorted order during the sort, but they are not in their final position, as they may have to be moved to make room for smaller items encountered later. The array is, however, fully sorted when the index reaches the right end.

Unlike that of selection sort, the running time of insertion sort depends on the initial order of the items in the input. For example, if the array is large and its entries are already in order (or nearly in order), then insertion sort is much, much faster than if the entries are randomly ordered or in reverse order.

Proposition B. Insertion sort uses $\sim N^2/4$ compares and $\sim N^2/4$ exchanges to sort a randomly ordered array of length N with distinct keys, on the average. The worst case is $\sim N^2/2$ compares and $\sim N^2/2$ exchanges and the best case is $N - 1$ compares and 0 exchanges.

Proof: Just as for PROPOSITION A, the number of compares and exchanges is easy to visualize in the N -by- N diagram that we use to illustrate the sort. We count entries below the diagonal—all of them, in the worst case, and none of them, in the best case. For randomly ordered arrays, we expect each item to go about halfway back, on the average, so we count one-half of the entries below the diagonal.

The number of compares is the number of exchanges plus an additional term equal to N minus the number of times the item inserted is the smallest so far. In the worst case (array in reverse order), this term is negligible in relation to the total; in the best case (array in order) it is equal to $N - 1$.

Insertion sort works well for certain types of nonrandom arrays that often arise in practice, even if they are huge. For example, as just mentioned, consider what happens when you use insertion sort on an array that is already sorted. Each item is immediately determined to be in its proper place in the array, and the total running time is linear. (The running time of selection sort is quadratic for such an array.) The same is true for arrays whose keys are all equal (hence the condition in PROPOSITION B that the keys must be distinct).

ALGORITHM 2.2 Insertion sort

```

public class Insertion
{
    public static void sort(Comparable[] a)
    { // Sort a[] into increasing order.
        int N = a.length;
        for (int i = 1; i < N; i++)
        { // Insert a[i] among a[i-1], a[i-2], a[i-3]....
            for (int j = i; j > 0 && less(a[j], a[j-1]); j--)
                exch(a, j, j-1);
        }
    }
    // See page 245 for less(), exch(), isSorted(), and main().
}

```

For each i from 1 to $N-1$, exchange $a[i]$ with the entries that are larger in $a[0]$ through $a[i-1]$. As the index i travels from left to right, the entries to its left are in sorted order in the array, so the array is fully sorted when i reaches the right end.

		a[]										
i	j	0	1	2	3	4	5	6	7	8	9	10
		S	O	R	T	E	X	A	M	P	L	E
1	0	O	S	R	T	E	X	A	M	P	L	E
2	1	O	R	S	T	E	X	A	M	P	L	E
3	3	O	R	S	T	E	X	A	M	P	L	E
4	0	E	O	R	S	T	X	A	M	P	L	E
5	5	E	O	R	S	T	X	A	M	P	L	E
6	0	A	E	O	R	S	T	X	M	P	L	E
7	2	A	E	M	O	R	S	T	X	P	L	E
8	4	A	E	M	O	P	R	S	T	X	L	E
9	2	A	E	L	M	O	P	R	S	T	X	E
10	2	A	E	E	L	M	O	P	R	S	T	X
		A	E	E	L	M	O	P	R	S	T	X

entries in gray
do not move

entry in red
is a[j]

entries in black
moved one position
right for insertion

Trace of insertion sort (array contents just after each insertion)

More generally, we consider the concept of a partially sorted array, as follows: An inversion is a pair of entries that are out of order in the array. For instance, E X A M P L E has 11 inversions: E-A, X-A, X-M, X-P, X-L, X-E, M-L, M-E, P-L, P-E, and L-E. If the number of inversions in an array is less than a constant multiple of the array size, we say that the array is partially sorted. Typical examples of partially sorted arrays are the following:

- An array where each entry is not far from its final position
- A small array appended to a large sorted array
- An array with only a few entries that are not in place

Insertion sort is an efficient method for such arrays; selection sort is not. Indeed, when the number of inversions is low, insertion sort is likely to be faster than any sorting method that we consider in this chapter.

Proposition C. The number of exchanges used by insertion sort is equal to the number of inversions in the array, and the number of compares is at least equal to the number of inversions and at most equal to the number of inversions plus the array size minus 1.

Proof: Every exchange involves two inverted adjacent entries and thus reduces the number of inversions by one, and the array is sorted when the number of inversions reaches zero. Every exchange corresponds to a compare, and an additional compare might happen for each value of i from 1 to $N-1$ (when $a[i]$ does not reach the left end of the array).

It is not difficult to speed up insertion sort substantially, by shortening its inner loop to move the larger entries to the right one position rather than doing full exchanges (thus cutting the number of array accesses in half). We leave this improvement for an exercise (see EXERCISE 2.1.25).

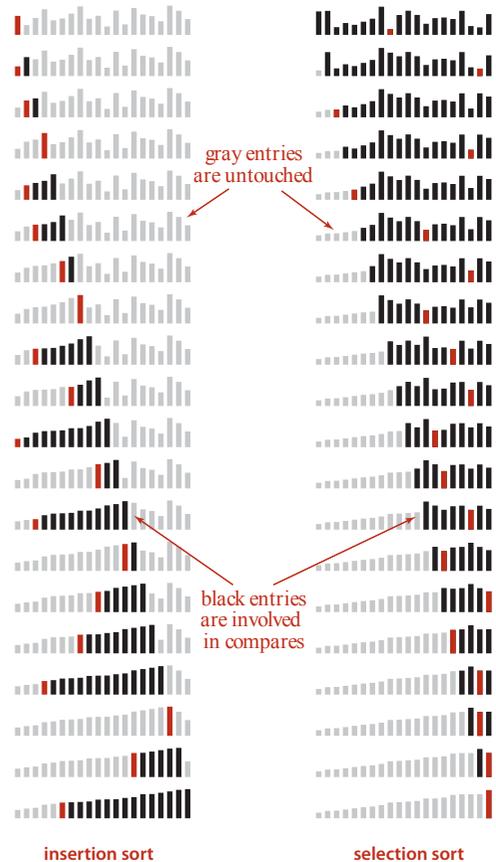
IN SUMMARY, insertion sort is an excellent method for partially sorted arrays and is also a fine method for tiny arrays. These facts are important not just because such arrays frequently arise in practice, but also because both types of arrays arise in intermediate stages of advanced sorting algorithms, so we will be considering insertion sort again in relation to such algorithms.

Visualizing sorting algorithms Throughout this chapter, we will be using a simple visual representation to help describe the properties of sorting algorithms. Rather than tracing the progress of a sort with key values such as letters, numbers, or words, we use vertical bars, to be sorted by their heights. The advantage of such a representation is that it can give insights into the behavior of a sorting method.

For example, you can see at a glance on the visual traces at right that insertion sort does not touch entries to the right of the scan pointer and selection sort does not touch entries to the left of the scan pointer. Moreover, it is clear from the visual traces that, since insertion sort also does not touch entries smaller than the inserted item, it uses about half the number of compares as selection sort, on the average.

With our StdDraw library, developing a visual trace is not much more difficult than doing a standard trace. We sort `Double` values, instrument the algorithm to call `show()` as appropriate (just as we do for a standard trace), and develop a version of `show()` that uses `StdDraw` to draw the bars instead of printing the results. The most complicated task is setting the scale for the y -axis so that the lines of the trace appear in the expected order. You are encouraged to work EXERCISE 2.1.18 in order to gain a better appreciation of the value of visual traces and the ease of creating them.

An even simpler task is to animate the trace so that you can see the array dynamically evolve to the sorted result. Developing an animated trace involves essentially the same process described in the previous paragraph, but without having to worry about the y -axis (just clear the window and redraw the bars each time). Though we cannot make the case on the printed page, such animated representations are also effective in gaining insight into how an algorithm works. You are also encouraged to work EXERCISE 2.1.17 to see for yourself.



Visual traces of elementary sorting algorithms

Comparing two sorting algorithms Now that we have two implementations, we are naturally interested in knowing which one is faster: selection sort (ALGORITHM 2.1) or insertion sort (ALGORITHM 2.2). Questions like this arise again and again and again in the study of algorithms and are a major focus throughout this book. We have discussed some fundamental ideas in CHAPTER 1, but we use this first case in point to illustrate our basic approach to answering such questions. Generally, following the approach introduced in SECTION 1.4, we compare algorithms by

- Implementing and debugging them
- Analyzing their basic properties
- Formulating a hypothesis about comparative performance
- Running experiments to validate the hypothesis

These steps are nothing more than the time-honored *scientific method*, applied to the study of algorithms.

In the present context, ALGORITHM 2.1 and ALGORITHM 2.2 are evidence of the first step; PROPOSITIONS A, B, and C constitute the second step; PROPERTY D on page 255 constitutes the third step; and the class `SortCompare` on page 256 enables the fourth step. These activities are all interrelated.

Our brief descriptions mask a substantial amount of effort that is required to properly implement, analyze, and test algorithms. Every programmer knows that such code is the product of a long round of debugging and refinement, every mathematician knows that proper analysis can be very difficult, and every scientist knows that formulating hypotheses and designing and executing experiments to validate them require great care. Full development of such results is reserved for experts studying our most important algorithms, but every programmer using an algorithm should be aware of the scientific context underlying its performance properties.

Having developed implementations, our next choice is to settle on an appropriate model for the input. For sorting, a natural model, which we have used for PROPOSITIONS A, B, and C, is to assume that the arrays are randomly ordered and that the key values are distinct. In applications where significant numbers of equal key values are present we will need a more complicated model.

How do we formulate a hypothesis about the running times of insertion sort and selection sort for randomly ordered arrays? Examining ALGORITHMS 2.1 and 2.2 and PROPOSITIONS A and B, it follows immediately that the running time of both algorithms should be quadratic for randomly ordered arrays. That is, the running time of insertion sort for such an input is proportional to some small constant times N^2 and the running time of selection sort is proportional to some other small constant times N^2 . The values of the two constants depend on the cost of compares and exchanges on the particular computer being used. For many types of data and for typical computers, it is reasonable

to assume that these costs are similar (though we will see a few significant exceptions). The following hypothesis follows directly:

Property D. The running times of insertion sort and selection sort are quadratic and within a small constant factor of one another for randomly ordered arrays of distinct values.

Evidence: This statement has been validated on many different computers over the past half-century. Insertion sort was about twice as fast as selection sort when the first edition of this book was written in 1980 and it still is today, even though it took several hours to sort 100,000 items with these algorithms then and just several seconds today. Is insertion sort a bit faster than selection sort on your computer? To find out, you can use the class `SortCompare` on the next page, which uses the `sort()` methods in the classes named as command-line arguments to perform the given number of experiments (sorting arrays of the given size) and prints the ratio of the observed running times of the algorithms.

To validate this hypothesis, we use `SortCompare` (see page 256) to perform the experiments. As usual, we use `Stopwatch` to compute the running time. The implementation of `time()` shown here does the job for the basic sorts in this chapter. The “randomly ordered” input model is embedded in the `timeRandomInput()` method in `SortCompare`, which generates random `Double` values, sorts them, and returns the total measured time of the sort for the given number of trials. Using random `Double` values between 0.0 and 1.0 is much simpler than the alternative of using a library function such as `StdRandom.shuffle()` and is effective because equal key values are very unlikely (see EXERCISE 2.5.31). As discussed in CHAPTER 1, the number of trials is taken as an argument both to take advantage of the law of large numbers (the more trials, the total running time divided by the number of trials is a more accurate estimate of the true average running time) and to help damp out system effects. You are encouraged to experiment with `SortCompare`

```
public static double time(String alg, Comparable[] a)
{
    Stopwatch timer = new Stopwatch();
    if (alg.equals("Insertion")) Insertion.sort(a);
    if (alg.equals("Selection")) Selection.sort(a);
    if (alg.equals("Shell"))     Shell.sort(a);
    if (alg.equals("Merge"))     Merge.sort(a);
    if (alg.equals("Quick"))     Quick.sort(a);
    if (alg.equals("Heap"))      Heap.sort(a);
    return timer.elapsedTime();
}
```

Timing one of the sort algorithms in this chapter on a given input

Comparing two sorting algorithms

```
public class SortCompare
{
    public static double time(String alg, Double[] a)
    { /* See text. */ }

    public static double timeRandomInput(String alg, int N, int T)
    { // Use alg to sort T random arrays of length N.
      double total = 0.0;
      Double[] a = new Double[N];
      for (int t = 0; t < T; t++)
      { // Perform one experiment (generate and sort an array).
        for (int i = 0; i < N; i++)
          a[i] = StdRandom.uniform();
        total += time(alg, a);
      }
      return total;
    }

    public static void main(String[] args)
    {
      String alg1 = args[0];
      String alg2 = args[1];
      int N = Integer.parseInt(args[2]);
      int T = Integer.parseInt(args[3]);
      double t1 = timeRandomInput(alg1, N, T); // total for alg1
      double t2 = timeRandomInput(alg2, N, T); // total for alg2
      StdOut.printf("For %d random Doubles\n  %s is", N, alg1);
      StdOut.printf(" %.1f times faster than %s\n", t2/t1, alg2);
    }
}
```

This client runs the two sorts named in the first two command-line arguments on arrays of N (the third command-line argument) random `Double` values between 0.0 and 1.0, repeating the experiment T (the fourth command-line argument) times, then prints the ratio of the total running times.

```
% java SortCompare Insertion Selection 1000 100
For 1000 random Doubles
Insertion is 1.7 times faster than Selection
```

on your computer to learn the extent to which its conclusion about insertion sort and selection sort is robust.

PROPERTY D is intentionally a bit vague—the value of the small constant factor is left unstated and the assumption that the costs of compares and exchanges are similar is left unstated—so that it can apply in a broad variety of situations. When possible, we try to capture essential aspects of the performance of each of the algorithms that we study in statements like this. As discussed in CHAPTER 1, each Property that we consider needs to be tested scientifically in a given situation, perhaps supplemented with a more refined hypothesis based upon a related Proposition (mathematical truth).

For practical applications, there is one further step, which is crucial: run experiments *to validate the hypothesis on the data at hand*. We defer consideration of this step to SECTION 2.5 and the exercises. In this case, if your sort keys are not distinct and/or not randomly ordered, PROPERTY D might not hold. You can randomly order an array with `StdRandom.shuffle()`, but applications with significant numbers of equal keys involve more careful analysis.

Our discussions of the analyses of algorithms are intended to be starting points, not final conclusions. If some other question about performance of the algorithms comes to mind, you can study it with a tool like `SortCompare`. Many opportunities to do so are presented in the exercises.

WE DO NOT DWELL further on the comparative performance of insertion sort and selection sort because we are much more interested in algorithms that can run a hundred or a thousand or a million times faster than either. Still, understanding these elementary algorithms is worthwhile for several reasons:

- They help us work out the ground rules.
- They provide performance benchmarks.
- They often are the method of choice in some specialized situations.
- They can serve as the basis for developing better algorithms.

For these reasons, we will use the same basic approach and consider elementary algorithms for every problem that we study throughout this book, not just sorting. Programs like `SortCompare` play a critical role in this incremental approach to algorithm development. At every step along the way, we can use such a program to help evaluate whether a new algorithm or an improved version of a known algorithm provides the performance gains that we expect.

Shellsort To exhibit the value of knowing properties of elementary sorts, we next consider a fast algorithm based on insertion sort. Insertion sort is slow for large unordered arrays because the only exchanges it does involve adjacent entries, so items can move through the array only one place at a time. For example, if the item with the smallest key happens to be at the end of the array, $N-1$ exchanges are needed to get that one item where it belongs. *Shellsort* is a simple extension of insertion sort that gains speed by allowing exchanges of array entries that are far apart, to produce partially sorted arrays that can be efficiently sorted, eventually by insertion sort.

The idea is to rearrange the array to give it the property that taking every h th entry (starting anywhere) yields a sorted subsequence. Such an array is said to be h -sorted. Put

$h = 4$

L E E A M H L E P S O L T S X R
 L ————— M ————— P ————— T
 E ————— H ————— S ————— S
 E ————— L ————— O ————— X
 A ————— E ————— L ————— R

An h -sorted sequence is h interleaved sorted subsequences

another way, an h -sorted array is h independent sorted subsequences, interleaved together. By h -sorting for some large values of h , we can move items in the array long distances and thus make it easier to h -sort for smaller values of h . Using such a procedure for any sequence of values of h that ends in 1 will produce a sorted array: that is shellsort. The implementation

in ALGORITHM 2.3 on the facing page uses the sequence of decreasing values $\frac{1}{2}(3^k-1)$, starting at the smallest increment greater than or equal to $\lceil N/3 \rceil$ and decreasing to 1. We refer to such a sequence as an *increment sequence*. ALGORITHM 2.3 computes its increment sequence; another alternative is to store an increment sequence in an array.

One way to implement shellsort would be, for each h , to use insertion sort independently on each of the h subsequences. Because the subsequences are independent, we can use an even simpler approach: when h -sorting the array, we insert each item among the previous items in its h -subsequence by exchanging it with those that have larger keys (moving them each one position to the right in the subsequence). We accomplish this task by using the insertion-sort code, but modified to decrement by h instead of 1 when moving through the array. This observation reduces the shellsort implementation to an insertion-sort-like pass through the array for each increment.

Shellsort gains efficiency by making a tradeoff between size and partial order in the subsequences. At the beginning, the subsequences are short; later in the sort, the subsequences are partially sorted. In both cases, insertion sort is the method of choice. The extent to which the subsequences are partially sorted is a variable factor that depends strongly on the increment sequence. Understanding shellsort's performance is a challenge. Indeed, ALGORITHM 2.3 is the only sorting method we consider whose performance on randomly ordered arrays has not been precisely characterized.

ALGORITHM 2.3 Shellsort

```

public class Shell
{
    public static void sort(Comparable[] a)
    { // Sort a[] into increasing order.
        int N = a.length;
        int h = 1;
        while (h < N/3) h = 3*h + 1; // 1, 4, 13, 40, 121, 364, 1093, ...
        while (h >= 1)
        { // h-sort the array.
            for (int i = h; i < N; i++)
            { // Insert a[i] among a[i-h], a[i-2*h], a[i-3*h]...
                for (int j = i; j >= h && less(a[j], a[j-h]); j -= h)
                    exch(a, j, j-h);
            }
            h = h/3;
        }
    }

    // See page 245 for less(), exch(), isSorted(), and main().
}

```

If we modify insertion sort (ALGORITHM 2.2) to h-sort the array and add an outer loop to decrease h through a sequence of increments starting at an increment as large as a constant fraction of the array length and ending at 1, we are led to this compact shellsort implementation.

```

% java SortCompare Shell Insertion 100000 100
For 100000 random Doubles
Shell is 600 times faster than Insertion

```

```

input  S H E L L S O R T E X A M P L E
13-sort P H E L L S O R T E X A M S L E
4-sort  L E E A M H L E P S O L T S X R
1-sort  A E E E H L L L M O P R S S T X

```

Shellsort trace (array contents after each pass)

How do we decide what increment sequence to use? In general, this question is a difficult one to answer. The performance of the algorithm depends not just on the number of increments, but also on arithmetical interactions among the increments such as

```

input  S H E L L S O R T E X A M P L E
13-sort P H E L L S O R T E X A M S L E
      P H E L L S O R T E X A M S L E
4-sort L H E L P S O R T E X A M S L E
      L H E L P S O R T E X A M S L E
      L H E L P S O R T E X A M S L E
      L H E L P S O R T E X A M S L E
      L H E L P S O R T E X A M S L E
      L E E L P H O R T S X A M S L E
      L E E A P H O L T S X R M S L E
      L E E A M H O L P S X R T S L E
      L E E A M H L L P S O R T S X E
      L E E A M H L E P S O L T S X R
1-sort E L E A M H L E P S O L T S X R
      E E L A M H L E P S O L T S X R
      A E E L M H L E P S O L T S X R
      A E E L M H L E P S O L T S X R
      A E E H L M L E P S O L T S X R
      A E E H L L M E P S O L T S X R
      A E E E H L L M P S O L T S X R
      A E E E H L L M P S O L T S X R
      A E E E H L L M O P S L T S X R
      A E E E H L L L M O P S T S X R
      A E E E H L L L M O P S S T X R
      A E E E H L L L M O P R S S T X
result A E E E H L L L M O P R S S T X

```

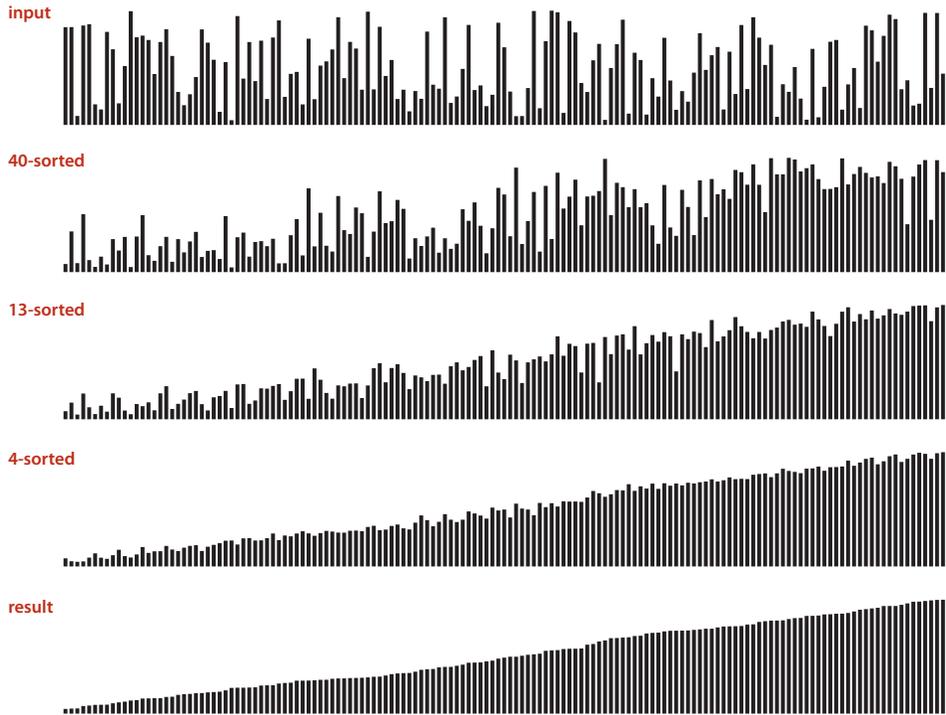
Detailed trace of shellsort (insertions)

size. Before reading further, try using `SortCompare` to compare shellsort with insertion sort and selection sort for array sizes that are increasing powers of 2 on your computer (see EXERCISE 2.1.27). You will see that shellsort makes it possible to address sorting

the size of their common divisors and other properties. Many different increment sequences have been studied in the literature, but no provably best sequence has been found. The increment sequence that is used in ALGORITHM 2.3 is easy to compute and use, and performs nearly as well as more sophisticated increment sequences that have been discovered that have provably better worst-case performance. Increment sequences that are substantially better still may be waiting to be discovered.

Shellsort is useful even for large arrays, particularly by contrast with selection sort and insertion sort. It also performs well on arrays that are in arbitrary order (not necessarily random). Indeed, constructing an array for which shellsort runs slowly for a particular increment sequence is usually a challenging exercise.

As you can learn with `SortCompare`, shellsort is much faster than insertion sort and selection sort, and its speed advantage increases with the array



Visual trace of shellsort

problems that could not be addressed with the more elementary algorithms. This example is our first practical illustration of an important principle that pervades this book: *achieving speedups that enable the solution of problems that could not otherwise be solved is one of the prime reasons to study algorithm performance and design.*

The study of the performance characteristics of shellsort requires mathematical arguments that are beyond the scope of this book. If you want to be convinced, start by thinking about how you would prove the following fact: *when an h -sorted array is k -sorted, it remains h -sorted.* As for the performance of ALGORITHM 2.3, the most important result in the present context is the knowledge that *the running time of shellsort is not necessarily quadratic*—for example, it is known that the worst-case number of compares for ALGORITHM 2.3 is proportional to $N^{3/2}$. That such a simple modification

to break the quadratic-running-time barrier is quite interesting, as doing so is a prime goal for many algorithm design problems.

No mathematical results are available about the average-case number of compares for shellsort for randomly ordered input. Increment sequences have been devised that drive the asymptotic growth of the worst-case number of compares down to $N^{4/3}$, $N^{5/4}$, $N^{6/5}$, . . . , but many of these results are primarily of academic interest because these functions are hard to distinguish from one another (and from a constant factor of N) for practical values of N .

In practice, you can safely take advantage of the past scientific study of shellsort just by using the increment sequence in ALGORITHM 2.3 (or one of the increment sequences in the exercises at the end of this section, which may improve performance by 20 to 40 percent). Moreover, you can easily validate the following hypothesis:

Property E. The number of compares used by shellsort with the increments 1, 4, 13, 40, 121, 364, . . . is bounded by a small multiple of N times the number of increments used.

Evidence: Instrumenting ALGORITHM 2.3 to count compares and divide by the number of increments used is a straightforward exercise (see EXERCISE 2.1.12). Extensive experiments suggest that the average number of compares per increment might be $N^{1/5}$, but it is quite difficult to discern the growth in that function unless N is huge. This property also seems to be rather insensitive to the input model.

EXPERIENCED PROGRAMMERS sometimes choose shellsort because it has acceptable running time even for moderately large arrays; it requires a small amount of code; and it uses no extra space. In the next few sections, we shall see methods that are more efficient, but they are perhaps only twice as fast (if that much) except for very large N , and they are more complicated. If you need a solution to a sorting problem, and are working in a situation where a system sort may not be available (for example, code destined for hardware or an embedded system), you can safely use shellsort, then determine sometime later whether it will be worthwhile to replace it with a more sophisticated method.

Q&A

Q. Sorting seems like a toy problem. Aren't many of the other things that we do with computers much more interesting?

A. Perhaps, but many of those interesting things are made possible by fast sorting algorithms. You will find many examples in SECTION 2.5 and throughout the rest of the book. Sorting is worth studying now because the problem is easy to understand, and you can appreciate the ingenuity behind the faster algorithms.

Q. Why so many sorting algorithms?

A. One reason is that the performance of many algorithms depends on the input values, so different algorithms might be appropriate for different applications having different kinds of input. For example, insertion sort is the method of choice for partially sorted or tiny arrays. Other constraints, such as space and treatment of equal keys, also come into play. We will revisit this question in SECTION 2.5.

Q. Why bother using the tiny helper methods `less()` and `exch()`?

A. They are basic abstract operations needed by any sort algorithm, and the code is easier to understand in terms of these abstractions. Moreover, they make the code directly portable to other settings. For example, much of the code in ALGORITHMS 2.1 and 2.2 is legal code in several other programming languages. Even in Java, we can use this code as the basis for sorting primitive types (which are not `Comparable`): simply implement `less()` with the code `v < w`.

Q. When I run `SortCompare`, I get different values each time that I run it (and those are different from the values in the book). Why?

A. For starters, you have a different computer from the one we used, not to mention a different operating system, Java runtime, and so forth. All of these differences might lead to slight differences in the machine code for the algorithms. Differences each time that you run it on your computer might be due to other applications that you are running or various other conditions. Running a very large number of trials should dampen the effect. The lesson is that small differences in algorithm performance are difficult to notice nowadays. That is a primary reason that we focus on large ones!

EXERCISES

- 2.1.1** Show, in the style of the example trace with ALGORITHM 2.1, how selection sort sorts the array E A S Y Q U E S T I O N.
- 2.1.2** What is the maximum number of exchanges involving any particular item during selection sort? What is the average number of exchanges involving an item?
- 2.1.3** Give an example of an array of N items that maximizes the number of times the test $a[j] < a[\text{min}]$ succeeds (and, therefore, min gets updated) during the operation of selection sort (ALGORITHM 2.1).
- 2.1.4** Show, in the style of the example trace with ALGORITHM 2.2, how insertion sort sorts the array E A S Y Q U E S T I O N.
- 2.1.5** For each of the two conditions in the inner for loop in insertion sort (ALGORITHM 2.2), describe an array of N items where that condition is always false when the loop terminates.
- 2.1.6** Which method runs faster for an array with all keys identical, selection sort or insertion sort?
- 2.1.7** Which method runs faster for an array in reverse order, selection sort or insertion sort?
- 2.1.8** Suppose that we use insertion sort on a randomly ordered array where items have only one of three values. Is the running time linear, quadratic, or something in between?
- 2.1.9** Show, in the style of the example trace with ALGORITHM 2.3, how shellsort sorts the array E A S Y S H E L L S O R T Q U E S T I O N.
- 2.1.10** Why not use selection sort for h -sorting in shellsort?
- 2.1.11** Implement a version of shellsort that keeps the increment sequence in an array, rather than computing it.
- 2.1.12** Instrument shellsort to print the number of compares divided by the array size for each increment. Write a test client that tests the hypothesis that this number is a small constant, by sorting arrays of random `Double` values, using array sizes that are increasing powers of 10, starting at 100.

CREATIVE PROBLEMS

2.1.13 *Deck sort.* Explain how you would put a deck of cards in order by suit (in the order spades, hearts, clubs, diamonds) and by rank within each suit, with the restriction that the cards must be laid out face down in a row, and the only allowed operations are to check the values of two cards and to exchange two cards (keeping them face down).

2.1.14 *Deque sort.* Explain how you would sort a deck of cards, with the restriction that the only allowed operations are to look at the values of the top two cards, to exchange the top two cards, and to move the top card to the bottom of the deck.

2.1.15 *Expensive exchange.* A clerk at a shipping company is charged with the task of rearranging a number of large crates in order of the time they are to be shipped out. Thus, the cost of compares is very low (just look at the labels) relative to the cost of exchanges (move the crates). The warehouse is nearly full—there is extra space sufficient to hold any one of the crates, but not two. What sorting method should the clerk use?

2.1.16 *Certification.* Write a `check()` method that calls `sort()` for a given array and returns `true` if `sort()` puts the array in order and leaves the same set of objects in the array as were there initially, `false` otherwise. Do not assume that `sort()` is restricted to move data only with `exch()`. You may use `Arrays.sort()` and assume that it is correct.

2.1.17 *Animation.* Add code to `Insertion`, `Selection` and `Shell` to make them draw the array contents as vertical bars like the visual traces in this section, redrawing the bars after each pass, to produce an animated effect, ending in a “sorted” picture where the bars appear in order of their height. *Hint:* Use a client like the one in the text that generates random `Double` values, insert calls to `show()` as appropriate in the sort code, and implement a `show()` method that clears the canvas and draws the bars.

2.1.18 *Visual trace.* Modify your solution to the previous exercise to make `Insertion`, `Selection` and `Shell` produce visual traces such as those depicted in this section. *Hint:* Judicious use of `setYscale()` makes this problem easy. *Extra credit:* Add the code necessary to produce red and gray color accents such as those in our figures.

2.1.19 *Shellsort worst case.* Construct an array of 100 elements containing the numbers 1 through 100 for which shellsort, with the increments 1 4 13 40, uses as large a number of compares as you can find.

2.1.20 *Shellsort best case.* What is the best case for shellsort? Justify your answer.

CREATIVE PROBLEMS *(continued)*

2.1.21 *Comparable transactions.* Using our code for `Date` (page 247) as a model, expand your implementation of `Transaction` (EXERCISE 1.2.13) so that it implements `Comparable`, such that transactions are kept in order by amount.

Solution:

```
public class Transaction implements Comparable<Transaction>
{
    ...
    private final double amount;
    ...
    public int compareTo(Transaction that)
    {
        if (this.amount > that.amount) return +1;
        if (this.amount < that.amount) return -1;
        return 0;
    }
    ...
}
```

2.1.22 *Transaction sort test client.* Write a class `SortTransactions` that consists of a static method `main()` that reads a sequence of transactions from standard input, sorts them, and prints the result on standard output (see EXERCISE 1.3.17).

Solution:

```
public class SortTransactions
{
    public static Transaction[] readTransactions()
    { /* See Exercise 1.3.17 */ }

    public static void main(String[] args)
    {
        Transaction[] transactions = readTransactions();
        Shell.sort(transactions);
        for (Transaction t : transactions)
            StdOut.println(t);
    }
}
```

EXPERIMENTS

2.1.23 *Deck sort.* Ask a few friends to sort a deck of cards (see EXERCISE 2.1.13). Observe them carefully and write down the method(s) that they use.

2.1.24 *Insertion sort with sentinel.* Develop an implementation of insertion sort that eliminates the $j > 0$ test in the inner loop by first putting the smallest item into position. Use `SortCompare` to evaluate the effectiveness of doing so. Note: It is often possible to avoid an index-out-of-bounds test in this way—the element that enables the test to be eliminated is known as a sentinel.

2.1.25 *Insertion sort without exchanges.* Develop an implementation of insertion sort that moves larger elements to the right one position with one array access per entry, rather than using `exch()`. Use `SortCompare` to evaluate the effectiveness of doing so.

2.1.26 *Primitive types.* Develop a version of insertion sort that sorts arrays of `int` values and compare its performance with the implementation given in the text (which sorts `Integer` values and implicitly uses autoboxing and auto-unboxing to convert).

2.1.27 *Shellsort is subquadratic.* Use `SortCompare` to compare shellsort with insertion sort and selection sort on your computer. Use array sizes that are increasing powers of 2, starting at 128.

2.1.28 *Equal keys.* Formulate and validate hypotheses about the running time of insertion sort and selection sort for arrays that contain just two key values, assuming that the values are equally likely to occur.

2.1.29 *Shellsort increments.* Run experiments to compare the increment sequence in ALGORITHM 2.3 with the sequence 1, 5, 19, 41, 109, 209, 505, 929, 2161, 3905, 8929, 16001, 36289, 64769, 146305, 260609 (which is formed by merging together the sequences $9 \cdot 4^k - 9 \cdot 2^k + 1$ and $4^k - 3 \cdot 2^k + 1$). See EXERCISE 2.1.11.

2.1.30 *Geometric increments.* Run experiments to determine a value of t that leads to the lowest running time of shellsort for random arrays for the increment sequence 1, $\lfloor t \rfloor$, $\lfloor t^2 \rfloor$, $\lfloor t^3 \rfloor$, $\lfloor t^4 \rfloor$, ... for $N = 10^6$. Give the values of t and the increment sequences for the best three values that you find.

EXPERIMENTS *(continued)*

The following exercises describe various clients for helping to evaluate sorting methods. They are intended as starting points for helping to understand performance properties, using random data. In all of them, use `time()`, as in `SortCompare`, so that you can get more accurate results by specifying more trials in the second command-line argument. We refer back to these exercises in later sections when evaluating more sophisticated methods.

2.1.31 *Doubling test.* Write a client that performs a doubling test for sort algorithms. Start at N equal to 1000, and print N , the predicted number of seconds, the actual number of seconds, and the ratio as N doubles. Use your program to validate that insertion sort and selection sort are quadratic for random inputs, and formulate and test a hypothesis for shellsort.

2.1.32 *Plot running times.* Write a client that uses `StdDraw` to plot the average running times of the algorithm for random inputs and various values of the array size. You may add one or two more command-line arguments. Strive to design a useful tool.

2.1.33 *Distribution.* Write a client that enters into an infinite loop running `sort()` on arrays of the size given as the third command-line argument, measures the time taken for each run, and uses `StdDraw` to plot the average running times. A picture of the distribution of the running times should emerge.

2.1.34 *Corner cases.* Write a client that runs `sort()` on difficult or pathological cases that might turn up in practical applications. Examples include arrays that are already in order, arrays in reverse order, arrays where all keys are the same, arrays consisting of only two distinct values, and arrays of size 0 or 1.

2.1.35 *Nonuniform distributions.* Write a client that generates test data by randomly ordering objects using other distributions than uniform, including the following:

- Gaussian
- Poisson
- Geometric
- Discrete (see EXERCISE 2.1.28 for a special case)

Develop and test hypotheses about the effect of such input on the performance of the algorithms in this section.

2.1.36 *Nonuniform data.* Write a client that generates test data that is not uniform, including the following:

- Half the data is 0s, half 1s.
- Half the data is 0s, half the remainder is 1s, half the remainder is 2s, and so forth.
- Half the data is 0s, half random `int` values.

Develop and test hypotheses about the effect of such input on the performance of the algorithms in this section.

2.1.37 *Partially sorted.* Write a client that generates partially sorted arrays, including the following:

- 95 percent sorted, last percent random values
- All entries within 10 positions of their final place in the array
- Sorted except for 5 percent of the entries randomly dispersed throughout the array

Develop and test hypotheses about the effect of such input on the performance of the algorithms in this section.

2.1.38 *Various types of items.* Write a client that generates arrays of items of various types with random key values, including the following:

- `String` key (at least ten characters), one `double` value
- `double` key, ten `String` values (all at least ten characters)
- `int` key, one `int[20]` value

Develop and test hypotheses about the effect of such input on the performance of the algorithms in this section.

2.2 MERGESORT

THE ALGORITHMS that we consider in this section are based on a simple operation known as merging: combining two ordered arrays to make one larger ordered array. This operation immediately leads to a simple recursive sort method known as mergesort: to sort an array, divide it into two halves, sort the two halves (recursively), and then merge the results. As you will see, one of mergesort's most attractive properties is that it guarantees to sort any array of N items in time proportional to $N \log N$. Its prime disadvantage is that it uses extra space proportional to N .

input	M	E	R	G	E	S	O	R	T	E	X	A	M	P	L	E	
sort left half	E	E	G	M	O	R	R	S		T	E	X	A	M	P	L	E
sort right half	E	E	G	M	O	R	R	S		A	E	E	L	M	P	T	X
merge results	A	E	E	E	E	G	L	M	M	O	P	R	R	S	T	X	

Mergesort overview

Abstract in-place merge The straightforward approach to implementing merging is to design a method that merges two disjoint ordered arrays of Comparable objects into a third array. This strategy is easy to implement: create an output array of the requisite size and then choose successively the smallest remaining item from the two input arrays to be the next item added to the output array.

However, when we mergesort a large array, we are doing a huge number of merges, so the cost of creating a new array to hold the output every time that we do a merge is problematic. It would be much more desirable to have an in-place method so that we could sort the first half of the array in place, then sort the second half of the array in place, then do the merge of the two halves by moving the items around within the array, without using a significant amount of other extra space. It is worthwhile to pause momentarily to consider how you might do that. At first blush, this problem seems to be one that must be simple to solve, but solutions that are known are quite complicated, especially by comparison to alternatives that use extra space.

Still, the abstraction of an in-place merge is useful. Accordingly, we use the method signature `merge(a, lo, mid, hi)` to specify a merge method that puts the result of merging the subarrays `a[lo..mid]` with `a[mid+1..hi]` into a single ordered array, leaving the result in `a[lo..hi]`. The code on the next page implements this merge method in just a few lines by copying everything to an auxiliary array and then merging back to the original. Another approach is described in EXERCISE 2.2.9.

Abstract in-place merge

```

public static void merge(Comparable[] a, int lo, int mid, int hi)
{ // Merge a[lo..mid] with a[mid+1..hi].
  int i = lo, j = mid+1;

  for (int k = lo; k <= hi; k++) // Copy a[lo..hi] to aux[lo..hi].
    aux[k] = a[k];

  for (int k = lo; k <= hi; k++) // Merge back to a[lo..hi].
    if (i > mid)                a[k] = aux[j++];
    else if (j > hi)            a[k] = aux[i++];
    else if (less(aux[j], aux[i])) a[k] = aux[j++];
    else                        a[k] = aux[i++];
}

```

This method merges by first copying into the auxiliary array `aux[]` then merging back to `a[]`. In the merge (the second for loop), there are four conditions: left half exhausted (take from the right), right half exhausted (take from the left), current key on right less than current key on left (take from the right), and current key on right greater than or equal to current key on left (take from the left).

		a[]												aux[]														
		k	0	1	2	3	4	5	6	7	8	9	i	j	0	1	2	3	4	5	6	7	8	9				
input			E	E	G	M	R	A	C	E	R	T			-	-	-	-	-	-	-	-	-	-	-			
copy			E	E	G	M	R	A	C	E	R	T			E	E	G	M	R	A	C	E	R	T				
													0	5														
	0	A											0	6	E	E	G	M	R	A	C	E	R	T				
	1	A	C										0	7	E	E	G	M	R		C	E	R	T				
	2	A	C	E									1	7	E	E	G	M	R			E	R	T				
	3	A	C	E	E								2	7		E	G	M	R			E	R	T				
	4	A	C	E	E	E							2	8			G	M	R			E	R	T				
	5	A	C	E	E	E	G						3	8			G	M	R				R	T				
	6	A	C	E	E	E	G	M					4	8				M	R					R	T			
	7	A	C	E	E	E	G	M	R					5	8					R						R	T	
	8	A	C	E	E	E	G	M	R	R															R	T		
	9	A	C	E	E	E	G	M	R	R	T		6	10														T
merged result			A	C	E	E	E	G	M	R	R	T																

Abstract in-place merge trace

Top-down mergesort ALGORITHM 2.4 is a recursive mergesort implementation based on this abstract in-place merge. It is one of the best-known examples of the utility of the *divide-and-conquer* paradigm for efficient algorithm design. This recursive code is the basis for an inductive proof that the algorithm sorts the array: if it sorts the two subarrays, it sorts the whole array, by merging together the subarrays.

To understand mergesort, it is worthwhile to consider carefully the dynamics of the method calls, shown in the trace at right. To sort $a[0..15]$, the `sort()` method calls itself to sort $a[0..7]$ then calls itself to sort $a[0..3]$ and $a[0..1]$ before finally doing the first merge of $a[0]$ with $a[1]$ after calling itself to sort $a[0]$ and then $a[1]$ (for brevity, we omit the calls for the base-case 1-entry sorts in the trace). Then the next merge is $a[2]$ with $a[3]$ and then $a[0..1]$ with $a[2..3]$ and so forth. From this trace, we see that the sort code simply provides an organized way to sequence the calls to the `merge()` method. This insight will be useful later in this section.

The recursive code also provides us with the basis for analyzing mergesort's running time. Because mergesort is a prototype of the divide-and-conquer algorithm design paradigm, we will consider this analysis in detail.

```

sort(a, 0, 15)
  sort left half (a, 0, 7)
    sort(a, 0, 3)
      sort(a, 0, 1)
        merge(a, 0, 0, 1)
      sort(a, 2, 3)
        merge(a, 2, 2, 3)
    merge(a, 0, 1, 3)
  sort(a, 4, 7)
    sort(a, 4, 5)
      merge(a, 4, 4, 5)
    sort(a, 6, 7)
      merge(a, 6, 6, 7)
    merge(a, 4, 5, 7)
  merge(a, 0, 3, 7)
  sort right half (a, 8, 15)
    sort(a, 8, 11)
      sort(a, 8, 9)
        merge(a, 8, 8, 9)
      sort(a, 10, 11)
        merge(a, 10, 10, 11)
    merge(a, 8, 9, 11)
  sort(a, 12, 15)
    sort(a, 12, 13)
      merge(a, 12, 12, 13)
    sort(a, 14, 15)
      merge(a, 14, 14, 15)
    merge(a, 12, 13, 15)
  merge(a, 8, 11, 15)
merge results merge(a, 0, 7, 15)

```

Top-down mergesort call trace

Proposition F. Top-down mergesort uses between $\frac{1}{2}N \lg N$ and $N \lg N$ compares to sort any array of length N .

Proof: Let $C(N)$ be the number of compares needed to sort an array of length N . We have $C(0) = C(1) = 0$ and for $N > 0$ we can write a recurrence relationship that directly mirrors the recursive `sort()` method to establish an upper bound:

$$C(N) \leq C(\lceil N/2 \rceil) + C(\lfloor N/2 \rfloor) + N.$$

The first term on the right is the number of compares to sort the left half of the array, the second term is the number of compares to sort the right half, and the third

ALGORITHM 2.4 Top-down mergesort

```

public class Merge
{
    private static Comparable[] aux;    // auxiliary array for merges

    public static void sort(Comparable[] a)
    {
        aux = new Comparable[a.length]; // Allocate space just once.
        sort(a, 0, a.length - 1);
    }

    private static void sort(Comparable[] a, int lo, int hi)
    { // Sort a[lo..hi].
        if (hi <= lo) return;
        int mid = lo + (hi - lo)/2;
        sort(a, lo, mid);    // Sort left half.
        sort(a, mid+1, hi);  // Sort right half.
        merge(a, lo, mid, hi); // Merge results (code on page 271).
    }
}

```

To sort a subarray $a[lo..hi]$ we divide it into two parts: $a[lo..mid]$ and $a[mid+1..hi]$, sort them independently (via recursive calls), and merge the resulting ordered subarrays to produce the result.

	lo	hi	a[]
	0	15	M E R G E S O R T E X A M P L E
merge(a, 0, 0, 1)	0	1	E M R G E S O R T E X A M P L E
merge(a, 2, 2, 3)	2	3	E M G R E S O R T E X A M P L E
merge(a, 0, 1, 3)	0	3	E G M R E S O R T E X A M P L E
merge(a, 4, 4, 5)	4	5	E G M R E S O R T E X A M P L E
merge(a, 6, 6, 7)	6	7	E G M R E S O R T E X A M P L E
merge(a, 4, 5, 7)	4	7	E G M R E O R S T E X A M P L E
merge(a, 0, 3, 7)	0	7	E E G M O R R S T E X A M P L E
merge(a, 8, 8, 9)	8	9	E E G M O R R S E T X A M P L E
merge(a, 10, 10, 11)	10	11	E E G M O R R S E T A X M P L E
merge(a, 8, 9, 11)	8	11	E E G M O R R S A E T X M P L E
merge(a, 12, 12, 13)	12	13	E E G M O R R S A E T X M P L E
merge(a, 14, 14, 15)	14	15	E E G M O R R S A E T X M P E L
merge(a, 12, 13, 15)	12	15	E E G M O R R S A E T X E L M P
merge(a, 8, 11, 15)	8	15	E E G M O R R S A E E L M P T X
merge(a, 0, 7, 15)	0	15	A E E E E G L M M O P R R S T X

Trace of merge results for top-down mergesort

term is the number of compares for the merge. The lower bound

$$C(N) \geq C(\lceil N/2 \rceil) + C(\lfloor N/2 \rfloor) + \lfloor N/2 \rfloor$$

follows because the number of compares for the merge is at least $\lfloor N/2 \rfloor$.

We derive an exact solution to the recurrence when equality holds and N is a power of 2 (say $N = 2^n$). First, since $\lfloor N/2 \rfloor = \lceil N/2 \rceil = 2^{n-1}$, we have

$$C(2^n) = 2C(2^{n-1}) + 2^n.$$

Dividing both sides by 2^n gives

$$C(2^n)/2^n = C(2^{n-1})/2^{n-1} + 1.$$

Applying the same equation to the first term on the right, we have

$$C(2^n)/2^n = C(2^{n-2})/2^{n-2} + 1 + 1.$$

Repeating the previous step $n - 1$ additional times gives

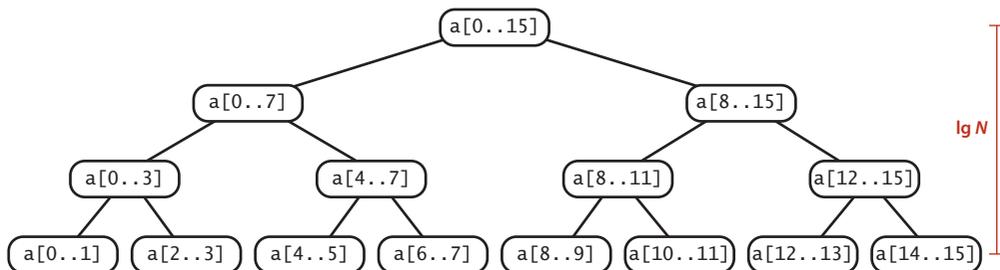
$$C(2^n)/2^n = C(2^0)/2^0 + n.$$

which, after multiplying both sides by 2^n , leaves us with the solution

$$C(N) = C(2^n) = n2^n = N \lg N.$$

Exact solutions for general N are more complicated, but it is not difficult to apply the same argument to the inequalities describing the bounds on the number of compares to prove the stated result for all values of N . This proof is valid no matter what the input values are and no matter in what order they appear.

Another way to understand PROPOSITION F is to examine the tree drawn below, where each node depicts a subarray for which `sort()` does a `merge()`. The tree has precisely n levels. For k from 0 to $n - 1$, the k th level from the top depicts 2^k subarrays, each of length 2^{n-k} , each of which thus requires at most 2^{n-k} compares for the merge. Thus we have $2^k \cdot 2^{n-k} = 2^n$ total cost for each of the n levels, for a total of $n2^n = N \lg N$.



Mergesort subarray dependence tree for $N = 16$

Proposition G. Top-down mergesort uses at most $6N \lg N$ array accesses to sort an array of length N .

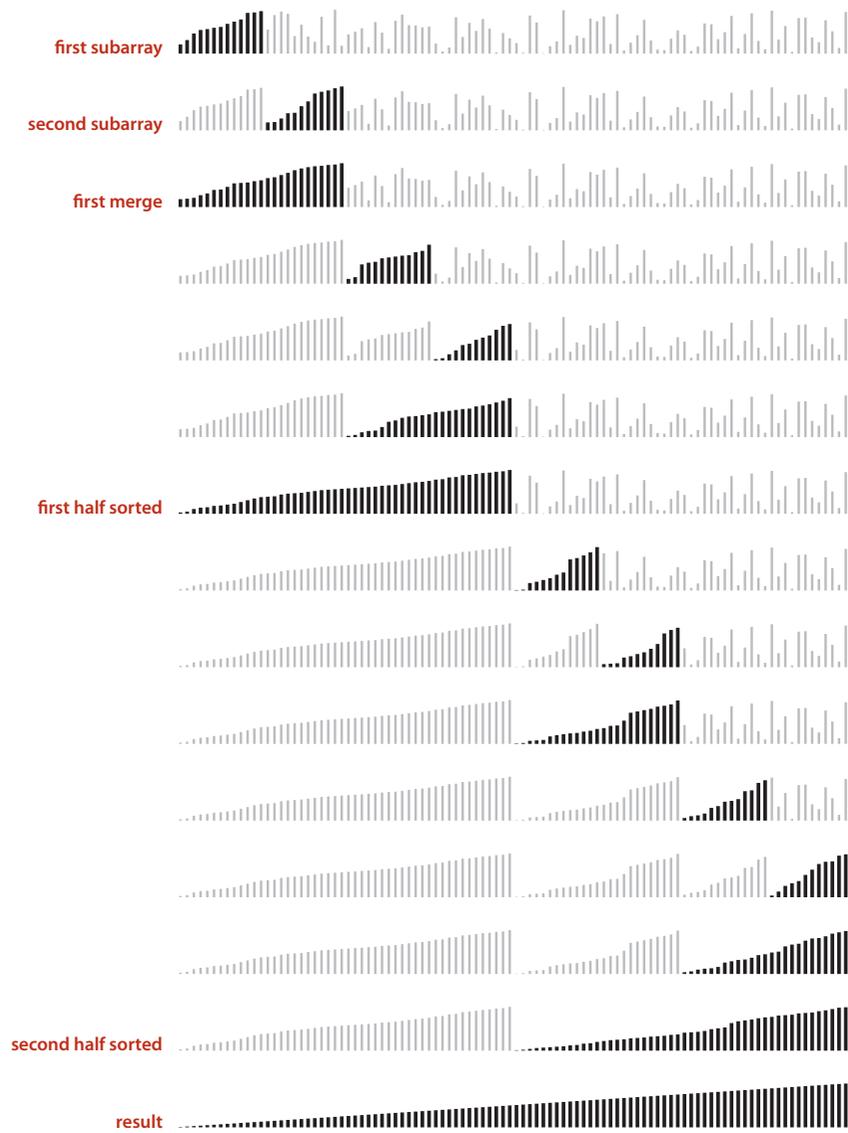
Proof: Each merge uses at most $6N$ array accesses ($2N$ for the copy, $2N$ for the move back, and at most $2N$ for compares). The result follows from the same argument as for PROPOSITION F.

PROPOSITIONS F and G tell us that we can expect the time required by mergesort to be proportional to $N \log N$. That fact brings us to a different level from the elementary methods in SECTION 2.1 because it tells us that we can sort huge arrays using just a logarithmic factor more time than it takes to examine every entry. You can sort millions of items (or more) with mergesort, but not with insertion sort or selection sort. The primary drawback of mergesort is that it requires extra space proportional to N , for the auxiliary array for merging. If space is at a premium, we need to consider another method. On the other hand, we can cut the running time of mergesort substantially with some carefully considered modifications to the implementation.

Use insertion sort for small subarrays. We can improve most recursive algorithms by handling small cases differently, because the recursion guarantees that the method will be used often for small cases, so improvements in handling them lead to improvements in the whole algorithm. In the case of sorting, we know that insertion sort (or selection sort) is simple and therefore likely to be faster than mergesort for tiny subarrays. As usual, a visual trace provides insight into the operation of mergesort. The visual trace on the next page shows the operation of a mergesort implementation with a cutoff for small subarrays. Switching to insertion sort for small subarrays (length 15 or less, say) will improve the running time of a typical mergesort implementation by 10 to 15 percent (see EXERCISE 2.2.23).

Test whether the array is already in order. We can reduce the running time to be linear for arrays that are already in order by adding a test to skip the call to `merge()` if `a[mid]` is less than or equal to `a[mid+1]`. With this change, we still do all the recursive calls, but the running time for any sorted subarray is linear (see EXERCISE 2.2.8).

Eliminate the copy to the auxiliary array. It is possible to eliminate the time (but not the space) taken to copy to the auxiliary array used for merging. To do so, we use two invocations of the sort method: one takes its input from the given array and puts the sorted output in the auxiliary array; the other takes its input from the auxiliary array and puts the sorted output in the given array. With this approach, in a bit of recursive trickery, we can arrange the recursive calls such that the computation switches the roles of the input array and the auxiliary array at each level (see EXERCISE 2.2.11).

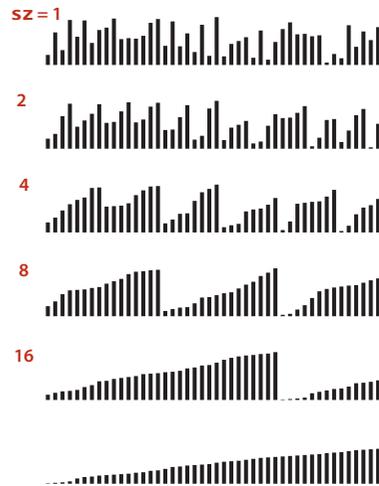


Visual trace of top-down mergesort with cutoff for small subarrays

It is appropriate to repeat here a point raised in CHAPTER 1 that is easily forgotten and needs reemphasis. Locally, we treat each algorithm in this book as if it were critical in some application. Globally, we try to reach general conclusions about which approach to recommend. Our discussion of such improvements is not necessarily a recommendation to always implement them, rather a warning not to draw absolute conclusions about performance from initial implementations. When addressing a new problem, your best bet is to use the simplest implementation with which you are comfortable and then refine it if it becomes a bottleneck. Addressing improvements that decrease running time just by a constant factor may not otherwise be worthwhile. You need to test the effectiveness of specific improvements by running experiments, as we indicate in exercises throughout.

In the case of mergesort, the three improvements just listed are simple to implement and are of interest when mergesort is the method of choice—for example, in situations discussed at the end of this chapter.

Bottom-up mergesort The recursive implementation of mergesort is prototypical of the *divide-and-conquer* algorithm design paradigm, where we solve a large problem by dividing it into pieces, solving the subproblems, then using the solutions for the pieces to solve the whole problem. Even though we are thinking in terms of merging together two large subarrays, the fact is that most merges are merging together tiny subarrays. Another way to implement mergesort is to organize the merges so that we do all the merges of tiny subarrays on one pass, then do a second pass to merge those subarrays in pairs, and so forth, continuing until we do a merge that encompasses the whole array. This method requires even less code than the standard recursive implementation. We start by doing a pass of 1-by-1 merges (considering individual items as subarrays of size 1), then a pass of 2-by-2 merges (merge subarrays of size 2 to make subarrays of size 4), then 4-by-4 merges, and so forth. The second subarray may be smaller than the first in the last merge on each pass (which is no problem for `merge()`), but otherwise all merges involve subarrays of equal size, doubling the sorted subarray size for the next pass.



Visual trace of bottom-up mergesort

Bottom-up mergesort

```

public class MergeBU
{
    private static Comparable[] aux;          // auxiliary array for merges
    // See page 271 for merge() code.

    public static void sort(Comparable[] a)
    { // Do lg N passes of pairwise merges.
        int N = a.length;
        aux = new Comparable[N];
        for (int sz = 1; sz < N; sz = sz+sz) // sz: subarray size
            for (int lo = 0; lo < N-sz; lo += sz+sz) // lo: subarray index
                merge(a, lo, lo+sz-1, Math.min(lo+sz+sz-1, N-1));
    }
}

```

Bottom-up mergesort consists of a sequence of passes over the whole array, doing *sz*-by-*sz* merges, starting with *sz* equal to 1 and doubling *sz* on each pass. The final subarray is of size *sz* only when the array size is an even multiple of *sz* (otherwise it is less than *sz*).

	a[i]															
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
sz=1	M	E	R	G	E	S	O	R	T	E	X	A	M	P	L	E
merge(a, 0, 0, 1)	E	M	R	G	E	S	O	R	T	E	X	A	M	P	L	E
merge(a, 2, 2, 3)	E	M	G	R	E	S	O	R	T	E	X	A	M	P	L	E
merge(a, 4, 4, 5)	E	M	G	R	E	S	O	R	T	E	X	A	M	P	L	E
merge(a, 6, 6, 7)	E	M	G	R	E	S	O	R	T	E	X	A	M	P	L	E
merge(a, 8, 8, 9)	E	M	G	R	E	S	O	R	E	T	X	A	M	P	L	E
merge(a, 10, 10, 11)	E	M	G	R	E	S	O	R	E	T	A	X	M	P	L	E
merge(a, 12, 12, 13)	E	M	G	R	E	S	O	R	E	T	A	X	M	P	L	E
merge(a, 14, 14, 15)	E	M	G	R	E	S	O	R	E	T	A	X	M	P	E	L
sz=2	E	G	M	R	E	S	O	R	E	T	A	X	M	P	E	L
merge(a, 0, 1, 3)	E	G	M	R	E	O	R	S	E	T	A	X	M	P	E	L
merge(a, 4, 5, 7)	E	G	M	R	E	O	R	S	A	E	T	X	M	P	E	L
merge(a, 8, 9, 11)	E	G	M	R	E	O	R	S	A	E	T	X	E	L	M	P
merge(a, 12, 13, 15)	E	G	M	R	E	O	R	S	A	E	T	X	E	L	M	P
sz=4	E	E	G	M	O	R	R	S	A	E	T	X	E	L	M	P
merge(a, 0, 3, 7)	E	E	G	M	O	R	R	S	A	E	E	L	M	P	T	X
merge(a, 8, 11, 15)	E	E	G	M	O	R	R	S	A	E	E	L	M	P	T	X
sz=8	A	E	E	E	E	G	L	M	M	O	P	R	R	S	T	X
merge(a, 0, 7, 15)	A	E	E	E	E	G	L	M	M	O	P	R	R	S	T	X

Trace of merge results for bottom-up mergesort

Proposition H. Bottom-up mergesort uses between $\frac{1}{2}N \lg N$ and $N \lg N$ compares and at most $6N \lg N$ array accesses to sort an array of length N .

Proof: The number of passes through the array is precisely $\lceil \lg N \rceil$ (that is precisely the value of n such that $2^{n-1} < N \leq 2^n$). For each pass, the number of array accesses is exactly $6N$ and the number of compares is at most N and no less than $N/2$.

WHEN THE ARRAY LENGTH IS A POWER OF 2, top-down and bottom-up mergesort perform precisely the same compares and array accesses, just in a different order. When the array length is not a power of 2, the sequence of compares and array accesses for the two algorithms will be different (see EXERCISE 2.2.5).

A version of bottom-up mergesort is the method of choice for sorting data organized in a linked list. Consider the list to be sorted sublists of size 1, then pass through to make sorted sublists of size 2 linked together, then size 4, and so forth. This method rearranges the links to sort the list in place (without creating any new list nodes).

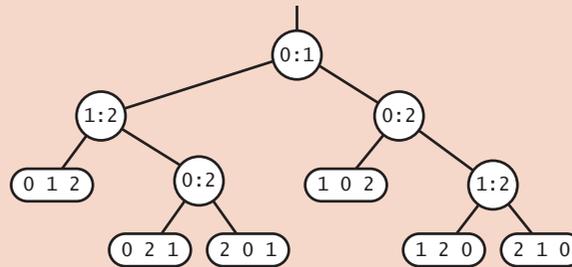
Both the top-down and bottom-up approaches to implementing a divide-and-conquer algorithm are intuitive. The lesson that you can take from mergesort is this: Whenever you encounter an algorithm based on one of these approaches, it is worth considering the other. Do you want to solve the problem by breaking it up into smaller problems (and solving them recursively) as in `Merge.sort()` or by building small solutions into larger ones as in `MergeBU.sort()`?

The complexity of sorting One important reason to know about mergesort is that we use it as the basis for proving a fundamental result in the field of computational complexity that helps us understand the intrinsic difficulty of sorting. In general, computational complexity plays an important role in the design of algorithms, and this result in particular is directly relevant to the design of sorting algorithms, so we next consider it in detail.

The first step in a study of complexity is to establish a model of computation. Generally, researchers strive to understand the simplest model relevant to a problem. For sorting, we study the class of compare-based algorithms that make their decisions about items only on the basis of comparing keys. A compare-based algorithm can do an arbitrary amount of computation between compares, but cannot get any information about a key except by comparing it with another one. Because of our restriction to the `Comparable` API, all of the algorithms in this chapter are in this class (note that we are ignoring the cost of array accesses), as are many algorithms that we might imagine. In CHAPTER 5, we consider algorithms that are not restricted to `Comparable` items.

Proposition I. No compare-based sorting algorithm can guarantee to sort N items with fewer than $\lg(N!) \sim N \lg N$ compares.

Proof: First, we assume that the keys are all distinct, since any algorithm must be able to sort such inputs. Now, we use a binary tree to describe the sequence of compares. Each node in the tree is either a leaf $(i_0 \ i_1 \ i_2 \ \dots \ i_{N-1})$ that indicates that the sort is complete and has discovered that the original inputs were in the order $a[i_0], a[i_1], \dots, a[i_{N-1}]$, or an internal node $(i:j)$ that corresponds to a compare operation between $a[i]$ and $a[j]$, with a left subtree corresponding to the sequence of compares in the case that $a[i]$ is less than $a[j]$, and a right subtree corresponding to what happens if $a[i]$ is greater than $a[j]$. Each path from the root to a leaf corresponds to the sequence of compares that the algorithm uses to establish the ordering given in the leaf. For example, here is a compare tree for $N = 3$:



We never explicitly construct such a tree—it is a mathematical device for describing the compares used by any algorithm.

The first key observation in the proof is that the tree must have at least $N!$ leaves because there are $N!$ different permutations of N distinct keys. If there are fewer than $N!$ leaves, then some permutation is missing from the leaves, and the algorithm would fail for that permutation.

The number of internal nodes on a path from the root to a leaf in the tree is the number of compares used by the algorithm for some input. We are interested in the length of the longest such path in the tree (known as the tree *height*) since it measures the worst-case number of compares used by the algorithm. Now, it is a basic combinatorial property of binary trees that a tree of height h has no more than 2^h leaves—the tree of height h with the maximum number of leaves is perfectly balanced, or complete. An example for $h = 4$ is diagrammed on the next page.

complete tree of height 4 (gray) has $2^4 = 16$ leaves

any other tree of height 4 (black) has fewer than 16 leaves

Combining the previous two paragraphs, we have shown that any compare-based sorting algorithm corresponds to a compare tree of height h with

$$N! \leq \text{number of leaves} \leq 2^h$$

at least $N!$ leaves

no more than 2^h leaves

The value of h is precisely the worst-case number of compares, so we can take the logarithm (base 2) of both sides of this equation and conclude that the number of compares used by any algorithm must be at least $\lg(N!)$. The approximation $\lg(N!) \sim N \lg N$ follows immediately from Stirling's approximation to the factorial function (see page 185).

This result serves as a guide for us to know, when designing a sorting algorithm, how well we can expect to do. For example, without such a result, one might set out to try to design a compare-based sorting algorithm that uses half as many compares as does mergesort, in the worst case. The lower bound in PROPOSITION I says that such an effort is futile—*no such algorithm exists*. It is an extremely strong statement that applies to any conceivable compare-based algorithm.

PROPOSITION H asserts that the number of compares used by mergesort in the worst case is $\sim N \lg N$. This result is an upper bound on the difficulty of the sorting problem in the sense that a better algorithm would have to guarantee to use a smaller number of compares. PROPOSITION I asserts that no sorting algorithm can guarantee to use fewer

than $\sim N \lg N$ compares. It is a lower bound on the difficulty of the sorting problem in the sense that even the best possible algorithm must use at least that many compares in the worst case. Together, they imply:

Proposition J. Mergesort is an asymptotically optimal compare-based sorting algorithm.

Proof: Precisely, we mean by this statement that *both the number of compares used by mergesort in the worst case and the minimum number of compares that any compare-based sorting algorithm can guarantee are $\sim N \lg N$* . PROPOSITIONS H and I establish these facts.

It is important to note that, like the model of computation, we need to precisely define what we mean by an optimal algorithm. For example, we might tighten the definition of optimality and insist that an optimal algorithm for sorting is one that uses precisely $\lg(N!)$ compares. We do not do so because we could not notice the difference between such an algorithm and (for example) mergesort for large N . Or, we might broaden the definition of optimality to include any sorting algorithm whose worst-case number of compares is *within a constant factor* of $N \lg N$. We do not do so because we might very well notice the difference between such an algorithm and mergesort for large N .

COMPUTATIONAL COMPLEXITY MAY SEEM RATHER ABSTRACT, but fundamental research on the intrinsic difficulty of solving computational problems hardly needs justification. Moreover, when it does apply, it is emphatically the case that computational complexity affects the development of good software. First, good upper bounds allow software engineers to provide performance guarantees; there are many documented instances where poor performance has been traced to someone using a quadratic sort instead of a linearithmic one. Second, good lower bounds spare us the effort of searching for performance improvements that are not attainable.

But the optimality of mergesort is not the end of the story and should not be misused to indicate that we need not consider other methods for practical applications. That is not the case because the theory in this section has a number of limitations. For example:

- Mergesort is not optimal with respect to space usage.
- The worst case may not be likely in practice.
- Operations other than compares (such as array accesses) may be important.
- One can sort certain data without using any compares.

Thus, we shall be considering several other sorting methods in this book.

Q&A

Q. Is mergesort faster than shellsort?

A. In practice, their running times are within a small constant factor of one another (when shellsort is using a well-tested increment sequence like the one in ALGORITHM 2.3), so comparative performance depends on the implementations.

```
% java SortCompare Merge Shell 100000
For 100000 random Double values
Merge is 1.2 times faster than Shell
```

In theory, no one has been able to prove that shellsort is linearithmic for random data, so there remains the possibility that the asymptotic growth of the average-case performance of shellsort is higher. Such a gap has been proven for worst-case performance, but it is not relevant in practice.

Q. Why not make the `aux[]` array local to `merge()`?

A. To avoid the overhead of creating an array for every merge, even the tiny ones. This cost would dominate the running time of mergesort (see EXERCISE 2.2.26). A more proper solution (which we avoid in the text to reduce clutter in the code) is to make `aux[]` local to `sort()` and pass it as an argument to `merge()` (see EXERCISE 2.2.9).

Q. How does mergesort fare when there are duplicate values in the array?

A. If all the items have the same value, the running time is linear (with the extra test to skip the merge when the array is sorted), but if there is more than one duplicate value, this performance gain is not necessarily realized. For example, suppose that the input array consists of N items with one value in odd positions and N items with another value in even positions. The running time is linearithmic for such an array (it satisfies the same recurrence as for items with distinct values), not linear.

EXERCISES

2.2.1 Give a trace, in the style of the trace given at the beginning of this section, showing how the keys A E Q S U Y E I N O S T are merged with the abstract in-place `merge()` method.

2.2.2 Give traces, in the style of the trace given with ALGORITHM 2.4, showing how the keys E A S Y Q U E S T I O N are sorted with top-down mergesort.

2.2.3 Answer EXERCISE 2.2.2 for bottom-up mergesort.

2.2.4 Does the abstract in-place merge produce proper output if and only if the two input subarrays are in sorted order? Prove your answer, or provide a counterexample.

2.2.5 Give the sequence of subarray sizes in the merges performed by both the top-down and the bottom-up mergesort algorithms, for $N = 39$.

2.2.6 Write a program to compute the exact value of the number of array accesses used by top-down mergesort and by bottom-up mergesort. Use your program to plot the values for N from 1 to 512, and to compare the exact values with the upper bound $6N \lg N$.

2.2.7 Show that the number of compares used by mergesort is monotonically increasing ($C(N+1) > C(N)$ for all $N > 0$).

2.2.8 Suppose that ALGORITHM 2.4 is modified to skip the call on `merge()` whenever $a[\text{mid}] \leq a[\text{mid}+1]$. Prove that the number of compares used to mergesort a sorted array is linear.

2.2.9 Use of a static array like `aux[]` is inadvisable in library software because multiple clients might use the class concurrently. Give an implementation of `Merge` that does not use a static array. Do not make `aux[]` local to `merge()` (see the Q&A for this section). *Hint*: Pass the auxiliary array as an argument to the recursive `sort()`.

CREATIVE PROBLEMS

2.2.10 *Faster merge.* Implement a version of `merge()` that copies the second half of `a[]` to `aux[]` in decreasing order and then does the merge back to `a[]`. This change allows you to remove the code to test that each of the halves has been exhausted from the inner loop. Note: The resulting sort is not stable (see page 341).

2.2.11 *Improvements.* Implement the three improvements to mergesort that are described in the text on page 275: Add a cutoff for small subarrays, test whether the array is already in order, and avoid the copy by switching arguments in the recursive code.

2.2.12 *Sublinear extra space.* Develop a merge implementation that reduces the extra space requirement to $\max(M, N/M)$, based on the following idea: Divide the array into N/M blocks of size M (for simplicity in this description, assume that N is a multiple of M). Then, (i) considering the blocks as items with their first key as the sort key, sort them using selection sort; and (ii) run through the array merging the first block with the second, then the second block with the third, and so forth.

2.2.13 *Lower bound for average case.* Prove that the expected number of compares used by any compare-based sorting algorithm must be at least $\sim N \lg N$ (assuming that all possible orderings of the input are equally likely). *Hint:* The expected number of compares is at least the external path length of the compare tree (the sum of the lengths of the paths from the root to all leaves), which is minimized when it is balanced.

2.2.14 *Merging sorted queues.* Develop a static method that takes two queues of sorted items as arguments and returns a queue that results from merging the queues into sorted order.

2.2.15 *Bottom-up queue mergesort.* Develop a bottom-up mergesort implementation based on the following approach: Given N items, create N queues, each containing one of the items. Create a queue of the N queues. Then repeatedly apply the merging operation of EXERCISE 2.2.14 to the first two queues and reinsert the merged queue at the end. Repeat until the queue of queues contains only one queue.

2.2.16 *Natural mergesort.* Write a version of bottom-up mergesort that takes advantage of order in the array by proceeding as follows each time it needs to find two arrays to merge: find a sorted subarray (by incrementing a pointer until finding an entry that is smaller than its predecessor in the array), then find the next, then merge them. Analyze the running time of this algorithm in terms of the array size and the number of

CREATIVE PROBLEMS *(continued)*

maximal increasing sequences in the array.

2.2.17 *Linked-list sort.* Implement a natural mergesort for linked lists. (This is the method of choice for sorting linked lists because it uses no extra space and is guaranteed to be linearithmic.)

2.2.18 *Shuffling a linked list.* Develop and implement a divide-and-conquer algorithm that randomly shuffles a linked list in linearithmic time and logarithmic extra space.

2.2.19 *Inversions.* Develop and implement a linearithmic algorithm for computing the number of inversions in a given array (the number of exchanges that would be performed by insertion sort for that array—see SECTION 2.1). This quantity is related to the Kendall tau distance; see SECTION 2.5.

2.2.20 *Index sort.* Develop and implement a version of mergesort that does not rearrange the array, but returns an `int[]` array `perm` such that `perm[i]` is the index of the *i*th smallest entry in the array.

2.2.21 *Triplicates.* Given three lists of *N* names each, devise a linearithmic algorithm to determine if there is any name common to all three lists, and if so, return the lexicographically first such name.

2.2.22 *3-way mergesort.* Suppose instead of dividing in half at each step, you divide into thirds, sort each third, and combine using a 3-way merge. What is the order of growth of the overall running time of this algorithm?

EXPERIMENTS

2.2.23 *Improvements.* Run empirical studies to evaluate the effectiveness of each of the three improvements to mergesort that are described in the text (see EXERCISE 2.2.11). Also, compare the performance of the merge implementation given in the text with the merge described in EXERCISE 2.2.10. In particular, empirically determine the best value of the parameter that decides when to switch to insertion sort for small subarrays.

2.2.24 *Sort-test improvement.* Run empirical studies for large randomly ordered arrays to study the effectiveness of the modification described in EXERCISE 2.2.8 for random data. In particular, develop a hypothesis about the average number of times the test (whether an array is sorted) succeeds, as a function of N (the original array size for the sort).

2.2.25 *Multiway mergesort.* Develop a mergesort implementation based on the idea of doing k -way merges (rather than 2-way merges). Analyze your algorithm, develop a hypothesis regarding the best value of k , and run experiments to validate your hypothesis.

2.2.26 *Array creation.* Use `SortCompare` to get a rough idea of the effect on performance on your machine of creating `aux[]` in `merge()` rather than in `sort()`.

2.2.27 *Subarray lengths.* Run mergesort for large random arrays, and make an empirical determination of the average length of the other subarray when the first subarray exhausts, as a function of N (the sum of the two subarray sizes for a given merge).

2.2.28 *Top-down versus bottom-up.* Use `SortCompare` to compare top-down and bottom-up mergesort for $N=10^3$, 10^4 , 10^5 , and 10^6 .

2.2.29 *Natural mergesort.* Determine empirically the number of passes needed in a natural mergesort (see EXERCISE 2.2.16) for random `Long` keys with $N=10^3$, 10^6 , and 10^9 . *Hint:* You do not need to implement a sort (or even generate full 64-bit keys) to complete this exercise.

2.3 QUICKSORT

THE SUBJECT OF THIS SECTION is the sorting algorithm that is probably used more widely than any other, *quicksort*. Quicksort is popular because it is not difficult to implement, works well for a variety of different kinds of input data, and is substantially faster than any other sorting method in typical applications. The quicksort algorithm's desirable features are that it is in-place (uses only a small auxiliary stack) and that it requires time proportional to $N \log N$ on the average to sort an array of length N . None of the algorithms that we have so far considered combine these two properties. Furthermore, quicksort has a shorter inner loop than most other sorting algorithms, which means that it is fast in practice as well as in theory. Its primary drawback is that it is fragile in the sense that some care is involved in the implementation to be sure to avoid bad performance. Numerous examples of mistakes leading to quadratic performance in practice are documented in the literature. Fortunately, the lessons learned from these mistakes have led to various improvements to the algorithm that make it of even broader utility, as we shall see.

The basic algorithm Quicksort is a divide-and-conquer method for sorting. It works by partitioning an array into two subarrays, then sorting the subarrays independently. Quicksort is complementary to mergesort: for mergesort, we break the array into two subarrays to be sorted and then combine the ordered subarrays to make the whole ordered array; for quicksort, we rearrange the array such that, when the two subarrays are sorted, the whole array is ordered. In the first instance, we do the two recursive calls before working on the whole array; in the second instance, we do the two recursive calls after working on the whole array. For mergesort, the array is divided in half; for quicksort, the position of the partition depends on the contents of the array.

input	Q	U	I	C	K	S	O	R	T	E	X	A	M	P	L	E
shuffle	K	R	A	T	E	L	E	P	U	I	M	Q	C	X	O	S
partition	E	C	A	I	E	K	L	P	U	T	M	Q	R	X	O	S
sort left	A	C	E	E	I	K	L	P	U	T	M	Q	R	X	O	S
sort right	A	C	E	E	I	K	L	M	O	P	Q	R	S	T	U	X
result	A	C	E	E	I	K	L	M	O	P	Q	R	S	T	U	X

Quicksort overview

ALGORITHM 2.5 Quicksort

```

public class Quick
{
    public static void sort(Comparable[] a)
    {
        StdRandom.shuffle(a);           // Eliminate dependence on input.
        sort(a, 0, a.length - 1);
    }

    private static void sort(Comparable[] a, int lo, int hi)
    {
        if (hi <= lo) return;
        int j = partition(a, lo, hi); // Partition (see page 291).
        sort(a, lo, j-1);             // Sort left part a[lo .. j-1].
        sort(a, j+1, hi);             // Sort right part a[j+1 .. hi].
    }
}

```

Quicksort is a recursive program that sorts a subarray $a[lo..hi]$ by using a `partition()` method that puts $a[j]$ into position and arranges the rest of the entries such that the recursive calls finish the sort.

	lo	j	hi	<u>0</u>	<u>1</u>	<u>2</u>	<u>3</u>	<u>4</u>	<u>5</u>	<u>6</u>	<u>7</u>	<u>8</u>	<u>9</u>	<u>10</u>	<u>11</u>	<u>12</u>	<u>13</u>	<u>14</u>	<u>15</u>
initial values				Q	U	I	C	K	S	O	R	T	E	X	A	M	P	L	E
random shuffle				K	R	A	T	E	L	E	P	U	I	M	Q	C	X	O	S
	0	5	15	E	C	A	I	E	K	L	P	U	T	M	Q	R	X	O	S
	0	3	4	E	C	A	E	I	K	L	P	U	T	M	Q	R	X	O	S
	0	2	2	A	C	E	E	I	K	L	P	U	T	M	Q	R	X	O	S
	0	0	1	A	C	E	E	I	K	L	P	U	T	M	Q	R	X	O	S
	1		1	A	C	E	E	I	K	L	P	U	T	M	Q	R	X	O	S
	4		4	A	C	E	E	I	K	L	P	U	T	M	Q	R	X	O	S
	6	6	15	A	C	E	E	I	K	L	P	U	T	M	Q	R	X	O	S
	7	9	15	A	C	E	E	I	K	L	M	O	P	T	Q	R	X	U	S
	7	7	8	A	C	E	E	I	K	L	M	O	P	T	Q	R	X	U	S
	8		8	A	C	E	E	I	K	L	M	O	P	T	Q	R	X	U	S
	10	13	15	A	C	E	E	I	K	L	M	O	P	S	Q	R	T	U	X
	10	12	12	A	C	E	E	I	K	L	M	O	P	R	Q	S	T	U	X
	10	11	11	A	C	E	E	I	K	L	M	O	P	Q	R	S	T	U	X
	10	10	10	A	C	E	E	I	K	L	M	O	P	Q	R	S	T	U	X
	14	14	15	A	C	E	E	I	K	L	M	O	P	Q	R	S	T	U	X
	15		15	A	C	E	E	I	K	L	M	O	P	Q	R	S	T	U	X
result				A	C	E	E	I	K	L	M	O	P	Q	R	S	T	U	X

The crux of the method is the partitioning process, which rearranges the array to make the following three conditions hold:

- The entry $a[j]$ is in its final place in the array, for some j .
- No entry in $a[l_0]$ through $a[j-1]$ is greater than $a[j]$.
- No entry in $a[j+1]$ through $a[h_i]$ is less than $a[j]$.

We achieve a complete sort by partitioning, then recursively applying the method.

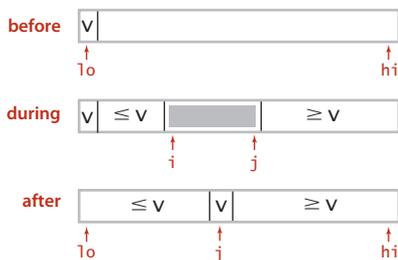
Because the partitioning process always fixes one item into its position, a formal proof by induction that the recursive method constitutes a proper sort is not difficult to develop: if the left subarray and the right subarray are both properly sorted, then the result array, made up of the left subarray (in order, with no entry larger than the partitioning item), the partitioning item, and the right subarray (in order, with no entry smaller than the partitioning item), is in order. ALGORITHM 2.5 is a recursive program

that implements this idea. It is a *randomized* algorithm, because it randomly shuffles the array before sorting it. Our reason for doing so is to be able to predict (and depend upon) its performance characteristics, as discussed below.

To complete the implementation, we need to implement the partitioning method. We use the following general strategy: First, we arbitrarily choose $a[l_0]$ to be the partitioning item—the one that will go into its final position. Next, we scan from the left end of the array until we find an entry greater than (or equal to) the partitioning item, and we scan from the right end of the array until we find an entry less than (or equal to) the partitioning item. The two items that

stopped the scans are out of place in the final partitioned array, so we exchange them. Continuing in this way, we ensure that no array entries to the left of the left index i are greater than the partitioning item, and no array entries to the right of the right index j are less than the partitioning item. When the scan indices cross, all that we need to do to complete the partitioning process is to exchange the partitioning item $a[l_0]$ with the rightmost entry of the left subarray ($a[j]$) and return its index j .

There are several subtle issues with respect to implementing quicksort that are reflected in this code and worthy of mention, because each either can lead to incorrect code or can significantly impact performance. Next, we discuss several of these issues. Later in this section, we will consider three important higher-level algorithmic improvements.



QuickSort partitioning overview

Quicksort partitioning

```
private static int partition(Comparable[] a, int lo, int hi)
{ // Partition into a[lo..i-1], a[i], a[i+1..hi].
  int i = lo, j = hi+1;           // left and right scan indices
  Comparable v = a[lo];          // partitioning item
  while (true)
  { // Scan right, scan left, check for scan complete, and exchange.
    while (less(a[++i], v)) if (i == hi) break;
    while (less(v, a[--j])) if (j == lo) break;
    if (i >= j) break;
    exch(a, i, j);
  }
  exch(a, lo, j);                // Put v = a[j] into position
  return j;                      // with a[lo..j-1] <= a[j] <= a[j+1..hi].
}
```

This code partitions on the item v in $a[lo]$. The main loop exits when the scan indices i and j cross. Within the loop, we increment i while $a[i]$ is less than v and decrement j while $a[j]$ is greater than v , then do an exchange to maintain the invariant property that no entries to the left of i are greater than v and no entries to the right of j are smaller than v . Once the indices meet, we complete the partitioning by exchanging $a[lo]$ with $a[j]$ (thus leaving the partitioning value in $a[j]$).

	i	j	v	a[]															
				0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
initial values	0	16		K	R	A	T	E	L	E	P	U	I	M	Q	C	X	O	S
scan left, scan right	1	12		K	<u>R</u>	A	T	E	L	E	P	U	I	M	Q	<u>C</u>	X	O	S
exchange	1	12		K	C	A	T	E	L	E	P	U	I	M	Q	R	X	O	S
scan left, scan right	3	9		K	C	<u>A</u>	T	E	L	E	P	U	<u>I</u>	M	Q	R	X	O	S
exchange	3	9		K	C	A	I	E	L	E	P	U	T	M	Q	R	X	O	S
scan left, scan right	5	6		K	C	A	I	<u>E</u>	L	<u>E</u>	P	U	T	M	Q	R	X	O	S
exchange	5	6		K	C	A	I	E	E	L	P	U	T	M	Q	R	X	O	S
scan left, scan right	6	5		K	C	A	I	E	<u>E</u>	<u>L</u>	P	U	T	M	Q	R	X	O	S
final exchange	6	5		E	C	A	I	E	K	L	P	U	T	M	Q	R	X	O	S
result		5		E	C	A	I	E	K	L	P	U	T	M	Q	R	X	O	S

Partitioning trace (array contents before and after each exchange)

Partitioning in place. If we use an extra array, partitioning is easy to implement, but not so much easier that it is worth the extra cost of copying the partitioned version back into the original. A novice Java programmer might even create a new spare array within the recursive method, for each partition, which would drastically slow down the sort.

Staying in bounds. If the smallest item or the largest item in the array is the partitioning item, we have to take care that the pointers do not run off the left or right ends of the array, respectively. Our `partition()` implementation has explicit tests to guard against this circumstance. The test `(j == 10)` is redundant, since the partitioning item is at `a[10]` and not less than itself. With a similar technique on the right it is not difficult to eliminate both tests (see EXERCISE 2.3.17).

Preserving randomness. The random shuffle puts the array in random order. Since it treats all items in the subarrays uniformly, ALGORITHM 2.5 has the property that its two subarrays are also in random order. This fact is crucial to the predictability of the algorithm's running time. An alternate way to preserve randomness is to choose a random item for partitioning within `partition()`.

Terminating the loop. Experienced programmers know to take special care to ensure that any loop must always terminate, and the partitioning loop for quicksort is no exception. Properly testing whether the pointers have crossed is a bit trickier than it might seem at first glance. A common error is to fail to take into account that the array might contain other items with the same key value as the partitioning item.

Handling items with keys equal to the partitioning item's key. It is best to stop the left scan for items with keys greater than *or equal to* the partitioning item's key and the right scan for items with key less than *or equal to* the partitioning item's key, as in ALGORITHM 2.5. Even though this policy might seem to create unnecessary exchanges involving items with keys equal to the partitioning item's key, it is crucial to avoiding quadratic running time in certain typical applications (see EXERCISE 2.3.11). Later, we discuss a better strategy for the case when the array contains a large number of items with equal keys.

Terminating the recursion. Experienced programmers also know to take special care to ensure that any recursive method must always terminate, and quicksort is again no exception. For instance, a common mistake in implementing quicksort involves not ensuring that one item is always put into position, then falling into an infinite recursive loop when the partitioning item happens to be the largest or smallest item in the array.

Performance characteristics Quicksort has been subjected to very thorough mathematical analysis, so that we can make precise statements about its performance. The analysis has been validated through extensive empirical experience, and is a useful tool in tuning the algorithm for optimum performance.

The inner loop of quicksort (in the partitioning method) increments an index and compares an array entry against a fixed value. This simplicity is one factor that makes quicksort quick: it is hard to envision a shorter inner loop in a sorting algorithm. For example, mergesort and shellsort are typically slower than quicksort because they also do data movement within their inner loops.

The second factor that makes quicksort quick is that it uses few compares. Ultimately, the efficiency of the sort depends on how well the partitioning divides the array, which in turn depends on the value of the partitioning item's key. Partitioning divides a large randomly ordered array into two smaller randomly ordered subarrays, but the actual split is equally likely (for distinct keys) to be anywhere in the array. Next, we consider the analysis of the algorithm, which allows us to see how this choice compares to the ideal choice.

The best case for quicksort is when each partitioning stage divides the array exactly in half. This circumstance would make the number of compares used by quicksort satisfy the divide-and-conquer recurrence $C_N = 2C_{N/2} + N$. The $2C_{N/2}$ term covers the cost of sorting the two subarrays; the N is the cost of examining each entry, using one partitioning index or the other. As in the proof of PROPOSITION F for mergesort, we know that this recurrence has the solution $C_N \sim N \lg N$. Although things do not always go this well, it is true that the partition falls in the middle *on the average*. Taking into account the precise probability of each partition position makes the recurrence more complicated and more difficult to solve, but the final result is similar. The proof of this result is the basis for our confidence in quicksort. If you are not mathematically inclined, you may wish to skip (and trust) it; if you are mathematically inclined, you may find it intriguing.

Proposition K. Quicksort uses $\sim 2N \ln N$ compares (and one-sixth that many exchanges) on the average to sort an array of length N with distinct keys.

Proof: Let C_N be the average number of compares needed to sort N items with distinct values. We have $C_0 = C_1 = 0$ and for $N > 1$ we can write a recurrence relationship that directly mirrors the recursive program:

$$C_N = N + 1 + (C_0 + C_1 + \dots + C_{N-2} + C_{N-1})/N + (C_{N-1} + C_{N-2} + \dots + C_0)/N$$

The first term is the cost of partitioning (always $N + 1$), the second term is the average cost of sorting the left subarray (which is equally likely to be any size from 0 to $N - 1$), and the third term is the average cost for the right subarray (which is the same as for the left subarray). Multiplying by N and collecting terms transforms this equation to

$$NC_N = N(N + 1) + 2(C_0 + C_1 + \dots + C_{N-2} + C_{N-1})$$

Subtracting the same equation for $N - 1$ from this equation gives

$$NC_N - (N - 1)C_{N-1} = 2N + 2C_{N-1}$$

Rearranging terms and dividing by $N(N + 1)$ leaves

$$C_N/(N + 1) = C_{N-1}/N + 2/(N + 1)$$

which telescopes to give the result

$$C_N \sim 2(N + 1)(1/3 + 1/4 + \dots + 1/(N + 1))$$

The parenthesized quantity is the discrete estimate of the area under the curve $1/x$ from 3 to $N + 1$ and $C_N \sim 2N \ln N$ by integration. Note that $2N \ln N \approx 1.39N \lg N$, so the average number of compares is only about 39 percent higher than in the best case.

A similar (but much more complicated) analysis is needed to establish the stated result for exchanges.

When keys may not be distinct, as is typical in practical applications, precise analysis is considerably more complicated, but it is not difficult to show that the average number of compares is no greater than C_N , even when duplicate keys may be present (on page 296, we will look at a way to improve quicksort in this case).

Despite its many assets, the basic quicksort program has one potential liability: it can be extremely inefficient if the partitions are unbalanced. For example, it could be the case that the first partition is on the smallest item, the second partition on the next smallest item, and so forth, so that the program will remove just one item for each call, leading to an excessive number of partitions of large subarrays. Avoiding this situation is the primary reason that we randomly shuffle the array before using quicksort. This action makes it so unlikely that bad partitions will happen consistently that we need not worry about the possibility.

Proposition L. Quicksort uses $\sim N^2/2$ compares in the worst case, but random shuffling protects against this case.

Proof: By the argument just given, the number of compares used when one of the subarrays is empty for every partition is

$$N + (N - 1) + (N - 2) + \dots + 2 + 1 = (N + 1) N / 2$$

This behavior means not only that the time required will be quadratic but also that the space required to handle the recursion will be linear, which is unacceptable for large arrays. But (with quite a bit more work) it is possible to extend the analysis that we did for the average to find that the standard deviation of the number of compares is about $.65N$, so the running time tends to the average as N grows and is unlikely to be far from the average. For example, even the rough estimate provided by Chebyshev's inequality says that the probability that the running time is more than ten times the average for an array with a million elements is less than $.00001$ (and the true probability is far smaller). The probability that the running time for a large array is close to quadratic is so remote that we can safely ignore the possibility (see EXERCISE 2.3.10). For example, the probability that quicksort will use as many compares as insertion sort or selection sort when sorting a large array on your computer is much less than the probability that your computer will be struck by lightning during the sort!

IN SUMMARY, you can be sure that the running time of ALGORITHM 2.5 will be within a constant factor of $1.39N \lg N$ whenever it is used to sort N items. The same is true of mergesort, but quicksort is typically faster because (even though it does 39 percent more compares) it does much less data movement. This mathematical assurance is probabilistic, but you can certainly rely upon it.

Algorithmic improvements Quicksort was invented in 1960 by C. A. R. Hoare, and many people have studied and refined it since that time. It is tempting to try to develop ways to improve quicksort: a faster sorting algorithm is computer science's "better mousetrap," and quicksort is a venerable method that seems to invite tinkering. Almost from the moment Hoare first published the algorithm, people began proposing ways to improve the algorithm. Not all of these ideas are fully successful, because the algorithm is so well-balanced that the effects of improvements can be more than offset by unexpected side effects, but a few of them, which we now consider, are quite effective.

If your sort code is to be used a great many times or to sort a huge array (or, in particular, if it is to be used as a library sort that will be used to sort arrays of unknown characteristics), then it is worthwhile to consider the improvements that are discussed in the next few paragraphs. As noted, you need to run experiments to determine the effectiveness of these improvements and to determine the best choice of parameters for your implementation. Typically, improvements of 20 to 30 percent are available.

Cutoff to insertion sort. As with most recursive algorithms, an easy way to improve the performance of quicksort is based on the following two observations:

- Quicksort is slower than insertion sort for tiny subarrays.
- Being recursive, quicksort's `sort()` is certain to call itself for tiny subarrays.

Accordingly, it pays to switch to insertion sort for tiny subarrays. A simple change to ALGORITHM 2.5 accomplishes this improvement: replace the statement

```
if (hi <= lo) return;
```

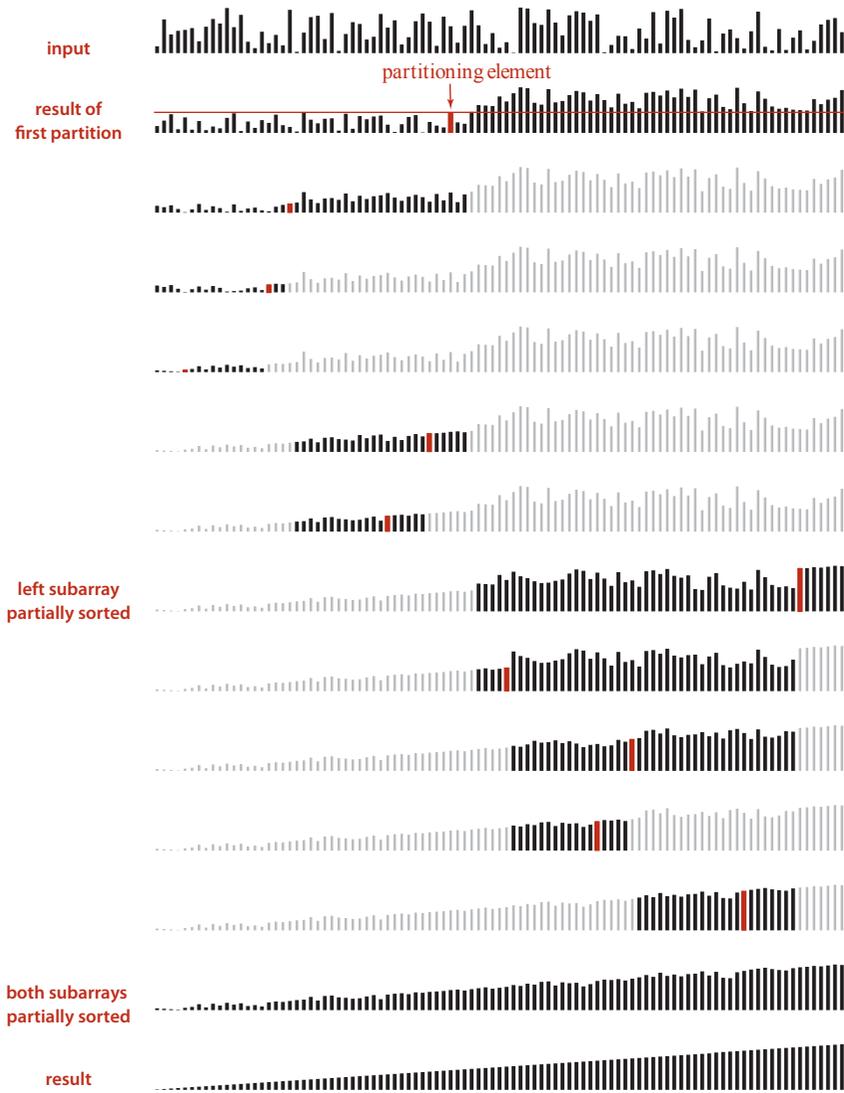
in `sort()` with a statement that invokes insertion sort for small subarrays:

```
if (hi <= lo + M) { Insertion.sort(a, lo, hi); return; }
```

The optimum value of the cutoff `M` is system-dependent, but any value between 5 and 15 is likely to work well in most situations (see EXERCISE 2.3.25).

Median-of-three partitioning. A second easy way to improve the performance of quicksort is to use the median of a small sample of items taken from the subarray as the partitioning item. Doing so will give a slightly better partition, but at the cost of computing the median. It turns out that most of the available improvement comes from choosing a sample of size 3 and then partitioning on the middle item (see EXERCISES 2.3.18 and 2.3.19). As a bonus, we can use the sample items as sentinels at the ends of the array and remove both array bounds tests in `partition()`.

Entropy-optimal sorting. Arrays with large numbers of duplicate keys arise frequently in applications. For example, we might wish to sort a large personnel file by year of birth, or perhaps to separate females from males. In such situations, the quicksort implementation that we have considered has acceptable performance, but it can be substantially improved. For example, a subarray that consists solely of items that are equal (just one key value) does not need to be processed further, but our implementation keeps partitioning down to small subarrays. In a situation where there are large numbers of duplicate keys in the input array, the recursive nature of quicksort ensures that subarrays consisting solely of items with keys that are equal will occur often. There is potential for significant improvement, from the linearithmic-time performance of the implementations seen so far to linear-time performance.



Quicksort with median-of-3 partitioning and cutoff for small subarrays

One straightforward idea is to partition the array into *three* parts, one each for items with keys smaller than, equal to, and larger than the partitioning item's key. Accomplishing this partitioning is more complicated than the 2-way partitioning that we have been using, and various different methods have been suggested for the task. It was a classical programming exercise popularized by E. W. Dijkstra as the *Dutch National Flag* problem, because it is like sorting an array with three possible key values, which might correspond to the three colors on the flag.

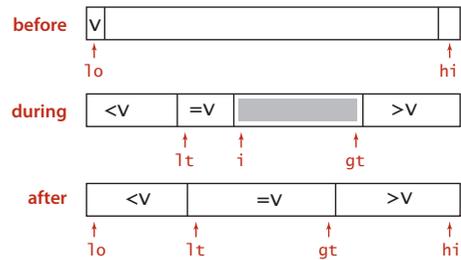
Dijkstra's solution to this problem leads to the remarkably simple partition code shown on the next page. It is based on a single left-to-right pass through the array that maintains a pointer `lt` such that `a[lo..lt-1]` is *less than* `v`, a pointer `gt` such that `a[gt+1..hi]` is *greater than* `v`, and a pointer `i` such that `a[lt..i-1]` are *equal to* `v` and `a[i..gt]` are not yet examined. Starting with `i` equal to `lo`, we process `a[i]` using the 3-way comparison given by the `Comparable` interface (instead of using `less()`) to directly handle the three possible cases:

- `a[i]` less than `v`: exchange `a[lt]` with `a[i]` and increment both `lt` and `i`
- `a[i]` greater than `v`: exchange `a[i]` with `a[gt]` and decrement `gt`
- `a[i]` equal to `v`: increment `i`

Each of these operations both maintains the invariant and decreases the value of `gt-i` (so that the loop terminates). Furthermore, every item encountered leads to an exchange except for those items with keys equal to the partitioning item's key.

Though this code was developed not long after quicksort in the 1970s, it fell out of favor because it uses many more exchanges than the standard 2-way partitioning method for the common case when the number of duplicate keys in the array is not high. In the 1990s J. Bentley and D. McIlroy developed a clever implementation that overcomes this problem (see EXERCISE 2.3.22), and observed that 3-way partitioning makes quicksort asymptotically faster than mergesort and other methods in practical situations involving large numbers of equal keys. Later, J. Bentley and R. Sedgwick developed a proof of this fact, which we discuss next.

But we proved that mergesort is optimal. How have we defeated that lower bound? The answer to this question is that PROPOSITION 1 in SECTION 2.2 addresses worst-case performance over all possible inputs, while now we are looking at worst-case performance with some information about the key values at hand. Mergesort does not guarantee optimal performance for any given distribution of duplicates in the input:



3-way partitioning overview

Quicksort with 3-way partitioning

```

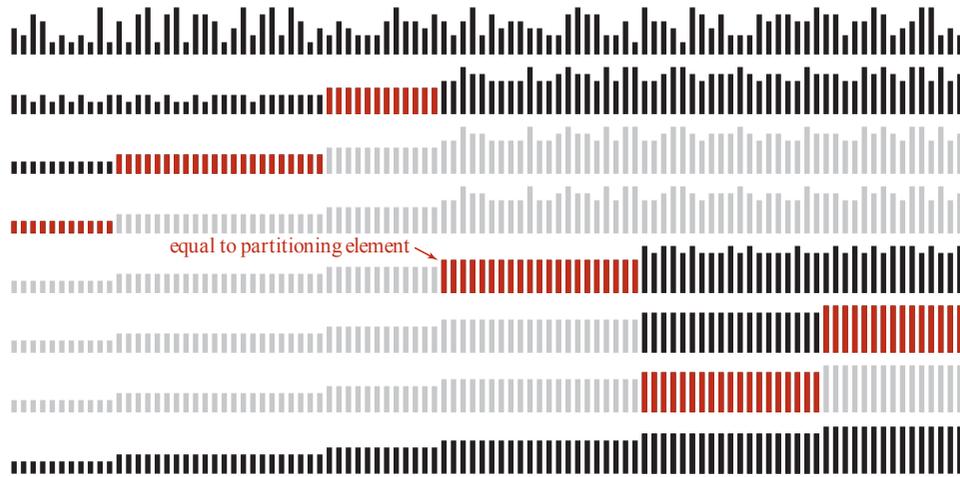
public class Quick3way
{
    private static void sort(Comparable[] a, int lo, int hi)
    { // See page 289 for public sort() that calls this method.
      if (hi <= lo) return;
      int lt = lo, i = lo+1, gt = hi;
      Comparable v = a[lo];
      while (i <= gt)
      {
          int cmp = a[i].compareTo(v);
          if (cmp < 0)  exch(a, lt++, i++);
          else if (cmp > 0) exch(a, i, gt--);
          else          i++;
      } // Now a[lo..lt-1] < v = a[lt..gt] < a[gt+1..hi].
      sort(a, lo, lt - 1);
      sort(a, gt + 1, hi);
    }
}

```

This sort code partitions to put keys equal to the partitioning element in place and thus does not have to include those keys in the subarrays for the recursive calls. It is far more efficient than the standard quicksort implementation for arrays with large numbers of duplicate keys (see text).

lt	i	gt	a[]												
			0	1	2	3	4	5	6	7	8	9	10	11	
0	0	11	R	B	W	W	R	W	B	R	R	W	B	R	
0	1	11	R	B	W	W	R	W	B	R	R	W	B	R	
1	2	11	B	R	W	W	R	W	B	R	R	W	B	R	
1	2	10	B	R	R	W	R	W	B	R	R	W	B	W	
1	3	10	B	R	R	W	R	W	B	R	R	W	B	W	
1	3	9	B	R	R	B	R	W	B	R	R	W	W	W	
2	4	9	B	B	R	R	R	W	B	R	R	W	W	W	
2	5	9	B	B	R	R	R	W	B	R	R	W	W	W	
2	5	8	B	B	R	R	R	W	B	R	R	W	W	W	
2	5	7	B	B	R	R	R	R	B	R	W	W	W	W	
2	6	7	B	B	R	R	R	R	B	R	W	W	W	W	
3	7	7	B	B	B	R	R	R	R	R	W	W	W	W	
3	8	7	B	B	B	R	R	R	R	R	W	W	W	W	
3	8	7	B	B	B	R	R	R	R	R	W	W	W	W	

3-way partitioning trace (array contents after each loop iteration)



Visual trace of quicksort with 3-way partitioning

for example, mergesort is linearithmic for a randomly ordered array that has only a constant number of distinct key values, but quicksort with 3-way partitioning is linear for such an array. Indeed, by examining the visual trace above, you can see that N times the number of key values is a conservative bound on the running time.

The analysis that makes these notions precise takes the distribution of key values into account. Given N keys with k distinct key values, for each i from 1 to k define f_i to be frequency of occurrence of the i th key value and p_i to be f_i / N , the probability that the i th key value is found when a random entry of the array is sampled. The *Shannon* entropy of the keys (a classic measure of their information content) is defined as

$$H = -(p_1 \lg p_1 + p_2 \lg p_2 + \dots + p_k \lg p_k).$$

Given any array of items to be sorted, we can calculate its entropy by counting the frequency of each key value. Remarkably, we can also derive from the entropy both a lower bound on the number of compares and an upper bound on the number of compares used by quicksort with 3-way partitioning.

Proposition M. No compare-based sorting algorithm can guarantee to sort N items with fewer than $NH - N$ compares, where H is the Shannon entropy, defined from the frequencies of key values.

Proof sketch: This result follows from a (relatively easy) generalization of the lower bound proof of PROPOSITION I in SECTION 2.2.

Proposition N. Quicksort with 3-way partitioning uses $\sim (2 \ln 2) N H$ compares to sort N items, where H is the Shannon entropy, defined from the frequencies of key values.

Proof sketch: This result follows from a (relatively difficult) generalization of the average-case analysis of quicksort in PROPOSITION K. As with distinct keys, this costs about 39 percent more than the optimum (but within a constant factor).

Note that $H = \lg N$ when the keys are all distinct (all the probabilities are $1/N$), which is consistent with PROPOSITION I in SECTION 2.2 and PROPOSITION K. The worst case for 3-way partitioning happens when the keys are distinct; when duplicate keys are present, it can do much better than mergesort. More important, these two properties together imply that quicksort with 3-way partitioning is entropy-optimal, in the sense that the average number of compares used by the best possible compare-based sorting algorithm and the average number of compares used by 3-way quicksort are within a constant factor of one another, for any given distribution of input key values.

As with standard quicksort, the running time tends to the average as the array size grows, and large deviations from the average are extremely unlikely, so that you can depend on 3-way quicksort's running time to be proportional to N times the entropy of the distribution of input key values. This property of the algorithm is important in practice because *it reduces the time of the sort from linearithmic to linear for arrays with large numbers of duplicate keys*. The order of the keys is immaterial, because the algorithm shuffles them to protect against the worst case. The distribution of keys defines the entropy and no compare-based algorithm can use fewer compares than defined by the entropy. This ability to adapt to duplicates in the input makes 3-way quicksort the algorithm of choice for a library sort—clients that sort arrays containing large numbers of duplicate keys are not unusual.

A CAREFULLY TUNED VERSION of quicksort is likely to run significantly faster on most computers for most applications than will any other compare-based sorting method. Quicksort is widely used throughout today's computational infrastructure because the mathematical models that we have discussed suggest that it will outperform other methods in practical applications, and extensive experiments and experience over the past several decades have validated that conclusion.

We will see in CHAPTER 5 that this is not quite the end of the story in the development of sorting algorithms, because it is possible to develop algorithms that do not use compares at all! But a version of quicksort turns out to be best in that situation, as well.

Q & A

Q. Is there some way to just divide the array into two halves, rather than letting the partitioning element fall where it may?

A. That is a question that stumped experts for over a decade. It is tantamount to finding the median key value in the array and then partitioning on that value. We discuss the problem of finding the median on page 346. It is possible to do so in linear time, but the cost of doing so with known algorithms (which are based on quicksort partitioning!) far exceeds the 39 percent savings available from splitting the array into equal parts.

Q. Randomly shuffling the array seems to take a significant fraction of the total time for the sort. Is doing so really worthwhile?

A. Yes. It protects against the worst case and makes the running time predictable. Hoare proposed this approach when he presented the algorithm in 1960—it is a prototypical (and among the first) randomized algorithm.

Q. Why all the focus on items with equal keys?

A. The issue directly impacts performance in practical situations. It was overlooked by many for decades, with the result that some older implementations of quicksort take quadratic time for arrays with large numbers of items with equal keys, which certainly do arise in applications. Better implementations such as ALGORITHM 2.5 take linearithmic time for such arrays, but improving that to linear-time as in the entropy-optimal sort at the end of this section is worthwhile in many situations.

EXERCISES

- 2.3.1** Show, in the style of the trace given with `partition()`, how that method partitions the array `E A S Y Q U E S T I O N`.
- 2.3.2** Show, in the style of the quicksort trace given in this section, how quicksort sorts the array `E A S Y Q U E S T I O N` (for the purposes of this exercise, ignore the initial shuffle).
- 2.3.3** What is the maximum number of times during the execution of `Quick.sort()` that the largest item can be exchanged, for an array of length N ?
- 2.3.4** Suppose that the initial random shuffle is omitted. Give six arrays of ten elements for which `Quick.sort()` uses the worst-case number of compares.
- 2.3.5** Give a code fragment that sorts an array that is known to consist of items having just two distinct keys.
- 2.3.6** Write a program to compute the exact value of C_N , and compare the exact value with the approximation $2N \ln N$, for $N = 100, 1,000$, and $10,000$.
- 2.3.7** Find the expected number of subarrays of size 0, 1, and 2 when quicksort is used to sort an array of N items with distinct keys. If you are mathematically inclined, do the math; if not, run some experiments to develop hypotheses.
- 2.3.8** About how many compares will `Quick.sort()` make when sorting an array of N items that are all equal?
- 2.3.9** Explain what happens when `Quick.sort()` is run on an array having items with just two distinct keys, and then explain what happens when it is run on an array having just three distinct keys.
- 2.3.10** *Chebyshev's inequality* says that the probability that a random variable is more than k standard deviations away from the mean is less than $1/k^2$. For $N = 1$ million, use Chebyshev's inequality to bound the probability that the number of compares used by quicksort is more than 100 billion ($.1 N^2$).
- 2.3.11** Suppose that we scan over items with keys equal to the partitioning item's key instead of stopping the scans when we encounter them. Show that the running time of this version of quicksort is quadratic for all arrays with just a constant number of distinct keys.

EXERCISES *(continued)*

2.3.12 Show, in the style of the trace given with the code, how the 3-way quicksort first partitions the array B A B A B A B A C A D A B R A.

2.3.13 What is the *recursive depth* of quicksort, in the best, worst, and average cases? This is the size of the stack that the system needs to keep track of the recursive calls. See EXERCISE 2.3.20 for a way to guarantee that the recursive depth is logarithmic in the worst case.

2.3.14 Prove that when running quicksort on an array with N distinct items, the probability of comparing the i th and j th smallest items is $2 / (j - i - 1)$. Then use this result to prove PROPOSITION K.

CREATIVE PROBLEMS

2.3.15 *Nuts and bolts.* (G. J. E. Rawlins) You have a mixed pile of N nuts and N bolts and need to quickly find the corresponding pairs of nuts and bolts. Each nut matches exactly one bolt, and each bolt matches exactly one nut. By fitting a nut and bolt together, you can see which is bigger, but it is not possible to directly compare two nuts or two bolts. Give an efficient method for solving the problem.

2.3.16 *Best case.* Write a program that produces a best-case array (with no duplicates) for `sort()` in ALGORITHM 2.5: an array of N items with distinct keys having the property that every partition will produce subarrays that differ in size by at most 1 (the same subarray sizes that would happen for an array of N equal keys). (For the purposes of this exercise, ignore the initial shuffle.)

The following exercises describe variants of quicksort. Each of them calls for an implementation, but naturally you will also want to use `SortCompare` for experiments to evaluate the effectiveness of each suggested modification.

2.3.17 *Sentinels.* Modify the code in ALGORITHM 2.5 to remove both bounds checks in the inner `while` loops. The test against the left end of the subarray is redundant since the partitioning item acts as a sentinel (v is never less than $a[l_0]$). To enable removal of the other test, put an item whose key is the largest in the whole array into $a[l_{\text{length}}-1]$ just after the shuffle. This item will never move (except possibly to be swapped with an item having the same key) and will serve as a sentinel in all subarrays involving the end of the array. Note: For a subarray that does not involve the end of the array, the leftmost entry to its right serves as a sentinel for the right end of the subarray.

2.3.18 *Median-of-3 partitioning.* Add median-of-3 partitioning to quicksort, as described in the text (see page 296). Run doubling tests to determine the effectiveness of the change.

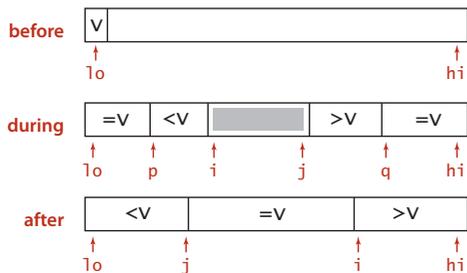
2.3.19 *Median-of-5 partitioning.* Implement a quicksort based on partitioning on the median of a random sample of five items from the subarray. Put the items of the sample at the appropriate ends of the array so that only the median participates in partitioning. Run doubling tests to determine the effectiveness of the change, in comparison both to the standard algorithm and to median-of-3 partitioning (see the previous exercise). *Extra credit:* Devise a median-of-5 algorithm that uses fewer than seven compares on any input.

CREATIVE PROBLEMS *(continued)*

2.3.20 *Nonrecursive quicksort.* Implement a nonrecursive version of quicksort based on a main loop where a subarray is popped from a stack to be partitioned, and the resulting subarrays are pushed onto the stack. Note: Push the larger of the subarrays onto the stack first, which guarantees that the stack will have at most $\lg N$ entries.

2.3.21 *Lower bound for sorting with equal keys.* Complete the first part of the proof of PROPOSITION M by following the logic in the proof of PROPOSITION I and using the observation that there are $N! / f_1! f_2! \dots f_k!$ different ways to arrange keys with k different values, where the i th value appears with frequency $f_i (= Np_i$, in the notation of PROPOSITION M), with $f_1 + \dots + f_k = N$.

2.3.22 *Fast 3-way partitioning.* (J. Bentley and D. McIlroy) Implement an entropy-optimal sort based on keeping



Bentley-McIlroy 3-way partitioning

of the subarray. Maintain indices p and q such that $a[lo..p-1]$ and $a[q+1..hi]$ are all equal to $a[lo]$, an index i such that $a[p..i-1]$ are all less than $a[lo]$, and an index j such that $a[j+1..q]$ are all greater than $a[lo]$. Add to the inner partitioning loop code to swap $a[i]$ with $a[p]$ (and increment p) if it is equal to v and to swap $a[j]$ with $a[q]$ (and decrement q) if it is equal to v before the usual comparisons of $a[i]$ and $a[j]$ with v . After the partitioning loop has terminated, add code to swap the items with equal keys into position.

Note: This code complements the code given in the

text, in the sense that it does extra swaps for keys equal to the partitioning item's key, while the code in the text does extra swaps for keys that are not equal to the partitioning item's key.

2.3.23 *Tukey's ninther.* Add to your implementation from EXERCISE 2.3.22 code to use the *Tukey ninther* to compute the partitioning item—choose three sets of three items, take the median of each, then use the median of the three medians as the partitioning item. Also, add a cutoff to insertion sort for small subarrays.

2.3.24 *Samplesort.* (W. Frazer and A. McKellar) Implement a quicksort based on using a sample of size $2^k - 1$. First, sort the sample, then arrange to have the recursive routine partition on the median of the sample and to move the two halves of the rest of the sample to each subarray, such that they can be used in the subarrays, without having to be sorted again. This algorithm is called *samplesort*.

EXPERIMENTS

2.3.25 *Cutoff to insertion sort.* Implement quicksort with a cutoff to insertion sort for subarrays with less than M elements, and empirically determine the value of M for which quicksort runs fastest in your computing environment to sort random arrays of N doubles, for $N = 10^3, 10^4, 10^5$, and 10^6 . Plot average running times for M from 0 to 30 for each value of M . Note: You need to add a three-argument `sort()` method to ALGORITHM 2.2 for sorting subarrays such that the call `Insertion.sort(a, lo, hi)` sorts the subarray `a[lo..hi]`.

2.3.26 *Subarray sizes.* Write a program that plots a histogram of the subarray sizes left for insertion sort when you run quicksort for an array of size N with a cutoff for subarrays of size less than M . Run your program for $M=10, 20$, and 50 and $N = 10^5$.

2.3.27 *Ignore small subarrays.* Run experiments to compare the following strategy for dealing with small subarrays with the approach described in EXERCISE 2.3.25: Simply ignore the small subarrays in quicksort, then run a single insertion sort after the quicksort completes. Note: You may be able to estimate the size of your computer's cache memory with these experiments, as the performance of this method is likely to degrade when the array does not fit in the cache.

2.3.28 *Recursion depth.* Run empirical studies to determine the average recursive depth used by quicksort with cutoff for arrays of size M , when sorting arrays of N distinct elements, for $M=10, 20$, and 50 and $N = 10^3, 10^4, 10^5$, and 10^6 .

2.3.29 *Randomization.* Run empirical studies to compare the effectiveness of the strategy of choosing a random partitioning item with the strategy of initially randomizing the array (as in the text). Use a cutoff for arrays of size M , and sort arrays of N distinct elements, for $M=10, 20$, and 50 and $N = 10^3, 10^4, 10^5$, and 10^6 .

2.3.30 *Corner cases.* Test quicksort on large nonrandom arrays of the kind described in EXERCISES 2.1.35 and 2.1.36 both with and without the initial random shuffle. How does shuffling affect its performance for these arrays?

2.3.31 *Histogram of running times.* Write a program that takes command-line arguments N and T , does T trials of the experiment of running quicksort on an array of N random `Double` values, and plots a histogram of the observed running times. Run your program for $N = 10^3, 10^4, 10^5$, and 10^6 , with T as large as you can afford to make the curves smooth. Your main challenge for this exercise is to appropriately scale the experimental results.



2.4 PRIORITY QUEUES

MANY APPLICATIONS REQUIRE that we process items having keys in order, but not necessarily in full sorted order and not necessarily all at once. Often, we collect a set of items, then process the one with the largest key, then perhaps collect more items, then process the one with the current largest key, and so forth. For example, you are likely to have a computer (or a cellphone) that is capable of running several applications at the same time. This effect is typically achieved by assigning a priority to events associated with applications, then always choosing to process next the highest-priority event. For example, most cellphones are likely to process an incoming call with higher priority than a game application.

An appropriate data type in such an environment supports two operations: remove *the maximum* and insert. Such a data type is called a *priority queue*. Using priority queues is similar to using queues (remove the oldest) and stacks (remove the newest), but implementing them efficiently is more challenging.

In this section, after a short discussion of elementary representations where one or both of the operations take linear time, we consider a classic priority-queue implementation based on the *binary heap* data structure, where items are kept in an array, subject to certain ordering constraints that allow for efficient (logarithmic-time) implementations of *remove the maximum* and insert.

Some important applications of priority queues include simulation systems, where the keys correspond to event times, to be processed in chronological order; job scheduling, where the keys correspond to priorities indicating which tasks are to be performed first; and numerical computations, where the keys represent computational errors, indicating in which order we should deal with them. We consider in CHAPTER 6 a detailed case study showing the use of priority queues in a particle-collision simulation.

We can use any priority queue as the basis for a sorting algorithm by inserting a sequence of items, then successively removing the smallest to get them out, in order. An important sorting algorithm known as *heapsort* also follows naturally from our heap-based priority-queue implementations. Later on in this book, we shall see how to use priority queues as building blocks for other algorithms. In CHAPTER 4, we shall see how priority queues are an appropriate abstraction for implementing several fundamental graph-searching algorithms; in CHAPTER 5, we shall develop a data-compression algorithm using methods from this section. These are but a few examples of the important role played by the priority queue as a tool in algorithm design.

API The priority queue is a prototypical abstract data type (see SECTION 1.2): it represents a set of values and operations on those values, and it provides a convenient abstraction that allows us to separate application programs (clients) from various implementations that we will consider in this section. As in SECTION 1.2, we precisely define the operations by specifying an applications programming interface (API) that provides the information needed by clients. Priority queues are characterized by the *remove the maximum* and *insert* operations, so we shall focus on them. We use the method names `delMax()` for *remove the maximum* and `insert()` for *insert*. By convention, we will compare keys only with a helper `less()` method, as we have been doing for sorting. Thus, if items can have duplicate keys, maximum means any item with the largest key value. To complete the API, we also need to add constructors (like the ones we used for stacks and queues) and a test if empty operation. For flexibility, we use a generic implementation with a parameterized type `Key` that implements the `Comparable` interface. This choice eliminates our distinction between items and keys and enables clearer and more compact descriptions of data structures and algorithms. For example, we refer to the “largest key” instead of the “largest item” or the “item with the largest key.”

For convenience in client code, the API includes three constructors, which enable clients to build priority queues of an initial fixed size (perhaps initialized with a given array of keys). To clarify client code, we will use a separate class `MinPQ` whenever appropriate, which is the same as `MaxPQ` except that it has a `delMin()` method that deletes and returns an item with the smallest key in the queue. Any `MaxPQ` implementation is easily converted into a `MinPQ` implementation and vice versa, simply by reversing the sense of the comparison in `less()`.

```
public class MaxPQ<Key extends Comparable<Key>>
```

<code>MaxPQ()</code>	<i>create a priority queue</i>
<code>MaxPQ(int max)</code>	<i>create a priority queue of initial capacity max</i>
<code>MaxPQ(Key[] a)</code>	<i>create a priority queue from the keys in a[]</i>
<code>void insert(Key v)</code>	<i>insert a key into the priority queue</i>
<code>Key max()</code>	<i>return the largest key</i>
<code>Key delMax()</code>	<i>return and remove the largest key</i>
<code>boolean isEmpty()</code>	<i>is the priority queue empty?</i>
<code>int size()</code>	<i>number of keys in the priority queue</i>

API for a generic priority queue

A priority-queue client. To appreciate the value of the priority-queue abstraction, consider the following problem: You have a huge input stream of N strings and associated integer values, and your task is to find the largest or smallest M integers (and associated strings) in the input stream. You might imagine the stream to be financial transactions, where your interest is to find the big ones, or pesticide levels in an agricultural product, where your interest is to find the small ones, or requests for service, or results from a scientific

experiment, or whatever. In some applications, the size of the input stream is so huge that it is best to consider it to be unbounded. One way to address this problem would be to sort the input stream and take the M largest keys from the result, but we have just stipulated that the input stream is too large for that. Another approach would be to compare each new key against the M largest seen so far, but that is also likely to be prohibitively expensive unless M is small. With priority queues, we can solve the problem with the `MinPQ` client `TopM` on the next page provided that we can develop efficient implementations of both `insert()` and `delMin()`. That is precisely our aim in this section. For the huge values of N that are likely to be encountered in our modern computational infrastructure, these implementations can make the difference between being able to address such a problem and not having the resources to do it at all.

Elementary implementations The basic data structures that we discussed in CHAPTER 1 provide us with four immediate starting points for implementing priority queues. We can use an array or a linked list, kept in order or unordered. These implementations are useful for small priority queues, situations where one of the two operations are predominant, or situations where some assumptions can be made about the order of the keys involved in the operations. Since these implementations are elementary, we will be content with brief descriptions here in the text and leave the code for exercises (see EXERCISE 2.4.3).

Array representation (unordered). Perhaps the simplest priority-queue implementation is based on our code for pushdown stacks in SECTION 1.3. The code for `insert` in the priority queue is the same as for `push` in the stack. To implement *remove the maximum*, we can add code like the inner loop of selection sort to exchange the maximum item with the item at the end and then delete that one, as we did with `pop()` for stacks. As with stacks, we can add resizing-array code to ensure that the data structure is always at least one-quarter full and never overflows.

client	order of growth	
	time	space
<i>sort client</i>	$N \log N$	N
<i>PQ client using elementary implementation</i>	NM	M
<i>PQ client using heap-based implementation</i>	$N \log M$	M

Costs of finding the largest M in a stream of N items

A priority-queue client

```

public class TopM
{
    public static void main(String[] args)
    { // Print the top M lines in the input stream.
        int M = Integer.parseInt(args[0]);
        MinPQ<Transaction> pq = new MinPQ<Transaction>(M+1);
        while (StdIn.hasNextLine())
        { // Create an entry from the next line and put on the PQ.
            pq.insert(new Transaction(StdIn.readLine()));
            if (pq.size() > M)
                pq.delMin(); // Remove minimum if M+1 entries on the PQ.
        } // Top M entries are on the PQ.

        Stack<Transaction> stack = new Stack<Transaction>();
        while (!pq.isEmpty()) stack.push(pq.delMin());
        for (Transaction t : stack) StdOut.println(t);
    }
}

```

Given an integer M from the command line and an input stream where each line contains a transaction, this `MinPQ` client prints the M lines whose numbers are the highest. It does so by using our `Transaction` class (see page 79, EXERCISE 1.2.19, and EXERCISE 2.1.21) to build a priority queue using

the numbers as keys, deleting the minimum after each insertion once the size of the priority queue reaches M . Once all the transactions have been processed, the top M come off the priority queue in increasing order, so this code puts them on a stack, then iterates through the stack to reverse the order and print them in decreasing order.

```

% more tinyBatch.txt
Turing      6/17/1990   644.08
vonNeumann  3/26/2002  4121.85
Dijkstra    8/22/2007  2678.40
vonNeumann  1/11/1999  4409.74
Dijkstra    11/18/1995  837.42
Hoare       5/10/1993  3229.27
vonNeumann  2/12/1994  4732.35
Hoare       8/18/1992  4381.21
Turing      1/11/2002   66.10
Thompson    2/27/2000  4747.08
Turing      2/11/1991  2156.86
Hoare       8/12/2003  1025.70
vonNeumann  10/13/1993 2520.97
Dijkstra    9/10/2000   708.95
Turing      10/12/1993 3532.36
Hoare       2/10/2005  4050.20

```

```

% java TopM 5 < tinyBatch.txt
Thompson    2/27/2000  4747.08
vonNeumann  2/12/1994  4732.35
vonNeumann  1/11/1999  4409.74
Hoare       8/18/1992  4381.21
vonNeumann  3/26/2002  4121.85

```

Array representation (ordered). Another approach is to add code for insert to move larger entries one position to the right, thus keeping the keys in the array in order (as in insertion sort). Thus, the largest entry is always at the end, and the code for *remove the maximum* in the priority queue is the same as for pop in the stack.

Linked-list representations. Similarly, we can start with our linked-list code for push-down stacks, modifying either the code for pop() to find and return the maximum or the code for push() to keep keys in reverse order and the code for pop() to unlink and return the first (maximum) item on the list.

data structure	insert	remove maximum
<i>ordered array</i>	N	1
<i>unordered array</i>	1	N
<i>heap</i>	$\log N$	$\log N$
<i>impossible</i>	1	1

Order of growth of worst-case running time for priority-queue implementations

Using unordered sequences is the prototypical *lazy* approach to this problem, where we defer doing work until necessary (to find the maximum); using ordered sequences is the prototypical *eager* approach to the problem, where we do as much work as we can up front (keep the list sorted on insertion) to make later operations efficient.

The significant difference between implementing stacks or queues and implementing priority queues has to do with performance. For stacks and queues, we were able to develop implementations of all the operations that take constant time; for priority queues, all of the elementary implementations just discussed have the property that either the insert or the *remove the maximum* operation takes linear time in the worst case. The *heap* data structure that we consider next enables implementations where *both* operations are guaranteed to be fast.

operation	argument	return value	size	contents (unordered)	contents (ordered)
insert	P		1	P	P
insert	Q		2	P Q	P Q
insert	E		3	P Q E	E P Q
remove max		Q	2	P E	E P
insert	X		3	P E X	E P X
insert	A		4	P E X A	A E P X
insert	M		5	P E X A M	A E M P X
remove max		X	4	P E M A	A E M P
insert	P		5	P E M A P	A E M P P
insert	L		6	P E M A P L	A E L M P P
insert	E		7	P E M A P L E	A E E L M P P
remove max		P	6	E E M A P L	A E E L M P

A sequence of operations on a priority queue

Heap definitions The *binary heap* is a data structure that can efficiently support the basic priority-queue operations. In a binary heap, the keys are stored in an array such that each key is guaranteed to be larger than (or equal to) the keys at two other specific positions. In turn, each of those keys must be larger than (or equal to) two additional keys, and so forth. This ordering is easy to see if we view the keys as being in a binary tree structure with edges from each key to the two keys known to be smaller.

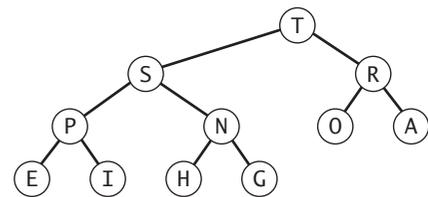
Definition. A binary tree is *heap-ordered* if the key in each node is larger than or equal to the keys in that node's two children (if any).

Equivalently, the key in each node of a heap-ordered binary tree is smaller than or equal to the key in that node's parent (if any). Moving up from any node, we get a nondecreasing sequence of keys; moving down from any node, we get a nonincreasing sequence of keys. In particular:

Proposition O. The largest key in a heap-ordered binary tree is found at the root.

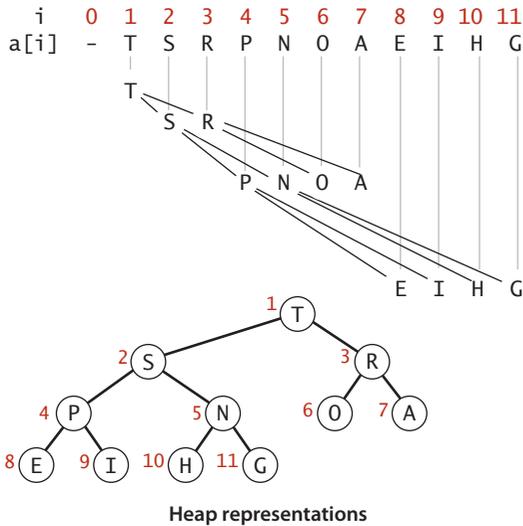
Proof: By induction on the size of the tree.

Binary heap representation. If we use a linked representation for heap-ordered binary trees, we would need to have three links associated with each key to allow travel up and down the tree (each node would have one pointer to its parent and one to each child). It is particularly convenient, instead, to use a complete binary tree like the one drawn at right. We draw such a structure by placing the root node and then proceeding down the page and from left to right, drawing and connecting two nodes beneath each node on the previous level until we have drawn N nodes. Complete trees provide the opportunity to use a compact array representation that does not involve explicit links. Specifically, we represent complete binary trees sequentially within an array by putting the nodes in level order, with the root at position 1, its children at positions 2 and 3, their children in positions 4, 5, 6, and 7, and so on.



A heap-ordered complete binary tree

Definition. A *binary heap* is a collection of keys arranged in a complete heap-ordered binary tree, represented in level order in an array (not using the first entry).



(For brevity, from now on we drop the “binary” modifier and use the term *heap* when referring to a binary heap.) In a heap, the parent of the node in position k is in position $\lfloor k/2 \rfloor$ and, conversely, the two children of the node in position k are in positions $2k$ and $2k + 1$. Instead of using explicit links (as in the binary tree structures that we will consider in CHAPTER 3), we can travel up and down by doing simple arithmetic on array indices: to move up the tree from $a[k]$ we set k to $k/2$; to move down the tree we set k to $2*k$ or $2*k+1$.

Complete binary trees represented as arrays (heaps) are rigid structures, but they have just enough flexibility to allow us to implement efficient priority-queue operations. Specifically, we

will use them to develop logarithmic-time insert and *remove the maximum* implementations. These algorithms take advantage of the capability to move up and down paths in the tree without pointers and have guaranteed logarithmic performance because of the following property of complete binary trees:

Proposition P. The height of a complete binary tree of size N is $\lfloor \lg N \rfloor$.

Proof: The stated result is easy to prove by induction or by noting that the height increases by 1 only when N is incremented to become a power of 2.

Algorithms on heaps We represent a heap of size N in private array `pq[]` of length $N + 1$, with `pq[0]` unused and the heap in `pq[1]` through `pq[N]`. As for sorting algorithms, we access keys only through private helper functions `less()` and `exch()`, but since all items are in the instance variable `pq[]`, we use the more compact implementations that do not involve passing the array name as a parameter. The heap operations that we consider work by first making a simple modification that could violate the heap condition, then traveling through the heap, modifying the heap as required to ensure that the heap condition is satisfied everywhere. We refer to this process as *reheapifying*, or *restoring heap order*.

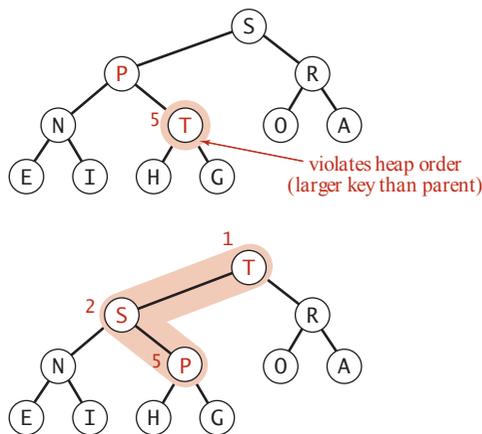
```
private boolean less(int i, int j)
{ return pq[i].compareTo(pq[j]) < 0; }

private void exch(int i, int j)
{ Key t = pq[i]; pq[i] = pq[j]; pq[j] = t; }
```

Compare and exchange methods for heap implementations

There are two cases. When the priority of some node is increased (or a new node is added at the bottom of a heap), we have to travel up the heap to restore the heap order. When the priority of some node is decreased (for example, if we replace the node at the root with a new node that has a smaller key), we have to travel down the heap to restore the heap order. First, we will consider how to implement these two basic auxiliary operations; then, we shall see how to use them to implement insert and *remove the maximum*.

Bottom-up reheapify (swim). If the heap order is violated because a node's key becomes larger than that node's parent's key, then we can make progress toward fixing the violation by exchanging the node with its parent.

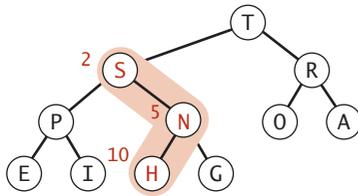
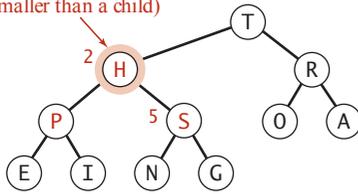


Bottom-up reheapify (swim)

After the exchange, the node is larger than both its children (one is the old parent, and the other is smaller than the old parent because it was a child of that node) but the node may still be larger than its parent. We can fix that violation in the same way, and so forth, moving up the heap until we reach a node with a larger key, or the root. Coding this process is straightforward when you keep in mind that the parent of the node at position k in a heap is at position $k/2$. The loop in `swim()` preserves the invariant that the only place the heap

order could be violated is when the node at position k might be larger than its parent. Therefore, when we get to a place where that node is not larger than its parent, we know that the heap order is satisfied throughout

violates heap order
(smaller than a child)



Top-down reheapify (sink)

the children of the node at position k in a heap are at positions $2k$ and $2k+1$. To justify the method's name, we think about the node, having too small a key, as having to sink to a lower level in the heap.

IF WE IMAGINE the heap to represent a cutthroat corporate hierarchy, with each of the children of a node representing subordinates (and the parent representing the immediate superior), then these operations have amusing interpretations. The `swim()` operation corresponds to a promising new manager arriving on the scene, being promoted up the chain of command (by exchanging jobs with any lower-qualified boss) until the new person encounters a higher-qualified boss. The `sink()` operation is analogous to the situation when the president of the company resigns and is replaced by someone from the outside. If the president's most powerful subordinate

```
private void swim(int k)
{
    while (k > 1 && less(k/2, k))
    {
        exch(k/2, k);
        k = k/2;
    }
}
```

Bottom-up reheapify (swim) implementation

key, as having to swim to a higher level in the heap.

Top-down reheapify (sink). If the heap order is violated because a node's key becomes smaller than one or both of that node's children's keys, then we can make progress toward fixing the violation by exchanging the node with the larger of its two children. This switch may cause a violation at the child; we fix that violation in the same way, and so forth, moving down the heap until we reach a node with both children smaller (or equal), or the bottom. The code again follows directly from

```
private void sink(int k)
{
    while (2*k <= N)
    {
        int j = 2*k;
        if (j < N && less(j, j+1)) j++;
        if (!less(k, j)) break;
        exch(k, j);
        k = j;
    }
}
```

Top-down reheapify (sink) implementation

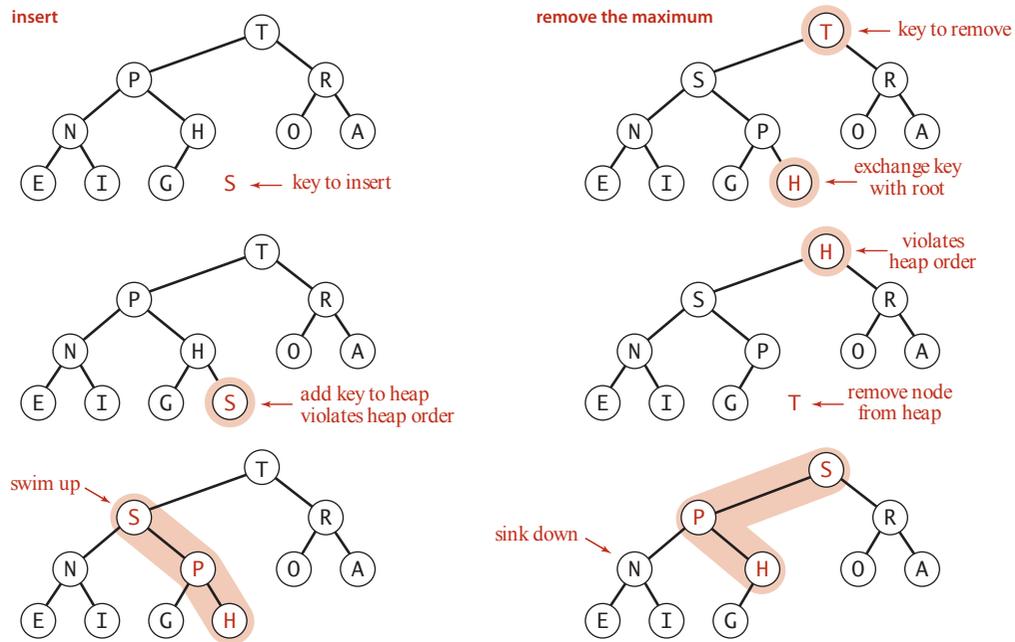
is stronger than the new person, they exchange jobs, and we move down the chain of command, demoting the new person and promoting others until the level of competence of the new person is reached, where there is no higher-qualified subordinate. These idealized scenarios may rarely be seen in the real world, but they may help you better understand basic operation on heaps.

These `sink()` and `swim()` operations provide the basis for efficient implementation of the priority-queue API, as diagrammed below and implemented in ALGORITHM 2.6.

Insert. We add the new key at the end of the array, increment the size of the heap, and then swim up through the heap with that key to restore the heap condition.

Remove the maximum. We take the largest item off the top, put the item from the end of the heap at the top, decrement the size of the heap, and then sink down through the heap with that key to restore the heap condition.

ALGORITHM 2.6 solves the basic problem that we posed at the beginning of this section: it is a priority-queue API implementation for which both insert and *delete the maximum* are guaranteed to take time logarithmic in the size of the queue.



Heap operations

ALGORITHM 2.6 Heap priority queue

```

public class MaxPQ<Key extends Comparable<Key>>
{
    private Key[] pq;           // heap-ordered complete binary tree
    private int N = 0;         // in pq[1..N] with pq[0] unused

    public MaxPQ(int maxN)
    { pq = (Key[]) new Comparable[maxN+1]; }

    public boolean isEmpty()
    { return N == 0; }

    public int size()
    { return N; }

    public void insert(Key v)
    {
        pq[++N] = v;
        swim(N);
    }

    public Key delMax()
    {
        Key max = pq[1];           // Retrieve max key from top.
        exch(1, N--);             // Exchange with last item.
        pq[N+1] = null;          // Avoid loitering.
        sink(1);                 // Restore heap property.
        return max;
    }

    // See pages 315–316 for implementations of these helper methods.
    private boolean less(int i, int j)
    private void exch(int i, int j)
    private void swim(int k)
    private void sink(int k)
}

```

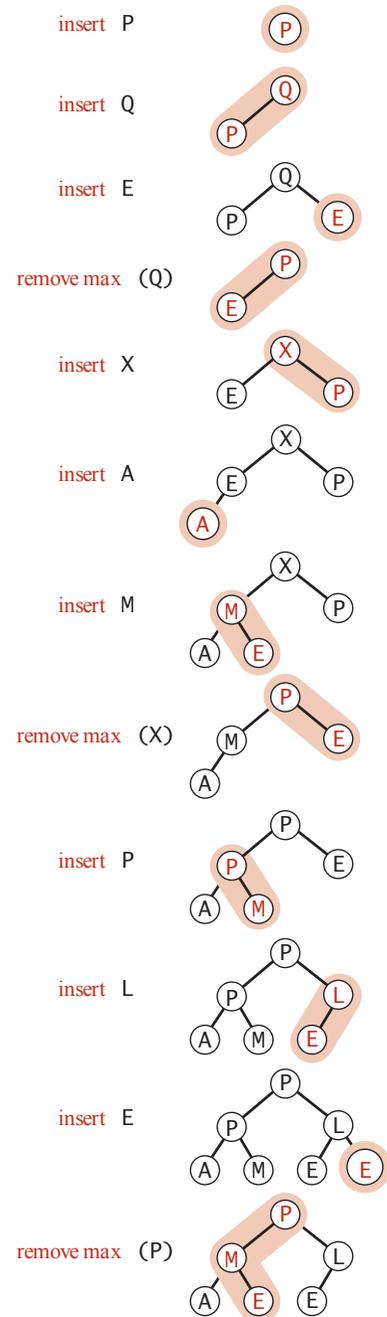
The priority queue is maintained in a heap-ordered complete binary tree in the array `pq[]` with `pq[0]` unused and the `N` keys in the priority queue in `pq[1]` through `pq[N]`. To implement `insert()`, we increment `N`, add the new element at the end, then use `swim()` to restore the heap order. For `delMax()`, we take the value to be returned from `pq[1]`, then move `pq[N]` to `pq[1]`, decrement the size of the heap, and use `sink()` to restore the heap condition. We also set the now-unused position `pq[N+1]` to `null` to allow the system to reclaim the memory associated with it. Code for dynamic array resizing is omitted, as usual (see SECTION 1.3). See EXERCISE 2.4.19 for the other constructors.

Proposition Q. In an N -key priority queue, the heap algorithms require no more than $1 + \lg N$ compares for insert and no more than $2 \lg N$ compares for *remove the maximum*.

Proof: By PROPOSITION B, both operations involve moving along a path between the root and the bottom of the heap whose number of links is no more than $\lg N$. The *remove the maximum* operation requires two compares for each node on the path (except at the bottom): one to find the child with the larger key, the other to decide whether that child needs to be promoted.

For typical applications that require a large number of intermixed insert and remove the maximum operations in a large priority queue, PROPOSITION Q represents an important performance breakthrough, summarized in the table shown on page 312. Where elementary implementations using an ordered array or an unordered array require linear time for one of the operations, a heap-based implementation provides a guarantee that both operations complete in logarithmic time. This improvement can make the difference between solving a problem and not being able to address it at all.

Multiway heaps. It is not difficult to modify our code to build heaps based on an array representation of complete heap-ordered ternary trees, with an entry at position k larger than or equal to entries at positions $3k-1$, $3k$, and $3k+1$ and smaller than or equal to entries at position $\lfloor (k+1)/3 \rfloor$, for all indices between 1 and N in an array of N items, and not much more difficult to use d -ary heaps for any given d . There is a tradeoff between the lower cost from the reduced tree height ($\log_d N$) and the higher cost of finding the largest of the d children at each node. This tradeoff is dependent on details of the implementation and the expected relative frequency of operations.



Priority queue operations in a heap

Array resizing. We can add a no-argument constructor, code for array doubling in `insert()`, and code for array halving in `delMax()`, just as we did for stacks in SECTION 1.3. Thus, clients need not be concerned about arbitrary size restrictions. The logarithmic time bounds implied by PROPOSITION Q are *amortized* when the size of the priority queue is arbitrary and the arrays are resized (see EXERCISE 2.4.22).

Immutability of keys. The priority queue contains objects that are created by clients but assumes that client code does not change the keys (which might invalidate the heap-order invariant). It is possible to develop mechanisms to enforce this assumption, but programmers typically do not do so because they complicate the code and are likely to degrade performance.

Index priority queue. In many applications, it makes sense to allow clients to refer to items that are already on the priority queue. One easy way to do so is to associate a unique integer index with each item. Moreover, it is often the case that clients have a universe of items of a known size N and perhaps are using (parallel) arrays to store information about the items, so other unrelated client code might already be using an integer index to refer to items. These considerations lead us to the following API:

```
public class IndexMinPQ<Key extends Comparable<Key>>
```

<code>IndexMinPQ(int maxN)</code>	<i>create a priority queue of capacity maxN with possible indices between 0 and maxN-1</i>
<code>void insert(int i, Key key)</code>	<i>insert key; associate it with index i</i>
<code>void changeKey(int k, Key key)</code>	<i>change the key associated with i to key</i>
<code>boolean contains(int i)</code>	<i>is index i associated with some key?</i>
<code>void delete(int i)</code>	<i>remove i and its associated key</i>
<code>Key minKey()</code>	<i>return a minimal key</i>
<code>int minIndex()</code>	<i>return a minimal key's index</i>
<code>int delMin()</code>	<i>remove a minimal key and return its index</i>
<code>boolean isEmpty()</code>	<i>is the priority queue empty?</i>
<code>int size()</code>	<i>number of keys in the priority queue</i>
<code>Key keyOf(int i)</code>	<i>return key associated with index i</i>

API for a generic priority queue with associated indices

A useful way of thinking of this data type is as implementing an array, but with fast access to the smallest entry in the array. Actually it does even better—it gives fast access to the minimum entry in a *specified subset* of an array’s entries (the ones that have been inserted). In other words, you can think of an `IndexMinPQ` named `pq` as representing a subset of an array `pq[0..N-1]` of items. Think of the call `pq.insert(i, key)` as adding `i` to the subset and setting `pq[i] = key` and the call `pq.changeKey(i, key)` as setting `pq[i] = key`, both also maintaining data structures needed to support the other operations, most importantly `delMin()` (remove and return the index of the minimum key) and `changeKey()` (change the item associated with an index that is already in the data structure—just as in `pq[i] = key`). These operations are important in many applications and are enabled by our ability to refer to the key (with the index). EXERCISE 2.4.33 describes how to extend ALGORITHM 2.6 to implement index priority queues with remarkable efficiency and with remarkably little code. Intuitively, when an item in the heap changes, we can restore the heap invariant with a sink operation (if the key decreases) and a swim operation (if the key increases). To perform the operations, we use the index to find the item in the heap. The ability to locate an item in the heap also allows us to add the `delete()` operation to the API.

Proposition Q (continued). In an index priority queue of size N , the number of compares required is proportional to at most $\log N$ for `insert`, `change priority`, `delete`, and `remove the minimum`.

Proof: Immediate from inspection of the code and the fact that all paths in a heap are of length at most $\sim \lg N$.

This discussion is for a minimum-oriented queue; as usual, we also implement on the booksite a maximum-oriented version `IndexMaxPQ`.

Index priority-queue client. The `IndexMinPQ` client `Multway` on page 322 solves the multiway merge problem: it merges together several sorted input streams into one sorted output stream. This problem arises in many applications: the streams might be the output of scientific instruments (sorted by time), lists of information from the web such as music or movies (sorted by title or artist name), commercial transactions (sorted by account number or time), or whatever. If you have the space, you might just read them all into an array and sort them, but with a priority queue, you can read input streams and put them in sorted order on the output *no matter how long they are*.

operation	order of growth of number of compares
<code>insert()</code>	$\log N$
<code>changeKey()</code>	$\log N$
<code>contains()</code>	1
<code>delete()</code>	$\log N$
<code>minKey()</code>	1
<code>minIndex()</code>	1
<code>delMin()</code>	$\log N$

Worst-case costs for an N -item heap-based indexed priority queue

Multiway merge priority-queue client

```

public class Multiway
{
    public static void merge(In[] streams)
    {
        int N = streams.length;
        IndexMinPQ<String> pq = new IndexMinPQ<String>(N);
        for (int i = 0; i < N; i++)
            if (!streams[i].isEmpty())
                pq.insert(i, streams[i].readString());
        while (!pq.isEmpty())
        {
            StdOut.println(pq.minKey());
            int i = pq.delMin();
            if (!streams[i].isEmpty())
                pq.insert(i, streams[i].readString());
        }
    }
    public static void main(String[] args)
    {
        int N = args.length;
        In[] streams = new In[N];
        for (int i = 0; i < N; i++)
            streams[i] = new In(args[i]);
        merge(streams);
    }
}

```

This `IndexMinPQ` client merges together the sorted input streams given as command-line arguments into a single sorted output stream on standard output (see text). Each stream index is associated with a key (the next string in the stream). After initialization, it enters a loop that prints the smallest string in the queue and removes the corresponding entry, then adds a new entry for the next string in that stream. For economy, the output is shown on one line below—the actual output is one string per line.

```

% more m1.txt
A B C F G I I Z
% more m2.txt
B D H P Q Q
% more m3.txt
A B E F J N

```

```

% java Multiway m1.txt m2.txt m3.txt
A A B B B C D E F F G H I I J N P Q Q Z

```

Heapsort We can use any priority queue to develop a sorting method. We insert all the items to be sorted into a minimum-oriented priority queue, then repeatedly use *remove the minimum* to remove them all in order. Using a priority queue represented as an unordered array in this way corresponds to doing a selection sort; using an ordered array corresponds to doing an insertion sort. What sorting method do we get if we use a heap? An entirely different one! Next, we use the heap to develop a classic elegant sorting algorithm known as *heapsort*.

Heapsort breaks into two phases: *heap construction*, where we reorganize the original array into a heap, and the *sortdown*, where we pull the items out of the heap in decreasing order to build the sorted result. For consistency with the code we have studied, we use a maximum-oriented priority queue and repeatedly remove the maximum. Focusing on the task of sorting, we abandon the notion of hiding the representation of the priority queue and use `swim()` and `sink()` directly. Doing so allows us to sort an array without needing any extra space, by maintaining the heap within the array to be sorted.

Heap construction. How difficult is the process of building a heap from N given items? Certainly we can accomplish this task in time proportional to $N \log N$, by proceeding from left to right through the array, using `swim()` to ensure that the items to the left of the scanning pointer make up a heap-ordered complete tree, like successive priority-queue insertions. A clever method that is much more efficient is to proceed from right to left, using `sink()` to make subheaps as we go. Every position in the array is the root of a small subheap; `sink()` works for such subheaps, as well. If the two children of a node are heaps, then calling `sink()` on that node makes the subtree rooted at the parent a heap. This process establishes the heap order inductively. The scan starts halfway back through the array because we can skip the subheaps of size 1. The scan ends at position 1, when we finish building the heap with one call to `sink()`. As the first phase of a sort, heap construction is a bit counterintuitive, because its goal is to produce a heap-ordered result, which has the largest item first in the array (and other larger items near the beginning), not at the end, where it is destined to finish.

Proposition R. Sink-based heap construction uses fewer than $2N$ compares and fewer than N exchanges to construct a heap from N items.

Proof: This fact follows from the observation that most of the heaps processed are small. For example, to build a heap of 127 items, we process 32 heaps of size 3, 16 heaps of size 7, 8 heaps of size 15, 4 heaps of size 31, 2 heaps of size 63, and 1 heap of size 127, so $32 \cdot 1 + 16 \cdot 2 + 8 \cdot 3 + 4 \cdot 4 + 2 \cdot 5 + 1 \cdot 6 = 120$ exchanges (twice as many compares) are required (at worst). See EXERCISE 2.4.20 for a complete proof.

ALGORITHM 2.7 Heapsort

```

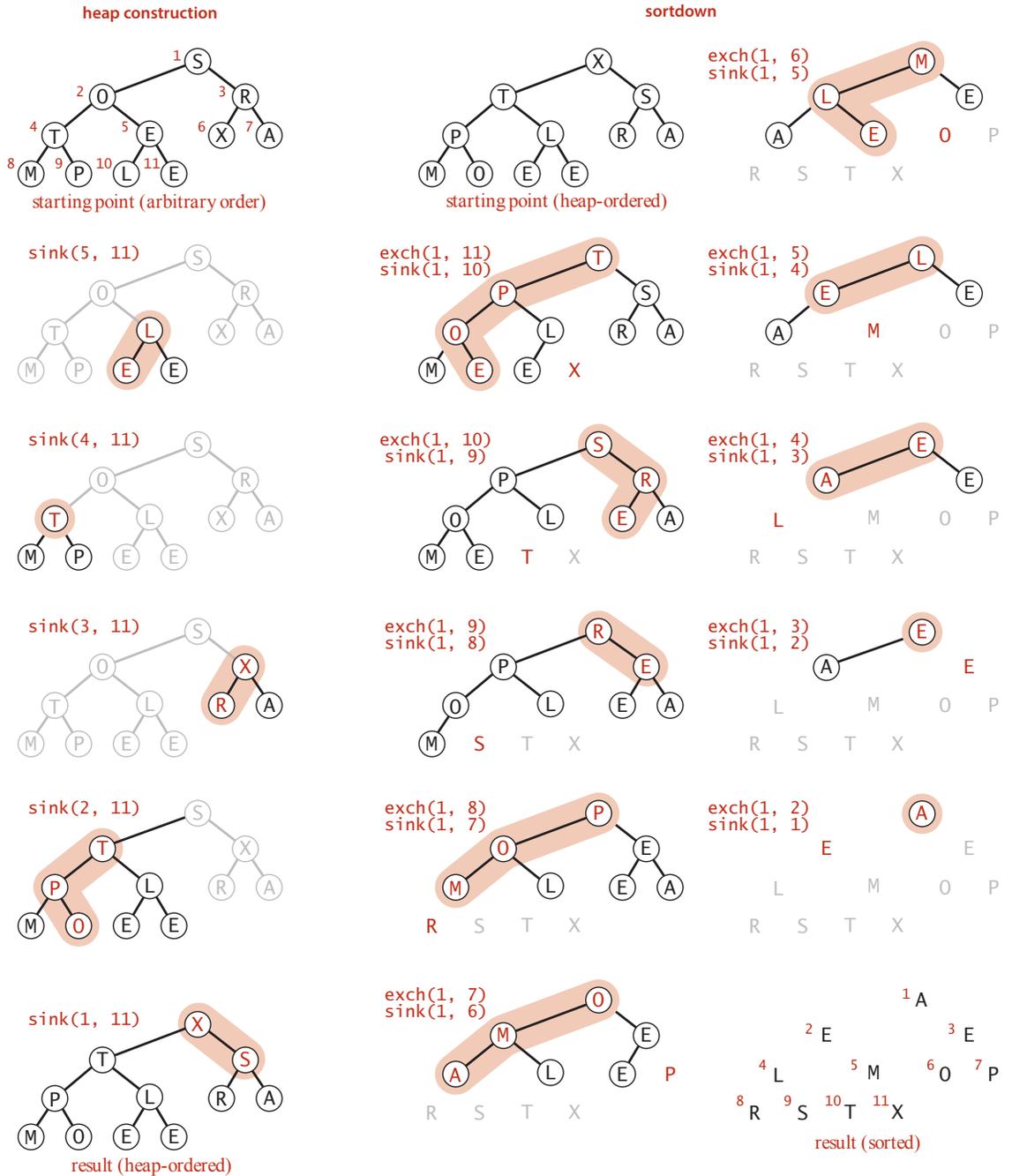
public static void sort(Comparable[] a)
{
    int N = a.length;
    for (int k = N/2; k >= 1; k--)
        sink(a, k, N);
    while (N > 1)
    {
        exch(a, 1, N--);
        sink(a, 1, N);
    }
}

```

This code sorts $a[1]$ through $a[N]$ using the `sink()` method (modified to take $a[]$ and N as arguments). The `for` loop constructs the heap; then the `while` loop exchanges the largest element $a[1]$ with $a[N]$ and then repairs the heap, continuing until the heap is empty. Decrementing the array indices in the implementations of `exch()` and `less()` gives an implementation that sorts $a[0]$ through $a[N-1]$, consistent with our other sorts.

		a[i]											
N	k	0	1	2	3	4	5	6	7	8	9	10	11
<i>initial values</i>		S	O	R	T	E	X	A	M	P	L	E	E
11	5	S	O	R	T	L	X	A	M	P	E	E	
11	4	S	O	R	T	L	X	A	M	P	E	E	
11	3	S	O	X	T	L	R	A	M	P	E	E	
11	2	S	T	X	P	L	R	A	M	O	E	E	
11	1	X	T	S	P	L	R	A	M	O	E	E	
<i>heap-ordered</i>		X	T	S	P	L	R	A	M	O	E	E	
10	1	T	P	S	O	L	R	A	M	E	E	X	
9	1	S	P	R	O	L	E	A	M	E	T	X	
8	1	R	P	E	O	L	E	A	M	S	T	X	
7	1	P	O	E	M	L	E	A	R	S	T	X	
6	1	O	M	E	A	L	E	P	R	S	T	X	
5	1	M	L	E	A	E	O	P	R	S	T	X	
4	1	L	E	E	A	M	O	P	R	S	T	X	
3	1	E	A	E	L	M	O	P	R	S	T	X	
2	1	E	A	E	L	M	O	P	R	S	T	X	
1	1	A	E	E	L	M	O	P	R	S	T	X	
<i>sorted result</i>		A	E	E	L	M	O	P	R	S	T	X	

Heapsort trace (array contents just after each sink)



Heapsort: constructing (left) and sorting down (right) a heap



Sortdown. Most of the work during heapsort is done during the second phase, where we remove the largest remaining item from the heap and put it into the array position vacated as the heap shrinks. This process is a bit like selection sort (taking the items in decreasing order instead of in increasing order), but it uses many fewer compares because the heap provides a much more efficient way to find the largest item in the unsorted part of the array.

Proposition S. Heapsort uses fewer than $2N \lg N + 2N$ compares (and half that many exchanges) to sort N items.

Proof: The $2N$ term covers the cost of heap construction (see PROPOSITION R). The $2N \lg N$ term follows from bounding the cost of each sink operation during the sort-down by $2 \lg N$ (see PROPOSITION Q).

ALGORITHM 2.7 is a full implementation based on these ideas, the classical *heapsort* algorithm, which was invented by J. W. J. Williams and refined by R. W. Floyd in 1964. Although the loops in this program seem to do different tasks (the first constructs the heap, and the second destroys the heap for the sortdown), they are both built around the `sink()` method. We provide an implementation outside of our priority-queue API to highlight the simplicity of the sorting algorithm (eight lines of code for `sort()` and another eight lines of code for `sink()`) and to make it an in-place sort.

As usual, you can gain some insight into the operation of the algorithm by studying a visual trace. At first, the process seems to do anything but sort, because large items are moving to the beginning of the array as the heap is being constructed. But then the method looks more like a mirror image of selection sort (except that it uses far fewer compares).

As for all of the other methods that we have studied, various people have investigated ways to improve heap-based priority-queue implementations and heapsort. We now briefly consider one of them.

Sink to the bottom, then swim. Most items reinserted into the heap during sortdown go all the way to the bottom. Floyd observed in 1964 that we can thus save time by avoiding the check for whether the item has reached its position, simply promoting the larger of the two children until the bottom is reached, then moving back up the heap to the proper position. This idea cuts the number of compares by a factor of 2 asymptotically—close to the number used by mergesort (for a randomly-ordered array). The method requires extra bookkeeping, and it is useful in practice only when the cost of compares is relatively high (for example, when we are sorting items with strings or other types of long keys).

HEAPSORT IS SIGNIFICANT in the study of the complexity of sorting (page 279) because it is the only method that we have seen that is optimal (within a constant factor) in its use of both time and space—it is guaranteed to use $\sim 2N \lg N$ compares and constant extra space in the worst case. When space is very tight (for example, in an embedded system or on a low-cost mobile device) it is popular because it can be implemented with just a few dozen lines (even in machine code) while still providing optimal performance. However, it is rarely used in typical applications on modern systems because it has poor cache performance: array entries are rarely compared with nearby array entries, so the number of cache misses is far higher than for quicksort, mergesort, and even shellsort, where most compares are with nearby entries.

On the other hand, the use of heaps to implement priority queues plays an increasingly important role in modern applications, because it provides an easy way to guarantee logarithmic running time for dynamic situations where large numbers of insert and *remove the maximum* operations are intermixed. We will encounter several examples later in this book.

Q&A

Q. I'm still not clear on the purpose of priority queues. Why exactly don't we just sort and then consider the items in increasing order in the sorted array?

A. In some data-processing examples such as `TopM` and `MuItiway`, the total amount of data is far too large to consider sorting (or even storing in memory). If you are looking for the top ten entries among a billion items, do you really want to sort a billion-entry array? With a priority queue, you can do it with a ten-entry priority queue. In other examples, all the data does not even exist together at any point in time: we take something from the priority queue, process it, and as a result of processing it perhaps add some more things to the priority queue.

Q. Why not use `Comparable`, as we do for sorts, instead of the generic `Item` in `MaxPQ`?

A. Doing so would require the client to cast the return value of `delMax()` to an actual type, such as `String`. Generally, casts in client code are to be avoided.

Q. Why not use `a[0]` in the heap representation?

A. Doing so simplifies the arithmetic a bit. It is not difficult to implement the heap methods based on a 0-based heap where the children of `a[0]` are `a[1]` and `a[2]`, the children of `a[1]` are `a[3]` and `a[4]`, the children of `a[2]` are `a[5]` and `a[6]`, and so forth, but most programmers prefer the simpler arithmetic that we use. Also, using `a[0]` as a sentinel value (in the parent of `a[1]`) is useful in some heap applications.

Q. Building a heap in heapsort by inserting items one by one seems simpler to me than the tricky bottom-up method described on page 323 in the text. Why bother?

A. For a sort implementation, it is 20 percent faster and requires half as much tricky code (no `swim()` needed). The difficulty of understanding an algorithm has not necessarily much to do with its simplicity, or its efficiency.

Q. What happens if I leave off the `extends Comparable<Key>` phrase in an implementation like `MaxPQ`?

A. As usual, the easiest way for you to answer a question of this sort for yourself is to simply try it. If you do so for `MaxPQ` you will get a compile-time error:

```
MaxPQ.java:21: cannot find symbol
  symbol   : method compareTo(Key)
```

which is Java's way of telling you that it does not know about `compareTo()` in `Item` because you neglected to declare that `Key` extends `Comparable<Key>`.

EXERCISES

2.4.1 Suppose that the sequence P R I O * R * * I * T * Y * * * Q U E * * * U * E (where a letter means insert and an asterisk means *remove the maximum*) is applied to an initially empty priority queue. Give the sequence of letters returned by the *remove the maximum* operations.

2.4.2 Criticize the following idea: To implement *find the maximum* in constant time, why not use a stack or a queue, but keep track of the maximum value inserted so far, then return that value for *find the maximum*?

2.4.3 Provide priority-queue implementations that support insert and *remove the maximum*, one for each of the following underlying data structures: unordered array, ordered array, unordered linked list, and ordered linked list. Give a table of the worst-case bounds for each operation for each of your four implementations.

2.4.4 Is an array that is sorted in decreasing order a max-oriented heap?

2.4.5 Give the heap that results when the keys E A S Y Q U E S T I O N are inserted in that order into an initially empty max-oriented heap.

2.4.6 Using the conventions of EXERCISE 2.4.1, give the sequence of heaps produced when the operations P R I O * R * * I * T * Y * * * Q U E * * * U * E are performed on an initially empty max-oriented heap.

2.4.7 The largest item in a heap must appear in position 1, and the second largest must be in position 2 or position 3. Give the list of positions in a heap of size 31 where the k th largest (i) can appear, and (ii) cannot appear, for $k=2, 3, 4$ (assuming the values to be distinct).

2.4.8 Answer the previous exercise for the k th smallest item.

2.4.9 Draw all of the different heaps that can be made from the five keys A B C D E, then draw all of the different heaps that can be made from the five keys A A A B B.

2.4.10 Suppose that we wish to avoid wasting one position in a heap-ordered array $pq[]$, putting the largest value in $pq[0]$, its children in $pq[1]$ and $pq[2]$, and so forth, proceeding in level order. Where are the parents and children of $pq[k]$?

2.4.11 Suppose that your application will have a huge number of insert operations, but only a few *remove the maximum* operations. Which priority-queue implementation do you think would be most effective: heap, unordered array, or ordered array?

EXERCISES *(continued)*

2.4.12 Suppose that your application will have a huge number of *find the maximum* operations, but a relatively small number of *insert* and *remove the maximum* operations. Which priority-queue implementation do you think would be most effective: heap, unordered array, or ordered array?

2.4.13 Describe a way to avoid the $j < N$ test in `sink()`.

2.4.14 What is the minimum number of items that must be exchanged during a *remove the maximum* operation in a heap of size N with no duplicate keys? Give a heap of size 15 for which the minimum is achieved. Answer the same questions for two and three successive *remove the maximum* operations.

2.4.15 Design a linear-time certification algorithm to check whether an array `pq[]` is a min-oriented heap.

2.4.16 For $N=32$, give arrays of items that make heapsort use as many and as few compares as possible.

2.4.17 Prove that building a minimum-oriented priority queue of size k then doing $N - k$ *replace the minimum* (*insert* followed by *remove the minimum*) operations leaves the k largest of the N items in the priority queue.

2.4.18 In MaxPQ, suppose that a client calls `insert()` with an item that is larger than all items in the queue, and then immediately calls `delMax()`. Assume that there are no duplicate keys. Is the resulting heap identical to the heap as it was before these operations? Answer the same question for two `insert()` operations (the first with a key larger than all keys in the queue and the second for a key larger than that one) followed by two `delMax()` operations.

2.4.19 Implement the constructor for MaxPQ that takes an array of items as argument, using the bottom-up heap construction method described on page 323 in the text.

2.4.20 Prove that sink-based heap construction uses fewer than $2N$ compares and fewer than N exchanges.

CREATIVE PROBLEMS

2.4.21 *Elementary data structures.* Explain how to use a priority queue to implement the stack, queue, and randomized queue data types from SECTION 1.3 and EXERCISE 1.3.35.

2.4.22 *Array resizing.* Add array resizing to MaxPQ, and prove bounds like those of PROPOSITION Q for array accesses, in an amortized sense.

2.4.23 *Multiway heaps.* Considering the cost of compares only, and assuming that it takes t compares to find the largest of t items, find the value of t that minimizes the coefficient of $N \lg N$ in the compare count when a t -ary heap is used in heapsort. First, assume a straightforward generalization of `sink()`; then, assume that Floyd's method can save one compare in the inner loop.

2.4.24 *Priority queue with explicit links.* Implement a priority queue using a heap-ordered binary tree, but use a triply linked structure instead of an array. You will need three links per node: two to traverse down the tree and one to traverse up the tree. Your implementation should guarantee logarithmic running time per operation, even if no maximum priority-queue size is known ahead of time.

2.4.25 *Computational number theory.* Write a program that prints out all integers of the form $a^3 + b^3$ where a and b are integers between 0 and N in sorted order, without using excessive space. That is, instead of computing an array of the N^2 sums and sorting them, build a minimum-oriented priority queue, initially containing $(0^3, 0, 0)$, $(1^3, 1, 0)$, $(2^3, 2, 0)$, \dots , $(N^3, N, 0)$. Then, while the priority queue is nonempty, *remove the smallest item* $(i^3 + j^3, i, j)$, print it, and then, if $j < N$, *insert the item* $(i^3 + (j+1)^3, i, j+1)$. Use this program to find all distinct integers a , b , c , and d between 0 and 10^6 such that $a^3 + b^3 = c^3 + d^3$.

2.4.26 *Heap without exchanges.* Because the `exch()` primitive is used in the `sink()` and `swim()` operations, the items are loaded and stored twice as often as necessary. Give more efficient implementations that avoid this inefficiency, a la insertion sort (see EXERCISE 2.1.25).

2.4.27 *Find the minimum.* Add a `min()` method to MaxPQ. Your implementation should use constant time and constant extra space.

2.4.28 *Selection filter.* Write a program similar to TopM that reads points (x, y, z) from standard input, takes a value M from the command line, and prints the M points that

CREATIVE PROBLEMS *(continued)*

are closest to the origin in Euclidean distance. Estimate the running time of your client for $N = 10^8$ and $M = 10^4$.

2.4.29 *Min/max priority queue.* Design a data type that supports the following operations: *insert*, *delete the maximum*, and *delete the minimum* (all in logarithmic time); and *find the maximum* and *find the minimum* (both in constant time). *Hint:* Use two heaps.

2.4.30 *Dynamic median-finding.* Design a data type that supports *insert* in logarithmic time, *find the median* in constant time, and *delete the median* in logarithmic time. *Hint:* Use a min-heap and a max-heap.

2.4.31 *Fast insert.* Develop a compare-based implementation of the `MinPQ` API such that *insert* uses $\sim \log \log N$ compares and *delete the minimum* uses $\sim 2 \log N$ compares. *Hint:* Use binary search on parent pointers to find the ancestor in `swim()`.

2.4.32 *Lower bound.* Prove that it is impossible to develop a compare-based implementation of the `MinPQ` API such that both *insert* and *delete the minimum* guarantee to use $\sim \log \log N$ compares per operation.

2.4.33 *Index priority-queue implementation.* Implement the basic operations in the index priority-queue API on page 320 by modifying `ALGORITHM 2.6` as follows: Change `pq[]` to hold indices, add an array `keys[]` to hold the key values, and add an array `qp[]` that is the inverse of `pq[]` — `qp[i]` gives the position of `i` in `pq[]` (the index `j` such that `pq[j]` is `i`). Then modify the code in `ALGORITHM 2.6` to maintain these data structures. Use the convention that `qp[i] = -1` if `i` is not on the queue, and include a method `contains()` that tests this condition. You need to modify the helper methods `exch()` and `less()` but not `sink()` or `swim()`.

Partial solution:

```

public class IndexMinPQ<Key extends Comparable<Key>>
{
    private int N;           // number of elements on PQ
    private int[] pq;       // binary heap using 1-based indexing
    private int[] qp;       // inverse: qp[pq[i]] = pq[qp[i]] = i
    private Key[] keys;     // items with priorities
    public IndexMinPQ(int maxN)
    {
        keys = (Key[]) new Comparable[maxN + 1];
        pq = new int[maxN + 1];
        qp = new int[maxN + 1];
        for (int i = 0; i <= maxN; i++) qp[i] = -1;
    }

    public boolean isEmpty()
    { return N == 0; }

    public boolean contains(int i)
    { return qp[i] != -1; }

    public void insert(int i, Key key)
    {
        N++;
        qp[i] = N;
        pq[N] = i;
        keys[i] = key;
        swim(N);
    }

    public Key minKey()
    { return keys[pq[1]]; }

    public int delMin()
    {
        int indexOfMin = pq[1];
        exch(1, N--);
        sink(1);
        keys[pq[N+1]] = null;
        qp[pq[N+1]] = -1;
        return indexOfMin;
    }
}

```

CREATIVE PROBLEMS *(continued)*

2.4.34 *Index priority-queue implementation (additional operations).* Add `minIndex()`, `changeKey()`, and `delete()` to your implementation of EXERCISE 2.4.33.

Solution:

```
public int minIndex()
{ return pq[1]; }

public void changeKey(int i, Key key)
{
    keys[i] = key;
    swim(qp[i]);
    sink(qp[i]);
}

public void delete(int i)
{
    int index = qp[i];
    exch(index, N--);
    swim(index);
    sink(index);
    keys[i] = null;
    qp[i] = -1;
}
```

2.4.35 *Sampling from a discrete probability distribution.* Write a class `Sample` with a constructor that takes an array `p[]` of `double` values as argument and supports the following two operations: `random()`—return an index `i` with probability $p[i]/T$ (where T is the sum of the numbers in `p[]`)—and `changeKey(i, v)`—change the value of `p[i]` to `v`. *Hint:* Use a complete binary tree where each node has implied weight $p[i]$. Store in each node the cumulative weight of all the nodes in its subtree. To generate a random index, pick a random number between 0 and T and use the cumulative weights to determine which branch of the subtree to explore. When updating `p[i]`, change all of the weights of the nodes on the path from the root to `i`. Avoid explicit pointers, as we do for heaps.

EXPERIMENTS

2.4.36 *Performance driver I.* Write a performance driver client program that uses `insert` to fill a priority queue, then uses `remove the maximum` to remove half the keys, then uses `insert` to fill it up again, then uses `remove the maximum` to remove all the keys, doing so multiple times on random sequences of keys of various lengths ranging from small to large; measures the time taken for each run; and prints out or plots the average running times.

2.4.37 *Performance driver II.* Write a performance driver client program that uses `insert` to fill a priority queue, then does as many `remove the maximum` and `insert` operations as it can do in 1 second, doing so multiple times on random sequences of keys of various lengths ranging from small to large; and prints out or plots the average number of `remove the maximum` operations it was able to do.

2.4.38 *Exercise driver.* Write an exercise driver client program that uses the methods in our priority-queue interface of ALGORITHM 2.6 on difficult or pathological cases that might turn up in practical applications. Simple examples include keys that are already in order, keys in reverse order, all keys the same, and sequences of keys having only two distinct values.

2.4.39 *Cost of construction.* Determine empirically the percentage of time heapsort spends in the construction phase for $N = 10^3$, 10^6 , and 10^9 .

2.4.40 *Floyd's method.* Implement a version of heapsort based on Floyd's sink-to-the-bottom-and-then-swim idea, as described in the text. Count the number of compares used by your program and the number of compares used by the standard implementation, for randomly ordered distinct keys with $N = 10^3$, 10^6 , and 10^9 .

2.4.41 *Multiway heaps.* Implement a version of heapsort based on complete heap-ordered 3-ary and 4-ary trees, as described in the text. Count the number of compares used by each and the number of compares used by the standard implementation, for randomly ordered distinct keys with $N = 10^3$, 10^6 , and 10^9 .

2.4.42 *Preorder heaps.* Implement a version of heapsort based on the idea of representing the heap-ordered tree in preorder rather than in level order. Count the number of compares used by your program and the number of compares used by the standard implementation, for randomly ordered keys with $N = 10^3$, 10^6 , and 10^9 .



2.5 APPLICATIONS

`SORTING ALGORITHMS` and priority queues are widely used in a broad variety of applications. Our purpose in this section is to briefly survey some of these applications, consider ways in which the efficient methods that we have considered play a critical role in such applications, and discuss some of the steps needed to make use of our sort and priority-queue code.

A prime reason why sorting is so useful is that it is much easier to search for an item in a sorted array than in an unsorted one. For over a century, people found it easy to look up someone's phone number in a phone book where items are sorted by last name. Now digital music players organize song files by artist name or song title; search engines display search results in descending order of importance; spreadsheets display columns sorted by a particular field; matrix-processing packages sort the real eigenvalues of a symmetric matrix in descending order; and so forth. Other tasks are also made easier once an array is in sorted order: from looking up an item in the sorted index in the back of this book; to removing duplicates from a long list such as a mailing list, a list of voters, or a list of websites; to performing statistical calculations such as removing outliers, finding the median, or computing percentiles.

Sorting also arises as a critical subproblem in many applications that appear to have nothing to do with sorting at all. Data compression, computer graphics, computational biology, supply-chain management, combinatorial optimization, social choice, and voting are but a few of many examples. The algorithms that we have considered in this chapter play a critical role in the development of effective algorithms in each of the later chapters in this book.

Most important is the system sort, so we begin by considering a number of practical considerations that come into play when building a sort for use by a broad variety of clients. While some of these topics are specific to Java, they each reflect challenges that need to be met in any system.

Our primary purpose is to demonstrate that, even though we have used mechanisms that are relatively simple, the sorting implementations that we are studying are widely applicable. The list of proven applications of fast sorting algorithms is vast, so we can consider just a small fraction of them: some scientific, some algorithmic, and some commercial. You will find many more examples in the exercises, and many more than that on the booksite. Moreover, we will often refer back to this chapter to effectively address the problems that we later consider *in this book!*

Sorting various types of data Our implementations sort arrays of `Comparable` objects. This Java convention allows us to use Java's callback mechanism to sort arrays of objects of any type that implements the `Comparable` interface. As described in SECTION 2.1, implementing `Comparable` amounts to defining a `compareTo()` method that implements a natural ordering for the type. We can use our code immediately to sort arrays of type `String`, `Integer`, `Double`, and other types such as `File` and `URL`, because these data types all implement `Comparable`. Being able to use the same code for all of those types is convenient, but typical applications involve working with data types that are defined for use within the application. Accordingly it is common to implement a `compareTo()` method for user-defined data types, so that they implement `Comparable`, thus enabling client code to sort arrays of that type (and build priority queues of values of that type).

Transaction example. A prototypical breeding ground for sorting applications is commercial data processing. For example, imagine that a company engaged in internet commerce maintains a record for each transaction involving a customer account that contains all of the pertinent information, such as the customer name, date, amount, and so forth. Nowadays, a successful company needs to be able to handle millions and millions of such transactions. As we saw in EXERCISE 2.1.21, it is reasonable to decide that a natural ordering of such transactions is that they be ordered by amount, which we can implement by adding an appropriate `compareTo()` method in the class definition. With such a definition, we could process an array `a[]` of `Transactions` by, for example, first sorting it with the call `Quick.sort(a)`. Our sorting methods know nothing about our `Transaction` data type, but Java's `Comparable` interface allows us to define a natural ordering so that we can use any of our methods to sort `Transaction` objects. Alternatively, we might specify that `Transaction` objects are to be ordered by date by implementing `compareTo()` to compare the `Date` fields. Since `Date` objects are themselves `Comparable`, we can just invoke the `compareTo()` method in `Date` rather than having to implement it from scratch. It is also reasonable to consider ordering this data by its customer field; arranging to allow clients the flexibility to switch among multiple different orders is an interesting challenge that we will soon consider.

```
public int compareTo(Transaction that)
{ return this.when.compareTo(that.when); }
```

Alternate `compareTo()` implementation for sorting transactions by date

Pointer sorting. The approach we are using is known in the classical literature as pointer sorting, so called because we process references to items and do not move the data itself. In programming languages such as C and C++, programmers explicitly decide whether to manipulate data or pointers to data; in Java, pointer manipulation is implicit. Except for primitive numeric types, we always manipulate references to objects (pointers), not the objects themselves. Pointer sorting adds a level of indirection: the array contains references to the objects to be sorted, not the objects themselves. We briefly consider some associated issues, in the context of sorting. With multiple reference arrays, we can have multiple different sorted representations of different parts of a single body of data (perhaps using multiple keys, as described below).

Keys are immutable. It stands to reason that an array might not remain sorted if a client is allowed to change the values of keys after the sort. Similarly, a priority queue can hardly be expected to operate properly if the client can change the values of keys between operations. In Java, it is wise to ensure that key values do not change by using immutable keys. Most of the standard data types that you are likely to use as keys, such as `String`, `Integer`, `Double`, and `File`, are immutable.

Exchanges are inexpensive. Another advantage of using references is that we avoid the cost of moving full items. The cost saving is significant for arrays with large items (and small keys) because the compare needs to access just a small part of the item, and most of the item is not even touched during the sort. The reference approach makes the cost of an exchange roughly equal to the cost of a compare for general situations involving arbitrarily large items (at the cost of the extra space for the references). Indeed, if the keys are long, the exchanges might even wind up being less costly than the compare. One way to study the performance of algorithms that sort arrays of numbers is to simply look at the total number of compares and exchanges they use, implicitly making the assumption that the cost of exchanges is the same as the cost of compares. Conclusions based on this assumption are likely to apply to a broad class of applications in Java, because we are sorting reference objects.

Alternate orderings. There are many applications where we want to use different orders for the objects that we are sorting, depending on the situation. The `Java Comparator` interface allows us to build multiple orders within a single class. It has a single public method `compare()` that compares two objects. If we have a data type that implements this interface, we can pass a `Comparator` to `sort()` (which passes it to `less()`) as in the example on the next page. The `Comparator` mechanism allows us to sort arrays of any type of object, using any total order that we wish to define for them. Using a `Comparator` instead of working with `Comparable` types better separates the definition of the type from the definition of what it means to compare two objects of

that type. Indeed, there are typically many possible ways to compare objects, and the `Comparator` mechanism allows us to choose among them. For instance, to sort an array `a[]` of strings without regard to whether characters are uppercase or lowercase you can just call `Insertion.sort(a, String.CASE_INSENSITIVE_ORDER)` which makes use of the `CASE_INSENSITIVE_ORDER` comparator defined in Java's `String` class. As you can imagine, the precise rules for ordering strings are complicated and quite different for various natural languages, so Java has many `String` comparators.

Items with multiple keys. In typical applications, items have multiple instance variables that might need to serve as sort keys. In our transaction example, one client may need to sort the transaction list by customer (for example, to bring together all transactions involving each customer); another client might need to sort the list by amount (for example, to identify high-value transactions); and other clients might need to use other fields as sort keys. The `Comparator` mechanism is precisely what we need to allow this flexibility. We can define multiple comparators, as in the alternate implementation of `Transaction` shown on the bottom of the next page. With this definition, a client can sort an array of `Transaction` objects by time with the call

```
Insertion.sort(a, new Transaction.WhenOrder())
```

or by amount with the call

```
Insertion.sort(a, new Transaction.HowMuchOrder()).
```

The sort does each compare through a callback to the `compare()` method in `Transaction` that is specified by the client code. To avoid the cost of making a new `Comparator` object for each sort, we could use `public static final` instance variables to define the comparators (as Java does for `CASE_INSENSITIVE_ORDER`).

```
public static void sort(Object[] a, Comparator c)
{
    int N = a.length;
    for (int i = 1; i < N; i++)
        for (int j = i; j > 0 && less(c, a[j], a[j-1]); j--)
            exch(a, j, j-1);
}

private static boolean less(Comparator c, Object v, Object w)
{ return c.compare(v, w) < 0; }

private static void exch(Object[] a, int i, int j)
{ Object t = a[i]; a[i] = a[j]; a[j] = t; }
```

Insertion sorting with a `Comparator`

Priority queues with comparators. The same flexibility to use comparators is also useful for priority queues. Extending our standard implementation in ALGORITHM 2.6 to support comparators involves the following steps:

- Import `java.util.Comparator`.
- Add to `MaxPQ` an instance variable `comparator` and a constructor that takes a comparator as argument and initializes `comparator` to that value.
- Add code to `less()` that checks whether `comparator` is `null` (and uses it if it is not `null`).

For example, with these changes, you could build different priority queues with `Transaction` keys, using the time, place, or account number for the ordering. If you remove the `Key extends Comparable<Key>` phrase from `MinPQ`, you even can support keys with no natural order.

```
import java.util.Comparator;

public class Transaction
{
    ...
    private final String who;
    private final Date when;
    private final double amount;
    ...
    public static class WhoOrder implements Comparator<Transaction>
    {
        public int compare(Transaction v, Transaction w)
        { return v.who.compareTo(w.who); }
    }

    public static class WhenOrder implements Comparator<Transaction>
    {
        public int compare(Transaction v, Transaction w)
        { return v.when.compareTo(w.when); }
    }

    public static class HowMuchOrder implements Comparator<Transaction>
    {
        public int compare(Transaction v, Transaction w)
        {
            if (v.amount < w.amount) return -1;
            if (v.amount > w.amount) return +1;
            return 0;
        }
    }
}
```

Comparator implementation for `Transaction` data type

Stability. A sorting method is stable if it preserves the relative order of equal keys in the array. This property is frequently important. For example, consider an internet commerce application where we have to process a large number of events that have locations and timestamps. To begin, suppose that we store events in an array as they arrive, so they are in order of the timestamp in the array. Now suppose that the application requires that the transactions be separated out by location for further processing. One easy way to do so is to sort the array by location. If the sort is unstable, the transactions for each city may not necessarily be in order by timestamp after the sort. Often, programmers who are unfamiliar with stability are surprised, when they first encounter the situation, by the way an unstable algorithm seems to scramble the data. Some of the sorting methods that we have considered in this chapter are stable (insertion sort and mergesort); many are not (selection sort, shellsort, quicksort, and heapsort). There are ways to trick any sort into stable behavior (see EXERCISE 2.5.18), but using a stable algorithm is generally preferable when stability is an essential requirement. It is easy to take stability for granted; actually, no practical method in common use achieves stability without using significant extra time or space (researchers have developed algorithms that do so, but applications programmers have judged them too complicated to be useful).

sorted by time	sorted by location (not stable)	sorted by location (stable)
Chicago 09:00:00	Chicago 09:25:52	Chicago 09:00:00
Phoenix 09:00:03	Chicago 09:03:13	Chicago 09:00:59
Houston 09:00:13	Chicago 09:21:05	Chicago 09:03:13
Chicago 09:00:59	Chicago 09:19:46	Chicago 09:19:32
Houston 09:01:10	Chicago 09:19:32	Chicago 09:19:46
Chicago 09:03:13	Chicago 09:00:00	Chicago 09:21:05
Seattle 09:10:11	Chicago 09:35:21	Chicago 09:25:52
Seattle 09:10:25	Chicago 09:00:59	Chicago 09:35:21
Phoenix 09:14:25	Houston 09:01:10	Houston 09:00:13
Chicago 09:19:32	Houston 09:00:13	Houston 09:01:10
Chicago 09:19:46	Phoenix 09:37:44	Phoenix 09:00:03
Chicago 09:21:05	Phoenix 09:00:03	Phoenix 09:14:25
Seattle 09:22:43	Phoenix 09:14:25	Phoenix 09:37:44
Seattle 09:22:54	Seattle 09:10:25	Seattle 09:10:11
Chicago 09:25:52	Seattle 09:36:14	Seattle 09:10:25
Chicago 09:35:21	Seattle 09:22:43	Seattle 09:22:43
Seattle 09:36:14	Seattle 09:10:11	Seattle 09:22:54
Phoenix 09:37:44	Seattle 09:22:54	Seattle 09:36:14

no
longer
sorted
by time

still
sorted
by time

Stability when sorting on a second key

Which sorting algorithm should I use? We have considered numerous sorting algorithms in this chapter, so this question is natural. Knowing which algorithm is best possible depends heavily on details of the application and implementation, but we have studied some general-purpose methods that can be nearly as effective as the best possible for a wide variety of applications.

The table at the bottom of this page is a general guide that summarizes the important characteristics of the sort algorithms that we have studied in this chapter. In all cases but shellsort (where the growth rate is only an estimate), insertion sort (where the growth rate depends on the order of the input keys), and both versions of quicksort (where the growth rate is probabilistic and may depend on the distribution of input key values), multiplying these growth rates by appropriate constants gives an effective way to predict running time. The constants involved are partly algorithm-dependent (for example, heapsort uses twice the number of compares as mergesort and both do many more array accesses than quicksort) but are primarily dependent on the implementation, the Java compiler, and your computer, which determine the number of machine instructions that are executed and the time that each requires. Most important, since they are constants, you can generally predict the running time for large N by running experiments for smaller N and extrapolating, using our standard doubling protocol.

algorithm	stable?	in place?	order of growth to sort N items		notes
			running time	extra space	
<i>selection sort</i>	no	yes	N^2	1	
<i>insertion sort</i>	yes	yes	between N and N^2	1	depends on order of items
<i>shellsort</i>	no	yes	$N \log N$? $N^{6/5}$?	1	
<i>quicksort</i>	no	yes	$N \log N$	$\lg N$	probabilistic guarantee
<i>3-way quicksort</i>	no	yes	between N and $N \log N$	$\lg N$	probabilistic, also depends on distribution of input keys
<i>mergesort</i>	yes	no	$N \log N$	N	
<i>heapsort</i>	no	yes	$N \log N$	1	

Performance characteristics of sorting algorithms

Property T. Quicksort is the fastest general-purpose sort.

Evidence: This hypothesis is supported by countless implementations of quicksort on countless computer systems since its invention decades ago. Generally, the reason that quicksort is fastest is that it has only a few instructions in its inner loop (and it does well with cache memories because it most often references data sequentially) so that its running time is $\sim c N \lg N$ with the value of c smaller than the corresponding constants for other linearithmic sorts. With 3-way partitioning, quicksort becomes linear for certain key distributions likely to arise in practice, where other sorts are linearithmic.

Thus, in most practical situations, quicksort is the method of choice. Still, given the broad reach of sorting and the broad variety of computers and systems, a flat statement like this is difficult to justify. For example, we have already seen one notable exception: if stability is important and space is available, mergesort might be best. We will see other exceptions in CHAPTER 5. With tools like `SortCompare` and a considerable amount of time and effort, you can do a more detailed study of comparative performance of these algorithms and the refinements that we have discussed for your computer, as discussed in several exercises at the end of this section. Perhaps the best way to interpret PROPERTY T is as saying that you certainly should seriously consider using quicksort in any sort application where running time is important.

Sorting primitive types. In some performance-critical applications, the focus may be on sorting numbers, so it is reasonable to avoid the costs of using references and sort primitive types instead. For example, consider the difference between sorting an array of `int` values and sorting an array of `Integer` values. In the former case, we exchange the numbers themselves and put them in order in the array; in the latter, we exchange references to `Integer` objects, which contain the numbers. If we are doing nothing more than sorting a huge array of numbers, we avoid paying the cost of storing an equal number of references plus the extra cost of accessing the numbers through the references, not to mention the cost of invoking `compareTo()` and `less()` methods. We can develop efficient versions of our sort codes for such purposes by replacing `Comparable` with the primitive type name, and redefining `less()` or just replacing calls to `less()` with code like `a[i] < a[j]` (see EXERCISE 2.1.26).

Java system sort. As an example of applying the information given in the table on page 342, consider Java's primary system sort method, `java.util.Arrays.sort()`. With overloading of argument types, this name actually represents a collection of methods:

- A different method for each primitive type
- A method for data types that implement `Comparable`
- A method that uses a `Comparator`

Java's systems programmers have chosen to use quicksort (with 3-way partitioning) to implement the primitive-type methods, and mergesort for reference-type methods. The primary practical implications of these choices are, as just discussed, to trade speed and memory usage (for primitive types) for guaranteed performance and stability (for reference types).

THE ALGORITHMS AND IDEAS that we have been considering are an essential part of many modern systems, including Java. When developing Java programs to address an application, you are likely to find that Java's `Arrays.sort()` implementations (perhaps supplemented by your own implementation(s) of `compareTo()` and/or `compare()`) will meet your needs, because you will be using 3-way quicksort or mergesort, both proven classic algorithms.

In this book, we generally will use our own `Quick.sort()` (usually) or `Merge.sort()` (when stability is important and space is not) in sort clients. You may feel free to use `Arrays.sort()` unless you have a good reason to use another specific method.

Reductions The idea that we can use sorting algorithms to solve other problems is an example of a basic technique in algorithm design known as reduction. We consider reduction in detail in CHAPTER 6 because of its importance in the theory of algorithms—in the meantime, we will consider several practical examples. A reduction is a situation where an algorithm developed for one problem is used to solve another. Applications programmers are quite used to the concept of reduction (whether or not it is explicitly articulated)—every time you make use of a method that solves problem B in order to solve problem A, you are doing a reduction from A to B. Indeed, one goal in implementing algorithms is to facilitate reductions by making the algorithms useful for as wide a variety as possible of applications. We begin with a few elementary examples for sorting. Many of these take the form of algorithmic puzzles where a quadratic brute-force algorithm is immediate. It is often the case that sorting the data first makes it easy to finish solving the problem in linear additional time, thus reducing the total cost from quadratic to linearithmic.

Duplicates. Are there any duplicate keys in an array of `Comparable` objects? How many distinct keys are there? Which value appears most frequently? For small arrays, these kinds of questions are easy to answer with a quadratic algorithm that compares each array entry with each other array entry. For large arrays, using a quadratic algorithm

is not feasible. With sorting, you can answer these questions in linearithmic time: first sort the array, then make a pass through the sorted array, taking note of duplicate keys that appear consecutively in the ordered array. For example, the code fragment at right counts the distinct keys in an array. With simple modifications to this code, you can answer the questions above and perform tasks such as printing all the distinct values, all the values that are duplicated, and so forth, even for huge arrays.

Rankings. A permutation (or ranking) is an array of N integers where each of the integers between 0 and $N - 1$ appears exactly once. The Kendall tau distance between two rankings is the number of pairs that are in different order in the two rankings. For example, the Kendall tau distance between 0 3 1 6 2 5 4 and 1 0 3 6 4 2 5 is four because the pairs

```
Quick.sort(a);
int count = 1; // Assume a.length > 0.
for (int i = 1; i < a.length; i++)
    if (a[i].compareTo(a[i-1]) != 0)
        count++;
```

Counting the distinct keys in a[]

0-1, 3-1, 2-4, 5-4 are in different relative order in the two rankings, but all other pairs are in the same relative order. This statistic is widely used: in sociology to study social choice and voting theory, in molecular biology to compare genes using expression profiles, and in ranking search engine results on the web, among many other applications. The Kendall tau distance between a permutation and the identity permutation (where each entry is equal to its index) is the number of inversions in the permutation, and a quadratic algorithm based on insertion sort to compute the distance is not difficult to devise (recall PROPOSITION C in SECTION 2.1). Efficiently computing the Kendall tau distance is an interesting exercise for a programmer (or a student!) who is familiar with the classical sorting algorithms that we have studied (see EXERCISE 2.5.19).

Priority-queue reductions. In SECTION 2.4, we considered two examples of problems that reduce to a sequence of operations on priority queues. TopM, on page 311, finds the M items in an input stream with the highest keys. Multiway, on page 322, merges M sorted input streams together to make a sorted output stream. Both of these problems are easily addressed with a priority queue of size M .

Median and order statistics. An important application related to sorting but for which a full sort is not required is the operation of finding the median of a collection of keys (the value with the property that half the keys are no larger and half the keys are no smaller). This operation is a common computation in statistics and in various other data-processing applications. Finding the median is a special case of selection: finding the k th smallest of a collection of numbers. Selection has many applications in the processing of experimental and other data. The use of the median and other order

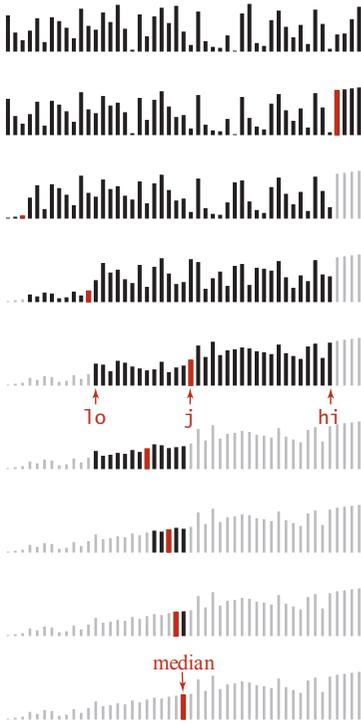
statistics to divide an array into smaller groups is common. Often, only a small part of a large array is to be saved for further processing; in such cases, a program that can select, say, the top 10 percent of the items of the array might be more appropriate than a full sort. Our TopM application of SECTION 2.4 solves this problem for an unbounded input stream, using a priority queue. An effective alternative to TopM when you have the items in an array is to just sort it: after the call `Quick.sort(a)` the k smallest items in the array are in the first k array positions for all k less than the array length. But this approach involves a sort, so the running time is linearithmic. Can we do better? Finding the k smallest items in an array is easy

when k is very small or very large, but more

challenging when k is a constant fraction of the array size, such as finding the median ($k = N/2$). You might be surprised to learn that it is possible to solve this problem in linear time, as in the `select()` method above (this implementation requires a client cast; for the more pedantic code needed to avoid this requirement, see the booksite). To do the job, `select()` maintains the variables `lo` and `hi` to delimit the subarray that contains the index k of the item to be selected and uses quick-sort partitioning to shrink the size of the subarray. Recall that `partition()` rearranges an array `a[lo]` through `a[hi]` and returns an integer j such that `a[lo]` through `a[j-1]` are less than or equal to `a[j]`, and `a[j+1]` through `a[hi]` are greater than or equal to `a[j]`. Now, if k is equal to j , then we are done. Otherwise, if $k < j$, then we need to continue working in the left subarray (by changing the value of `hi` to `j-1`); if $k > j$, then we need to continue working in the right subarray (by changing `lo` to `j+1`). The loop maintains the invariant that no entry to the left of `lo` is larger and no entry to the right of `hi` is smaller than any element within `a[lo..hi]`. After partitioning, we preserve this invariant and shrink the interval until it

```
public static Comparable
select(Comparable[] a, int k)
{
    StdRandom.shuffle(a);
    int lo = 0, hi = a.length - 1;
    while (hi > lo)
    {
        int j = partition(a, lo, hi);
        if (j == k) return a[k];
        else if (j > k) hi = j - 1;
        else if (j < k) lo = j + 1;
    }
    return a[k];
}
```

Selecting the k smallest items in `a[]`



Partitioning to find the median

consists just of k . Upon termination, $a[k]$ contains the $(k+1)$ st smallest entry, $a[0]$ through $a[k-1]$ are all smaller than (or equal to) $a[k]$, and $a[k+1]$ through the end of the array are all greater than (or equal to) $a[k]$. To gain some insight into why this is a linear-time algorithm, suppose that partitioning divides the array exactly in half each time. Then the number of compares is $N + N/2 + N/4 + N/8 + \dots$, terminating when the k th smallest item is found. This sum is less than $2N$. As with quicksort, it takes a bit of math to find the true bound, which is a bit higher. Also as with quicksort, the analysis depends on partitioning on a random item, so that the guarantee is probabilistic.

Proposition U. Partitioning-based selection is a linear-time algorithm, on average.

Proof: An analysis similar to, but significantly more complex than, the proof of PROPOSITION K for quicksort leads to the result that the average number of compares is $\sim 2N + 2k \ln(N/k) + 2(N-k) \ln(N/(N-k))$, which is linear for any allowed value of k . For example, this formula says that finding the median ($k = N/2$) requires $\sim (2 + 2 \ln 2)N$ compares, on the average. Note that the worst case is quadratic but randomization protects against that possibility, as with quicksort.

Designing a selection algorithm that is guaranteed to use a linear number of compares in the worst case is a classic result in computational complexity, but it has not yet led to a useful practical algorithm.

A brief survey of sorting applications Direct applications of sorting are familiar, ubiquitous, and far too numerous for us to list them all. You sort your music by song title or by artist name, your email or phone calls by time or origin, and your photos by date. Universities sort student accounts by name or ID. Credit card companies sort millions or even billions of transactions by date or amount. Scientists sort not only experimental data by time or other identifier but also to enable detailed simulations of the natural world, from the motion of particles or heavenly bodies to the structure of materials to social interactions and relationships. Indeed, it is difficult to identify a computational application that does not involve sorting! To elaborate upon this point, we describe in this section examples of applications that are more complicated than the reductions just considered, including several that we will examine in detail later in this book.

Commercial computing. The world is awash in information. Government organizations, financial institutions, and commercial enterprises organize much of this information by sorting it. Whether the information is accounts to be sorted by name or number, transactions to be sorted by date or amount, mail to be sorted by postal code or address, files to be sorted by name or date, or whatever, processing such data is sure to involve a sorting algorithm somewhere along the way. Typically, such information is organized in huge databases, sorted by multiple keys for efficient search. An effective strategy that is widely used is to collect new information, add it to the database, sort it on the keys of interest, and merge the sorted result for each key into the existing database. The methods that we have discussed have been used effectively since the early days of computing to build a huge infrastructure of sorted data and methods for processing it that serve as the basis for all of this commercial activity. Arrays having millions or even billions of entries are routinely processed today—without linearithmic sorting algorithms, such arrays could not be sorted, making such processing extremely difficult or impossible.

Search for information. Keeping data in sorted order makes it possible to efficiently search through it using the classic *binary search* algorithm (see CHAPTER 1). You will also see that the same scheme makes it easy to quickly handle many other kinds of queries. How many items are smaller than a given item? Which items fall within a given range? In CHAPTER 3, we consider such questions. We also consider in detail various extensions to sorting and binary search that allow us to intermix such queries with operations that insert and remove objects from the set, still guaranteeing logarithmic performance for all operations.

Operations research. The field of *operations research* (OR) develops and applies mathematical models for problem-solving and decision-making. We will see several examples in this book of relationships between OR and the study of algorithms, beginning here with the use of sorting in a classic OR problem known as *scheduling*. Suppose that we have N jobs to complete, where job j requires t_j seconds of processing time. We need to complete all of the jobs but want to maximize customer satisfaction by minimizing the average completion time of the jobs. The *shortest processing time first* rule, where we schedule the jobs in increasing order of processing time, is known to accomplish this goal. Therefore we can sort the jobs by processing time or put them on a minimum-oriented priority queue. With various other constraints and restrictions, we get various other scheduling problems, which frequently arise in industrial applications and are well-studied. As another example, consider the load-balancing problem, where we have M identical processors and N jobs to complete, and our goal is to schedule all of the jobs on the processors so that the time at which the last job completes is as early as possible. This specific problem is NP-hard (see CHAPTER 6) so we do not expect to find a practical way to compute an optimal schedule. One method that is known to produce a good schedule is the *longest processing time first* rule, where we consider the jobs in descending order of processing time, assigning each job to the processor that becomes available first. To implement this algorithm, we first sort the jobs in reverse order. Then we maintain a priority queue of M processors, where the priority is the sum of the processing times of its jobs. At each step, we delete the processor with the minimum priority, add the next job to the processor, and reinsert that processor into the priority queue.

Event-driven simulation. Many scientific applications involve simulation, where the point of the computation is to model some aspect of the real world in order to be able to better understand it. Before the advent of computing, scientists had little choice but to build mathematical models for this purpose; such models are now well-complemented by computational models. Doing such simulations efficiently can be challenging, and use of appropriate algorithms certainly can make the difference between being able to complete the simulation in a reasonable amount of time and being stuck with the choice of accepting inaccurate results or waiting for the simulation to do the computation necessary to get accurate results. We will consider in CHAPTER 6 a detailed example that illustrates this point.

Numerical computations. Scientific computing is often concerned with accuracy (how close are we to the true answer?). Accuracy is extremely important when we are performing millions of computations with estimated values such as the floating-point representation of real numbers that we commonly use on computers. Some numerical algorithms use priority queues and sorting to control accuracy in calculations. For

example, one way to do numerical integration (quadrature), where the goal is to estimate the area under a curve, is to maintain a priority queue with accuracy estimates for a set of subintervals that comprise the whole interval. The process is to remove the least accurate subinterval, split it in half (thus achieving better accuracy for the two halves), and put the two halves back onto the priority queue, continuing until a desired tolerance is reached.

Combinatorial search. A classic paradigm in artificial intelligence and in coping with intractable problems is to define a set of *configurations* with well-defined moves from one configuration to the next and a priority associated with each move. Also defined is a start configuration and a goal configuration (which corresponds to having solved the problem). The well-known *A** algorithm is a problem-solving process where we put the start configuration on the priority queue, then do the following until reaching the goal: remove the highest-priority configuration and add to the queue all configurations that can be reached from that with one move (excluding the one just removed). As with event-driven simulation, this process is tailor-made for priority queues. It reduces solving the problem to defining an effective priority function. See EXERCISE 2.5.32 for an example.

BEYOND SUCH DIRECT APPLICATIONS (and we have only indicated a small fraction of those), sorting and priority queues are an essential abstraction in algorithm design, so they will surface frequently throughout this book. We next list some examples of applications from later in the book. All of these applications depend upon the efficient implementations of sorting algorithms and the priority-queue data type that we have considered in this chapter.

Prim's algorithm and Dijkstra's algorithm are classical algorithms from CHAPTER 4. That chapter is about algorithms that process *graphs*, a fundamental model for items and edges that connect pairs of items. The basis for these and several other algorithms is *graph search*, where we proceed from item to item along edges. Priority queues play a fundamental role in organizing graph searches, enabling efficient algorithms.

Kruskal's algorithm is another classic algorithm for graphs whose edges have weights that depends upon processing the edges in order of their weight. Its running time is dominated by the cost of the sort.

Huffman compression is a classic data compression algorithm that depends upon processing a set of items with integer weights by combining the two smallest to produce a new one whose weight is the sum of its two constituents. Implementing this opera-

tion is immediate, using a priority queue. Several other data-compression schemes are based upon sorting.

String-processing algorithms, which are of critical importance in modern applications in cryptology and in genomics, are often based on sorting (generally using one of the specialized string sorts discussed in CHAPTER 5). For example, we will discuss in CHAPTER 6 algorithms for finding the longest repeated substring in a given string that is based on first sorting suffixes of the strings.

Q & A

Q. Is there a priority-queue data type in the Java library?

A. Yes, see `java.util.PriorityQueue`.

Q. Does stability matter when sorting arrays primitive types in Java?

A. Stability is a (mostly) meaningless concept when applied to primitive types because you cannot distinguish between two equal `int` or `double` values. There is one exotic exception that arises because there are multiple representations of NaN but they are considered equal when sorting.

EXERCISES

2.5.1 Consider the following implementation of the `compareTo()` method for `String`. How does the third line help with efficiency?

```
public int compareTo(String that)
{
    if (this == that) return 0; // this line
    int n = Math.min(this.length(), that.length());
    for (int i = 0; i < n; i++)
    {
        if (this.charAt(i) < that.charAt(i)) return -1;
        else if (this.charAt(i) > that.charAt(i)) return +1;
    }
    return this.length() - that.length();
}
```

2.5.2 Write a program that reads a list of words from standard input and prints all two-word compound words in the list. For example, if `after`, `thought`, and `afterthought` are in the list, then `afterthought` is a compound word.

2.5.3 Criticize the following implementation of a class intended to represent account balances. Why is `compareTo()` a flawed implementation of the `Comparable` interface?

```
public class Balance implements Comparable<Balance>
{
    ...
    private double amount;
    public int compareTo(Balance that)
    {
        if (this.amount < that.amount - 0.005) return -1;
        if (this.amount > that.amount + 0.005) return +1;
        return 0;
    }
    ...
}
```

Describe a way to fix this problem.

2.5.4 Implement a method `String[] dedup(String[] a)` that returns the objects in `a[]` in sorted order, with duplicates removed.

2.5.5 Explain why selection sort is not stable.

EXERCISES *(continued)*

2.5.6 Implement a recursive version of `select()`.

2.5.7 About how many compares are required, on the average, to find the smallest of N items using `select()`?

2.5.8 Write a program `Frequency` that reads strings from standard input and prints the number of times each string occurs, in descending order of frequency.

2.5.9 Develop a data type that allows you to write a client that can sort a file such as the one shown at right.

2.5.10 Create a data type `Version` that represents a software version number, such as `115.1.1`, `115.10.1`, `115.10.2`. Implement the `Comparable` interface so that `115.1.1` is less than `115.10.1`, and so forth.

2.5.11 One way to describe the result of a sorting algorithm is to specify a permutation `p[]` of the numbers `0` to `a.length-1`, such that `p[i]` specifies where the key originally in `a[i]` ends up. Give the permutations that describe the results of insertion sort, selection sort, shellsort, mergesort, quicksort, and heapsort for an array of seven equal keys.

input (DJI A volumes for each day)

```
1-Oct-28      3500000
2-Oct-28      3850000
3-Oct-28      4060000
4-Oct-28      4330000
5-Oct-28      4360000
...
30-Dec-99     554680000
31-Dec-99     374049984
3-Jan-00      931800000
4-Jan-00     1009000000
5-Jan-00     1085500032
...
```

output

```
19-Aug-40    130000
26-Aug-40    160000
24-Jul-40    200000
10-Aug-42    210000
23-Jun-42    210000
...
23-Jul-02    2441019904
17-Jul-02    2566500096
15-Jul-02    2574799872
19-Jul-02    2654099968
24-Jul-02    2775559936
```

CREATIVE PROBLEMS

2.5.12 *Scheduling.* Write a program `SPT.java` that reads job names and processing times from standard input and prints a schedule that minimizes average completion time using the shortest processing time first rule, as described on page 349.

2.5.13 *Load balancing.* Write a program `LPT.java` that takes an integer `M` as a command-line argument, reads job names and processing times from standard input and prints a schedule assigning the jobs to `M` processors that approximately minimizes the time when the last job completes using the longest processing time first rule, as described on page 349.

2.5.14 *Sort by reverse domain.* Write a data type `Domain` that represents domain names, including an appropriate `compareTo()` method where the natural order is in order of the reverse domain name. For example, the reverse domain of `cs.princeton.edu` is `edu.princeton.cs`. This is useful for web log analysis. *Hint:* Use `s.split("\\.")` to split the string `s` into tokens, delimited by dots. Write a client that reads domain names from standard input and prints the reverse domains in sorted order.

2.5.15 *Spam campaign.* To initiate an illegal spam campaign, you have a list of email addresses from various domains (the part of the email address that follows the `@` symbol). To better forge the return addresses, you want to send the email from another user at the same domain. For example, you might want to forge an email from `wayne@princeton.edu` to `rs@princeton.edu`. How would you process the email list to make this an efficient task?

2.5.16 *Unbiased election.* In order to thwart bias against candidates whose names appear toward the end of the alphabet, California sorted the candidates appearing on its 2003 gubernatorial ballot by using the following order of characters:

R W Q O J M V A H B S G Z X N T C I E K U P D Y F L

Create a data type where this is the natural order and write a client `California` with a single static method `main()` that sorts strings according to this ordering. Assume that each string is composed solely of uppercase letters.

2.5.17 *Check stability.* Extend your `check()` method from EXERCISE 2.1.16 to call `sort()` for a given array and return `true` if `sort()` sorts the array in order in a stable manner, `false` otherwise. Do not assume that `sort()` is restricted to move data only with `exch()`.

CREATIVE PROBLEMS *(continued)*

2.5.18 *Force stability.* Write a wrapper method that makes any sort stable by creating a new key type that allows you to append each key's index to the key, call `sort()`, then restore the original key after the sort.

2.5.19 *Kendall tau distance.* Write a program `KendallTau.java` that computes the Kendall tau distance between two permutations in linearithmic time.

2.5.20 *Idle time.* Suppose that a machine processes N jobs. Write a program that, given the list of job start and finish times, finds the largest interval where the machine is idle and the largest interval where the machine is not idle.

2.5.21 *Multidimensional sort.* Write a `Vector` data type for use in having the sorting methods sort multidimensional vectors of d integers, putting the vectors in order by first component, those with equal first component in order by second component, those with equal first and second components in order by third component, and so forth.

2.5.22 *Stock market trading.* Investors place buy and sell orders for a particular stock on an electronic exchange, specifying a maximum buy or minimum sell price that they are willing to pay, and how many shares they wish to trade at that price. Develop a program that uses priority queues to match up buyers and sellers and test it through simulation. Maintain two priority queues, one for buyers and one for sellers, executing trades whenever a new order can be matched with an existing order or orders.

2.5.23 *Sampling for selection.* Investigate the idea of using sampling to improve selection. *Hint:* Using the median may not always be helpful.

2.5.24 *Stable priority queue.* Develop a stable priority-queue implementation (which returns duplicate keys in the same order in which they were inserted).

2.5.25 *Points in the plane.* Write three `static` comparators for the `Point2D` data type of page 77, one that compares points by their x coordinate, one that compares them by their y coordinate, and one that compares them by their distance from the origin. Write two `non-static` comparators for the `Point2D` data type, one that compares them by their distance to a specified point and one that compares them by their polar angle with respect to a specified point.

2.5.26 *Simple polygon.* Given N points in the plane, draw a simple polygon with the

N points as vertices. *Hint*: Find the point p with the smallest y coordinate, breaking ties with the smallest x coordinate. Connect the points in increasing order of the polar angle they make with p .

2.5.27 *One-dimensional intervals.* Write three comparators for the `Interval1D` data type of page 77, one that compares intervals by their left endpoint, one that compares intervals by their right endpoint, and one that compares intervals by their length.

2.5.28 *Sort files by name.* Write a program `FileSorter` that takes the name of a directory as a command-line argument and prints out all of the files in that directory, sorted by file name. *Hint*: Use the `File` data type.

2.5.29 *Sort files by size and date of last modification.* Write comparators for the type `File` to order by increasing/decreasing order of file size, ascending/descending order of file name, and ascending/descending order of last modification date. Use these comparators in a program `LS` that takes a command-line argument and lists the files in the current directory according to a specified order, e.g., `"-t"` to sort by timestamp. Support multiple flags to break ties. Be sure to use a stable sort.

2.5.30 *Boerner's theorem.* True or false: If you sort each column of a matrix, then sort each row, the columns are still sorted. Justify your answer.

EXPERIMENTS

2.5.31 *Distinct values.* Write a client that takes integers M , N , and T as command-line arguments, then uses the code given in the text to perform T trials of the following experiment: Generate N random `int` values between 0 and $M - 1$ and count the number of distinct values. Run your program for $T = 10$ and $N = 10^3, 10^4, 10^5$, and 10^6 , with $M = N/2$, and N , and $2N$. Probability theory says that the number of distinct values should be about $M(1 - e^{-\alpha})$ where $\alpha = N/M$ —print a table to help you confirm that your experiments validate that formula.

2.5.32 *8 puzzle.* The 8 puzzle is a game invented and popularized by Noyes Palmer Chapman in the 1870s. It is played on a 3-by-3 grid with 8 tiles labeled 1 through 8 and a blank square. Your goal is to rearrange the tiles so that they are in order. You are permitted to slide one of the available tiles horizontally or vertically (but not diagonally) into the blank square. Write a program that solves the puzzle using the A^* algorithm. Start by using as priority the sum of the number of moves made to get to this board position plus the number of tiles in the wrong position. (Note that the number of moves you must make from a given board position is at least as big as the number of tiles in the wrong place.) Investigate substituting other functions for the number of tiles in the wrong position, such as the sum of the Manhattan distance from each tile to its correct position, or the sums of the squares of these distances.

2.5.33 *Random transactions.* Develop a generator that takes an argument N , generates N random `Transaction` objects (see EXERCISES 2.1.21 and 2.1.22), using assumptions about the transactions that you can defend. Then compare the performance of shellsort, mergesort, quicksort, and heapsort for sorting N transactions, for $N=10^3, 10^4, 10^5$, and 10^6 .

This page intentionally left blank

THREE



Searching

3.1	Symbol Tables	362
3.2	Binary Search Trees	396
3.3	Balanced Search Trees	424
3.4	Hash Tables	458
3.5	Applications.	486

Modern computing and the internet have made accessible a vast amount of information. The ability to efficiently search through this information is fundamental to processing it. This chapter describes classical *searching* algorithms that have proven to be effective in numerous diverse applications for decades. Without algorithms like these, the development of the computational infrastructure that we enjoy in the modern world would not have been possible.

We use the term symbol table to describe an abstract mechanism where we save information (a value) that we can later search for and retrieve by specifying a key. The nature of the keys and the values depends upon the application. There can be a huge number of keys and a huge amount of information, so implementing an efficient symbol table is a significant computational challenge.

Symbol tables are sometimes called dictionaries, by analogy with the time-honored system of providing definitions for words by listing them alphabetically in a reference book. In an English-language dictionary, a key is a word and its value is the entry associated with the word that contains the definition, pronunciation, and etymology. Symbol tables are also sometimes called indices, by analogy with another time-honored system of providing access to terms by listing them alphabetically at the end of a book such as a textbook. In a book index, a key is a term of interest and its value is the list of page numbers that tell readers where to find that term in the book.

After describing the basic APIs and two fundamental implementations, we consider three classic data structures that can support efficient symbol-table implementations: binary search trees, red-black trees, and hash tables. We conclude with several extensions and applications, many of which would not be feasible without the efficient algorithms that you will learn about in this chapter.

3.1 SYMBOL TABLES

The primary purpose of a symbol table is to associate a value with a key. The client can insert key-value pairs into the symbol table with the expectation of later being able to *search* for the value associated with a given key, from among all of the key-value pairs that have been put into the table. This chapter describes several ways to structure this data so as to make efficient not just the insert and *search* operations, but several other convenient operations as well. To implement a symbol table, we need to define an underlying data structure and then specify algorithms for insert, search, and other operations that create and manipulate the data structure.

Search is so important to so many computer applications that symbol tables are available as high-level abstractions in many programming environments, including Java—we shall discuss Java’s symbol-table implementations in SECTION 3.5. The table below gives some examples of keys and values that you might use in typical applications. We consider some illustrative reference clients soon, and SECTION 3.5 is devoted to showing you how to use symbol tables effectively in your own clients. We also use symbol tables in developing other algorithms throughout the book.

Definition. A symbol table is a data structure for key-value pairs that supports two operations: insert (put) a new pair into the table and *search* for (get) the value associated with a given key.

application	purpose of search	key	value
<i>dictionary</i>	find definition	word	definition
<i>book index</i>	find relevant pages	term	list of page numbers
<i>file share</i>	find song to download	name of song	computer ID
<i>account management</i>	process transactions	account number	transaction details
<i>web search</i>	find relevant web pages	keyword	list of page names
<i>compiler</i>	find type and value	variable name	type and value

Typical symbol-table applications

API The symbol table is a prototypical abstract data type (see CHAPTER 1): it represents a well-defined set of values and operations on those values, enabling us to develop clients and implementations separately. As usual, we precisely define the operations by specifying an applications programming interface (API) that provides the contract between client and implementation:

<code>public class ST<Key, Value></code>	
<code>ST()</code>	<i>create a symbol table</i>
<code>void put(Key key, Value val)</code>	<i>put key-value pair into the table (remove key from table if value is null)</i>
<code>Value get(Key key)</code>	<i>value paired with key (null if key is absent)</i>
<code>void delete(Key key)</code>	<i>remove key (and its value) from table</i>
<code>boolean contains(Key key)</code>	<i>is there a value paired with key?</i>
<code>boolean isEmpty()</code>	<i>is the table empty?</i>
<code>int size()</code>	<i>number of key-value pairs in the table</i>
<code>Iterable<Key> keys()</code>	<i>all the keys in the table</i>

API for a generic basic symbol table

Before examining client code, we consider several design choices for our implementations to make our code consistent, compact, and useful.

Generics. As we did with sorting, we will consider the methods without specifying the types of the items being processed, using generics. For symbol tables, we emphasize the separate roles played by keys and values in search by specifying the key and value types explicitly instead of viewing keys as implicit in items as we did for priority queues in SECTION 2.4. After we have considered some of the characteristics of this basic API (for example, note that there is no mention of order among the keys), we will consider an extension for the typical case when keys are `Comparable`, which enables numerous additional methods.

Duplicate keys. We adopt the following conventions in all of our implementations:

- Only one value is associated with each key (no duplicate keys in a table).
- When a client puts a key-value pair into a table already containing that key (and an associated value), the new value replaces the old one.

These conventions define the associative array abstraction, where you can think of a symbol table as being just like an array, where keys are indices and values are array

entries. In a conventional array, keys are integer indices that we use to quickly access array values; in an associative array (symbol table), keys are of arbitrary type, but we can still use them to quickly access values. Some programming languages (not Java) provide special support that allows programmers to use code such as `st[key]` for `st.get(key)` and `st[key] = val` for `st.put(key, val)` where `key` and `val` are objects of arbitrary type.

Null keys. Keys must not be `null`. As with many mechanisms in Java, use of a null key results in an exception at runtime (see the third Q&A on page 387).

Null values. We also adopt the convention that no key can be associated with the value `null`. This convention is directly tied to our specification in the API that `get()` should return `null` for keys not in the table, effectively associating the value `null` with every key not in the table. This convention has two (intended) consequences: First, we can test whether or not the symbol table defines a value associated with a given key by testing whether `get()` returns `null`. Second, we can use the operation of calling `put()` with `null` as its second (value) argument to implement deletion, as described in the next paragraph.

Deletion. Deletion in symbol tables generally involves one of two strategies: *lazy* deletion, where we associate keys in the table with `null`, then perhaps remove all such keys at some later time; and *eager* deletion, where we remove the key from the table immediately. As just discussed, the code `put(key, null)` is an easy (lazy) implementation of `delete(key)`. When we give an (eager) implementation of `delete()`, it is intended to replace this default. In our symbol-table implementations that do not use the default `delete()`, the `put()` implementations on the booksite begin with the defensive code

```
if (val == null) { delete(key); return; }
```

to ensure that no key in the table is associated with `null`. For economy, we do not include this code in the book (and we do not call `put()` with a `null` value in client code).

Shorthand methods. For clarity in client code, we include the methods `contains()` and `isEmpty()` in the API, with the default one-line implementations shown here. For economy, we do not repeat this code, but we assume it to be present in all implementations of the symbol-table API and use these methods freely in client code.

method	default implementation
<code>void delete(Key key)</code>	<code>put(key, null);</code>
<code>boolean contains(Key key)</code>	<code>return get(key) != null;</code>
<code>boolean isEmpty()</code>	<code>return size() == 0;</code>

Default implementations

Iteration. To enable clients to process all the keys and values in the table, we might add the phrase `implements Iterable<Key>` to the first line of the API to specify that every implementation must implement an `iterator()` method that returns an iterator having appropriate implementations of `hasNext()` and `next()`, as described for stacks and queues in SECTION 1.3. For symbol tables, we adopt a simpler alternative approach, where we specify a `keys()` method that returns an `Iterable<Key>` object for clients to use to iterate through the keys. Our reason for doing so is to maintain consistency with methods that we will define for ordered symbol tables that allow clients to iterate through a specified subset of keys in the table.

Key equality. Determining whether or not a given key is in a symbol table is based on the concept of *object equality*, which we discussed at length in SECTION 1.2 (see page 102). Java's convention that all objects inherit an `equals()` method and its implementation of `equals()` both for standard types such as `Integer`, `Double`, and `String` and for more complicated types such as `File` and `URL` is a head start—when using these types of data, you can just use the built-in implementation. For example, if `x` and `y` are `String` values, then `x.equals(y)` is `true` if and only if `x` and `y` have the same length and are identical in each character position. For such client-defined keys, you need to override `equals()`, as discussed in SECTION 1.2. You can use our implementation of `equals()` for `Date` (page 103) as a template to develop `equals()` for a type of your own. As discussed for priority queues on page 320, a best practice is to make `Key` types immutable, because consistency cannot otherwise be guaranteed.

Ordered symbol tables In typical applications, keys are `Comparable` objects, so the option exists of using the code `a.compareTo(b)` to compare two keys `a` and `b`. Several symbol-table implementations take advantage of order among the keys that is implied by `Comparable` to provide efficient implementations of the `put()` and `get()` operations. More important, in such implementations, we can think of the symbol table as *keeping the keys in order* and consider a significantly expanded API that defines numerous natural and useful operations involving relative key order. For example, suppose that your keys are times of the day. You might be interested in knowing the earliest or the latest time, the set of keys that fall between two given times, and so forth. In most cases, such operations are not difficult to implement with the same data structures and methods underlying the `put()` and `get()` implementations. Specifically, for applications where keys are `Comparable`, we implement in this chapter the following API:

<code>public class ST<Key extends Comparable<Key>, Value></code>	
<code>ST()</code>	<i>create an ordered symbol table</i>
<code>void put(Key key, Value val)</code>	<i>put key-value pair into the table (remove key from table if value is null)</i>
<code>Value get(Key key)</code>	<i>value paired with key (null if key is absent)</i>
<code>void delete(Key key)</code>	<i>remove key (and its value) from table</i>
<code>boolean contains(Key key)</code>	<i>is there a value paired with key?</i>
<code>boolean isEmpty()</code>	<i>is the table empty?</i>
<code>int size()</code>	<i>number of key-value pairs</i>
<code>Key min()</code>	<i>smallest key</i>
<code>Key max()</code>	<i>largest key</i>
<code>Key floor(Key key)</code>	<i>largest key less than or equal to key</i>
<code>Key ceiling(Key key)</code>	<i>smallest key greater than or equal to key</i>
<code>int rank(Key key)</code>	<i>number of keys less than key</i>
<code>Key select(int k)</code>	<i>key of rank k</i>
<code>void deleteMin()</code>	<i>delete smallest key</i>
<code>void deleteMax()</code>	<i>delete largest key</i>
<code>int size(Key lo, Key hi)</code>	<i>number of keys in [lo..hi]</i>
<code>Iterable<Key> keys(Key lo, Key hi)</code>	<i>keys in [lo..hi], in sorted order</i>
<code>Iterable<Key> keys()</code>	<i>all keys in the table, in sorted order</i>

API for a generic ordered symbol table

Your signal that one of our programs is implementing this API is the presence of the `Key` extends `Comparable<Key>` generic type variable in the class declaration, which specifies that the code depends upon the keys being `Comparable` and implements the richer set of operations. Together, these operations define for client programs an *ordered symbol table*.

Minimum and maximum. Perhaps the most natural queries for a set of ordered keys are to ask for the smallest and largest keys. We have already encountered these operations, in our discussion of priority queues in SECTION 2.4. In ordered symbol tables, we also have methods to delete the maximum and minimum keys (and their associated values). With this capability, the symbol table can operate like the `IndexMinPQ()` class that we discussed in SECTION 2.4. The primary differences are that equal keys are allowed in priority queues but not in symbol tables and that ordered symbol tables support a much larger set of operations.

Floor and ceiling. Given a key, it is often useful to be able to perform the *floor* operation (find the largest key that is less than or equal to the given key) and the *ceiling* operation (find the smallest key that is greater than or equal to the given key). The nomenclature comes from functions defined on real numbers (the floor of a real number x is the largest integer that is smaller than or equal to x and the ceiling of a real number x is the smallest integer that is greater than or equal to x).

Rank and selection. The basic operations for determining where a new key fits in the order are the rank operation (find the number of keys less than a given key) and the select operation (find the key with a given rank). To test your understanding of their meaning, confirm for yourself that both $i == \text{rank}(\text{select}(i))$ for all i between 0 and `size()-1` and all keys in the table satisfy $\text{key} == \text{select}(\text{rank}(\text{key}))$. We have already encountered the need for these operations, in our discussion of sort applications in SECTION 2.5. For symbol tables, our challenge is to perform these operations quickly, intermixed with insertions, deletions, and searches.

	keys	values
<code>min()</code> →	09:00:00	Chicago
	09:00:03	Phoenix
	09:00:13	Houston
<code>get(09:00:13)</code> →	09:00:59	Chicago
	09:01:10	Houston
<code>floor(09:05:00)</code> →	09:03:13	Chicago
	09:10:11	Seattle
<code>select(7)</code> →	09:10:25	Seattle
	09:14:25	Phoenix
	09:19:32	Chicago
	09:19:46	Chicago
<code>keys(09:15:00, 09:25:00)</code> →	09:21:05	Chicago
	09:22:43	Seattle
	09:22:54	Seattle
	09:25:52	Chicago
<code>ceiling(09:30:00)</code> →	09:35:21	Chicago
	09:36:14	Seattle
<code>max()</code> →	09:37:44	Phoenix

`size(09:15:00, 09:25:00)` is 5
`rank(09:10:25)` is 7

Examples of ordered symbol-table operations

Range queries. How many keys fall within a given range (between two given keys)? Which keys fall in a given range? The two-argument `size()` and `keys()` methods that answer these questions are useful in many applications, particularly in large databases. The capability to handle such queries is one prime reason that ordered symbol tables are so widely used in practice.

Exceptional cases. When a method is to return a key and there is no key fitting the description in the table, our convention is to throw an exception (an alternate approach, which is also reasonable, would be to return `null` in such cases). For example, `min()`, `max()`, `deleteMin()`, `deleteMax()`, `floor()`, and `ceiling()` all throw exceptions if the table is empty, as does `select(k)` if `k` is less than 0 or not less than `size()`.

Shorthand methods. As we have already seen with `isEmpty()` and `contains()` in our basic API, we keep some redundant methods in the API for clarity in client code. For economy in the text, we assume that the following default implementations are included in any implementation of the ordered symbol-table API unless otherwise specified:

method	default implementation
<code>void deleteMin()</code>	<code>delete(min());</code>
<code>void deleteMax()</code>	<code>delete(max());</code>
<code>int size(Key lo, Key hi)</code>	<pre> if (hi.compareTo(lo) < 0) return 0; else if (contains(hi)) return rank(hi) - rank(lo) + 1; else return rank(hi) - rank(lo); </pre>
<code>Iterable<Key> keys()</code>	<code>return keys(min(), max());</code>

Default implementations of redundant order-based symbol-table methods

Key equality (revisited). The best practice in Java is to make `compareTo()` consistent with `equals()` in all `Comparable` types. That is, for every pair of values `a` and `b` in any given `Comparable` type, it should be the case that `(a.compareTo(b) == 0)` and `a.equals(b)` have the same value. To avoid any potential ambiguities, we avoid the use of `equals()` in ordered symbol-table implementations. Instead, we use `compareTo()` exclusively to compare keys: we take the boolean expression `a.compareTo(b) == 0` to

mean “Are a and b equal?” Typically, such a test marks the successful end of a search for a in the symbol table (by finding b). As you saw with sorting algorithms, Java provides standard implementations of `compareTo()` for many commonly used types of keys, and it is not difficult to develop a `compareTo()` implementation for your own data types (see SECTION 2.5).

Cost model. Whether we use `equals()` (for symbol tables where keys are not `Comparable`) or `compareTo()` (for ordered symbol tables with `Comparable` keys), we use the term compare to refer to the operation of comparing a symbol-table entry against a search key. In most symbol-table implementations, this operation is in the inner loop. In the few cases where that is not the case, we also count array accesses.

SYMBOL-TABLE IMPLEMENTATIONS are generally characterized by their underlying data structures and their implementations of `get()` and `put()`. We do not always provide implementations of all of the other methods in the text because many of them make good exercises to test your understanding of the underlying data structures. To distinguish implementations, we add a descriptive prefix to ST that refers to the implementation in the class name of symbol-table implementations. In clients, we use ST to call on a reference implementation unless we wish to refer to a specific implementation. You will gradually develop a better feeling for the rationale behind the methods in the APIs in the context of the numerous clients and symbol-table implementations that we present and discuss throughout this chapter and throughout the rest of this book. We also discuss alternatives to the various design choices that we have described here in the Q&A and exercises.

Searching cost model.

When studying symbol-table implementations, we count compares (equality tests or key comparisons). In (rare) cases where compares are not in the inner loop, we count array accesses.

Sample clients While we defer detailed consideration of applications to SECTION 3.5, it is worthwhile to consider some client code before considering implementations. Accordingly, we now consider two clients: a test client that we use to trace algorithm behavior on small inputs and a performance client that we use to motivate the development of efficient implementations.

Test client. For tracing our algorithms on small inputs we assume that all of our implementations use the test client below, which takes a sequence of strings from standard input, builds a symbol table that associates the value i with the i th string in the input, and then prints the table. In the traces in the text, we assume that the input is a sequence

```
public static void main(String[] args)
{
    ST<String, Integer> st;
    st = new ST<String, Integer>();
    for (int i = 0; !StdIn.isEmpty(); i++)
    {
        String key = StdIn.readString();
        st.put(key, i);
    }
    for (String s : st.keys())
        StdOut.println(s + " " + st.get(s));
}
```

Basic symbol-table test client

keys	S	E	A	R	C	H	E	X	A	M	P	L	E
values	0	1	2	3	4	5	6	7	8	9	10	11	12

	output for basic symbol table (one possibility)	output for ordered symbol table
	L 11	A 8
	P 10	C 4
	M 9	E 12
	X 7	H 5
	H 5	L 11
	C 4	M 9
	R 3	P 10
	A 8	R 3
	E 12	S 0
	S 0	X 7

Keys, values, and output for test client

of single-character strings. Most often, we use the string "S E A R C H E X A M P L E". By our conventions, this client associates the key S with the value 0, the key R with the value 3, and so forth. But E is associated with the value 12 (not 1 or 6) and A with the value 8 (not 2) because our associative-array convention implies that each key is associated with the value used in the most recent call to `put()`. For basic (unordered) implementations, the order of the keys in the output of this test client is not specified (it depends on characteristics of the implementation); for an ordered symbol table the test client prints the keys in sorted order. This client is an example of an indexing client, a special case of a fundamental symbol-table application that we discuss in SECTION 3.5.

Performance client. `FrequencyCounter` (on the next page) is a symbol-table client that finds the number of occurrences of each string (having at least as many characters as a given threshold length) in a sequence of strings from standard input, then iterates through the keys to find the one that occurs the most frequently. This client is an example of a dictionary client, an application that we discuss in more detail in SECTION 3.5. This client answers a simple question: Which word (no shorter than a given length) occurs most frequently in a given text? Throughout this chapter, we measure performance of this client with three reference inputs: the first five lines of C. Dickens's *Tale of Two Cities* (`tinyTale.txt`), the full text of the book (`tale.txt`), and a popular database of 1 million sentences taken at random from the web that is known as the *Leipzig Corpora Collection* (`leipzig1M.txt`). For example, here is the content of `tinyTale.txt`:

```
% more tinyTale.txt
it was the best of times it was the worst of times
it was the age of wisdom it was the age of foolishness
it was the epoch of belief it was the epoch of incredulity
it was the season of light it was the season of darkness
it was the spring of hope it was the winter of despair
```

Small test input

This text has 60 words taken from 20 distinct words, four of which occur ten times (the highest frequency). Given this input, `FrequencyCounter` will print out any of `it`, `was`, `the`, or `of` (the one chosen may vary, depending upon characteristics of the symbol-table implementation), followed by the frequency, 10.

To study performance for the larger inputs, it is clear that two measures are of interest: Each word in the input is used as a search key once, so the total number of words in the text is certainly relevant. Second, each distinct word in the input is put into the

	tinyTale.txt		tale.txt		leipzig1M.txt	
	words	distinct	words	distinct	words	distinct
all words	60	20	135,635	10,679	21,191,455	534,580
at least 8 letters	3	3	14,350	5,737	4,239,597	299,593
at least 10 letters	2	2	4,582	2,260	1,610,829	165,555

Characteristics of larger test input streams

A symbol-table client

```
public class FrequencyCounter
{
    public static void main(String[] args)
    {
        int minlen = Integer.parseInt(args[0]); // key-length cutoff
        ST<String, Integer> st = new ST<String, Integer>();
        while (!StdIn.isEmpty())
        { // Build symbol table and count frequencies.
            String word = StdIn.readString();
            if (word.length() < minlen) continue; // Ignore short keys.
            if (!st.contains(word)) st.put(word, 1);
            else
                st.put(word, st.get(word) + 1);
        }
        // Find a key with the highest frequency count.
        String max = "";
        st.put(max, 0);
        for (String word : st.keys())
            if (st.get(word) > st.get(max))
                max = word;
        StdOut.println(max + " " + st.get(max));
    }
}
```

This ST client counts the frequency of occurrence of the strings in standard input, then prints out one that occurs with highest frequency. The command-line argument specifies a lower bound on the length of keys considered.

```
% java FrequencyCounter 1 < tinyTale.txt
it 10

% java FrequencyCounter 8 < tale.txt
business 122

% java FrequencyCounter 10 < leipzig1M.txt
government 24763
```

symbol table (and the total number of distinct words in the input gives the size of the table after all keys have been inserted), so the total number of distinct words in the input stream is certainly relevant. We need to know both of these quantities in order to estimate the running time of `FrequencyCounter` (for a start, see EXERCISE 3.1.6). We defer details until we consider some algorithms, but you should have in mind a general idea of the needs of typical applications like this one. For example, running `FrequencyCounter` on `leipzig1M.txt` for words of length 8 or more involves millions of searches in a table with hundreds of thousands of keys and values. A server on the web might need to handle billions of transactions on tables with millions of keys and values.

THE BASIC QUESTION THAT THIS CLIENT and these examples raise is the following: Can we develop a symbol-table implementation that can handle a huge number of `get()` operations on a large table, which itself was built with a large number of intermixed `get()` and `put()` operations? If you are only doing a few searches, any implementation will do, but you cannot make use of a client like `FrequencyCounter` for large problems without a good symbol-table implementation. `FrequencyCounter` is surrogate for a very common situation. Specifically, it has the following characteristics, which are shared by many other symbol-table clients:

- Search and insert operations are intermixed.
- The number of distinct keys is not small.
- Substantially more searches than inserts are likely.
- Search and insert patterns, though unpredictable, are not random.

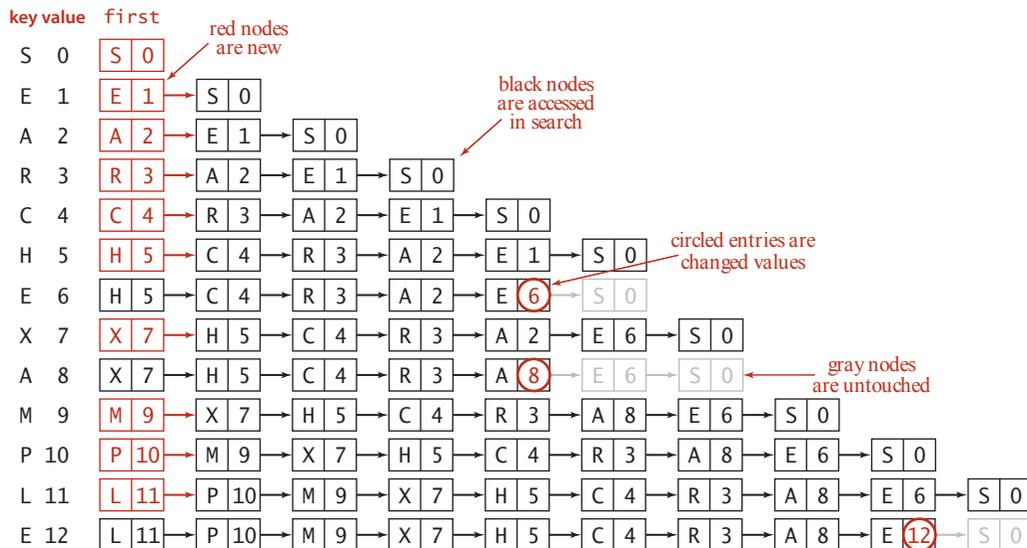
Our goal is to develop symbol-table implementations that make it feasible to use such clients to solve typical practical problems.

Next, we consider two elementary implementations and their performance for `FrequencyCounter`. Then, in the next several sections, you will learn classic implementations that can achieve excellent performance for such clients, even for huge input streams and tables.

Sequential search in an unordered linked list One straightforward option for the underlying data structure for a symbol table is a linked list of nodes that contain keys and values, as in the code on the facing page. To implement `get()`, we scan through the list, using `equals()` to compare the search key with the key in each node in the list. If we find the match, we return the associated value; if not, we return `null`. To implement `put()`, we also scan through the list, using `equals()` to compare the client key with the key in each node in the list. If we find the match, we update the value associated with that key to be the value given in the second argument; if not, we create a new node with the given key and value and insert it at the beginning of the list. This method is known as *sequential search*: we search by considering the keys in the table one after another, using `equals()` to test for a match with our search key.

ALGORITHM 3.1 (SequentialSearchST) is an implementation of our basic symbol-table API that uses standard list-processing mechanisms, which we have used for elementary data structures in CHAPTER 1. We have left the implementations of `size()`, `keys()`, and `eager delete()` for exercises. You are encouraged to work these exercises to cement your understanding of the linked-list data structure and the basic symbol-table API.

Can this linked-list-based implementation handle applications that need large tables, such as our sample clients? As we have noted, analyzing symbol-table algorithms is more complicated than analyzing sorting algorithms because of the difficulty of



Trace of linked-list ST implementation for standard indexing client

ALGORITHM 3.1 Sequential search (in an unordered linked list)

```

public class SequentialSearchST<Key, Value>
{
    private Node first;          // first node in the linked list

    private class Node
    { // linked-list node
        Key key;
        Value val;
        Node next;
        public Node(Key key, Value val, Node next)
        {
            this.key = key;
            this.val = val;
            this.next = next;
        }
    }

    public Value get(Key key)
    { // Search for key, return associated value.
        for (Node x = first; x != null; x = x.next)
            if (key.equals(x.key))
                return x.val; // search hit
        return null; // search miss
    }

    public void put(Key key, Value val)
    { // Search for key. Update value if found; grow table if new.
        for (Node x = first; x != null; x = x.next)
            if (key.equals(x.key))
                { x.val = val; return; } // Search hit: update val.
        first = new Node(key, val, first); // Search miss: add new node.
    }
}

```

This ST implementation uses a private Node inner class to keep the keys and values in an unordered linked list. The `get()` implementation searches the list sequentially to find whether the key is in the table (and returns the associated value if so). The `put()` implementation also searches the list sequentially to check whether the key is in the table. If so, it updates the associated value; if not, it creates a new node with the given key and value and inserts it at the beginning of the list. Implementations of `size()`, `keys()`, and eager `delete()` are left for exercises.

characterizing the sequence of operations that might be invoked by a given client. As noted for `FrequencyCounter`, the most common situation is that, while search and insert patterns are unpredictable, they certainly are not random. For this reason, we pay careful attention to worst-case performance. For economy, we use the term *search hit* to refer to a successful search and *search miss* to refer to an unsuccessful search.

Proposition A. Search misses and insertions in an (unordered) linked-list symbol table having N key-value pairs both require N compares, and search hits N compares in the worst case.

Proof: When searching for a key that is not in the list, we test every key in the table against the search key. Because of our policy of disallowing duplicate keys, we need to do such a search before each insertion.

Corollary. Inserting N distinct keys into an initially empty linked-list symbol table uses $\sim N^2/2$ compares.

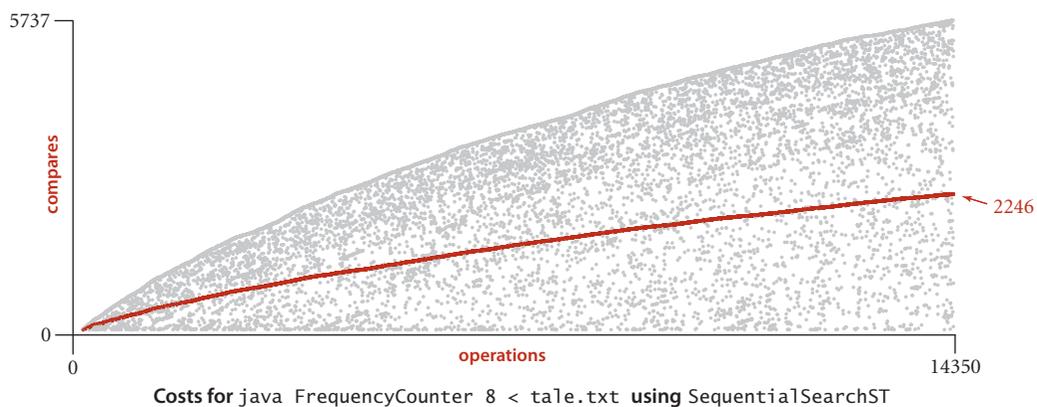
It is true that searches for keys that are in the table need not take linear time. One useful measure is to compute the total cost of searching for all of the keys in the table, divided by N . This quantity is precisely the expected number of compares required for a search under the condition that searches for each key in the table are equally likely. We refer to such a search as a *random search hit*. Though client search patterns are not likely to be random, they often are well-described by this model. It is easy to show that the average number of compares for a random search hit is $\sim N/2$: the `get()` method in ALGORITHM 3.1 uses 1 compare to find the first key, 2 compares to find the second key, and so forth, for an average cost of $(1 + 2 + \dots + N)/N = (N + 1)/2 \sim N/2$.

This analysis strongly indicates that a linked-list implementation with sequential search is too slow for it to be used to solve huge problems such as our reference inputs with clients like `FrequencyCounter`. The total number of compares is proportional to the product of the number of searches and the number of inserts, which is more than 10^9 for `Tale of Two Cities` and more than 10^{14} for the `Leipzig Corpora`.

As usual, to validate analytic results, we need to run experiments. As an example, we study the operation of `FrequencyCounter` with command-line argument `8` for `tale.txt`, which involves 14,350 `put()` operations (recall that every word in the input leads to a `put()`, to update its frequency, and we ignore the cost of easily avoided calls to `contains()`). The symbol table grows to 5,737 keys, so about one-third of the

operations increase the size of the table; the rest are searches. To visualize the performance, we use `VisualAccumulator` (see page 95) to plot two points corresponding to each `put()` operation as follows: for the i th `put()` operation we plot a gray point with x coordinate i and y coordinate the number of key compares it uses and a red point with x coordinate i and y coordinate the cumulative average number of key compares used for the first i `put()` operations. As with any scientific data, there is a great deal of information in this data for us to investigate (this plot has 14,350 gray points and 14,350 red points). In this context, our primary interest is that the plot validates our hypothesis that about half the list is accessed for the average `put()` operation. The actual total is slightly lower than half, but this fact (and the precise shape of the curves) is perhaps best explained by characteristics of the application, not our algorithms (see EXERCISE 3.1.36).

While detailed characterization of performance for particular clients can be complicated, specific hypotheses are easy to formulate and to test for `FrequencyCounter` with our reference inputs or with randomly ordered inputs, using a client like the `DoublingTest` client that we introduced in CHAPTER 1. We will reserve such testing for exercises and for the more sophisticated implementations that follow. If you are not already convinced that we need faster implementations, be sure to work these exercises (or just run `FrequencyCounter` with `SequentialSearchST` on `leipzig1M.txt`!).



Binary search in an ordered array Next, we consider a full implementation of our ordered symbol-table API. The underlying data structure is a pair of parallel arrays, one for the keys and one for the values. ALGORITHM 3.2 (BinarySearchST) on the facing page keeps Comparable keys in order in the array, then uses array indexing to enable fast implementation of `get()` and other operations.

The heart of the implementation is the `rank()` method, which returns the number of keys smaller than a given key. For `get()`, the rank tells us precisely where the key is to be found if it is in the table (and, if it is not there, that it is not in the table).

For `put()`, the rank tells us precisely where to update the value when the key is in the table, and precisely where to put the key when the key is not in the table. We move all larger keys over one position to make room (working from back to front) and insert the given key and value into the proper positions in their respective arrays. Again, studying BinarySearchST in conjunction with a trace of our test client is an instructive introduction to this data structure.

This code maintains parallel arrays of keys and values (see EXERCISE 3.1.12 for an alternative). As with our implementations of generic stacks and queues in CHAPTER 1, this code carries the inconvenience of having to create a `Key` array of type `Comparable` and a `Value` array of type `Object`, and to cast them back to `Key[]` and `Value[]` in the constructor. As usual, we can use array resizing so that clients do not have to be concerned with the size of the array (noting, as you shall see, that this method is too slow to use with large arrays).

key	value	keys[]										N	vals[]									
		0	1	2	3	4	5	6	7	8	9		0	1	2	3	4	5	6	7	8	9
S	0	S										1	0									
E	1	E	S									2	1	0								
A	2	A	E	S								3	2	1	0							
R	3	A	E	R	S							4	2	1	3	0						
C	4	A	C	E	R	S						5	2	4	1	3	0					
H	5	A	C	E	H	R	S					6	2	4	1	5	3	0				
E	6	A	C	E	H	R	S					6	2	4	6	5	3	0				
X	7	A	C	E	H	R	S	X				7	2	4	6	5	3	0	7			
A	8	A	C	E	H	R	S	X				7	8	4	6	5	3	0	7			
M	9	A	C	E	H	M	R	S	X			8	8	4	6	5	9	3	0	7		
P	10	A	C	E	H	M	P	R	S	X		9	8	4	6	5	9	10	3	0	7	
L	11	A	C	E	H	L	M	P	R	S	X	10	8	4	6	5	11	9	10	3	0	7
E	12	A	C	E	H	L	M	P	R	S	X	10	8	4	12	5	11	9	10	3	0	7
		A	C	E	H	L	M	P	R	S	X		8	4	12	5	11	9	10	3	0	7

Trace of ordered-array ST implementation for standard indexing client

ALGORITHM 3.2 Binary search (in an ordered array)

```

public class BinarySearchST<Key extends Comparable<Key>, Value>
{
    private Key[] keys;
    private Value[] vals;
    private int N;

    public BinarySearchST(int capacity)
    { // See Algorithm 1.1 for standard array-resizing code.
        keys = (Key[]) new Comparable[capacity];
        vals = (Value[]) new Object[capacity];
    }

    public int size()
    { return N; }

    public Value get(Key key)
    {
        if (isEmpty()) return null;
        int i = rank(key);
        if (i < N && keys[i].compareTo(key) == 0) return vals[i];
        else return null;
    }

    public int rank(Key key)
    // See page 381.

    public void put(Key key, Value val)
    { // Search for key. Update value if found; grow table if new.
        int i = rank(key);
        if (i < N && keys[i].compareTo(key) == 0)
        { vals[i] = val; return; }
        for (int j = N; j > i; j--)
        { keys[j] = keys[j-1]; vals[j] = vals[j-1]; }
        keys[i] = key; vals[i] = val;
        N++;
    }

    public void delete(Key key)
    // See Exercise 3.1.16 for this method.
}

```

This ST implementation keeps the keys and values in parallel arrays. The `put()` implementation moves larger keys one position to the right before growing the table as in the array-based stack implementation in SECTION 1.3. Array-resizing code is omitted here.

Binary search. The reason that we keep keys in an ordered array is so that we can use array indexing to dramatically reduce the number of compares required for each search, using the classic and venerable *binary search* algorithm that we used as an example in CHAPTER 1. We maintain indices into the sorted key array that delimit the subarray that might contain the search key. To search, we compare the search key against the key in the middle of the subarray. If the search key is less than the key in the middle, we search in the left half of the subarray; if the search key is greater than the key in the middle, we search in the right half of the subarray; otherwise the key in the middle is equal to the search key. The `rank()` code on the facing page uses binary search to complete the symbol-table implementation just discussed. This implementation is worthy of careful study. To study it, we start with the equivalent recursive code at left. A call to `rank(key, 0, N-1)` does the same sequence of compares as a call to the nonrecursive implementation in ALGORITHM 3.2, but this alternate version better exposes the structure of the algorithm, as discussed in SECTION 1.1. This recursive `rank()` preserves the following properties:

```
public int rank(Key key, int lo, int hi)
{
    if (hi < lo) return lo;
    int mid = lo + (hi - lo) / 2;
    int cmp = key.compareTo(keys[mid]);
    if (cmp < 0)
        return rank(key, lo, mid-1);
    else if (cmp > 0)
        return rank(key, mid+1, hi);
    else return mid;
}
```

Recursive binary search

- If key is in the table, `rank()` returns its index in the table, which is the same as the number of keys in the table that are smaller than key.
- If key is not in the table, `rank()` also returns the number of keys in the table that are smaller than key.

Taking the time to convince yourself that the nonrecursive `rank()` in ALGORITHM 3.2 operates as expected (either by proving that it is equivalent to the recursive version or by proving directly that the loop always terminates with the value of `lo` precisely equal to the number of keys in the table that are smaller than key) is a worthwhile exercise for any programmer. (*Hint*: Note that `lo` starts at 0 and never decreases.)

Other operations. Since the keys are kept in an ordered array, most of the order-based operations are compact and straightforward, as you can see from the codepage 382. For example, a call to `select(k)` just returns `keys[k]`. We have left `delete()` and `floor()` as exercises. You are encouraged to study the implementation of `ceiling()` and the two-argument `keys()` and to work these exercises to cement your understanding of the ordered symbol-table API and this implementation.

ALGORITHM 3.2 (continued) Binary search in an ordered array (iterative)

```

public int rank(Key key)
{
    int lo = 0, hi = N-1;
    while (lo <= hi)
    {
        int mid = lo + (hi - lo) / 2;
        int cmp = key.compareTo(keys[mid]);
        if (cmp < 0) hi = mid - 1;
        else if (cmp > 0) lo = mid + 1;
        else return mid;
    }
    return lo;
}

```

This method uses the classic method described in the text to compute the number of keys in the table that are smaller than key. Compare key with the key in the middle: if it is equal, return its index; if it is less, look in the left half; if it is greater, look in the right half.

		keys[]									
		0	1	2	3	4	5	6	7	8	9
successful search for P											
	lo hi mid										
	0 9 4	A	C	E	H	L	M	P	R	S	X
	5 9 7	A	C	E	H	L	M	P	R	S	X
	5 6 5	A	C	E	H	L	M	P	R	S	X
	6 6 6	A	C	E	H	L	M	P	R	S	X
unsuccessful search for Q											
	lo hi mid										
	0 9 4	A	C	E	H	L	M	P	R	S	X
	5 9 7	A	C	E	H	L	M	P	R	S	X
	5 6 5	A	C	E	H	L	M	P	R	S	X
	7 6 6	A	C	E	H	L	M	P	R	S	X

entries in black are a[lo..hi]
 entry in red is a[mid]
 loop exits with keys[mid] = P: return 6
 loop exits with lo > hi: return 7

Trace of binary search for rank in an ordered array

ALGORITHM 3.2 (continued) Ordered symbol-table operations for binary search

```
public Key min()
{ return keys[0]; }

public Key max()
{ return keys[N-1]; }

public Key select(int k)
{ return keys[k]; }

public Key ceiling(Key key)
{
    int i = rank(key);
    return keys[i];
}

public Key floor(Key key)
// See Exercise 3.1.17.

public Key delete(Key key)
// See Exercise 3.1.16.

public Iterable<Key> keys(Key lo, Key hi)
{
    Queue<Key> q = new Queue<Key>();
    for (int i = rank(lo); i < rank(hi); i++)
        q.enqueue(keys[i]);
    if (contains(hi))
        q.enqueue(keys[rank(hi)]);
    return q;
}
```

These methods, along with the methods of EXERCISE 3.1.16 and EXERCISE 3.1.17, complete the implementation of our (ordered) symbol-table API using binary search in an ordered array. The `min()`, `max()`, and `select()` methods are trivial, just amounting to returning the appropriate key from its known position in the array. The `rank()` method, which is the basis of binary search, plays a central role in the others. The `floor()` and `delete()` implementations, left for exercises, are more complicated, but still straightforward.

Analysis of binary search The recursive implementation of `rank()` also leads to an immediate argument that binary search guarantees fast search, because it corresponds to a recurrence relation that describes an upper bound on the number of compares.

Proposition B. Binary search in an ordered array with N keys uses no more than $\lg N + 1$ compares for a search (successful or unsuccessful).

Proof: This analysis is similar to (but simpler than) the analysis of mergesort (PROPOSITION F in CHAPTER 2). Let $C(N)$ be the number of compares to search for a key in a symbol table of size N . We have $C(0) = 0$, $C(1) = 1$, and for $N > 0$ we can write a recurrence relationship that directly mirrors the recursive method:

$$C(N) \leq C(\lfloor N/2 \rfloor) + 1.$$

Whether the search goes to the left or to the right, the size of the subarray is no more than $\lfloor N/2 \rfloor$, and we use one compare to check for equality and to choose whether to go left or right. When N is one less than a power of 2 (say $N = 2^n - 1$), this recurrence is not difficult to solve. First, since $\lfloor N/2 \rfloor = 2^{n-1} - 1$, we have

$$C(2^n - 1) \leq C(2^{n-1} - 1) + 1.$$

Applying the same equation to the first term on the right, we have

$$C(2^n - 1) \leq C(2^{n-2} - 1) + 1 + 1.$$

Repeating the previous step $n - 2$ additional times gives

$$C(2^n - 1) \leq C(2^0) + n$$

which leaves us with the solution

$$C(N) = C(2^n - 1) \leq n + 1 < \lg N + 1.$$

Exact solutions for general N are more complicated, but it is not difficult to extend this argument to establish the stated property for all values of N (see EXERCISE 3.1.20). With binary search, we achieve a logarithmic-time search guarantee.

The implementation just given for `ceiling()` is based on a single call to `rank()`, and the default two-argument `size()` implementation calls `rank()` twice, so this proof also establishes that these operations (and `floor()`) are supported in guaranteed logarithmic time (`min()`, `max()`, and `select()` are constant-time operations).

method	order of growth of running time
put()	N
get()	$\log N$
delete()	N
contains()	$\log N$
size()	1
min()	1
max()	1
floor()	$\log N$
ceiling()	$\log N$
rank()	$\log N$
select()	1
deleteMin()	N
deleteMax()	1

Despite its guaranteed logarithmic search, BinarySearchST still does not enable us to use clients like FrequencyCounter to solve huge problems, because the put() method is too slow. Binary search reduces the number of compares, but not the running time, because its use does not change the fact that the number of array accesses required to build a symbol table in an ordered array is quadratic in the size of the array when keys are randomly ordered (and in typical practical situations where the keys, while not random, are well-described by this model).

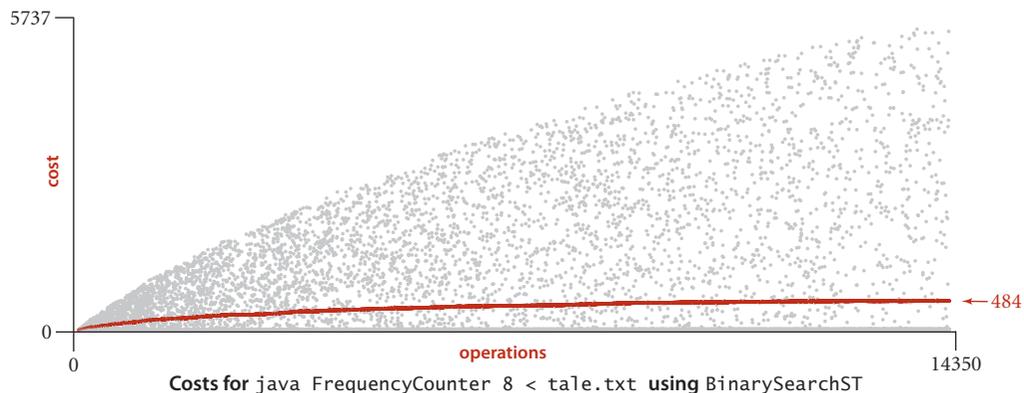
Proposition B (continued). Inserting a new key into an ordered array of size N uses $\sim 2N$ array accesses in the worst case, so inserting N keys into an initially empty table uses $\sim N^2$ array accesses in the worst case.

Proof: Same as for PROPOSITION A.

BinarySearchST costs

For Tale of Two Cities, with over 10^4 distinct keys, the cost to build the table is nearly 10^8 array accesses, and for the Leipzig project, with over 10^6 distinct keys, the cost to build the table is over 10^{11} array accesses. While not quite prohibitive on modern computers, these costs are extremely (and unnecessarily) high.

Returning to the cost of the put() operations for FrequencyCounter for words of length 8 or more, we see a reduction in the average cost from 2,246 compares (plus array accesses) per operation for SequentialSearchST to 484 for BinarySearchST. As before, this cost is even better than would be predicted by analysis, and the extra improvement is likely again explained by properties of the application (see EXERCISE 3.1.36). This improvement is impressive, but we can do much better, as you shall see.



Preview Binary search is typically far better than sequential search and is the method of choice in numerous practical applications. For a static table (with no insert operations allowed), it is worthwhile to initialize and sort the table, as in the version of binary search that we considered in CHAPTER 1 (see page 99). Even when the bulk of the key-value pairs is known before the bulk of the searches (a common situation in applications), it is worthwhile to add to `BinarySearchST` a constructor that initializes and sorts the table (see EXERCISE 3.1.12). Still, binary search is infeasible for use in many other applications. For example, it fails for our Leipzig Corpora application because searches and inserts are intermixed and the table size is too large. As we have emphasized, typical modern search clients require symbol tables that can support fast implementations of *both* search and insert. That is, we need to be able to build huge tables where we can insert (and perhaps remove) key-value pairs in unpredictable patterns, intermixed with searches.

The table below summarizes performance characteristics for the elementary symbol-table implementations considered in this section. The table entries give the leading term of the cost (number of array accesses for binary search, number of compares for the others), which implies the order of growth of the running time.

algorithm (data structure)	worst-case cost (after N inserts)		average-case cost (after N random inserts)		efficiently support ordered operations?
	search	insert	search hit	insert	
<i>sequential search</i> (<i>unordered linked list</i>)	N	N	$N/2$	N	no
<i>binary search</i> (<i>ordered array</i>)	$\lg N$	$2N$	$\lg N$	N	yes

Cost summary for basic symbol-table implementations

The central question is whether we can devise algorithms and data structures that achieve logarithmic performance for *both* search and insert. The answer is a resounding yes! Providing that answer is the main thrust of this chapter. Along with the fast sort capability discussed in CHAPTER 2, fast symbol-table search/insert is one of the most important contributions of algorithmics and one of the most important steps toward the development of the rich computational infrastructure that we now enjoy.

How can we achieve this goal? To support efficient insertion, it seems that we need a linked structure. But a singly linked list forecloses the use of binary search, because the efficiency of binary search depends on our ability to get to the middle of any subarray

quickly via indexing (and the only way to get to the middle of a singly linked list is to follow links). To combine the efficiency of binary search with the flexibility of linked structures, we need more complicated data structures. That combination is provided both by *binary search trees*, the subject of the next two sections, and by *hash tables*, the subject of SECTION 3.4.

We consider six symbol-table implementations in this chapter, so a brief preview is in order. The table below is a list of the data structures, along with the primary reasons in favor of and against using each for an application. They appear in the order in which we consider them.

We will get into more detail on properties of the algorithms and implementations as we discuss them, but the brief characterizations in this table will help you keep them in a broader context as you learn them. The bottom line is that we have several fast symbol-table implementations that can be and are used to great effect in countless applications.

underlying data structure	implementation	pros	cons
<i>linked list</i> (<i>sequential search</i>)	SequentialSearchST	best for tiny STs	slow for large STs
<i>ordered array</i> (<i>binary search</i>)	BinarySearchST	optimal search and space, order-based ops	slow insert
<i>binary search tree</i>	BST	easy to implement, order-based ops	no guarantees space for links
<i>balanced BST</i>	RedBlackBST	optimal search and insert, order-based ops	space for links
<i>hash table</i>	SeparateChainingHashST LinearProbingHashST	fast search/insert for common types of data	need hash for each type no order-based ops space for links/empty

Pros and cons of symbol-table implementations

Q&A

Q. Why not use an `Item` type that implements `Comparable` for symbol tables, in the same way as we did for priority queues in SECTION 2.4, instead of having separate keys and values?

A. That is also a reasonable option. These two approaches illustrate two different ways to associate information with keys—we can do so implicitly by building a data type that includes the key or explicitly by separating keys from values. For symbol tables, we have chosen to highlight the associative array abstraction. Note also that a client specifies just a key in search, not a key-value aggregation.

Q. Why bother with `equals()`? Why not just use `compareTo()` throughout?

A. Not all data types lead to key values that are easy to compare, even though having a symbol table still might make sense. To take an extreme example, you may wish to use pictures or songs as keys. There is no natural way to compare them, but we can certainly test equality (with some work).

Q. Why not allow keys to take the value `null`?

A. We assume that `Key` is an `Object` because we use it to invoke `compareTo()` or `equals()`. But a call like `a.compareTo(b)` would cause a null pointer exception if `a` is `null`. By ruling out this possibility, we allow for simpler client code.

Q. Why not use a method like the `less()` method that we used for sorting?

A. Equality plays a special role in symbol tables, so we also would need a method for testing equality. To avoid proliferation of methods that have essentially the same function, we adopt the built-in Java methods `equals()` and `compareTo()`.

Q. Why not declare `key[]` as `Object[]` (instead of `Comparable[]`) in `BinarySearchST` before casting, in the same way that `val[]` is declared as `Object`?

A. Good question. If you do so, you will get a `ClassCastException` because keys need to be `Comparable` (to ensure that entries in `key[]` have a `compareTo()` method). Thus, declaring `key[]` as `Comparable[]` is required. Delving into the details of programming-language design to explain the reasons would take us somewhat off topic. We use precisely this idiom (and nothing more complicated) in any code that uses `Comparable` generics and arrays throughout this book.

Q&A *(continued)*

Q. What if we need to associate multiple values with the same key? For example, if we use `Date` as a key in an application, wouldn't we have to process equal keys?

A. Maybe, maybe not. For example, you can't have two trains arrive at the station on the same track at the same time (but they could arrive on different tracks at the same time). There are two ways to handle the situation: use some other information to disambiguate or make the value a `Queue` of values having the same key. We consider applications in detail in SECTION 3.5.

Q. Presorting the table as discussed on page 385 seems like a good idea. Why relegate that to an exercise (see EXERCISE 3.1.12)?

A. Indeed, this may be the method of choice in some applications. But adding a slow insert method to a data structure designed for fast search “for convenience” is a performance trap, because an unsuspecting client might intermix searches and inserts in a huge table and experience quadratic performance. Such traps are all too common, so that “buyer beware” is certainly appropriate when using software developed by others, particularly when interfaces are too wide. This problem becomes acute when a large number of methods are included “for convenience” leaving performance traps throughout, while a client might expect efficient implementations of all methods. Java's `ArrayList` class is an example (see EXERCISE 3.5.27).

EXERCISES

3.1.1 Write a client that creates a symbol table mapping letter grades to numerical scores, as in the table below, then reads from standard input a list of letter grades and computes and prints the GPA (the average of the numbers corresponding to the grades).

A+	A	A-	B+	B	B-	C+	C	C-	D	F
4.33	4.00	3.67	3.33	3.00	2.67	2.33	2.00	1.67	1.00	0.00

3.1.2 Develop a symbol-table implementation `ArrayST` that uses an (unordered) array as the underlying data structure to implement our basic symbol-table API.

3.1.3 Develop a symbol-table implementation `OrderedSequentialSearchST` that uses an ordered linked list as the underlying data structure to implement our ordered symbol-table API.

3.1.4 Develop `Time` and `Event` ADTs that allow processing of data as in the example illustrated on page 367.

3.1.5 Implement `size()`, `delete()`, and `keys()` for `SequentialSearchST`.

3.1.6 Give the number of calls to `put()` and `get()` issued by `FrequencyCounter`, as a function of the number W of words and the number D of distinct words in the input.

3.1.7 What is the average number of distinct keys that `FrequencyCounter` will find among N random nonnegative integers less than 1,000, for $N=10, 10^2, 10^3, 10^4, 10^5,$ and 10^6 ?

3.1.8 What is the most frequently used word of ten letters or more in *Tale of Two Cities*?

3.1.9 Add code to `FrequencyCounter` to keep track of the last call to `put()`. Print the last word inserted and the number of words that were processed in the input stream prior to this insertion. Run your program for `ta1e.txt` with length cutoffs 1, 8, and 10.

3.1.10 Give a trace of the process of inserting the keys `E A S Y Q U E S T I O N` into an initially empty table using `SequentialSearchST`. How many compares are involved?

3.1.11 Give a trace of the process of inserting the keys `E A S Y Q U E S T I O N` into an initially empty table using `BinarySearchST`. How many compares are involved?

3.1.12 Modify `BinarySearchST` to maintain one array of `Item` objects that contain keys and values, rather than two parallel arrays. Add a constructor that takes an array of

EXERCISES *(continued)*

Item values as argument and uses mergesort to sort the array.

3.1.13 Which of the symbol-table implementations in this section would you use for an application that does 10^3 `put()` operations and 10^6 `get()` operations, randomly intermixed? Justify your answer.

3.1.14 Which of the symbol-table implementations in this section would you use for an application that does 10^6 `put()` operations and 10^3 `get()` operations, randomly intermixed? Justify your answer.

3.1.15 Assume that searches are 1,000 times more frequent than insertions for a `BinarySearchST` client. Estimate the percentage of the total time that is devoted to insertions, when the number of searches is 10^3 , 10^6 , and 10^9 .

3.1.16 Implement the `delete()` method for `BinarySearchST`.

3.1.17 Implement the `floor()` method for `BinarySearchST`.

3.1.18 Prove that the `rank()` method in `BinarySearchST` is correct.

3.1.19 Modify `FrequencyCounter` to print all of the values having the highest frequency of occurrence, not just one of them. *Hint*: Use a `Queue`.

3.1.20 Complete the proof of PROPOSITION B (show that it holds for all values of N). *Hint*: Start by showing that $C(N)$ is monotonic: $C(N) \leq C(N+1)$ for all $N > 0$.

CREATIVE PROBLEMS

3.1.21 *Memory usage.* Compare the memory usage of `BinarySearchST` with that of `SequentialSearchST` for N key-value pairs, under the assumptions described in SECTION 1.4. Do not count the memory for the keys and values themselves, but do count references to them. For `BinarySearchST`, assume that array resizing is used, so that the array is between 25 percent and 100 percent full.

3.1.22 *Self-organizing search.* A self-organizing search algorithm is one that rearranges items to make those that are accessed frequently likely to be found early in the search. Modify your search implementation for EXERCISE 3.1.2 to perform the following action on every search hit: move the key-value pair found to the beginning of the list, moving all pairs between the beginning of the list and the vacated position to the right one position. This procedure is called the move-to-front heuristic.

3.1.23 *Analysis of binary search.* Prove that the maximum number of compares used for a binary search in a table of size N is precisely the number of bits in the binary representation of N , because the operation of shifting 1 bit to the right converts the binary representation of N into the binary representation of $\lfloor N/2 \rfloor$.

3.1.24 *Interpolation search.* Suppose that arithmetic operations are allowed on keys (for example, they may be `Double` or `Integer` values). Write a version of binary search that mimics the process of looking near the beginning of a dictionary when the word begins with a letter near the beginning of the alphabet. Specifically, if k_x is the key value sought, k_{lo} is the key value of the first key in the table, and k_{hi} is the key value of the last key in the table, look first $\lfloor (k_x - k_{lo}) / (k_{hi} - k_{lo}) \rfloor$ -way through the table, not half-way. Test your implementation against `BinarySearchST` for `FrequencyCounter`.

3.1.25 *Software caching.* Since the default implementation of `contains()` calls `get()`, the inner loop of `FrequencyCounter`

```
if (!st.contains(word)) st.put(word, 1);
else                    st.put(word, st.get(word) + 1);
```

leads to two or three searches for the same key. To enable clear client code like this without sacrificing efficiency, we can use a technique known as software *caching*, where we save the location of the most recently accessed key in an instance variable. Modify `SequentialSearchST` and `BinarySearchST` to take advantage of this idea.

CREATIVE PROBLEMS *(continued)*

3.1.26 *Frequency count from a dictionary.* Modify `FrequencyCounter` to take the name of a dictionary file as its argument, count frequencies of the words from standard input that are also in that file, and print two tables of the words with their frequencies, one sorted by frequency, the other sorted in the order found in the dictionary file.

3.1.27 *Small tables.* Suppose that a `BinarySearchST` client has S search operations and N distinct keys. Give the order of growth of S such that the cost of building the table is the same as the cost of all the searches.

3.1.28 *Ordered insertions.* Modify `BinarySearchST` so that inserting a key that is larger than all keys in the table takes constant time (so that building a table by calling `put()` for keys that are in order takes linear time).

3.1.29 *Test client.* Write a test client for `BinarySearchST` that tests the implementations of `min()`, `max()`, `floor()`, `ceiling()`, `select()`, `rank()`, `deleteMin()`, `deleteMax()`, and `keys()` that are given in the text. Start with the standard indexing client given on page 370. Add code to take additional command-line arguments, as appropriate.

3.1.30 *Certification.* Add assert statements to `BinarySearchST` to check algorithm invariants and data structure integrity after every insertion and deletion. For example, every index i should always be equal to `rank(select(i))` and the array should always be in order.

EXPERIMENTS

3.1.31 *Performance driver.* Write a performance driver program that uses `put()` to fill a symbol table, then uses `get()` such that each key in the table is hit an average of ten times and there is about the same number of misses, doing so multiple times on random sequences of string keys of various lengths ranging from 2 to 50 characters; measures the time taken for each run; and prints out or plots the average running times.

3.1.32 *Exercise driver.* Write an exercise driver program that uses the methods in our ordered symbol-table API on difficult or pathological cases that might turn up in practical applications. Simple examples include key sequences that are already in order, key sequences in reverse order, key sequences where all keys are the same, and keys consisting of only two distinct values.

3.1.33 *Driver for self-organizing search.* Write a driver program for self-organizing search implementations (see EXERCISE 3.1.22) that uses `put()` to fill a symbol table with N keys, then does $10N$ successful searches according to a predefined probability distribution. Use this driver to compare the running time of your implementation from EXERCISE 3.1.22 with `BinarySearchST` for $N = 10^3, 10^4, 10^5,$ and 10^6 using the probability distribution where search hits the i th smallest key with probability $1/2^i$.

3.1.34 *Zipf's law.* Do the previous exercise for the probability distribution where search hits the i th smallest key with probability $1/(iH_N)$ where H_N is a Harmonic number (see page 185). This distribution is called *Zipf's law*. Compare the move-to-front heuristic with the optimal arrangement for the distributions in the previous exercise, which is to keep the keys in increasing order (decreasing order of their expected frequency).

3.1.35 *Performance validation I.* Run doubling tests that use the first N words of Tale of Two Cities for various values of N to test the hypothesis that the running time of `FrequencyCounter` is quadratic when it uses `SequentialSearchST` for its symbol table.

3.1.36 *Performance validation II.* Explain why the performance of `BinarySearchST` and `SequentialSearchST` for `FrequencyCounter` is even better than predicted by analysis.

3.1.37 *Put/get ratio.* Determine empirically the ratio of the amount of time that `BinarySearchST` spends on `put()` operations to the time that it spends on `get()` operations when `FrequencyCounter` is used to find the frequency of occurrence of values

EXPERIMENTS *(continued)*

in 1 million random M -bit `int` values, for $M = 10, 20,$ and 30 . Answer the same question for `tale.txt` and compare the results.

3.1.38 *Amortized cost plots.* Develop instrumentation for `FrequencyCounter`, `SequentialSearchST`, and `BinarySearchST` so that you can produce plots like the ones in this section showing the cost of each `put()` operation during the computation.

3.1.39 *Actual timings.* Instrument `FrequencyCounter` to use `Stopwatch` and `StdDraw` to make a plot where the x -axis is the number of calls on `get()` or `put()` and the y -axis is the total running time, with a point plotted of the cumulative time after each call. Run your program for `Tale of Two Cities` using `SequentialSearchST` and again using `BinarySearchST` and discuss the results. Note: Sharp jumps in the curve may be explained by *caching*, which is beyond the scope of this question.

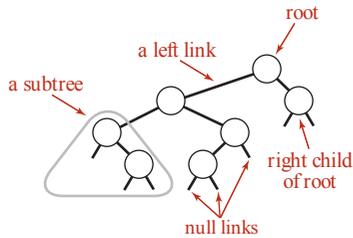
3.1.40 *Crossover to binary search.* Find the values of N for which binary search in a symbol table of size N becomes 10, 100, and 1,000 times faster than sequential search. Predict the values with analysis and verify them experimentally.

3.1.41 *Crossover to interpolation search.* Find the values of N for which interpolation search in a symbol table of size N becomes 1, 2, and 10 times faster than binary search, assuming the keys to be random 32-bit integers (see EXERCISE 3.1.24). Predict the values with analysis, and verify them experimentally.

This page intentionally left blank

3.2 BINARY SEARCH TREES

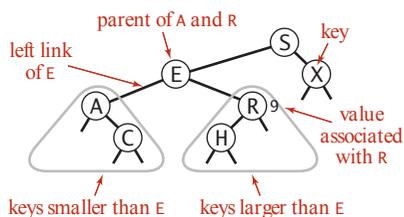
IN THIS SECTION, we will examine a symbol-table implementation that combines the flexibility of insertion in a linked list with the efficiency of search in an ordered array. Specifically, using two links per node (instead of the one link per node found in linked lists) leads to an efficient symbol-table implementation based on the binary search tree data structure, which qualifies as one of the most fundamental algorithms in computer science.



Anatomy of a binary tree

To begin, we define basic terminology. We are working with data structures made up of nodes that contain links that are either null or references to other nodes. In a binary tree, we have the restriction that every node is pointed to by just one other node, which is called its parent (except for one node, the root, which has no nodes pointing to it), and that each node has exactly two links, which are called its left and *right* links, that point to nodes called its *left child* and *right child*, respectively. Although links point to nodes, we can view each link as pointing to a binary tree, the tree whose root is the referenced node. Thus, we can define a binary tree as either a null link or a node with a left link and a right link, each references to (disjoint) subtrees that are themselves binary trees. In a *binary search tree*, each node also has a key and a value, with an ordering restriction to support efficient search.

Definition. A *binary search tree* (BST) is a binary tree where each node has a Comparable key (and an associated value) and satisfies the restriction that the key in any node is larger than the keys in all nodes in that node's left subtree and smaller than the keys in all nodes in that node's right subtree.



Anatomy of a binary search tree

We draw BSTs with keys in the nodes and use terminology such as “A is the left child of E” that associates nodes with keys. Lines connecting the nodes represent links, and we give the value associated with a key in black, beside the nodes (suppressing the value as dictated by context). Each node's links connect it to nodes below it on the page, except for null links, which are short segments at the bottom. As usual, our examples use the single-letter keys that are generated by our indexing test client.

Basic implementation ALGORITHM 3.3 defines the BST data structure that we use throughout this section to implement the ordered symbol-table API. We begin by considering this classic data structure definition and the characteristic associated implementations of the `get()` (search) and `put()` (insert) methods.

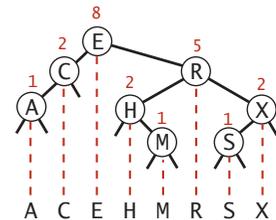
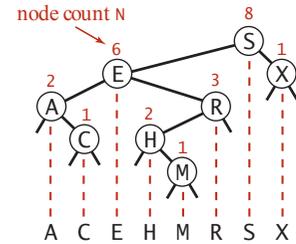
Representation. We define a private nested class to define nodes in BSTs, just as we did for linked lists. Each node contains a key, a value, a left link, a right link, and a node count (when relevant, we include node counts in red above the node in our drawings). The left link points to a BST for items with smaller keys, and the right link points to a BST for items with larger keys. The instance variable `N` gives the node count in the subtree rooted at the node. This field facilitates the implementation of various ordered symbol-table operations, as you will see. The private `size()` method in ALGORITHM 3.3 is implemented to assign the value 0 to null links, so that we can maintain this field by making sure that the invariant

$$\text{size}(x) = \text{size}(x.\text{left}) + \text{size}(x.\text{right}) + 1$$

holds for every node `x` in the tree.

A BST represents a set of keys (and associated values), and there are many different BSTs that represent the same set. If we project the keys in a BST such that all keys in each node's left subtree appear to the left of the key in the node and all keys in each node's right subtree appear to the right of the key in the node, then we always get the keys in sorted order. We take advantage of the flexibility inherent in having many BSTs represent this sorted order to develop efficient algorithms for building and using BSTs.

Search. As usual, when we search for a key in a symbol table, we have one of two possible outcomes. If a node containing the key is in the table, we have a *search hit*, so we return the associated value. Otherwise, we have a *search miss* (and return `null`). A recursive algorithm to search for a key in a BST follows immediately from the recursive structure: if the tree is empty, we have a search miss; if the search key is equal to the key at the root, we have a search hit. Otherwise, we search (recursively) in the appropriate subtree, moving left if the search key is smaller, right if it is larger. The recursive `get()` method on page 399 implements this algorithm directly. It takes a node (root of a subtree) as first argument and a key as second argument, starting with the root of the tree and the search key. The code maintains the invariant that no parts of the tree other than the subtree rooted at the current node can have a node whose key is equal to the search key. Just as the size of the interval in binary search shrinks by about half on each iteration,



Two BSTs that represent the same set of keys

ALGORITHM 3.3 Binary search tree symbol table

```

public class BST<Key extends Comparable<Key>, Value>
{
    private Node root;                // root of BST

    private class Node
    {
        private Key key;              // key
        private Value val;            // associated value
        private Node left, right;     // links to subtrees
        private int N;                // # nodes in subtree rooted here

        public Node(Key key, Value val, int N)
        { this.key = key; this.val = val; this.N = N; }
    }

    public int size()
    { return size(root); }

    private int size(Node x)
    {
        if (x == null) return 0;
        else return x.N;
    }

    public Value get(Key key)
    // See page 399.

    public void put(Key key, Value val)
    // See page 399.

    // See page 407 for min(), max(), floor(), and ceiling().
    // See page 409 for select() and rank().
    // See page 411 for delete(), deleteMin(), and deleteMax().
    // See page 413 for keys().

}

```

This implementation of the ordered symbol-table API uses a binary search tree built from Node objects that each contain a key, associated value, two links, and a node count N. Each Node is the root of a subtree containing N nodes, with its left link pointing to a Node that is the root of a subtree with smaller keys and its right link pointing to a Node that is the root of a subtree with larger keys. The instance variable root points to the Node at the root of the BST (which has all the keys and associated values in the symbol table). Implementations of other methods appear throughout this section.

ALGORITHM 3.3 (continued) Search and insert for BSTs

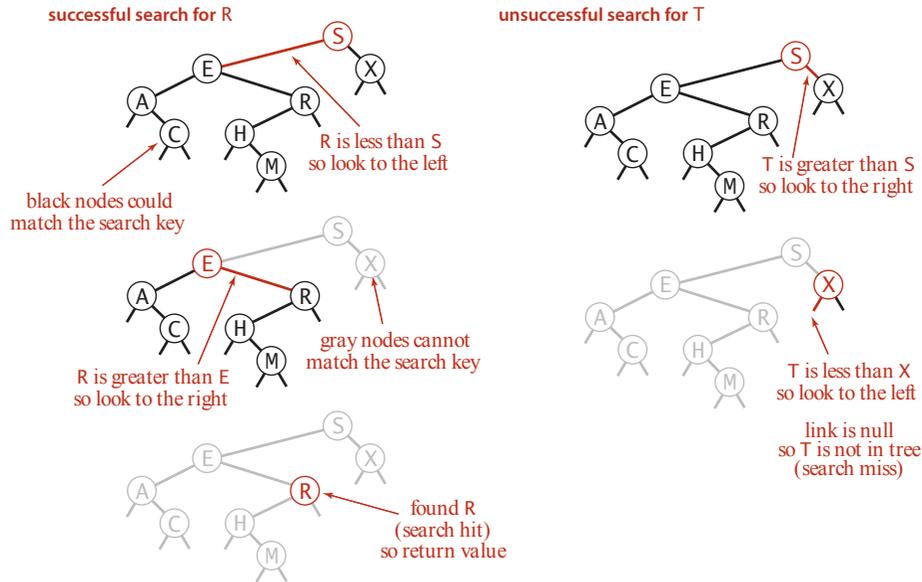
```
public Value get(Key key)
{ return get(root, key); }

private Value get(Node x, Key key)
{ // Return value associated with key in the subtree rooted at x;
  // return null if key not present in subtree rooted at x.
  if (x == null) return null;
  int cmp = key.compareTo(x.key);
  if (cmp < 0) return get(x.left, key);
  else if (cmp > 0) return get(x.right, key);
  else return x.val;
}

public void put(Key key, Value val)
{ // Search for key. Update value if found; grow table if new.
  root = put(root, key, val);
}

private Node put(Node x, Key key, Value val)
{
  // Change key's value to val if key in subtree rooted at x.
  // Otherwise, add new node to subtree associating key with val.
  if (x == null) return new Node(key, val, 1);
  int cmp = key.compareTo(x.key);
  if (cmp < 0) x.left = put(x.left, key, val);
  else if (cmp > 0) x.right = put(x.right, key, val);
  else x.val = val;
  x.N = size(x.left) + size(x.right) + 1;
  return x;
}
```

These implementations of `get()` and `put()` for the symbol-table API are characteristic recursive BST methods that also serve as models for several other implementations that we consider later in the chapter. Each method can be understood as both working code and a proof by induction of the inductive hypothesis in the opening comment.



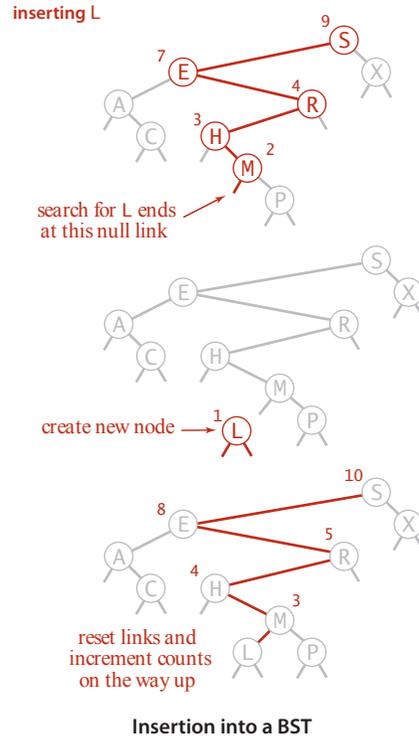
Search hit (left) and search miss (right) in a BST

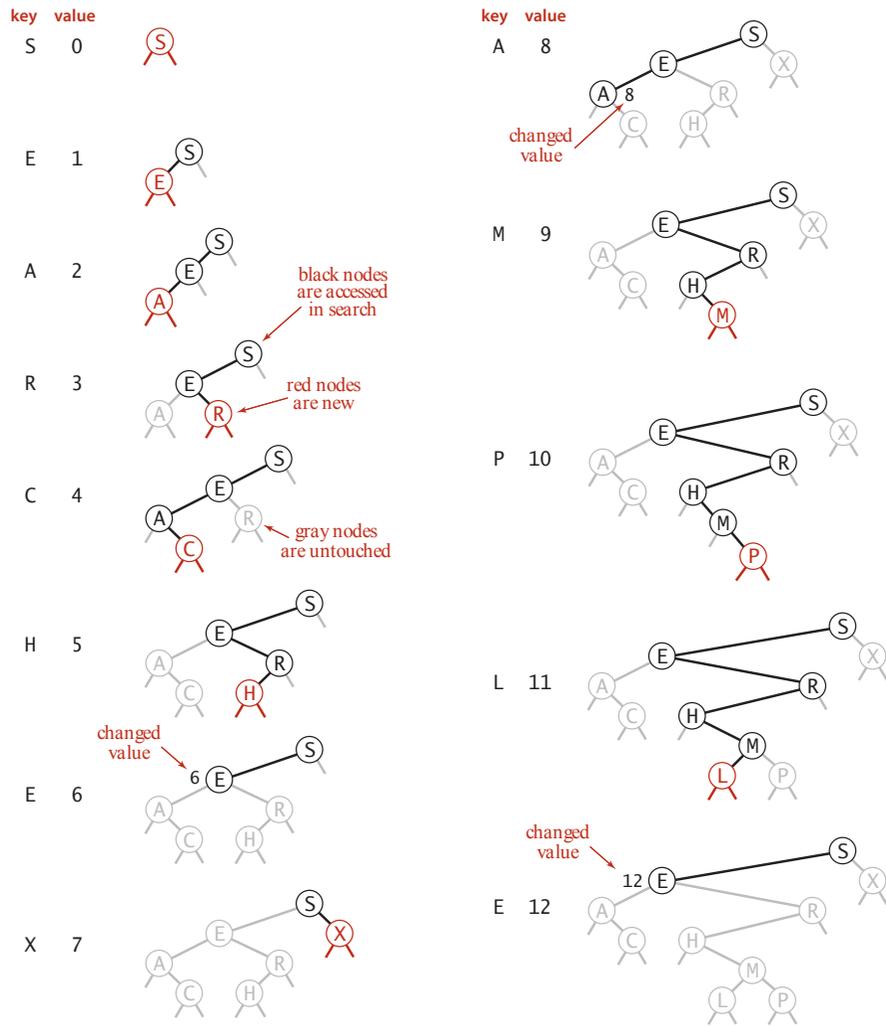
the size of the subtree rooted at the current node when searching in a BST shrinks when we go down the tree (by about half, ideally, but at least by one). The procedure stops either when a node containing the search key is found (search hit) or when the current subtree becomes empty (search miss). Starting at the top, the search procedure at each node involves a recursive invocation for one of that node's children, so the search defines a path through the tree. For a search hit, the path terminates at the node containing the key. For a search miss, the path terminates at a null link.

Insert. The search code in ALGORITHM 3.3 is almost as simple as binary search; that simplicity is an essential feature of BSTs. A more important essential feature of BSTs is that insert is not much more difficult to implement than *search*. Indeed, a search for a key not in the tree ends at a null link, and all that we need to do is replace that link with a new node containing the key (see the diagram on the next page). The recursive `put()` method in ALGORITHM 3.3 accomplishes this task using logic similar to that we used for the recursive search: if the tree is empty, we return a new node containing the key and value; if the search key is less than the key at the root, we set the left link to the result of inserting the key into the left subtree; otherwise, we set the right link to the result of inserting the key into the right subtree.

Recursion. It is worthwhile to take the time to understand the dynamics of these recursive implementations. You can think of the code before the recursive calls as happening on the way down the tree: it compares the given key against the key at each node and moves right or left accordingly. Then, think of the code after the recursive calls as happening on the way up the tree. For `get()` this amounts to a series of return statements, but for `put()`, it corresponds to resetting the link of each parent to its child on the search path and to incrementing the counts on the way up the path. In simple BSTs, the only new link is the one at the bottom, but resetting the links higher up on the path is as easy as the test to avoid setting them. Also, we just need to increment the node count on each node on the path, but we use more general code that sets each to one plus the sum of the counts in its subtrees. Later in this section and in the next section, we shall study more advanced algorithms that are naturally expressed with this same recursive scheme but that can change more links on the search paths and need the more general node-count-update code. Elementary BSTs are often implemented with nonrecursive code (see EXERCISE 3.2.13)—we use recursion in our implementations both to make it easy for you to convince yourself that the code is operating as described and to prepare the groundwork for more sophisticated algorithms.

A CAREFUL STUDY of the trace for our standard indexing client that is shown on the next page will give you a feeling for the way in which binary search trees grow. New nodes are attached to null links at the bottom of the tree; the tree structure is not otherwise changed. For example, the root has the first key inserted, one of the children of the root has the second key inserted, and so forth. Because each node has two links, the tree tends to grow out, rather than just down. Moreover, only the keys on the path from the root to the sought or inserted key are examined, so the number of keys examined becomes a smaller and smaller fraction of the number of keys in the tree as the tree size increases.

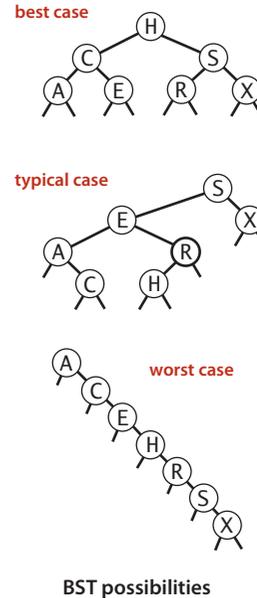




BST trace for standard indexing client

Analysis The running times of algorithms on binary search trees depend on the shapes of the trees, which, in turn, depend on the order in which keys are inserted. In the best case, a tree with N nodes could be perfectly balanced, with $\sim \lg N$ nodes between the root and each null link. In the worst case there could be N nodes on the search path. The balance in typical trees turns out to be much closer to the best case than the worst case.

For many applications, the following simple model is reasonable: We assume that the keys are (uniformly) random, or, equivalently, that they are inserted in random order. Analysis of this model stems from the observation that BSTs are dual to quicksort. The node at the root of the tree corresponds to the first partitioning item in quicksort (no keys to the left are larger, and no keys to the right are smaller) and the subtrees are built recursively, corresponding to quicksort's recursive subarray sorts. This observation leads us to the analysis of properties of the trees.



Proposition C. Search hits in a BST built from N random keys require $\sim 2 \ln N$ (about $1.39 \lg N$) compares, on the average.

Proof: The number of compares used for a search hit ending at a given node is 1 plus the depth. Adding the depths of all nodes, we get a quantity known as the *internal path length* of the tree. Thus, the desired quantity is 1 plus the average internal path length of the BST, which we can analyze with the same argument that we used for PROPOSITION K in SECTION 2.3: Let C_N be the internal path length of a BST built from inserting N randomly ordered distinct keys, so that the average cost of a search hit is $1 + C_N/N$. We have $C_0 = C_1 = 0$ and for $N > 1$ we can write a recurrence relationship that directly mirrors the recursive BST structure:

$$C_N = N - 1 + (C_0 + C_{N-1})/N + (C_1 + C_{N-2})/N + \dots + (C_{N-1} + C_0)/N$$

The $N - 1$ term takes into account that the root contributes 1 to the path length of each of the other $N - 1$ nodes in the tree; the rest of the expression accounts for the subtrees, which are equally likely to be any of the N sizes. After rearranging terms, this recurrence is nearly identical to the one that we solved in SECTION 2.3 for quicksort, and we can derive the approximation $C_N \sim 2N \ln N$.

Proposition D. Insertions and search misses in a BST built from N random keys require $\sim 2 \ln N$ (about $1.39 \lg N$) compares, on the average.

Proof: Insertions and search misses take one more compare, on the average, than search hits. This fact is not difficult to establish by induction (see EXERCISE 3.2.16).

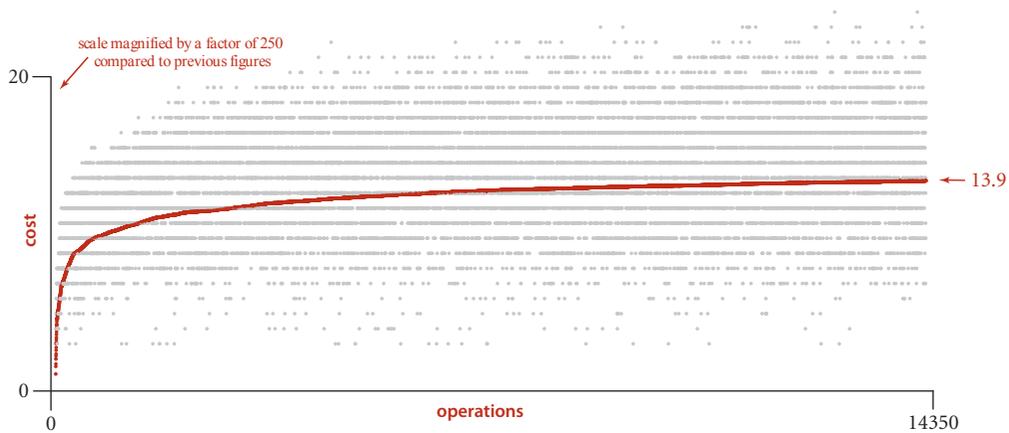
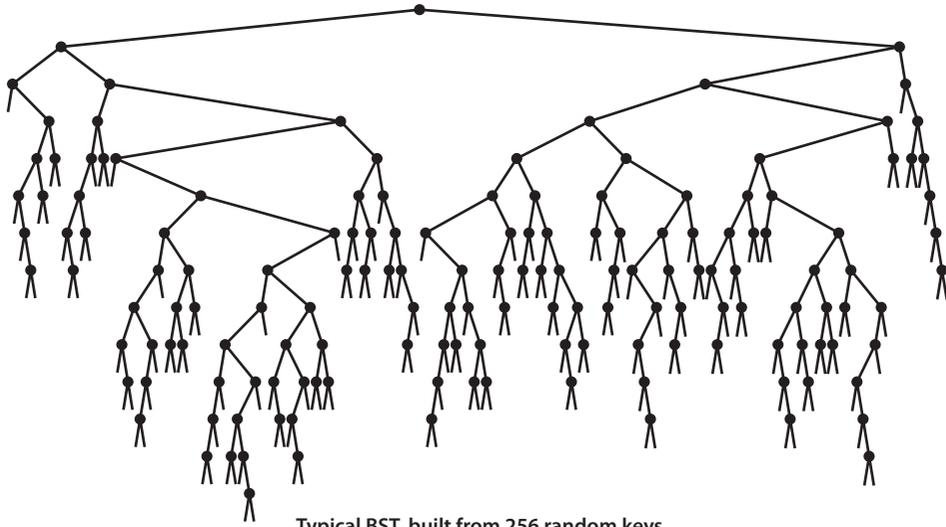
PROPOSITION C says that we should expect the BST search cost for random keys to be about 39 percent higher than that for binary search. PROPOSITION D says that the extra cost is well worthwhile, because the cost of inserting a new key is also expected to be logarithmic—flexibility not available with binary search in an ordered array, where the number of array accesses required for an insertion is typically linear. As with quicksort, the standard deviation of the number of compares is known to be low, so that these formulas become increasingly accurate as N increases.

Experiments. How well does our random-key model match what is found in typical symbol-table clients? As usual, this question has to be studied carefully for particular practical applications, because of the large potential variation in performance. Fortunately, for many clients, the model is quite good for BSTs.

For our example study of the cost of the `put()` operations for `FrequencyCounter` for words of length 8 or more, we see a reduction in the average cost from 484 array accesses or compares per operation for `BinarySearchST` to 13 for BST, again providing a quick validation of the logarithmic performance predicted by the theoretical model. More extensive experiments for larger inputs are illustrated in the table on the next page. On the basis of PROPOSITIONS C and D, it is reasonable to predict that this number should be about twice the natural logarithm of the table size, because the preponderance of operations are searches in a nearly full table. This prediction has at least the following inherent inaccuracies:

- Many operations are for smaller tables.
- The keys are not random.
- The table size may be too small for the approximation $2 \ln N$ to be accurate.

Nevertheless, as you can see in the table, this prediction is accurate for our `FrequencyCounter` test cases to within a few compares. Actually, most of the difference can be explained by refining the mathematics in the approximation (see EXERCISE 3.2.35).



Costs for java FrequencyCounter 8 < tale.txt using BST

	tale.txt				leipzig1M.txt			
	words	distinct	compares		words	distinct	compares	
			model	actual			model	actual
all words	135,635	10,679	18.6	17.5	21,191,455	534,580	23.4	22.1
8+ letters	14,350	5,737	17.6	13.9	4,239,597	299,593	22.7	21.4
10+ letters	4,582	2,260	15.4	13.1	1,610,829	165,555	20.5	19.3

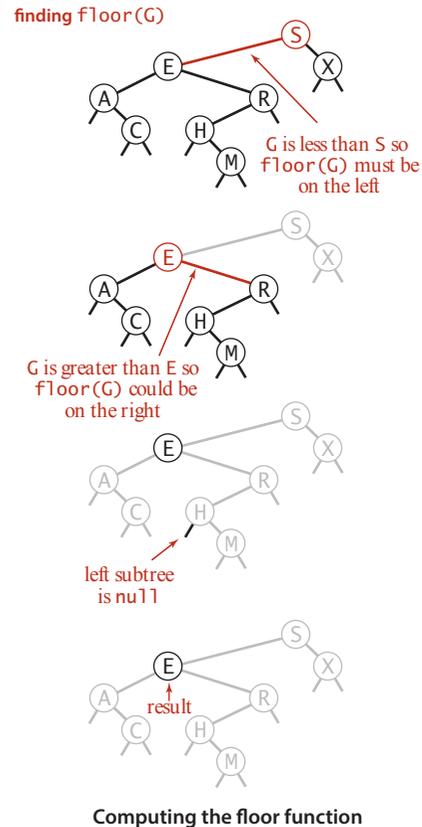
Average number of compares per put() for FrequencyCounter using BST

Order-based methods and deletion An important reason that BSTs are widely used is that they allow us to *keep the keys in order*. As such, they can serve as the basis for implementing the numerous methods in our ordered symbol-table API (see page 366) that allow clients to access key-value pairs not just by providing the key, but also by relative key order. Next, we consider implementations of the various methods in our ordered symbol-table API.

Minimum and maximum. If the left link of the root is null, the smallest key in a BST is the key at the root; if the left link is not null, the smallest key in the BST is the smallest key in the subtree rooted at the node referenced by the left link. This statement is both a description of the recursive `min()` method on page 407 and an inductive proof that it finds the smallest key in the BST. The computation is equivalent to a simple iteration (move left until finding a null link), but we use recursion for consistency. We might have the recursive method return a `Key` instead of a `Node`, but we will later have a need to use this method to access the `Node` containing the minimum key. Finding the maximum key is similar, moving to the right instead of to the left.

Floor and ceiling. If a given key `key` is *less than* the key at the root of a BST, then the floor of key (the largest key in the BST less than or equal to key) must be in the left subtree. If key is *greater than* the key at the root, then the floor of key could be in the right subtree, but only if there is a key smaller than or equal to key in the right subtree; if not (or if key is equal to the key at the root), then the key at the root is the floor of key. Again, this description serves both as the basis for the recursive `floor()` method and for an inductive proof that it computes the desired result. Interchanging right and left (and less and greater) gives `ceiling()`.

Selection. Selection in a BST works in a manner analogous to the partition-based method of selection in an array that we studied in SECTION 2.5. We maintain in BST nodes the variable `N` that counts the number of keys in the subtree rooted at that node precisely to support this operation.



ALGORITHM 3.3 (continued) Min, max, floor, and ceiling in BSTs

```
public Key min()
{
    return min(root).key;
}

private Node min(Node x)
{
    if (x.left == null) return x;
    return min(x.left);
}

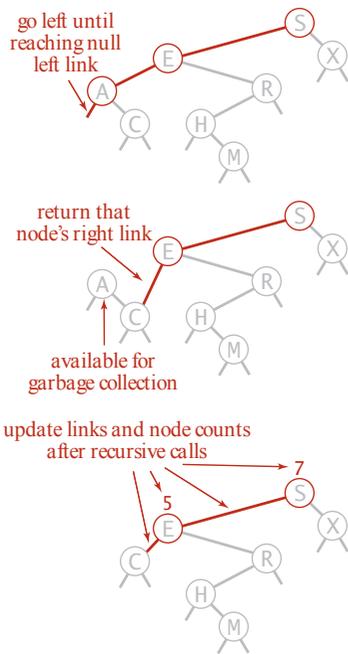
public Key floor(Key key)
{
    Node x = floor(root, key);
    if (x == null) return null;
    return x.key;
}

private Node floor(Node x, Key key)
{
    if (x == null) return null;
    int cmp = key.compareTo(x.key);
    if (cmp == 0) return x;
    if (cmp < 0) return floor(x.left, key);
    Node t = floor(x.right, key);
    if (t != null) return t;
    else return x;
}
```

Each client method calls a corresponding private method that takes an additional link (to a `Node`) as argument and returns `null` or a `Node` containing the desired `Key` via the recursive procedure described in the text. The `max()` and `ceiling()` methods are the same as `min()` and `floor()` (respectively) with `right` and `left` (and `<` and `>`) interchanged.

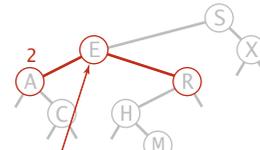
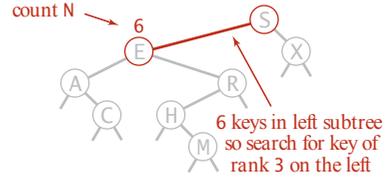
Suppose that we seek the key of rank k (the key such that precisely k other keys in the BST are smaller). If the number of keys t in the left subtree is larger than k , we look (recursively) for the key of rank k in the left subtree; if t is equal to k , we return the key at the root; and if t is smaller than k , we look (recursively) for the key of rank $k - t - 1$ in the right subtree. As usual, this description serves both as the basis for the recursive `select()` method on the facing page and for a proof by induction that it works as expected.

Rank. The inverse method `rank()` that returns the rank of a given key is similar: if the given key is equal to the key at the root, we return the number of keys t in the left subtree; if the given key is less than the key at the root, we return the rank of the key in the left subtree (recursively computed); and if the given key is larger than the key at the root, we return t plus one (to count the key at the root) plus the rank of the key in the right subtree (recursively computed).

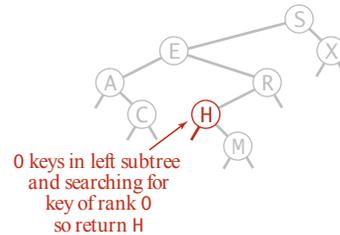
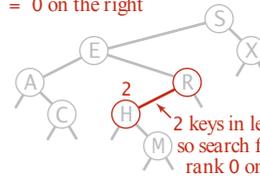


Deleting the minimum in a BST

finding `select(3)`
the key of rank 3



2 keys in left subtree so search for key of rank $3-2-1 = 0$ on the right



Selection in a BST

Delete the minimum/maximum. The most difficult BST operation to implement is the `delete()` method that removes a key-value pair from the symbol table. As a warmup, consider `deleteMin()` (remove the key-value pair with the smallest key). As with `put()` we write a recursive method that takes a link to a `Node` as argument and returns a link to a `Node`, so that we can reflect changes to the tree by assigning the result to the link used as argument. For `deleteMin()` we go left until finding a `Node` that has a null left link and then replace the link to that node by its right link (simply by returning the right link in the recursive method). The deleted node, with no link now pointing to it, is

ALGORITHM 3.3 (continued) Selection and rank in BSTs

```
public Key select(int k)
{
    return select(root, k).key;
}

private Node select(Node x, int k)
{
    // Return Node containing key of rank k.
    if (x == null) return null;
    int t = size(x.left);
    if (t > k) return select(x.left, k);
    else if (t < k) return select(x.right, k-t-1);
    else return x;
}

public int rank(Key key)
{ return rank(key, root); }

private int rank(Key key, Node x)
{
    // Return number of keys less than key in the subtree rooted at x.
    if (x == null) return 0;
    int cmp = key.compareTo(x.key);
    if (cmp < 0) return rank(key, x.left);
    else if (cmp > 0) return 1 + size(x.left) + rank(key, x.right);
    else return size(x.left);
}
```

This code uses the same recursive scheme that we have been using throughout this chapter to implement the `select()` and `rank()` methods. It depends on using the private `size()` method given at the beginning of this section that returns the number of nodes in the subtree rooted at a node.

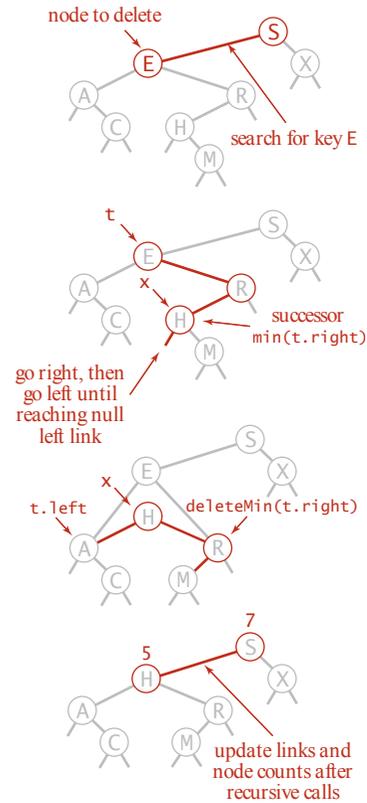
available for garbage collection. Our standard recursive setup accomplishes, after the deletion, the task of setting the appropriate link in the parent and updating the counts in all nodes in the path to the root. The symmetric method works for `deleteMax()`.

Delete. We can proceed in a similar manner to delete any node that has one child (or no children), but what can we do to delete a node that has two children? We are left with two links, but have a place in the parent node for only one of them. An answer to this dilemma, first proposed by T. Hibbard in 1962, is to delete a node x by replacing it with its successor. Because x has a right child, its successor is the node with the smallest key in its right subtree. The replacement preserves order in the tree because there are no keys between x .key and the successor's key. We can accomplish the task of replacing x by its successor in four (!) easy steps:

- Save a link to the node to be deleted in `t`.
- Set `x` to point to its successor `min(t.right)`.
- Set the right link of `x` (which is supposed to point to the BST containing all the keys larger than `x.key`) to `deleteMin(t.right)`, the link to the BST containing all the keys that are larger than `x.key` after the deletion.
- Set the left link of `x` (which was null) to `t.left` (all the keys that are less than both the deleted key and its successor).

Our standard recursive setup accomplishes, after the recursive calls, the task of setting the appropriate link in the parent and decrementing the node counts in the nodes on the path to the root (again, we accomplish the task of updating the counts by setting the counts in each node on the search path to be one plus the sum of the counts in its children). While this method does the job, it has a flaw that might cause performance problems in some practical situations. The problem is that the choice of using the successor is arbitrary and not symmetric. Why not use the predecessor? In practice, it is worthwhile to choose at random between the predecessor and the successor. See EXERCISE 3.2.42 for details.

deleting E



Deletion in a BST

ALGORITHM 3.3 (continued) Deletion in BSTs

```

public void deleteMin()
{
    root = deleteMin(root);
}

private Node deleteMin(Node x)
{
    if (x.left == null) return x.right;
    x.left = deleteMin(x.left);
    x.N = size(x.left) + size(x.right) + 1;
    return x;
}

public void delete(Key key)
{ root = delete(root, key); }

private Node delete(Node x, Key key)
{
    if (x == null) return null;
    int cmp = key.compareTo(x.key);
    if (cmp < 0) x.left = delete(x.left, key);
    else if (cmp > 0) x.right = delete(x.right, key);
    else
    {
        if (x.right == null) return x.left;
        if (x.left == null) return x.right;
        Node t = x;
        x = min(t.right); // See page 407.
        x.right = deleteMin(t.right);
        x.left = t.left;
    }
    x.N = size(x.left) + size(x.right) + 1;
    return x;
}

```

These methods implement eager Hibbard deletion in BSTs, as described in the text on the facing page. The `delete()` code is compact, but tricky. Perhaps the best way to understand it is to read the description at left, try to write the code yourself on the basis of the description, then compare your code with this code. This method is typically effective, but performance in large-scale applications can become a bit problematic (see EXERCISE 3.2.42). The `deleteMax()` method is the same as `deleteMin()` with right and left interchanged.

Range queries. To implement the `keys()` method that returns the keys in a given range, we begin with a basic recursive BST traversal method, known as *inorder traversal*. To illustrate the method, we consider the task of printing all the keys in a BST in order. To do so, print all the keys in the left subtree (which are less than the key at the root by

```
private void print(Node x)
{
    if (x == null) return;
    print(x.left);
    StdOut.println(x.key);
    print(x.right);
}
```

Printing the keys in a BST in order

definition of BSTs), then print the key at the root, then print all the keys in the right subtree (which are greater than the key at the root by definition of BSTs), as in the code at left. As usual, the description serves as an argument by induction that this code prints the keys in order. To implement the two-argument `keys()` method that returns to a client all the keys in a specified range, we modify this code to add each key that is in the range to a `Queue`, and to skip the recursive calls for subtrees that cannot contain keys in the range. As with `BinarySearchST`, the

fact that we gather the keys in a `Queue` is hidden from the client. The intention is that clients should process all the keys in the range of interest using Java's *foreach* construct rather than needing to know what data structure we use to implement `Iterable<Key>`.

Analysis. How efficient are the order-based operations in BSTs? To study this question, we consider the *tree height* (the maximum depth of any node in the tree). Given a tree, its height determines the worst-case cost of all BST operations (except for range search which incurs additional cost proportional to the number of keys returned).

Proposition E. In a BST, all operations take time proportional to the height of the tree, in the worst case.

Proof: All of these methods go down one or two paths in the tree. The length of any path is no more than the height, by definition.

We expect the tree height (the worst-case cost) to be higher than the average internal path length that we defined on page 403 (which averages in the short paths as well), but how much higher? This question may seem to you to be similar to the questions answered by `PROPOSITION C` and `PROPOSITION D`, but it is far more difficult to answer, certainly beyond the scope of this book. The average height of a BST built from random keys was shown to be logarithmic by J. Robson in 1979, and L. Devroye later showed that the value approaches $2.99 \lg N$ for large N . Thus, if the insertions in our application are well-described by the random-key model, we are well on the way toward our goal of developing a symbol-table implementation that supports all of these operations

ALGORITHM 3.3 (continued) Range searching in BSTs

```

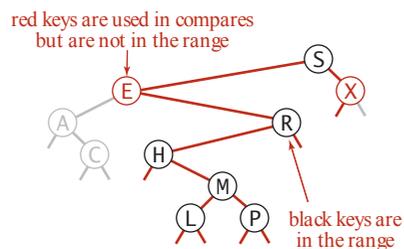
public Iterable<Key> keys()
{ return keys(min(), max()); }

public Iterable<Key> keys(Key lo, Key hi)
{
    Queue<Key> queue = new Queue<Key>();
    keys(root, queue, lo, hi);
    return queue;
}

private void keys(Node x, Queue<Key> queue, Key lo, Key hi)
{
    if (x == null) return;
    int cmplo = lo.compareTo(x.key);
    int cmphi = hi.compareTo(x.key);
    if (cmplo < 0) keys(x.left, queue, lo, hi);
    if (cmplo <= 0 && cmphi >= 0) queue.enqueue(x.key);
    if (cmphi > 0) keys(x.right, queue, lo, hi);
}

```

To enqueue all the keys from the tree rooted at a given node that fall in a given range onto a queue, we (recursively) enqueue all the keys from the left subtree (if any of them could fall in the range), then enqueue the node at the root (if it falls in the range), then (recursively) enqueue all the keys from the right subtree (if any of them could fall in the range).

searching in the range [F..T]**Range search in a BST**

in logarithmic time. We can expect that no path in a tree built from random keys is longer than $3 \lg N$, but what can we expect if the keys are not random? In the next section, you will see why this question is moot in practice because of *balanced BSTs*, which guarantee that the BST height will be logarithmic regardless of the order in which keys are inserted.

IN SUMMARY, BSTs are not difficult to implement and can provide fast search and insert for practical applications of all kinds if the key insertions are well-approximated by the random-key model. For our examples (and for many practical applications) BSTs make the difference between being able to accomplish the task at hand and not being able to do so. Moreover, many programmers choose BSTs for symbol-table implementations because they also support fast rank, select, delete, and range query operations. Still, as we have emphasized, the bad worst-case performance of BSTs may not be tolerable in some situations. Good performance of the basic BST implementation is dependent on the keys being sufficiently similar to random keys that the tree is not likely to contain many long paths. With quicksort, we were able to randomize; with a symbol-table API, we do not have that freedom, because the client controls the mix of operations. Indeed, the worst-case behavior is not unlikely in practice—it arises when a client inserts keys in order or in reverse order, a sequence of operations that some client certainly might attempt in the absence of any explicit warnings to avoid doing so. This possibility is a primary reason to seek better algorithms and data structures, which we consider next.

algorithm (data structure)	worst-case cost (after N inserts)		average-case cost (after N random inserts)		efficiently support ordered operations?
	search	insert	search hit	insert	
<i>sequential search</i> (<i>unordered linked list</i>)	N	N	$N/2$	N	no
<i>binary search</i> (<i>ordered array</i>)	$\lg N$	N	$\lg N$	$N/2$	yes
<i>binary tree search</i> (<i>BST</i>)	N	N	$1.39 \lg N$	$1.39 \lg N$	yes

Cost summary for basic symbol-table implementations (updated)

Q&A

- Q.** I've seen BSTs before, but not using recursion. What are the tradeoffs?
- A.** Generally, recursive implementations are a bit easier to verify for correctness; non-recursive implementations are a bit more efficient. See EXERCISE 3.2.13 for an implementation of `get()`, the one case where you might notice the improved efficiency. If trees are unbalanced, the depth of the function-call stack could be a problem in a recursive implementation. Our primary reason for using recursion is to ease the transition to the balanced BST implementations of the next section, which definitely are easier to implement and debug with recursion.
- Q.** Maintaining the node count field in `Node` seems to require a lot of code. Is it really necessary? Why not maintain a single instance variable containing the number of nodes in the tree to implement the `size()` client method?
- A.** The `rank()` and `select()` methods need to have the size of the subtree rooted at each node. If you are not using these ordered operations, you can streamline the code by eliminating this field (see EXERCISE 3.2.12). Keeping the node count correct for all nodes is admittedly error-prone, but also a good check for debugging. You might also use a recursive method to implement `size()` for clients, but that would take linear time to count all the nodes and is a dangerous choice because you might experience poor performance in a client program, not realizing that such a simple operation is so expensive.

EXERCISES

3.2.1 Draw the BST that results when you insert the keys E A S Y Q U E S T I O N, in that order (associating the value i with the i th key, as per the convention in the text) into an initially empty tree. How many compares are needed to build the tree?

3.2.2 Inserting the keys in the order A X C S E R H into an initially empty BST gives a worst-case tree where every node has one null link, except one at the bottom, which has two null links. Give five other orderings of these keys that produce worst-case trees.

3.2.3 Give five orderings of the keys A X C S E R H that, when inserted into an initially empty BST, produce the best-case tree.

3.2.4 Suppose that a certain BST has keys that are integers between 1 and 10, and we search for 5. Which sequence below cannot be the sequence of keys examined?

- a. 10, 9, 8, 7, 6, 5
- b. 4, 10, 8, 6, 5
- c. 1, 10, 2, 9, 3, 8, 4, 7, 6, 5
- d. 2, 7, 3, 8, 4, 5
- e. 1, 2, 10, 4, 8, 5

3.2.5 Suppose that we have an estimate ahead of time of how often search keys are to be accessed in a BST, and the freedom to insert them in any order that we desire. Should the keys be inserted into the tree in increasing order, decreasing order of likely frequency of access, or some other order? Explain your answer.

3.2.6 Add to BST a method `height()` that computes the height of the tree. Develop two implementations: a recursive method (which takes linear time and space proportional to the height), and a method like `size()` that adds a field to each node in the tree (and takes linear space and constant time per query).

3.2.7 Add to BST a recursive method `avgCompares()` that computes the average number of compares required by a random search hit in a given BST (the internal path length of the tree divided by its size, plus one). Develop two implementations: a recursive method (which takes linear time and space proportional to the height), and a method like `size()` that adds a field to each node in the tree (and takes linear space and constant time per query).

3.2.8 Write a static method `optCompares()` that takes an integer argument N and computes the number of compares required by a random search hit in an optimal (perfectly

balanced) BST with N nodes, where all the null links are on the same level if the number of links is a power of 2 or on one of two levels otherwise.

3.2.9 Draw all the different BST shapes that can result when N keys are inserted into an initially empty tree, for $N = 2, 3, 4, 5,$ and 6 .

3.2.10 Write a test client for BST that tests the implementations of `min()`, `max()`, `floor()`, `ceiling()`, `select()`, `rank()`, `delete()`, `deleteMin()`, `deleteMax()`, and `keys()` that are given in the text. Start with the standard indexing client given on page 370. Add code to take additional command-line arguments, as appropriate.

3.2.11 How many binary tree shapes of N nodes are there with height N ? How many different ways are there to insert N distinct keys into an initially empty BST that result in a tree of height N ? (See EXERCISE 3.2.2.)

3.2.12 Develop a BST implementation that omits `rank()` and `select()` and does not use a count field in `Node`.

3.2.13 Give nonrecursive implementations of `get()` and `put()` for BST.

Partial solution: Here is an implementation of `get()`:

```
public Value get(Key key)
{
    Node x = root;
    while (x != null)
    {
        int cmp = key.compareTo(x.key);
        if (cmp == 0) return x.val;
        else if (cmp < 0) x = x.left;
        else if (cmp > 0) x = x.right;
    }
    return null;
}
```

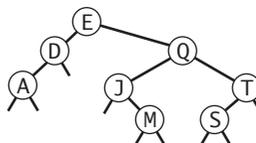
The implementation of `put()` is more complicated because of the need to save a pointer to the parent node to link in the new node at the bottom. Also, you need a separate pass to check whether the key is already in the table because of the need to update the counts. Since there are many more searches than inserts in performance-critical implementations, using this code for `get()` is justified; the corresponding change for `put()` might not be noticed.

EXERCISES (continued)

3.2.14 Give nonrecursive implementations of `min()`, `max()`, `floor()`, `ceiling()`, `rank()`, and `select()`.

3.2.15 Give the sequences of nodes examined when the methods in BST are used to compute each of the following quantities for the tree drawn at right.

- a. `floor("Q")`
- b. `select(5)`
- c. `ceiling("Q")`
- d. `rank("J")`
- e. `size("D", "T")`
- f. `keys("D", "T")`



3.2.16 Define the *external path length* of a tree to be the sum of the number of nodes on the paths from the root to all null links. Prove that the difference between the external and internal path lengths in any binary tree with N nodes is $2N$ (see PROPOSITION C).

3.2.17 Draw the sequence of BSTs that results when you delete the keys from the tree of EXERCISE 3.2.1, one by one, in the order they were inserted.

3.2.18 Draw the sequence of BSTs that results when you delete the keys from the tree of EXERCISE 3.2.1, one by one, in alphabetical order.

3.2.19 Draw the sequence of BSTs that results when you delete the keys from the tree of EXERCISE 3.2.1, one by one, by successively deleting the key at the root.

3.2.20 Prove that the running time of the two-argument `keys()` in a BST is at most proportional to the tree height plus the number of keys in the range.

3.2.21 Add a BST method `randomKey()` that returns a random key from the symbol table in time proportional to the tree height, in the worst case.

3.2.22 Prove that if a node in a BST has two children, its successor has no left child and its predecessor has no right child.

3.2.23 Is `delete()` commutative? (Does deleting x , then y give the same result as deleting y , then x ?)

3.2.24 Prove that no compare-based algorithm can build a BST using fewer than $\lg(N!) \sim N \lg N$ compares.

CREATIVE PROBLEMS

3.2.25 *Perfect balance.* Write a program that inserts a set of keys into an initially empty BST such that the tree produced is equivalent to binary search, in the sense that the sequence of compares done in the search for any key in the BST is the same as the sequence of compares used by binary search for the same key.

3.2.26 *Exact probabilities.* Find the probability that each of the trees in EXERCISE 3.2.9 is the result of inserting N random distinct elements into an initially empty tree.

3.2.27 *Memory usage.* Compare the memory usage of BST with the memory usage of `BinarySearchST` and `SequentialSearchST` for N key-value pairs, under the assumptions described in SECTION 1.4 (see EXERCISE 3.1.21). Do not count the memory for the keys and values themselves, but do count references to them. Then draw a diagram that depicts the precise memory usage of a BST with `String` keys and `Integer` values (such as the ones built by `FrequencyCounter`), and then estimate the memory usage (in bytes) for the BST built when `FrequencyCounter` uses BST for `Tale of Two Cities`.

3.2.28 *Software caching.* Modify BST to keep the most recently accessed `Node` in an instance variable so that it can be accessed in constant time if the next `put()` or `get()` uses the same key (see EXERCISE 3.1.25).

3.2.29 *Tree traversal with constant extra memory.* Design an algorithm that performs an inorder tree traversal of a BST using only a constant amount of extra memory. *Hint:* On the way down the tree, make the child point to the parent and reverse it on the way back up the tree.

3.2.30 *BST reconstruction.* Given the preorder (or postorder) traversal of a BST (not including null nodes), design an algorithm to reconstruct the BST.

CREATIVE PROBLEMS *(continued)*

3.2.31 Certification. Write a method `isBST()` that takes a `Node` as argument and returns `true` if the argument node is the root of a binary search tree, `false` otherwise. *Hint:* Write a helper method that takes a `Node` and two `Keys` as arguments and returns `true` if the argument node is the root of a binary search tree with all keys strictly between the two argument keys, `false` otherwise.

Solution:

```
private boolean isBST()
{ return isBST(root, null, null) }

private boolean isBST(Node x, Key min, Key max)
{
    if (x == null) return true;
    if (min != null && x.key.compareTo(min) <= 0) return false;
    if (max != null && x.key.compareTo(max) >= 0) return false;
    return isBST(x.left, min, x.key)
        && isBST(x.right, x.key, max);
}
```

3.2.32 Subtree count check. Write a recursive method that takes a `Node` as argument and returns `true` if the subtree count field `N` is consistent in the data structure rooted at that node, `false` otherwise.

3.2.33 Select/rank check. Write a method that checks, for all `i` from 0 to `size()-1`, whether `i` is equal to `rank(select(i))` and, for all keys in the BST, whether `key` is equal to `select(rank(key))`.

3.2.34 Threading. Your goal is to support an extended API `DoublyThreadedBST` that supports the following additional operations in constant time:

```
Key next(Key key)    key that follows key (null if key is the maximum)
Key prev(Key key)   key that precedes key (null if key is the minimum)
```

To do so, add fields `pred` and `succ` to `Node` that contain links to the predecessor and successor nodes, and modify `put()`, `deleteMin()`, `deleteMax()`, and `delete()` to maintain these fields.

3.2.35 *Refined analysis.* Refine the mathematical model to better explain the experimental results in the table given in the text. Specifically, show that the average number of compares for a successful search in a tree built from random keys approaches the limit $2 \ln N + 2\gamma - 3 \approx 1.39 \lg N - 1.85$ as N increases, where $\gamma = .57721\dots$ is Euler's constant. *Hint:* Referring to the quicksort analysis in SECTION 2.3, use the fact that the integral of $1/x$ approaches $\ln N + \gamma$.

3.2.36 *Iterator.* Is it possible to write a nonrecursive version of `keys()` that uses space proportional to the tree height (independent of the number of keys in the range)?

3.2.37 *Level-order traversal.* Write a method `printLevel()` that takes a `Node` as argument and prints the keys in the subtree rooted at that node in level order (in order of their distance from the root, with nodes on each level in order from left to right). *Hint:* Use a `Queue`.

3.2.38 *Tree drawing.* Add a method `draw()` to `BST` that draws `BST` figures in the style of the text. *Hint:* Use instance variables to hold node coordinates, and use a recursive method to set the values of these variables.

EXPERIMENTS

3.2.39 *Average case.* Run empirical studies to estimate the average and standard deviation of the number of compares used for search hits and for search misses in a BST built by running 100 trials of the experiment of inserting N random keys into an initially empty tree, for $N = 10^4$, 10^5 , and 10^6 . Compare your results against the formula for the average given in EXERCISE 3.2.35.

3.2.40 *Height.* Run empirical studies to estimate average BST height by running 100 trials of the experiment of inserting N random keys into an initially empty tree, for $N = 10^4$, 10^5 , and 10^6 . Compare your results against the $2.99 \lg N$ estimate that is described in the text.

3.2.41 *Array representation.* Develop a BST implementation that represents the BST with three arrays (preallocated to the maximum size given in the constructor): one with the keys, one with array indices corresponding to left links, and one with array indices corresponding to right links. Compare the performance of your program with that of the standard implementation.

3.2.42 *Hibbard deletion degradation.* Write a program that takes an integer N from the command line, builds a random BST of size N , then enters into a loop where it deletes a random key (using the code `delete(select(StdRandom.uniform(N)))`) and then inserts a random key, iterating the loop N^2 times. After the loop, measure and print the average length of a path in the tree (the internal path length divided by N , plus 1). Run your program for $N = 10^2$, 10^3 , and 10^4 to test the somewhat counterintuitive hypothesis that this process increases the average path length of the tree to be proportional to the square root of N . Run the same experiments for a `delete()` implementation that makes a random choice whether to use the predecessor or the successor node.

3.2.43 *Put/get ratio.* Determine empirically the ratio of the amount of time that BST spends on `put()` operations to the time that it spends on `get()` operations when `FrequencyCounter` is used to find the frequency of occurrence of values in 1 million randomly-generated integers.

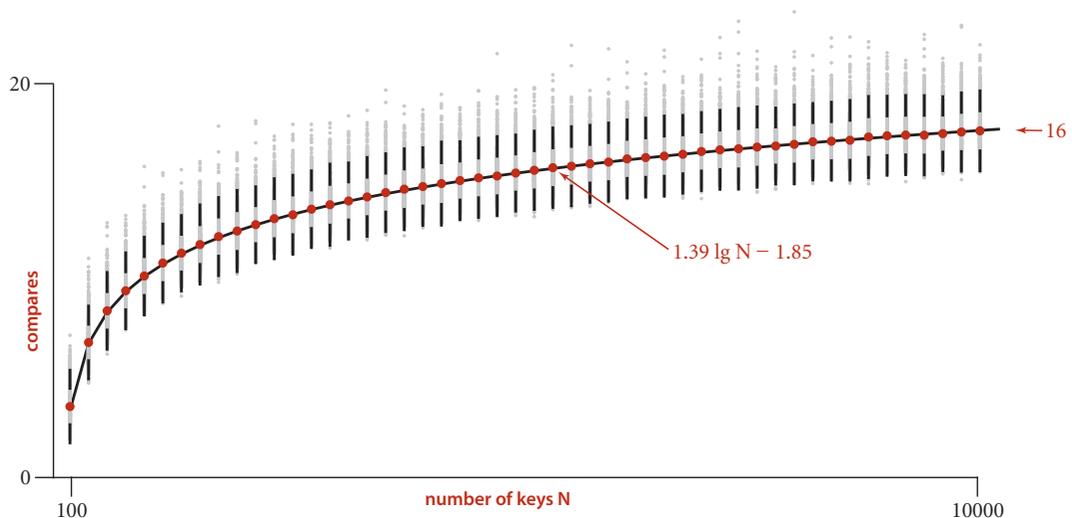
3.2.44 *Cost plots.* Instrument BST so that you can produce plots like the ones in this section showing the cost of each `put()` operation during the computation (see EXERCISE 3.1.38).

3.2.45 *Actual timings.* Instrument `FrequencyCounter` to use `Stopwatch` and `StdDraw` to make a plot where the x axis is the number of calls on `get()` or `put()` and the y axis

is the total running time, with a point plotted of the cumulative time after each call. Run your program for Tale of Two Cities using `SequentialSearchST` and again using `BinarySearchST` and again using `BST` and discuss the results. Note: Sharp jumps in the curve may be explained by *caching*, which is beyond the scope of this question (see EXERCISE 3.1.39).

3.2.46 *Crossover to binary search trees.* Find the values of N for which using a binary search tree to build a symbol table of N random double keys becomes 10, 100, and 1,000 times faster than binary search. Predict the values with analysis and verify them experimentally.

3.2.47 *Average search time.* Run empirical studies to compute the average and standard deviation of the average length of a path to a random node (internal path length divided by tree size, plus 1) in a BST built by insertion of N random keys into an initially empty tree, for N from 100 to 10,000. Do 1,000 trials for each tree size. Plot the results in a Tufte plot, like the one at the bottom of this page, fit with a curve plotting the function $1.39 \lg N - 1.85$ (see EXERCISE 3.2.35 and EXERCISE 3.2.39).

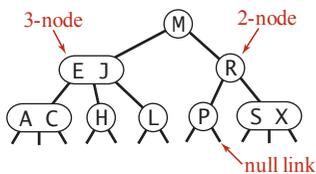


Average path length to a random node in a BST built from random keys

3.3 BALANCED SEARCH TREES

The algorithms in the previous section work well for a wide variety of applications, but they have poor worst-case performance. We introduce in this section a type of binary search tree where costs are guaranteed to be logarithmic, no matter what sequence of keys is used to construct them. Ideally, we would like to keep our binary search trees perfectly balanced. In an N -node tree, we would like the height to be $\sim \lg N$ so that we can guarantee that all searches can be completed in $\sim \lg N$ compares, just as for binary search (see PROPOSITION B). Unfortunately, maintaining perfect balance for dynamic insertions is too expensive. In this section, we consider a data structure that slightly relaxes the perfect balance requirement to provide guaranteed logarithmic performance not just for the insert and *search* operations in our symbol-table API but also for all of the ordered operations (except range search).

2-3 search trees The primary step to get the flexibility that we need to guarantee balance in search trees is to allow the nodes in our trees to hold more than one key. Specifically, referring to the nodes in a standard BST as 2-nodes (they hold two links and one key), we now also allow 3-nodes, which hold three links and two keys. Both 2-nodes and 3-nodes have one link for each of the intervals subtended by its keys.



Anatomy of a 2-3 search tree

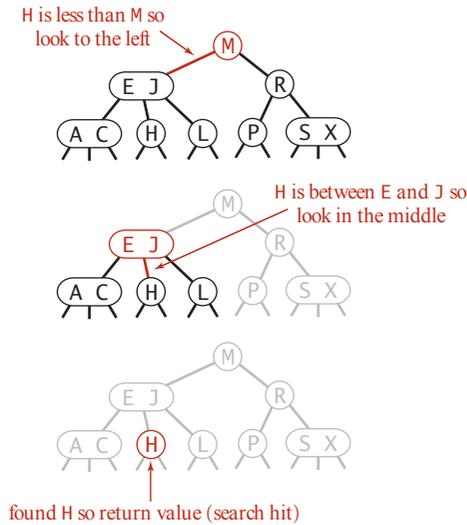
Definition. A 2-3 search tree is a tree that is either empty or

- A 2-node, with one key (and associated value) and two links, a left link to a 2-3 search tree with smaller keys, and a right link to a 2-3 search tree with larger keys
- A 3-node, with two keys (and associated values) and *three* links, a left link to a 2-3 search tree with smaller keys, a middle link to a 2-3 search tree with keys between the node's keys, and a right link to a 2-3 search tree with larger keys

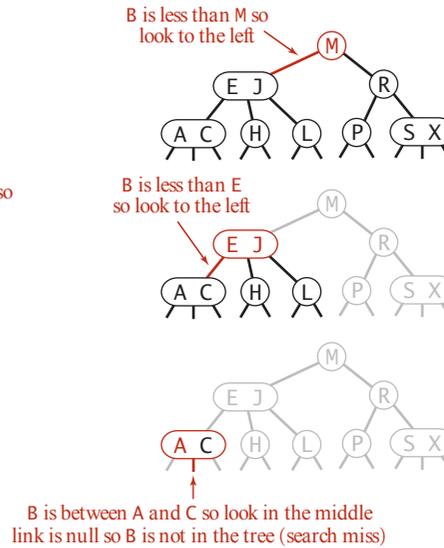
As usual, we refer to a link to an empty tree as a null link.

A perfectly balanced 2-3 search tree is one whose null links are all the same distance from the root. To be concise, we use the term *2-3 tree* to refer to a perfectly balanced 2-3 search tree (the term denotes a more general structure in other contexts). Later, we shall see efficient ways to define and implement the basic operations on 2-nodes, 3-nodes, and 2-3 trees; for now, let us assume that we can manipulate them conveniently and see how we can use them as search trees.

successful search for H



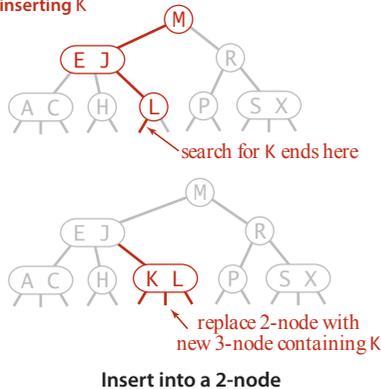
unsuccessful search for B



Search hit (left) and search miss (right) in a 2-3 tree

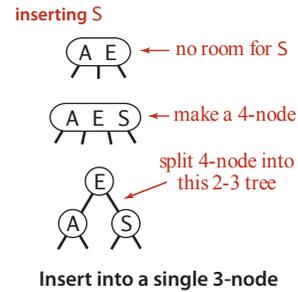
Search. The search algorithm for keys in a 2-3 tree directly generalizes the search algorithm for BSTs. To determine whether a key is in the tree, we compare it against the keys at the root. If it is equal to any of them, we have a search hit; otherwise, we follow the link from the root to the subtree corresponding to the interval of key values that could contain the search key. If that link is null, we have a search miss; otherwise we recursively search in that subtree.

inserting K



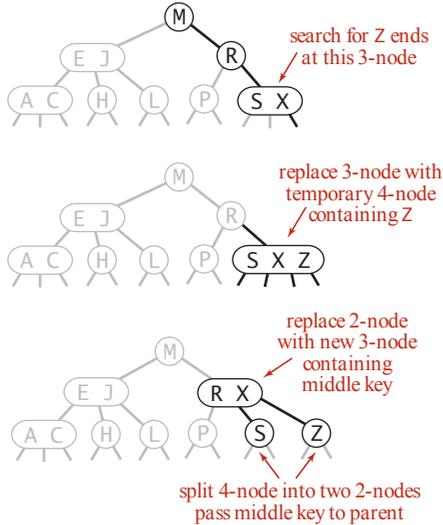
Insert into a 2-node. To insert a new key in a 2-3 tree, we might do an unsuccessful search and then hook on a new node with the key at the bottom, as we did with BSTs, but the new tree would not remain perfectly balanced. The primary reason that 2-3 trees are useful is that we can do insertions and still maintain perfect balance. It is easy to accomplish this task if the node at which the search terminates is a 2-node: we just replace the node with a 3-node containing its key and the new key to be inserted. If the node where the search terminates is a 3-node, we have more work to do.

Insert into a tree consisting of a single 3-node. As a first warmup before considering the general case, suppose that we want to insert into a tiny 2-3 tree consisting of just a single 3-node. Such a tree has two keys, but no room for a new key in its one node. To be able to perform the insertion, we temporarily put the new key into a 4-node, a natural extension of our node type that has three keys and four links. Creating the 4-node is convenient because it is easy to convert it into a 2-3 tree made up of three 2-nodes, one with the middle key (at the root), one with the smallest of the three keys (pointed to by the left link of the root), and one with the largest of the three keys (pointed to by the right link of the root). Such a tree is a 3-node BST and also a perfectly balanced 2-3 search tree, with all the null links at the same distance from the root. Before the insertion, the height of the tree is 0; after the insertion, the height of the tree is 1. This case is simple, but it is worth considering because it illustrates height growth in 2-3 trees.



Insert into a 3-node whose parent is a 2-node. As a second warmup, suppose that the search ends at a 3-node at the bottom whose parent is a 2-node. In this case, we can still make room for the new key *while maintaining perfect balance in the tree*, by making a temporary 4-node as just described, then splitting the 4-node as just described, but then, instead of creating a new node to hold the middle key, moving the middle key to the node's parent. You can think of the transformation as replacing the link to the old 3-node in the parent by the middle key with links on either side to the new 2-nodes. By our assumption, there is room for doing so in the parent: the parent was a 2-node (with one key and two links) and becomes a 3-node (with two keys and three links). Also, this transformation does not affect the defining properties of (perfectly balanced) 2-3 trees. The tree remains ordered because the middle key is moved to the parent, and it remains perfectly balanced: if all null links are the same distance from the root before the insertion, they are all the same distance from the root after the insertion. Be certain that you understand this transformation—it is the crux of 2-3 tree dynamics.

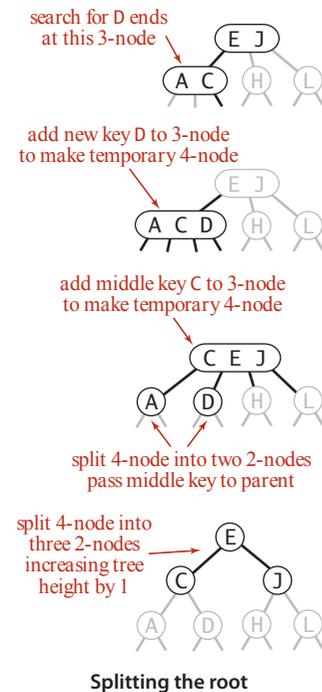
inserting Z



Insert into a 3-node whose parent is a 3-node. Now suppose that the search ends at a node whose parent is a 3-node. Again, we make a temporary 4-node as just described, then split it and insert its middle key into the parent. The parent was a 3-node, so we replace it with a temporary new 4-node containing the middle key from the 4-node split. Then, we perform precisely *the same transformation on that node*. That is, we split the new 4-node and insert its middle key into its parent. Extending to the general case is clear: we continue up the tree, splitting 4-nodes and inserting their middle keys in their parents until reaching a 2-node, which we replace with a 3-node that does not need to be further split, or until reaching a 3-node at the root.

Splitting the root. If we have 3-nodes along the whole path from the insertion point to the root, we

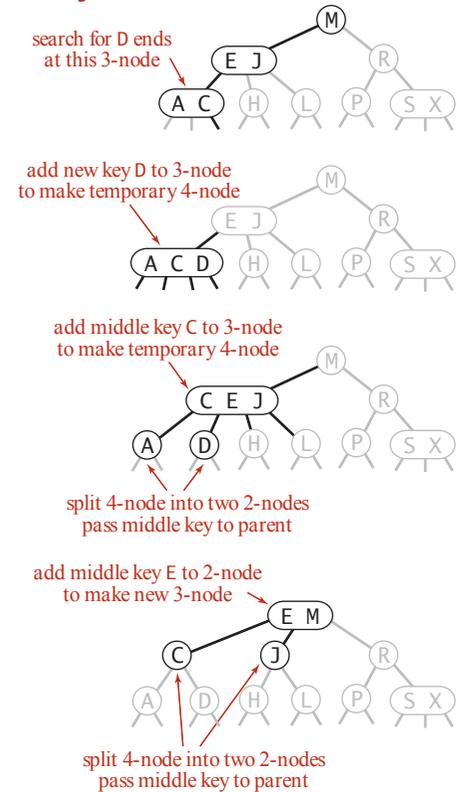
inserting D



end up with a temporary 4-node at the root. In this case we can proceed in precisely the same way as for insertion into a tree consisting of a single 3-node. We split the temporary 4-node into three 2-nodes, increasing the height of the tree by 1. Note that this last transformation preserves perfect balance because it is performed at the root.

Local transformations. Splitting a temporary 4-node in a 2-3 tree involves one of six transformations, summarized at the bottom of the next page. The 4-node may be the root; it may be the left child or the right child of a 2-node; or it may be the left child, middle child, or right child of a 3-node. The basis of the 2-3 tree insertion algorithm is that all of these transformations are purely local: no part of the tree needs to be examined or modified other than the specified nodes and links. The number of

inserting D

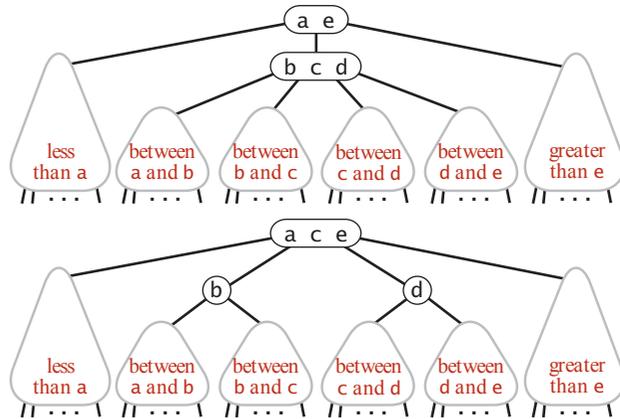


Insert into a 3-node whose parent is a 3-node

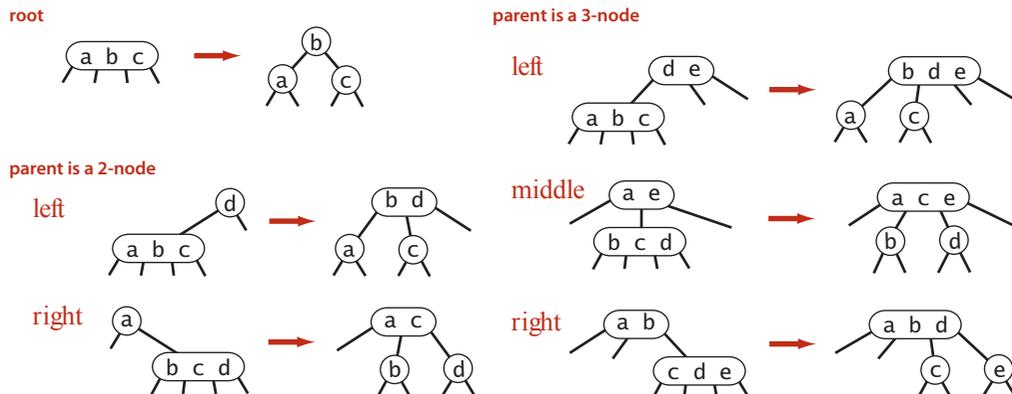
links changed for each transformation is bounded by a small constant. In particular, the transformations are effective when we find the specified patterns *anywhere* in the tree, not just at the bottom. Each of the transformations passes up one of the keys from a 4-node to that node's parent in the tree and then restructures links accordingly, without touching any other part of the tree.

Global properties. Moreover, these local transformations

preserve the global properties that the tree is ordered and perfectly balanced: the number of links on the path from the root to any null link is the same. For reference, a complete diagram illustrating this point for the case that the 4-node is the middle child of a 3-node is shown above. If the length of every path from a root to a null link is h before the transformation, then it is h after the transformation. *Each transformation preserves this property*, even while splitting the 4-node into two 2-nodes and while changing the parent from a 2-node to a 3-node or from a 3-node into a temporary 4-node. When the root splits into three 2-nodes, the length of every path from the root to a null link increases by 1. If you are not fully convinced, work EXERCISE 3.3.7, which asks you to



Splitting a 4-node is a local transformation that preserves order and perfect balance



Splitting a temporary 4-node in a 2-3 tree (summary)

extend the diagrams at the top of the previous page for the other five cases to illustrate the same point. Understanding that every local transformation preserves order and perfect balance in the whole tree is the key to understanding the algorithm.

UNLIKE STANDARD BSTS, which grow down from the top, 2-3 trees grow up from the bottom. If you take the time to carefully study the figure on the next page, which gives the sequence of 2-3 trees that is produced by our standard indexing test client and the sequence of 2-3 trees that is produced when the same keys are inserted in increasing order, you will have a good understanding of the way that 2-3 trees are built. Recall that in a BST, the increasing-order sequence for 10 keys results in a worst-case tree of height 9. In the 2-3 trees, the height is 2.

The preceding description is sufficient to define a symbol-table implementation with 2-3 trees as the underlying data structure. Analyzing 2-3 trees is different from analyzing BSTs because our primary interest is in worst-case performance, as opposed to average-case performance (where we analyze expected performance under the random-key model). In symbol-table implementations, we normally have no control over the order in which clients insert keys into the table and worst-case analysis is one way to provide performance guarantees.

Proposition F. Search and insert operations in a 2-3 tree with N keys are guaranteed to visit at most $\lg N$ nodes.

Proof: The height of an N -node 2-3 tree is between $\lceil \log_3 N \rceil = \lceil (\lg N) / (\lg 3) \rceil$ (if the tree is all 3-nodes) and $\lfloor \lg N \rfloor$ (if the tree is all 2-nodes) (see EXERCISE 3.3.4).

Thus, we can guarantee good worst-case performance with 2-3 trees. The amount of time required at each node by each of the operations is bounded by a constant, and both operations examine nodes on just one path, so the total cost of any search or insert is guaranteed to be logarithmic. As you can see from comparing the 2-3 tree depicted at the bottom of page 431 with the BST formed from the same keys on page 405, a perfectly balanced 2-3 tree strikes a remarkably flat posture. For example, the height of a 2-3 tree that contains 1 billion keys is between 19 and 30. It is quite remarkable that we can guarantee to perform arbitrary search and insertion operations among 1 billion keys by examining at most 30 nodes.

However, we are only part of the way to an implementation. Although it is possible to write code that performs transformations on distinct data types representing 2- and 3-nodes, most of the tasks that we have described are inconvenient to implement in

insert S



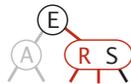
E



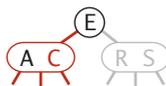
A



R



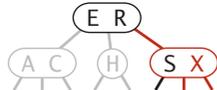
C



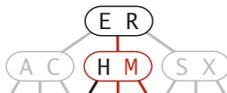
H



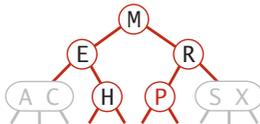
X



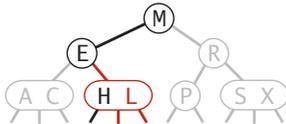
M



P



L



standard indexing client

insert A



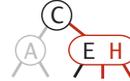
C



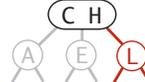
E



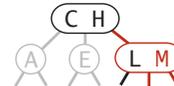
H



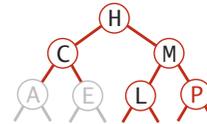
L



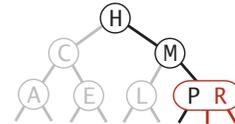
M



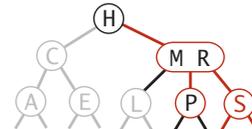
P



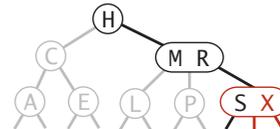
R



S



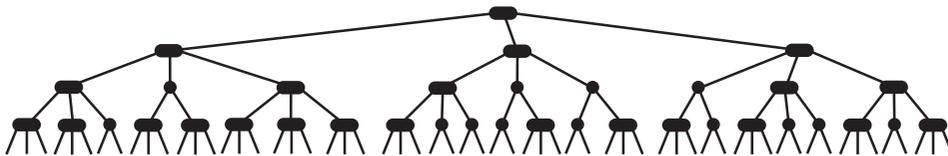
X



same keys in increasing order

2-3 construction traces

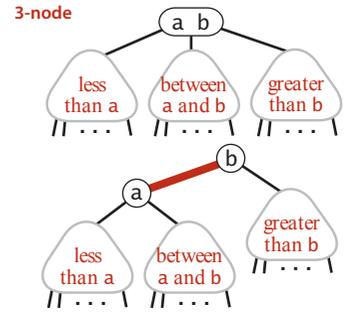
this direct representation because there are numerous different cases to be handled. We would need to maintain two different types of nodes, compare search keys against each of the keys in the nodes, copy links and other information from one type of node to another, convert nodes from one type to another, and so forth. Not only is there a substantial amount of code involved, but the overhead incurred could make the algorithms slower than standard BST search and insert. The primary purpose of balancing is to provide insurance against a bad worst case, but we would prefer the overhead cost for that insurance to be low. Fortunately, as you will see, we can do the transformations in a uniform way using little overhead.



Typical 2-3 tree built from random keys

Red-black BSTs The insertion algorithm for 2-3 trees just described is not difficult to understand; now, we will see that it is also not difficult to implement. We will consider a simple representation known as a *red-black BST* that leads to a natural implementation. In the end, not much code is required, but understanding how and why the code gets the job done requires a careful look.

Encoding 3-nodes. The basic idea behind red-black BSTs is to encode 2-3 trees by starting with standard BSTs (which are made up of 2-nodes) and adding extra information to encode 3-nodes. We think of the links as being of two different types: red links, which bind together two 2-nodes to represent 3-nodes, and black links, which bind together the 2-3 tree. Specifically, we represent 3-nodes as two 2-nodes connected by a single red link that leans left (one of the 2-nodes is the left child of the other). One advantage of using such a representation is that it allows us to use our `get()` code for standard BST search *without modification*. Given any 2-3 tree, we can immediately derive a corresponding BST, just by converting each node as specified. We refer to BSTs that represent 2-3 trees in this way as *red-black BSTs*.



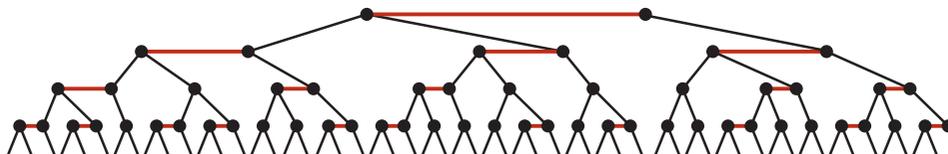
Encoding a 3-node with two 2-nodes connected by a left-leaning red link

An equivalent definition. Another way to proceed is to *define* red-black BSTs as BSTs having red and black links and satisfying the following three restrictions:

- Red links lean left.
- No node has two red links connected to it.
- The tree has perfect black balance: every path from the root to a null link has the same number of black links—we refer to this number as the tree's *black height*.

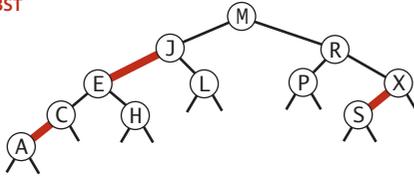
There is a 1-1 correspondence between red-black BSTs defined in this way and 2-3 trees.

A 1-1 correspondence. If we draw the red links horizontally in a red-black BST, all of the null links are the same distance from the root, and if we then collapse together the nodes connected by red links, the result is a 2-3 tree. Conversely, if we draw 3-nodes in

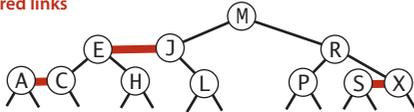


A red-black tree with horizontal red links is a 2-3 tree

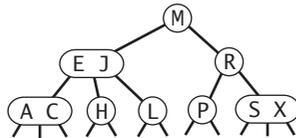
red-black BST



horizontal red links



2-3 tree

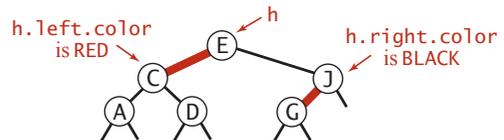


1-1 correspondence between red-black BSTs and 2-3 trees

Color representation. For convenience, since each node is pointed to by precisely one link (from its parent), we encode the color of links in nodes, by adding a boolean instance variable `color` to our `Node` data type, which is `true` if the link from the parent is red and `false` if it is black. By convention, null links are black. For clarity in our code, we define constants `RED` and `BLACK` for use in setting and testing this variable. We use a private method `isRed()` to test the color of a node's link to its parent. When we refer to the color of a node, we are referring to the color of the link pointing to it, and vice versa.

Rotations. The implementation that we will consider might allow right-leaning red links or two red links in a row during an operation, but it always corrects these conditions before completion, through judicious use of an operation called rotation that switches the orientation of

a 2-3 tree as two 2-nodes connected by a red link that leans left, then no node has two red links connected to it, and the tree has perfect black balance, since the black links correspond to the 2-3 tree links, which are perfectly balanced by definition. Whichever way we choose to define them, red-black BSTs are *both* BSTs and 2-3 trees. Thus, if we can implement the 2-3 tree insertion algorithm by maintaining the 1-1 correspondence, then we get the best of both worlds: the simple and efficient search method from standard BSTs and the efficient insertion-balancing method from 2-3 trees.



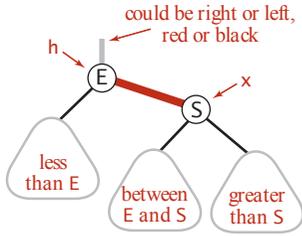
```
private static final boolean RED = true;
private static final boolean BLACK = false;

private class Node
{
    Key key;           // key
    Value val;        // associated data
    Node left, right; // subtrees
    int N;            // # nodes in this subtree
    boolean color;    // color of link from
                    // parent to this node

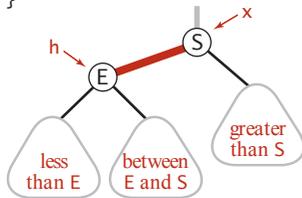
    Node(Key key, Value val, int N, boolean color)
    {
        this.key = key;
        this.val = val;
        this.N = N;
        this.color = color;
    }
}

private boolean isRed(Node x)
{
    if (x == null) return false;
    return x.color == RED;
}
```

Node representation for red-black BSTs



```
Node rotateLeft(Node h)
{
    Node x = h.right;
    h.right = x.left;
    x.left = h;
    x.color = h.color;
    h.color = RED;
    x.N = h.N;
    h.N = 1 + size(h.left)
        + size(h.right);
    return x;
}
```



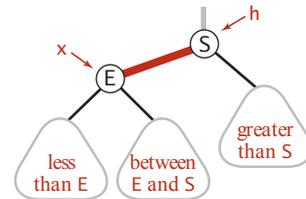
Left rotate (right link of h)

red links. First, suppose that we have a right-leaning red link that needs to be rotated to lean to the left (see the diagram at left). This operation is called a left rotation. We organize the computation as a method that takes a link to a red-black BST as argument and, assuming that link to be to a Node *h* whose right link is red, makes the necessary adjustments and returns a link to a node that is the root of a red-black BST for the same set of keys whose left link is red. If you check each of the lines of code against the before/after drawings in the diagram, you will find this operation is easy to understand: we are switching from having the smaller of the two keys at the root to having the larger of the two keys at the root. Implementing a *right* rotation that converts a left-leaning red link to a right-leaning one amounts to the same code, with left and right interchanged (see the diagram at right below).

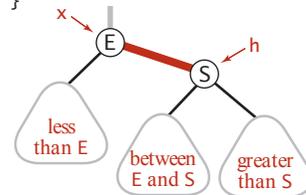
Resetting the link in the parent after a rotation. Whether left or right, every rotation leaves us with a link. We always use the link returned by `rotateRight()` or `rotateLeft()` to reset the appropriate link in the parent (or the root of the tree). That may be a right or a left link, but we can always use it to reset the link in the parent. This link may be red or black—both `rotateLeft()` and `rotateRight()` preserve its color by setting `x.color` to `h.color`. This might allow two red links in a row to occur within the tree, but our algorithms will also use rotations to correct this condition when it arises. For example, the code

```
h = rotateLeft(h);
```

rotates left a right-leaning red link that is to the right of node *h*, setting *h* to point to the root of the resulting subtree (which contains all the same nodes as the subtree pointed to by *h* before the rotation, but a different root). The ease of writing this type of code is the primary reason we use recursive implementations of BST methods, as it makes doing rotations an easy supplement to normal insertion, as you will see.



```
Node rotateRight(Node h)
{
    Node x = h.left;
    h.left = x.right;
    x.right = h;
    x.color = h.color;
    h.color = RED;
    x.N = h.N;
    h.N = 1 + size(h.left)
        + size(h.right);
    return x;
}
```



Right rotate (left link of h)

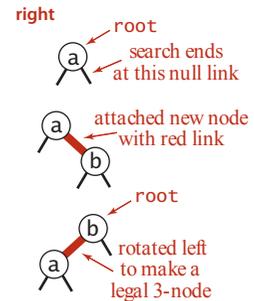
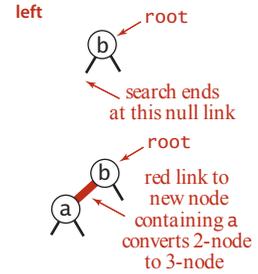
WE CAN USE ROTATIONS to help maintain the 1-1 correspondence between 2-3 trees and red-black BSTs as new keys are inserted because they preserve the two defining properties of red-black BSTs: order and perfect black balance. That is, we can use rotations on a red-black BST without having to worry about losing its order or its perfect black balance. Next, we see how to use rotations to preserve the other two defining properties of red-black BSTs (no consecutive red links on any path and no right-leaning red links). We warm up with some easy cases.

Insert into a single 2-node. A red-black BST with 1 key is just a single 2-node. Inserting the second key immediately shows the need for having a rotation operation. If the new key is smaller than the key in the tree, we just make a new (red) node with the new key and we are done: we have a red-black BST that is equivalent to a single 3-node. But if the new key is larger than the key in the tree, then attaching a new (red) node gives a right-leaning red link, and the code `root = rotateLeft(root)`; completes the insertion by rotating the red link to the left and updating the tree root link. The result in both cases is the red-black representation of a single 3-node, with two keys, one left-leaning red link, and black height 0.

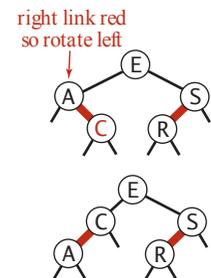
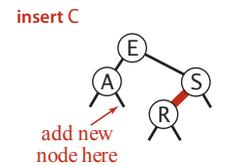
Insert into a 2-node at the bottom. We insert keys into a red-black BST as usual into a BST, adding a new node at the bottom (respecting the order), but always connected to its parent with a red link. If the parent is a 2-node, then the same two cases just discussed are effective. If the new node is attached to the left link, the parent simply becomes a 3-node; if it is attached to a right link, we have a 3-node leaning the wrong way, but a left rotation finishes the job.

Insert into a tree with two keys (in a 3-node). This case reduces to three subcases: the new key is either less than both keys in the tree, between them, or greater than both of them. Each of the cases introduces a node with two red links connected to it; our goal is to correct this condition.

- The simplest of the three cases is when the new key is larger than the two in the tree and is therefore attached on the rightmost link of the 3-node, making a balanced tree with the middle key at the root, connected with red links to nodes containing a smaller and a larger key. If we flip the colors of those two links from red to black, then we have a balanced tree of (black) height 1 with three nodes, exactly what we need to maintain our 1-1 correspondence to 2-3 trees. The other two cases eventually reduce to this case.

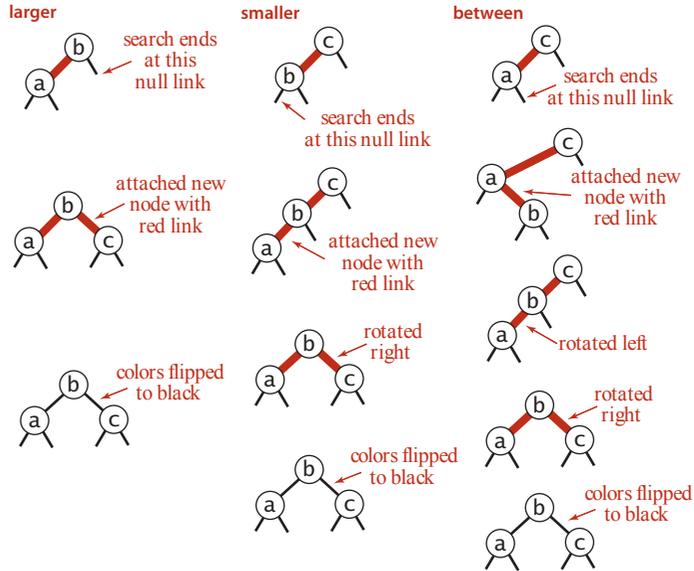


Insert into a single 2-node (two cases)

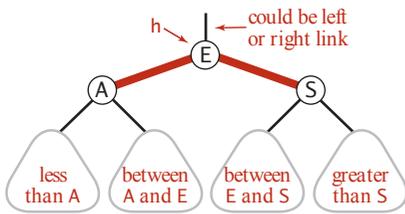


Insert into a 2-node at the bottom

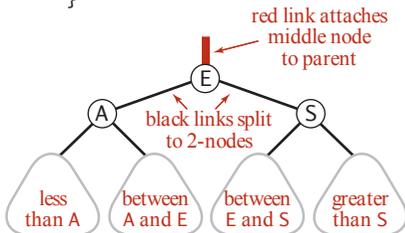
- If the new key is smaller than the two keys in the tree and goes on the left link, then we have two red links in a row, both leaning to the left, which we can reduce to the previous case (middle key at the root, connected to the others by two red links) by rotating the top link to the right.
- If the new key goes between the two keys in the tree, we



Insert into a single 3-node (three cases)



```
void flipColors(Node h)
{
    h.color = RED;
    h.left.color = BLACK;
    h.right.color = BLACK;
}
```



Flipping colors to split a 4-node

again have two red links in a row, a right-leaning one below a left-leaning one, which we can reduce to the previous case (two red links in a row, to the left) by rotating left the bottom link.

In summary, we achieve the desired result by doing zero, one, or two rotations followed by flipping the colors of the two children of the root. As with 2-3 trees, *be certain that you understand these transformations*, as they are the key to red-black tree dynamics.

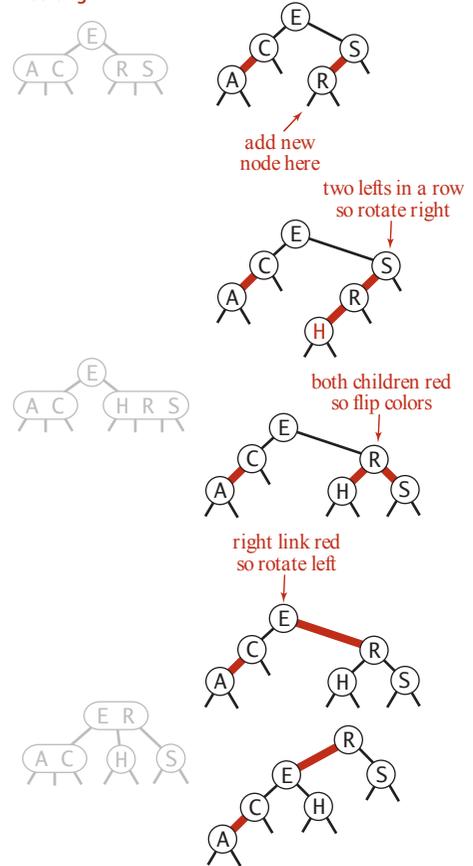
Flipping colors. To flip the colors of the two red children of a node, we use a method `flipColors()`, shown at left. In addition to flipping the colors of the children from red to black, we also flip the color of the parent from black to red. A critically important characteristic of this operation is that, like rotations, it is a local transformation that preserves perfect black balance in the tree. Moreover, this convention immediately leads us to a full implementation, as we describe next.

Keeping the root black. In the case just considered (insert into a single 3-node), the color flip will color the root red. This can also happen in larger trees. Strictly speaking, a red root implies that the root is part of a 3-node, but that is not the case, so we color the root black after each insertion. Note that the black height of the tree increases by 1 whenever the root is involved in a color flip, where its children's colors are both flipped from red to black.

Insert into a 3-node at the bottom. Now suppose that we add a new node at the bottom that is connected to a 3-node. The same three cases just discussed arise. Either the new link is connected to the right link of the 3-node (in which case we just flip colors) or to the left link of the 3-node (in which case we need to rotate the top link right and flip colors) or to the middle link of the 3-node (in which case we rotate left the bottom link, then rotate right the top link, then flip colors). Flipping the colors makes the link to the middle node red, which amounts to passing it up to its parent, putting us back in the same situation with respect to the parent, which we can fix by moving up the tree.

Passing a red link up the tree. The 2-3 tree insertion algorithm calls for us to split the 3-node, passing the middle key up to be inserted into its parent, continuing until encountering a 2-node or the root. In every case we have considered, we precisely accomplish this objective: after doing any necessary rotations, we flip colors, which turns the middle node to red. From the point of view of the parent of that node, that link becoming red can be handled in precisely the same manner as if the red link came from attaching a new node: we pass up a red link to the middle node. The three cases summarized in the diagram on the next page precisely capture the operations necessary in a red-black tree to implement the key operation in 2-3 tree insertion: to insert into a 3-node, create a temporary 4-node, split it, and pass a red link to the middle key up to its parent. Continuing the same process, we pass a red link up the tree until reaching a 2-node or the root.

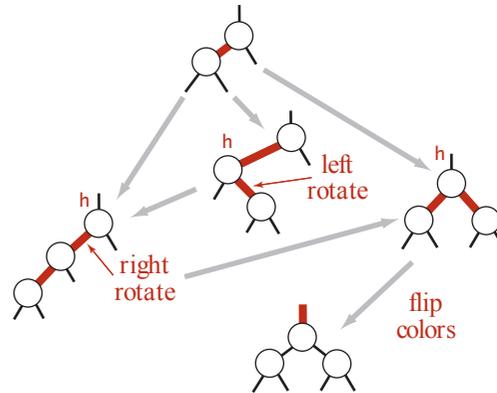
inserting H



Insert into a 3-node at the bottom

IN SUMMARY, WE CAN MAINTAIN OUR 1-1 correspondence between 2-3 trees and red-black BSTs during insertion by judicious use of three simple operations: left rotate, right rotate, and color flip. We can accomplish the insertion by performing the following operations, one after the other, on each node as we pass up the tree from the point of insertion:

- If the right child is red and the left child is black, rotate left.
- If both the left child and its left child are red, rotate right.
- If both children are red, flip colors.



Passing a red link up a red-black BST

It certainly is worth your while to check that this sequence handles each of the cases just described. Note that the first operation handles both the rotation necessary to lean the 3-node to the left when the parent is a 2-node and the rotation necessary to lean the bottom link to the left when the new red link is the middle link in a 3-node.

Implementation Since the balancing operations are to be performed on the way up the tree from the point of insertion, implementing them is easy in our standard recursive implementation: we just do them after the recursive calls, as shown in ALGORITHM 3.4. The three operations listed in the previous paragraph each can be accomplished with a single `if` statement that tests the colors of two nodes in the tree. Even though it involves a small amount of code, this implementation would be quite difficult to understand without the two layers of abstraction that we have developed (2-3 trees and red-black BSTs) to implement it. At a cost of testing three to five node colors (and perhaps doing a rotation or two or flipping colors when a test succeeds), we get BSTs that have nearly perfect balance.

The traces for our standard indexing client and for the same keys inserted in increasing order are given on page 440. Considering these examples simply in terms of our three operations on red-black trees, as we have been doing, is an instructive exercise. Another instructive exercise is to check the correspondence with 2-3 trees that the algorithm maintains (using the figure for the same keys given on page 430). In both cases, you can test your understanding of the algorithm by considering the transformations (two color flips and two rotations) that are needed when `P` is inserted into the red-black BST (see EXERCISE 3.3.12).

ALGORITHM 3.4 Insert for red-black BSTs

```

public class RedBlackBST<Key extends Comparable<Key>, Value>
{
    private Node root;

    private class Node // BST node with color bit (see page 433)
    {
        private boolean isRed(Node h) // See page 433.
        private Node rotateLeft(Node h) // See page 434.
        private Node rotateRight(Node h) // See page 434.
        private void flipColors(Node h) // See page 436.

        private int size() // See page 398.

        public void put(Key key, Value val)
        { // Search for key. Update value if found; grow table if new.
            root = put(root, key, val);
            root.color = BLACK;
        }

        private Node put(Node h, Key key, Value val)
        {
            if (h == null) // Do standard insert, with red link to parent.
                return new Node(key, val, 1, RED);

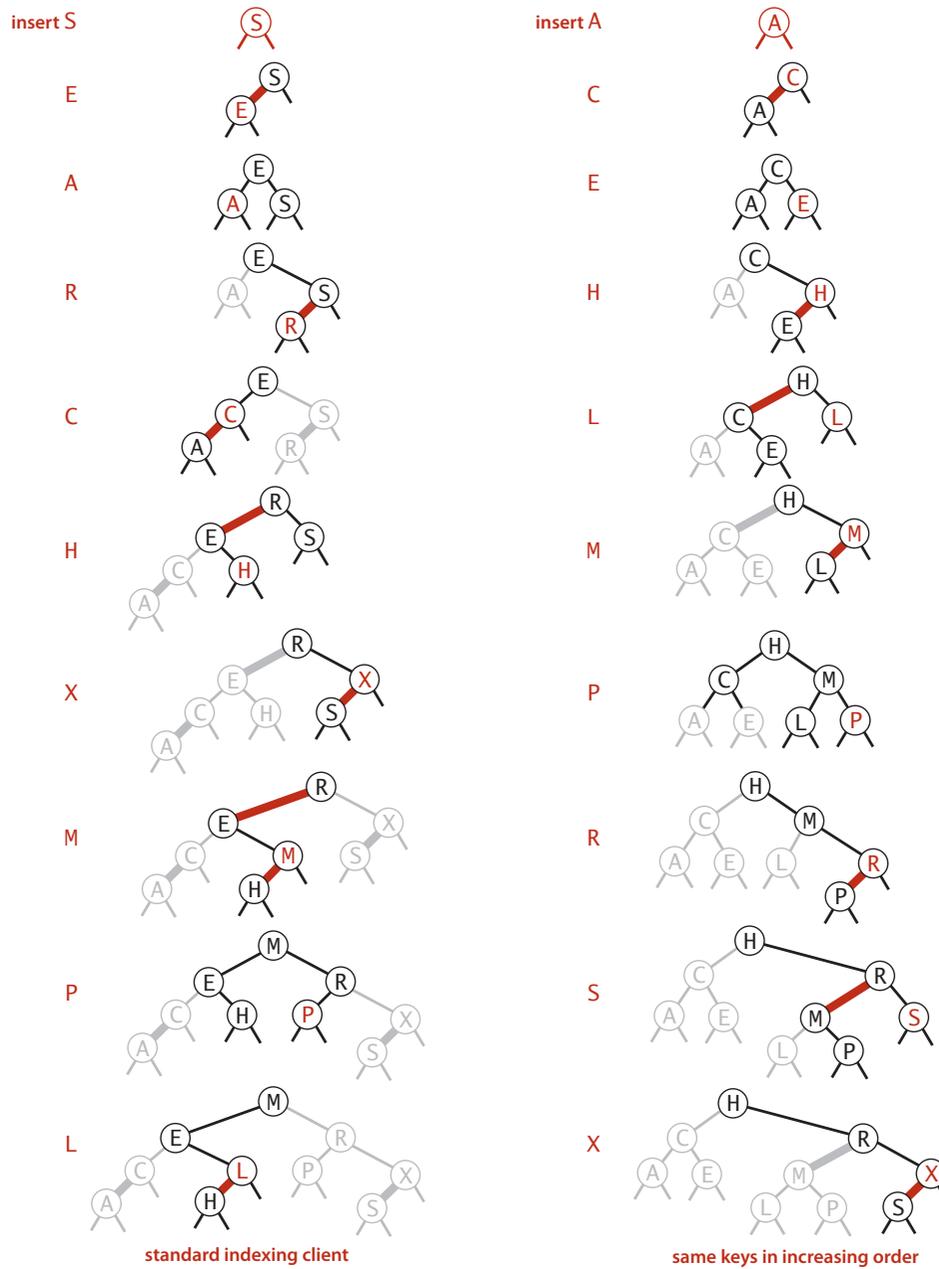
            int cmp = key.compareTo(h.key);
            if (cmp < 0) h.left = put(h.left, key, val);
            else if (cmp > 0) h.right = put(h.right, key, val);
            else h.val = val;

            if (isRed(h.right) && !isRed(h.left)) h = rotateLeft(h);
            if (isRed(h.left) && isRed(h.left.left)) h = rotateRight(h);
            if (isRed(h.left) && isRed(h.right)) flipColors(h);

            h.N = size(h.left) + size(h.right) + 1;
            return h;
        }
    }
}

```

The code for the recursive `put()` for red-black BSTs is identical to `put()` in elementary BSTs except for the three `if` statements after the recursive calls, which provide near-perfect balance in the tree by maintaining a 1-1 correspondence with 2-3 trees, on the way up the search path. The first rotates left any right-leaning 3-node (or a right-leaning red link at the bottom of a temporary 4-node); the second rotates right the top link in a temporary 4-node with two left-leaning red links; and the third flips colors to pass a red link up the tree (see text).

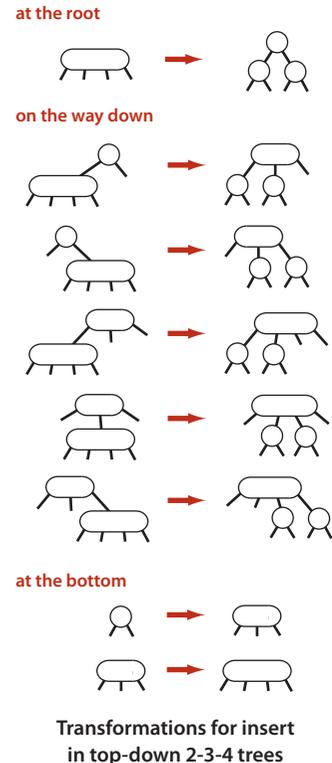


Red-black BST construction traces

Deletion Since `put()` in ALGORITHM 3.4 is already one of the most intricate methods that we consider in this book, and the implementations of `deleteMin()`, `deleteMax()`, and `delete()` for red-black BSTs are a bit more complicated, we defer their full implementations to exercises. Still, the basic approach is worthy of study. To describe it, we begin by returning to 2-3 trees. As with insertion, we can define a sequence of local transformations that allow us to delete a node while still maintaining perfect balance. The process is somewhat more complicated than for insertion, because we do the transformations both on the way down the search path, when we introduce temporary 4-nodes (to allow for a node to be deleted), and also on the way up the search path, where we split any leftover 4-nodes (in the same manner as for insertion).

Top-down 2-3-4 trees. As a first warmup for deletion, we consider a simpler algorithm that does transformations both on the way down the path and on the way up the path: an insertion algorithm for 2-3-4 trees, where the temporary 4-nodes that we saw in 2-3 trees can persist in the tree. The insertion algorithm is based on doing transformations on the way down the path to maintain the invariant that the current node is not a 4-node (so we are assured that there will be room to insert the new key at the bottom) and transformations on the way up the path to balance any 4-nodes that may have been created. The transformations on the way down are precisely the same transformations that we used for splitting 4-nodes in 2-3 trees. If the root is a 4-node, we split it into three 2-nodes, increasing the height of the tree by 1. On the way down the tree, if we encounter a 4-node with a 2-node as parent, we split the 4-node into two 2-nodes and pass the middle key to the parent, making it a 3-node; if we encounter a 4-node with a 3-node as parent, we split the 4-node into two 2-nodes and pass the middle key to the parent, making it a 4-node. We do not need to worry about encountering a 4-node with a 4-node as parent by virtue of the invariant. At the bottom, we have, again by virtue of the invariant, a 2-node or a 3-node, so we have room to insert the new key. To implement this algorithm with red-black BSTs, we

- Represent 4-nodes as a balanced subtree of three 2-nodes, with both the left and right child connected to the parent with a red link
- Split 4-nodes on the way down the tree with color flips
- Balance 4-nodes on the way up the tree with rotations, as for insertion



Remarkably, you can implement top-down 2-3-4 trees by moving one line of code in `put()` in ALGORITHM 3.4: move the `colorFlip()` call (and accompanying test) to before the recursive calls (between the test for null and the comparison). This algorithm has some advantages over 2-3 trees in applications where multiple processes have access to the same tree, because it always is operating within a link or two of the current node. The deletion algorithms that we describe next are based on a similar scheme and are effective for these trees as well as for 2-3 trees.

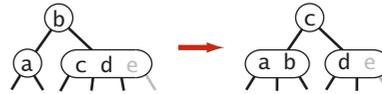
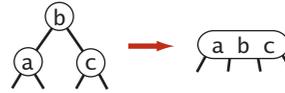
Delete the minimum. As a second warmup for deletion, we consider the operation of deleting the minimum from a 2-3 tree. The basic idea is based on the observation that we can easily delete a key from a 3-node at the bottom of the tree, but not from a 2-node. Deleting the key from a 2-node leaves a node with no keys; the natural thing to do would be to replace the node with a null link, but that operation would violate the perfect balance condition. So, we adopt the following approach: to ensure that we do not end up on a 2-node, we perform appropriate transformations on the way down the tree to preserve the invariant that the current node is not a 2-node (it might be a 3-node or a temporary 4-node). First, at the root, there are two possibilities: if the root is a 2-node and both children are 2-nodes, we can just convert the three nodes to a 4-node; otherwise we can

borrow from the right sibling if necessary to ensure that the left child of the root is not a 2-node. Then, on the way down the tree, one of the following cases must hold:

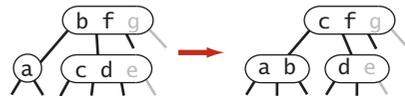
- If the left child of the current node is not a 2-node, there is nothing to do.
- If the left child is a 2-node and its immediate sibling is not a 2-node, move a key from the sibling to the left child.
- If the left child and its immediate sibling are 2-nodes, then combine them with the smallest key in the parent to make a 4-node, changing the parent from a 3-node to a 2-node or from a 4-node to a 3-node.

Following this process as we traverse left links to the bottom, we wind up on a 3-node or a 4-node with the smallest key, so we can just remove it, converting the 3-node to a

at the root



on the way down



at the bottom



Transformations for delete the minimum

2-node or the 4-node to a 3-node. Then, on the way up the tree, we split any unused temporary 4-nodes.

Delete. The same transformations along the search path just described for deleting the minimum are effective to ensure that the current node is not a 2-node during a search for any key. If the search key is at the bottom, we can just remove it. If the key is not at the bottom, then we have to exchange it with its successor as in regular BSTs. Then, since the current node is not a 2-node, we have reduced the problem to deleting the minimum in a subtree whose root is not a 2-node, and we can use the procedure just described for that subtree. After the deletion, as usual, we split any remaining 4-nodes on the search path on the way up the tree.

SEVERAL OF THE EXERCISES at the end of this section are devoted to examples and implementations related to these deletion algorithms. People with an interest in developing or understanding implementations need to master the details covered in these exercises. People with a general interest in the study of algorithms need to recognize that these methods are important because they represent the first symbol-table implementation that we have seen where *search*, *insert*, and *delete* are all guaranteed to be efficient, as we will establish next.

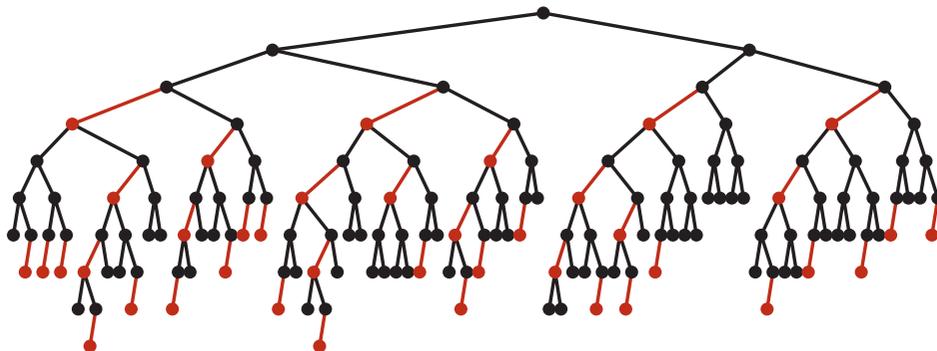
Properties of red-black BSTs Studying the properties of red-black BSTs is a matter of checking the correspondence with 2-3 trees and then applying the analysis of 2-3 trees. The end result is that *all symbol-table operations in red-black BSTs are guaranteed to be logarithmic in the size of the tree* (except for range search, which additionally costs time proportional to the number of keys returned). We repeat and emphasize this point because of its importance.

Analysis. First, we establish that red-black BSTs, while not perfectly balanced, are always nearly so, regardless of the order in which the keys are inserted. This fact immediately follows from the 1-1 correspondence with 2-3 trees and the defining property of 2-3 trees (perfect balance).

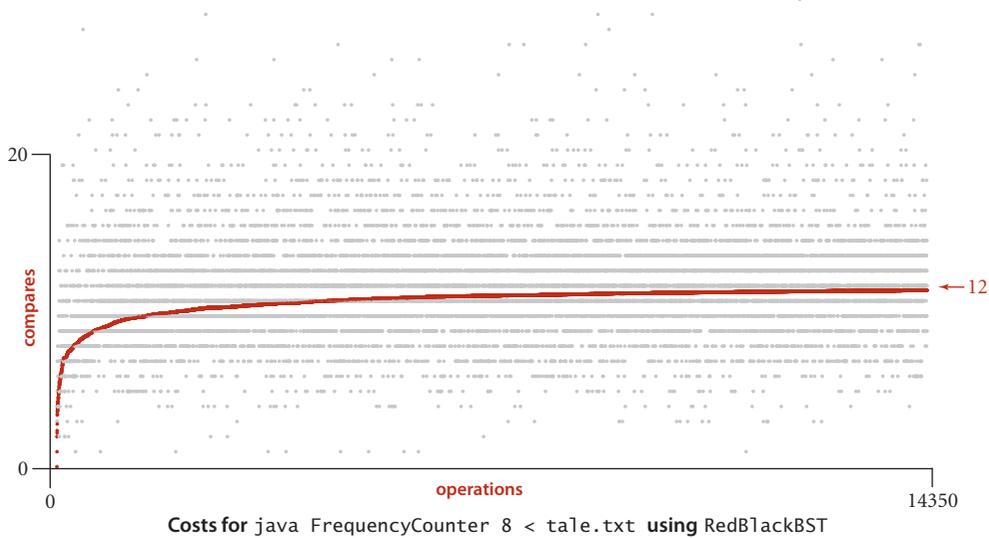
Proposition G. The height of a red-black BST with N nodes is no more than $2 \lg N$.

Proof sketch: The worst case is a 2-3 tree that is all 2-nodes except that the leftmost path is made up of 3-nodes. The path taking left links from the root is twice as long as the paths of length $\sim \lg N$ that involve just 2-nodes. It is possible, but not easy, to develop key sequences that cause the construction of red-black BSTs whose average path length is the worst-case $2 \lg N$. If you are mathematically inclined, you might enjoy exploring this issue by working EXERCISE 3.3.24.

This upper bound is conservative: experiments involving both random insertions and insertion sequences found in typical applications support the hypothesis that each search in a red-black BST of N nodes uses about $1.0 \lg N - 0.5$ compares, on the average. Moreover, you are not likely to encounter a substantially higher average number of compares in practice.



Typical red-black BST built from random keys (null links omitted)



The `get()` method in red-black BSTs does not examine the node color, so the balancing mechanism adds no overhead; search is faster than in elementary BSTs because the tree is balanced. Each key is inserted just once, but may be involved in many, many search operations, so the end result is that we get search times that are close to optimal (because the trees are nearly balanced and no work for balancing is done during the searches) at relatively little cost (unlike binary search, insertions are guaranteed to be logarithmic). The inner loop of the search is a compare followed by updating a link, which is quite short, like the inner loop of binary search (compare and index arithmetic). This implementation is the first we have seen with logarithmic guarantees for both search and insert, and it has a tight inner loop, so its use is justified in a broad variety of applications, including library implementations.

Ordered symbol-table API. One of the most appealing features of red-black BSTs is that the complicated code is limited to the `put()` (and deletion) methods. Our code for the minimum/maximum, select, rank, floor, ceiling and range queries in standard BSTs can be used *without any change*, since it operates on BSTs and has no need to refer to the node color. ALGORITHM 3.4, together with these methods (and the deletion methods), leads to a complete implementation of our ordered symbol-table API. Moreover, all of the methods benefit from the near-perfect balance in the tree because they all require time proportional to the tree height, at most. Thus PROPOSITION G, in combination with PROPOSITION E, suffices to establish a logarithmic performance guarantee for all of them.

Proposition I. In a red-black BST, the following operations take logarithmic time in the worst case: search, insertion, finding the minimum, finding the maximum, floor, ceiling, rank, select, delete the minimum, delete the maximum, delete, and range count.

Proof: We have just discussed `get()`, `put()`, and the deletion operations. For the others, the code from SECTION 3.2 can be used verbatim (it just ignores the node color). Guaranteed logarithmic performance follows from PROPOSITIONS E and G, and the fact that each algorithm performs a constant number of operations on each node examined.

On reflection, it is quite remarkable that we are able to achieve such guarantees. In a world awash with information, where people maintain tables with trillions or quadrillions of entries, the fact is that we can guarantee to complete any one of these operations in such tables with just a few dozen compares.

algorithm (data structure)	worst-case cost (after N inserts)		average-case cost (after N random inserts)		efficiently support ordered operations?
	search	insert	search hit	insert	
<i>sequential search</i> (unordered linked list)	N	N	$N/2$	N	no
<i>binary search</i> (ordered array)	$\lg N$	N	$\lg N$	$N/2$	yes
<i>binary tree search</i> (BST)	N	N	$1.39 \lg N$	$1.39 \lg N$	yes
<i>2-3 tree search</i> (red-black BST)	$2 \lg N$	$2 \lg N$	$1.00 \lg N$	$1.00 \lg N$	yes

Cost summary for symbol-table implementations (updated)

Q&A

Q. Why not let the 3-nodes lean either way and also allow 4-nodes in the trees?

A. Those are fine alternatives, used by many for decades. You can learn about several of these alternatives in the exercises. The left-leaning convention reduces the number of cases and therefore requires substantially less code.

Q. Why not use an array of Key values to represent 2-, 3-, and 4-nodes with a single Node type?

A. Good question. That is precisely what we do for B-trees (see CHAPTER 6), where we allow many more keys per node. For the small nodes in 2-3 trees, the overhead for the array is too high a price to pay.

Q. When we split a 4-node, we sometimes set the color of the right node to RED in `rotateRight()` and then immediately set it to BLACK in `flipColors()`. Isn't that wasteful?

A. Yes, and we also sometimes unnecessarily recolor the middle node. In the grand scheme of things, resetting a few extra bits is not in the same league with the improvement from linear to logarithmic that we get for all operations, but in performance-critical applications, you can put the code for `rotateRight()` and `flipColors()` inline and eliminate those extra tests. We use those methods for deletion, as well, and find them slightly easier to use, understand, and maintain by making sure that they preserve perfect black balance.

EXERCISES

3.3.1 Draw the 2-3 tree that results when you insert the keys E A S Y Q U T I O N in that order into an initially empty tree.

3.3.2 Draw the 2-3 tree that results when you insert the keys Y L P M X H C R A E S in that order into an initially empty tree.

3.3.3 Find an insertion order for the keys S E A R C H X M that leads to a 2-3 tree of height 1.

3.3.4 Prove that the height of a 2-3 tree with N keys is between $\sim \log_3 N \approx .63 \lg N$ (for a tree that is all 3-nodes) and $\sim \lg N$ (for a tree that is all 2-nodes).

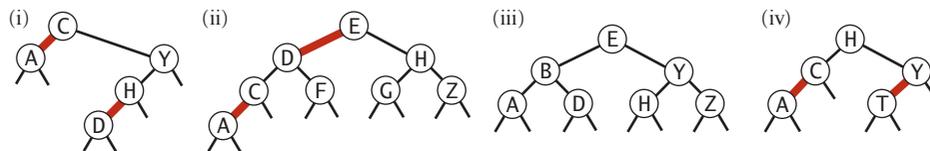
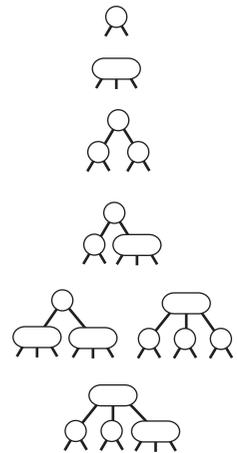
3.3.5 The figure at right shows all the structurally different 2-3 trees with N keys, for N from 1 up to 6 (ignore the order of the subtrees). Draw all the structurally different trees for $N = 7, 8, 9,$ and 10 .

3.3.6 Find the probability that each of the 2-3 trees in EXERCISE 3.3.5 is the result of the insertion of N random distinct keys into an initially empty tree.

3.3.7 Draw diagrams like the one at the top of page 428 for the other five cases in the diagram at the bottom of that page.

3.3.8 Show all possible ways that one might represent a 4-node with three 2-nodes bound together with red links (not necessarily left-leaning).

3.3.9 Which of the following are red-black BSTs?



3.3.10 Draw the red-black BST that results when you insert items with the keys E A S Y Q U T I O N in that order into an initially empty tree.

3.3.11 Draw the red-black BST that results when you insert items with the keys Y L P M X H C R A E S in that order into an initially empty tree.

EXERCISES (continued)

3.3.12 Draw the red-black BST that results after each transformation (color flip or rotation) during the insertion of P for our standard indexing client.

3.3.13 True or false: If you insert keys in increasing order into a red-black BST, the tree height is monotonically increasing.

3.3.14 Draw the red-black BST that results when you insert letters A through K in order into an initially empty tree, then describe what happens in general when trees are built by insertion of keys in ascending order (see also the figure in the text).

3.3.15 Answer the previous two questions for the case when the keys are inserted in descending order.

3.3.16 Show the result of inserting n into the red-black BST drawn at right (only the search path is shown, and you need to include only these nodes in your answer).

3.3.17 Generate two random 16-node red-black BSTs. Draw them (either by hand or with a program). Compare them with the (unbalanced) BSTs built with the same keys.

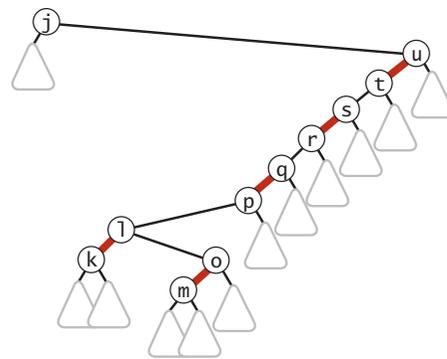
3.3.18 Draw all the structurally different red-black BSTs with N keys, for N from 2 up to 10 (see EXERCISE 3.3.5).

3.3.19 With 1 bit per node for color, we can represent 2-, 3-, and 4-nodes. How many bits per node would we need to represent 5-, 6-, 7-, and 8-nodes with a binary tree?

3.3.20 Compute the internal path length in a perfectly balanced BST of N nodes, when N is a power of 2 minus 1.

3.3.21 Create a test client for RedBlackBST, based on your solution to EXERCISE 3.2.10.

3.3.22 Find a sequence of keys to insert into a BST and into a red-black BST such that the height of the BST is less than the height of the red-black BST, or prove that no such sequence is possible.



CREATIVE PROBLEMS

3.3.23 *2-3 trees without balance restriction.* Develop an implementation of the basic symbol-table API that uses 2-3 trees that are not necessarily balanced as the underlying data structure. Allow 3-nodes to lean either way. Hook the new node onto the bottom with a black link when inserting into a 3-node at the bottom. Run experiments to develop a hypothesis estimating the average path length in a tree built from N random insertions.

3.3.24 *Worst case for red-black BSTs.* Show how to construct a red-black BST demonstrating that, in the worst case, almost all the paths from the root to a null link in a red-black BST of N nodes are of length $2 \lg N$.

3.3.25 *Top-down 2-3-4 trees.* Develop an implementation of the basic symbol-table API that uses balanced 2-3-4 trees as the underlying data structure, using the red-black representation and the insertion method described in the text, where 4-nodes are split by flipping colors on the way down the search path and balancing on the way up.

3.3.26 *Single top-down pass.* Develop a modified version of your solution to EXERCISE 3.3.25 that does not use recursion. Complete all the work splitting and balancing 4-nodes (and balancing 3-nodes) on the way down the tree, finishing with an insertion at the bottom.

3.3.27 *Allow right-leaning red links.* Develop a modified version of your solution to EXERCISE 3.3.25 that allows right-leaning red links in the tree.

3.3.28 *Bottom-up 2-3-4 trees.* Develop an implementation of the basic symbol-table API that uses balanced 2-3-4 trees as the underlying data structure, using the red-black representation and a bottom-up insertion method based on the same recursive approach as ALGORITHM 3.4. Your insertion method should split only the sequence of 4-nodes (if any) on the bottom of the search path.

3.3.29 *Optimal storage.* Modify `RedBlackBST` so that it does not use any extra storage for the color bit, based on the following trick: To color a node red, swap its two links. Then, to test whether a node is red, test whether its left child is larger than its right child. You have to modify the compares to accommodate the possible link swap, and this trick replaces bit compares with key compares that are presumably more expensive, but it shows that the bit in the nodes can be eliminated, if necessary.

3.3.30 *Software caching.* Modify `RedBlackBST` to keep the most recently accessed Node in an instance variable so that it can be accessed in constant time if the next `put()` or

CREATIVE PROBLEMS *(continued)*

`get()` uses the same key (see EXERCISE 3.1.25).

3.3.31 *Tree drawing.* Add a method `draw()` to `RedBlackBST` that draws red-black BST figures in the style of the text (see EXERCISE 3.2.38)

3.3.32 *AVL trees.* An AVL tree is a BST where the height of every node and that of its sibling differ by at most 1. (The oldest balanced tree algorithms are based on using rotations to maintain height balance in AVL trees.) Show that coloring red links that go from nodes of even height to nodes of odd height in an AVL tree gives a (perfectly balanced) 2-3-4 tree, where red links are not necessarily left-leaning. *Extra credit:* Develop an implementation of the symbol-table API that uses this as the underlying data structure. One approach is to keep a height field in each node, using rotations after the recursive calls to adjust the height as necessary; another is to use the red-black representation and use methods like `moveRedLeft()` and `moveRedRight()` in EXERCISE 3.3.39 and EXERCISE 3.3.40.

3.3.33 *Certification.* Add to `RedBlackBST` a method `is23()` to check that no node is connected to two red links and that there are no right-leaning red links and a method `isBalanced()` to check that all paths from the root to a null link have the same number of black links. Combine these methods with code from `isBST()` in EXERCISE 3.2.31 to create a method `isRedBlackBST()` that checks that the tree is a red-black BST.

3.3.34 *All 2-3 trees.* Write code to generate all structurally different 2-3 trees of height 2, 3, and 4. There are 2, 7, and 122 such trees, respectively. (*Hint:* Use a symbol table.)

3.3.35 *2-3 trees.* Write a program `TwoThreeST.java` that uses two node types to implement 2-3 search trees directly.

3.3.36 *2-3-4-5-6-7-8 trees.* Describe algorithms for search and insertion in balanced 2-3-4-5-6-7-8 search trees.

3.3.37 *Memoryless.* Show that red-black BSTs are not memoryless: for example, if you insert a key that is smaller than all the keys in the tree and then immediately delete the minimum, you may get a different tree.

3.3.38 *Fundamental theorem of rotations.* Show that any BST can be transformed into any other BST on the same set of keys by a sequence of left and right rotations.

3.3.39 *Delete the minimum.* Implement the `deleteMin()` operation for red-black BSTs by maintaining the correspondence with the transformations given in the text for moving down the left spine of the tree while maintaining the invariant that the current node is not a 2-node.

Solution:

```
private Node moveRedLeft(Node h)
{ // Assuming that h is red and both h.left and h.left.left
  // are black, make h.left or one of its children red.
  flipColors(h);
  if (isRed(h.right.left))
  {
    h.right = rotateRight(h.right);
    h = rotateLeft(h);
  }
  return h;
}

public void deleteMin()
{
  if (!isRed(root.left) && !isRed(root.right))
    root.color = RED;
  root = deleteMin(root);
  if (!isEmpty()) root.color = BLACK;
}

private Node deleteMin(Node h)
{
  if (h.left == null)
    return null;
  if (!isRed(h.left) && !isRed(h.left.left))
    h = moveRedLeft(h);
  h.left = deleteMin(h.left);
  return balance(h);
}
```

This code assumes a `balance()` method that consists of the line of code

```
if (isRed(h.right)) h = rotateLeft(h);
```

CREATIVE PROBLEMS *(continued)*

followed by the last five lines of the recursive `put()` in ALGORITHM 3.4 and a `flipColors()` implementation that complements the three colors, instead of the method given in the text for insertion. For deletion, we set the parent to BLACK and the two children to RED.

3.3.40 *Delete the maximum.* Implement the `deleteMax()` operation for red-black BSTs. Note that the transformations involved differ slightly from those in the previous exercise because red links are left-leaning.

Solution:

```
private Node moveRedRight(Node h)
{ // Assuming that h is red and both h.right and h.right.left
  // are black, make h.right or one of its children red.
  flipColors(h)
  if (isRed(h.left.left))
    h = rotateRight(h);
  return h;
}

public void deleteMax()
{
  if (!isRed(root.left) && !isRed(root.right))
    root.color = RED;
  root = deleteMax(root);
  if (!isEmpty()) root.color = BLACK;
}

private Node deleteMax(Node h)
{
  if (isRed(h.left))
    h = rotateRight(h);
  if (h.right == null)
    return null;
  if (!isRed(h.right) && !isRed(h.right.left))
    h = moveRedRight(h);
  h.right = deleteMax(h.right);
  return balance(h);
}
```

3.3.41 Delete. Implement the `delete()` operation for red-black BSTs, combining the methods of the previous two exercises with the `delete()` operation for BSTs.

Solution:

```
public void delete(Key key)
{
    if (!isRed(root.left) && !isRed(root.right))
        root.color = RED;
    root = delete(root, key);
    if (!isEmpty()) root.color = BLACK;
}

private Node delete(Node h, Key key)
{
    if (key.compareTo(h.key) < 0)
    {
        if (!isRed(h.left) && !isRed(h.left.left))
            h = moveRedLeft(h);
        h.left = delete(h.left, key);
    }
    else
    {
        if (isRed(h.left))
            h = rotateRight(h);
        if (key.compareTo(h.key) == 0 && (h.right == null))
            return null;
        if (!isRed(h.right) && !isRed(h.right.left))
            h = moveRedRight(h);
        if (key.compareTo(h.key) == 0)
        {
            Node x = min(h.right);
            h.key = x.key;
            h.val = x.val;
            h.right = deleteMin(h.right);
        }
        else h.right = delete(h.right, key);
    }
    return balance(h);
}
```

EXPERIMENTS

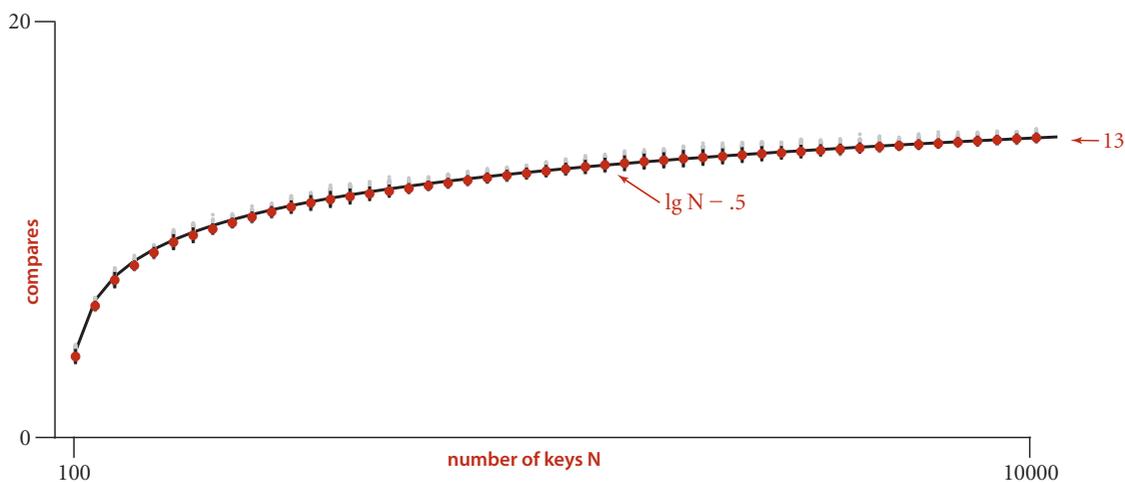
3.3.42 *Count red nodes.* Write a program that computes the percentage of red nodes in a given red-black BST. Test your program by running at least 100 trials of the experiment of inserting N random keys into an initially empty tree, for $N = 10^4$, 10^5 , and 10^6 , and formulate an hypothesis.

3.3.43 *Cost plots.* Instrument `RedBlackBST` so that you can produce plots like the ones in this section showing the cost of each `put()` operation during the computation (see EXERCISE 3.1.38).

3.3.44 *Average search time.* Run empirical studies to compute the average and standard deviation of the average length of a path to a random node (internal path length divided by tree size, plus 1) in a red-black BST built by insertion of N random keys into an initially empty tree, for N from 1 to 10,000. Do at least 1,000 trials for each tree size. Plot the results in a Tufte plot, like the one at the bottom of this page, fit with a curve plotting the function $\lg N - .5$.

3.3.45 *Count rotations.* Instrument your program for EXERCISE 3.3.43 to plot the number of rotations and node splits that are used to build the trees. Discuss the results.

3.3.46 *Height.* Instrument your program for EXERCISE 3.3.43 to plot the height of red-black BSTs. Discuss the results.



Average path length to a random node in a red-black BST built from random keys

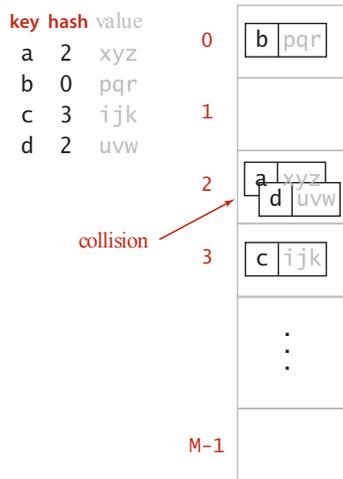
This page intentionally left blank

3.4 HASH TABLES

If keys are small integers, we can use an array to implement an unordered symbol table, by interpreting the key as an array index so that we can store the value associated with key i in array entry i , ready for immediate access. In this section, we consider *hashing*, an extension of this simple method that handles more complicated types of keys. We reference key-value pairs using arrays by doing arithmetic operations to transform keys into array indices.

Search algorithms that use hashing consist of two separate parts. The first part is to compute a *hash function* that transforms the search key into an array index. Ideally,

different keys would map to different indices. This ideal is generally beyond our reach, so we have to face the possibility that two or more different keys may hash to the same array index. Thus, the second part of a hashing search is a collision-resolution process that deals with this situation. After describing ways to compute hash functions, we shall consider two different approaches to collision resolution: *separate chaining* and *linear probing*.



Hashing: the crux of the problem

Hashing is a classic example of a time-space tradeoff. If there were no memory limitation, then we could do any search with only one memory access by simply using the key as an index in a (potentially huge) array. This ideal often cannot be achieved, however, because the amount of memory required is prohibitive when the number of possible key values is huge. On the other hand, if there were no time limitation, then we can get by with only a minimum amount of memory by using sequential search in an unordered array. Hashing provides a way to use a reasonable amount of both memory and time to strike a balance

between these two extremes. Indeed, it turns out that we can trade off time and memory in hashing algorithms by adjusting parameters, not by rewriting code. To help choose values of the parameters, we use classical results from probability theory.

Probability theory is a triumph of mathematical analysis that is beyond the scope of this book, but the hashing algorithms we consider that take advantage of the knowledge gained from that theory are quite simple, and widely used. With hashing, you can implement search and insert for symbol tables that require constant (amortized) time per operation in typical applications, making it the method of choice for implementing basic symbol tables in many situations.

Hash functions The first problem that we face is the computation of the hash function, which transforms keys into array indices. If we have an array that can hold M key-value pairs, then we need a *hash function* that can transform any given key into an index into that array: an integer in the range $[0, M-1]$. We seek a hash function that both is easy to compute and uniformly distributes the keys: for each key, every integer between 0 and $M-1$ should be equally likely (independently for every key). This ideal is somewhat mysterious; to understand hashing, we begin by thinking carefully about how to implement such a function.

In principle, any key can be represented as a sequence of bits, so we might design a generic hash function that maps sequences of bits to integers in the desired range. In practice, programmers implement hash functions based on higher-level representations. For example, if the key involves a number, such as a social security number, we could start with that number; if the key involves a string, such as a person's name, we need to convert the string into a number; and if the key has multiple parts, such as a mailing address, we need to combine the parts somehow. For many common types of keys, we can make use of default implementations provided by Java. We briefly discuss potential implementations for various types of keys so that you can see what is involved because you do need to provide implementations for key types that you create.

Typical example. Suppose that we have an application where the keys are U.S. social security numbers. A social security number such as 123-45-6789 is a nine-digit number divided into three fields. The first field identifies the geographical area where the number was issued (for example, social security numbers whose first field is 035 are from Rhode Island and numbers whose first field is 214 are from Maryland) and the other two fields identify the individual. There are a billion (10^9) different social security numbers, but suppose that our application will need to process just a few hundred keys, so that we could use a hash table of size $M = 1,000$. One possible approach to implementing a hash function is to use three digits from the key. Using three digits from the third field is likely to be preferable to using the three digits in the first field (since customers may not be uniformly dispersed over geographic areas), but a better approach is to use all nine digits to make an `int` value, then consider hash functions for integers, described next.

Positive integers. The most commonly used method for hashing integers is called *modular hashing*: we choose the array size M to be prime and, for any positive integer key k , compute the remainder when dividing k by M . This function is very easy to compute (`k % M`, in Java) and is effective in dispersing the keys evenly between 0 and

key	hash ($M = 100$)	hash ($M = 97$)
212	12	18
618	18	36
302	2	11
940	40	67
702	2	23
704	4	25
612	12	30
606	6	24
772	72	93
510	10	25
423	23	35
650	50	68
317	17	26
907	7	34
507	7	22
304	4	13
714	14	35
857	57	81
801	1	25
900	0	27
413	13	25
701	1	22
418	18	30
601	1	19

Modular hashing

$M - 1$. If M is not prime, it may be the case that not all of the bits of the key play a role, which amounts to missing an opportunity to disperse the values evenly. For example, if the keys are base-10 numbers and M is 10^k , then only the k least significant digits are used. As a simple example where such a choice might be problematic, suppose that the keys are telephone area codes and $M = 100$. For historical reasons, most area codes in the United States have middle digit 0 or 1, so this choice strongly favors the values less than 20, where the use of the prime value 97 better disperses them (a prime value not close to 100 would do even better). Similarly, IP addresses that are used in the internet are binary numbers that are not random for similar historical reasons as for telephone area codes, so we need to use a table size that is a prime (in particular, not a power of 2) if we want to use modular hashing to disperse them.

Floating-point numbers. If the keys are real numbers between 0 and 1, we might just multiply by M and round off to the nearest integer to get an index between 0 and $M - 1$. Although this approach is intuitive, it is defective because it gives more weight to the most significant bits of the keys; the least significant bits play no role. One way to address this situation is to use modular hashing on the binary representation of the key (this is what Java does).

Strings. Modular hashing works for long keys such as strings, too: we simply treat them as huge integers. For example, the code at left computes a modular hash function for a `String` `s`: recall that `charAt()` returns a `char` value in Java, which is a 16-bit nonnegative integer. If R is greater than any character value, this computation would

```
int hash = 0;
for (int i = 0; i < s.length(); i++)
    hash = (R * hash + s.charAt(i)) % M;
```

Hashing a string key

be equivalent to treating the `String` as an N -digit base- R integer, computing the remainder that results when dividing that number by M . A classic algorithm known as *Horner's method* gets the job done with N multiplications, additions, and remainder operations. If the value of R is sufficiently

small that no overflow occurs, the result is an integer between 0 and $M - 1$, as desired. The use of a small prime integer such as 31 ensures that the bits of all the characters play a role. Java's default implementation for `String` uses a method like this.

Compound keys. If the key type has multiple integer fields, we can typically mix them together in the way just described for `String` values. For example, suppose that search keys are of type `Date`, which has three integer fields: day (two-digit day), month (two-digit month), and year (four-digit year). We compute the number

```
int hash = (((day * R + month) % M) * R + year) % M;
```

which, if the value of R is sufficiently small that no overflow occurs, is an integer between 0 and $M-1$, as desired. In this case, we could save the cost of the inner $\% M$ operation by choosing a moderate prime value such as 31 for R . As with strings, this method generalizes to handle any number of fields.

Java conventions. Java helps us address the basic problem that every type of data needs a hash function by ensuring that every data type inherits a method called `hashCode()` that returns a 32-bit integer. The implementation of `hashCode()` for a data type must be *consistent with equals*. That is, if `a.equals(b)` is true, then `a.hashCode()` must have the same numerical value as `b.hashCode()`. Conversely, if the `hashCode()` values are different, then we know that the objects are not equal. If the `hashCode()` values are the same, the objects may or may not be equal, and we must use `equals()` to decide which condition holds. This convention is a basic requirement for clients to be able to use `hashCode()` for symbol tables. Note that it implies that you must override both `hashCode()` and `equals()` if you need to hash with a user-defined type. The default implementation returns the machine address of the key object, which is seldom what you want. Java provides `hashCode()` implementations that override the defaults for many common types (including `String`, `Integer`, `Double`, `File`, and `URL`).

Converting a `hashCode()` to an array index. Since our goal is an array index, not a 32-bit integer, we combine `hashCode()` with modular hashing in our implementations to produce integers between 0 and $M-1$, as follows:

```
private int hash(Key x)
{ return (x.hashCode() & 0x7fffffff) % M; }
```

This code masks off the sign bit (to turn the 32-bit number into a 31-bit nonnegative integer) and then computes the remainder when dividing by M , as in modular hashing. Programmers commonly use a prime number for the hash table size M when using code like this, to attempt to make use of all the bits of the hash code. Note: To avoid confusion, we omit all of these calculations in our hashing examples and use instead the hash values in the table at right.

key	S	E	A	R	C	H	X	M	P	L
hash ($M = 5$)	2	0	0	4	4	4	2	4	3	3
hash ($M = 16$)	6	10	4	14	5	4	15	1	14	6

Hash values for keys in examples

User-defined `hashCode()`. Client code expects that `hashCode()` disperses the keys uniformly among the possible 32-bit result values. That is, for any object x , you can write `x.hashCode()` and, in principle, expect to get any one of the 2^{32} possible 32-bit values with equal likelihood. Java's `hashCode()` implementations for `String`, `Integer`, `Double`, `File`, and `URL` aspire to this functionality; for your own type, you have to try to do it on your own. The `Date` example that we considered on page 460 illustrates

```
public class Transaction
{
    ...
    private final String who;
    private final Date when;
    private final double amount;

    public int hashCode()
    {
        int hash = 17;
        hash = 31 * hash + who.hashCode();
        hash = 31 * hash + when.hashCode();
        hash = 31 * hash
            + ((Double) amount).hashCode();
        return hash;
    }
    ...
}
```

Implementing hashCode() in a user-defined type

one way to proceed: make integers from the instance variables and use modular hashing. In Java, the convention that all data types inherit a hashCode() method enables an even simpler approach: use the hashCode() method for the instance variables to convert each to a 32-bit int value and then do the arithmetic, as illustrated at left for Transaction. For primitive-type instance variables, note that a cast to a wrapper type is necessary to access the hashCode() method. Again, the precise values of the multiplier (31 in our example) is not particularly important.

Software caching. If computing the hash code is expensive, it may be worthwhile to

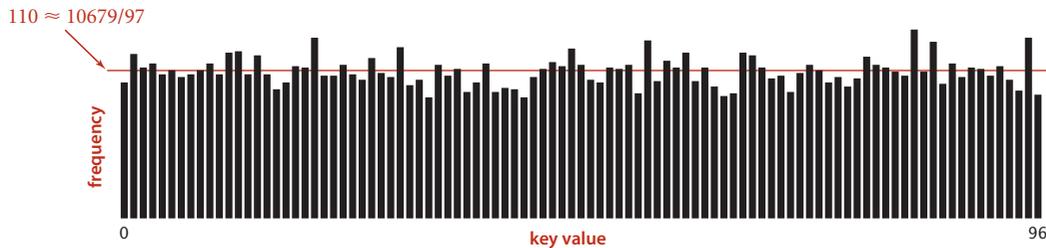
cache the hash for each key. That is, we maintain an instance variable hash in the key type that contains the value of hashCode() for each key object (see EXERCISE 3.4.25). On the first call to hashCode(), we have to compute the full hash code (and set the value of hash), but subsequent calls on hashCode() simply return the value of hash. Java uses this technique to reduce the cost of computing hashCode() for String objects.

IN SUMMARY, WE HAVE THREE PRIMARY REQUIREMENTS in implementing a good hash function for a given data type:

- It should be consistent—equal keys must produce the same hash value.
- It should be *efficient to compute*.
- It should *uniformly distribute the set of keys*.

Satisfying these requirements simultaneously in Java is a job for experts. As with many built-in capabilities, Java programmers who use hashing assume that hashCode() does the job, absent any evidence to the contrary.

Still, you should be vigilant whenever using hashing in situations where good performance is critical, because a bad hash function is a classic example of a performance bug: everything will work properly, but much more slowly than expected. Perhaps the easiest way to ensure uniformity is to make sure that all the bits of the key play an equal role in computing every hash value; perhaps the most common mistake in implementing hash functions is to ignore significant numbers of the key bits. Whatever the implementation, it is wise to test any hash function that you use, when performance is important. Which takes more time: computing a hash function or comparing two keys? Does your



Hash value frequencies for words in *Tale of Two Cities* (10,679 keys, $M = 97$)

hash function spread a typical set of keys uniformly among the values between 0 and $M-1$? Doing simple experiments that answer these questions can protect future clients from unfortunate surprises. For example, the histogram above shows that our `hash()` implementation using the `hashCode()` from Java's `String` data type produces a reasonable dispersion of the words for our *Tale of Two Cities* example.

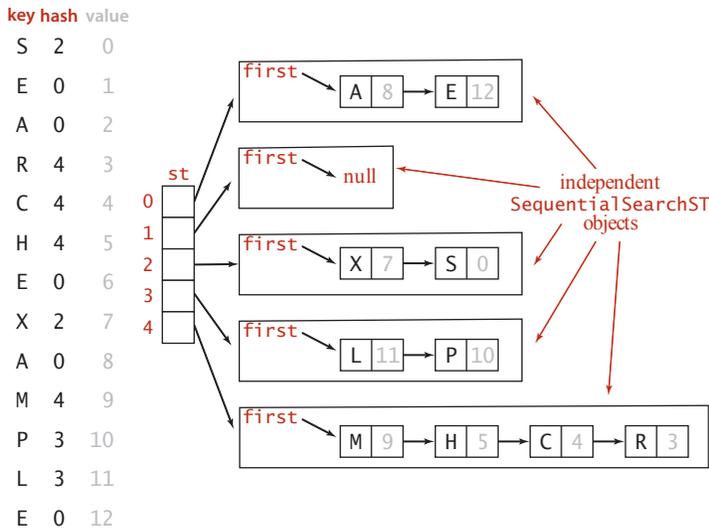
Underlying this discussion is a fundamental assumption that we make when using hashing; it is an idealized model that we do not actually expect to achieve, but it guides our thinking when implementing hashing algorithms and facilitates their analyses:

Assumption J (*uniform hashing assumption*). The hash functions that we use uniformly and independently distribute keys among the integer values between 0 and $M-1$.

Discussion: With all of the arbitrary choices we have made, the Java hash functions that we have considered do not satisfy these conditions; nor can any deterministic hash function. The idea of constructing hash functions that uniformly and independently distribute keys leads to deep issues in theoretical computer science. In 1977, L. Carter and M. Wegman described how to construct a universal family of hash functions. If a hash function is chosen at random from a universal family, the hash function uniformly distributes the keys, but only with partial independence. Although weaker than full independence, the partial independence is sufficient to establish performance guarantees similar to those stated in PROPOSITIONS K and M.

ASSUMPTION J IS A USEFUL WAY TO THINK about hashing for two primary reasons. First, it is a worthy goal when designing hash functions that guides us away from making arbitrary decisions that might lead to an excessive number of collisions. Second, we will use it to develop hypotheses about the performance of hashing algorithms—even when hash functions are not known to satisfy ASSUMPTION J, we can perform computational experiments and validate that they achieve the predicted performance.

Hashing with separate chaining A hash function converts keys into array indices. The second component of a hashing algorithm is collision resolution: a strategy for handling the case when two or more keys to be inserted hash to the same index. A straightforward and general approach to collision resolution is to build, for each of the M array indices, a linked list of the key-value pairs whose keys hash to that index. This method is known as *separate chaining* because items that collide are chained together in separate linked lists. The basic idea is to choose M to be sufficiently large that the lists are sufficiently short to enable efficient search through a two-step process: hash to find the list that could contain the key, then sequentially search through that list for the key.



Hashing with separate chaining for standard indexing client

One way to proceed is to expand `SequentialSearchST` (ALGORITHM 3.1) to implement separate chaining using linked-list primitives (see EXERCISE 3.4.2). A simpler (though slightly less efficient) way to proceed is to adopt a more general approach: we build, for each of the M array indices, a symbol table of the keys that hash to that index, thus reusing code that we have already developed. The implementation `SeparateChainingHashST` in ALGORITHM 3.5 maintains an array of `SequentialSearchST` objects and implements `get()` and `put()` by computing a hash function to choose which

`SequentialSearchST` object can contain the key and then using `get()` and `put()` (respectively) from `SequentialSearchST` to complete the job.

Since we have M lists and N keys, the average length of the lists is always N/M , no matter how the keys are distributed among the lists. For example, suppose that all the items fall onto the first list—the average length of the lists is $(N + 0 + 0 + 0 + \dots + 0)/M = N/M$. However the keys are distributed on the lists, the sum of the list lengths is N and the average is N/M . Separate chaining is useful in practice because *each* list is extremely likely to have about N/M key-value pairs. In typical situations, we can verify this consequence of ASSUMPTION J and count on fast search and insert.

ALGORITHM 3.5 Hashing with separate chaining

```

public class SeparateChainingHashST<Key, Value>
{
    private int M; // hash table size
    private SequentialSearchST<Key, Value>[] st; // array of ST objects

    public SeparateChainingHashST()
    { this(997); }

    public SeparateChainingHashST(int M)
    { // Create M linked lists.
      this.M = M;
      st = (SequentialSearchST<Key, Value>[]) new SequentialSearchST[M];
      for (int i = 0; i < M; i++)
        st[i] = new SequentialSearchST();
    }

    private int hash(Key key)
    { return (key.hashCode() & 0x7fffffff) % M; }

    public Value get(Key key)
    { return (Value) st[hash(key)].get(key); }

    public void put(Key key, Value val)
    { st[hash(key)].put(key, val); }

    public Iterable<Key> keys()
    // See Exercise 3.4.19.
}

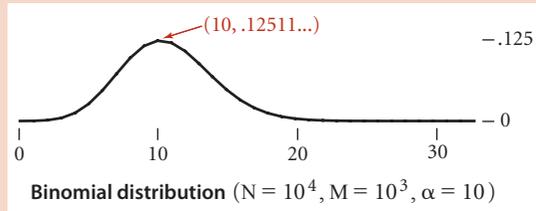
```

This basic symbol-table implementation maintains an array of linked lists, using a hash function to choose a list for each key. For simplicity, we use `SequentialSearchST` methods. We need a cast when creating `st[]` because Java prohibits arrays with generics. The default constructor specifies 997 lists, so that for large tables, this code is about a factor of 1,000 faster than `SequentialSearchST`. This quick solution is an easy way to get good performance when you have some idea of the number of key-value pairs to be `put()` by a client. A more robust solution is to use array resizing to make sure that the lists are short no matter how many key-value pairs are in the table (see page 474 and EXERCISE 3.4.18).

Proposition K. In a separate-chaining hash table with M lists and N keys, the probability (under ASSUMPTION J) that the number of keys in a list is within a small constant factor of N/M is extremely close to 1.

Proof sketch: ASSUMPTION J makes this an application of classical probability theory. We sketch the proof, for readers who are familiar with basic probabilistic analysis. The probability that a given list will contain exactly k keys is given by the binomial distribution

$$\binom{N}{k} \left(\frac{1}{M}\right)^k \left(\frac{M-1}{M}\right)^{N-k}$$

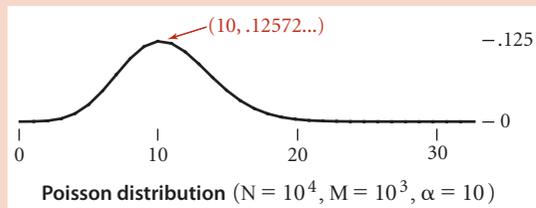


by the following argument: Choose k out of the N keys. Those k keys hash to the given list with probability $1/M$, and the other $N - k$ keys do not hash to the given list with probability $1 - (1/M)$. In terms of $\alpha = N/M$, we can rewrite this expression as

$$\binom{N}{k} \left(\frac{\alpha}{N}\right)^k \left(1 - \frac{\alpha}{N}\right)^{N-k}$$

which (for small α) is closely approximated by the classical Poisson distribution

$$\frac{\alpha^k e^{-\alpha}}{k!}$$



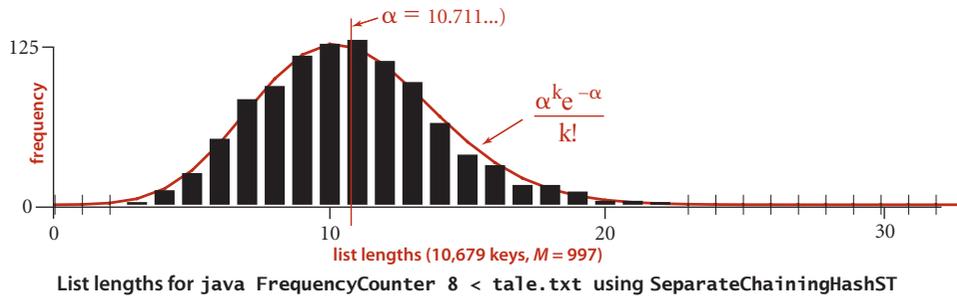
It follows that the probability that a list has more than t keys on it is bounded by the quantity $(\alpha e/t)^t e^{-\alpha}$. This probability is extremely small for practical ranges of the parameters. For example, if the average length of the lists is 10, the probability that we will hash to some list with more than 20 keys on it is less than $(10 e/2)^2 e^{-10} \approx 0.0084$, and if the average length of the lists is 20, the probability that we will hash to some list with more than 40 keys on it is less than $(20 e/2)^2 e^{-20} \approx 0.0000016$. This concentration result does not guarantee that every list will be short. Indeed it is known that, if α is a constant, the average length of the longest list grows with $\log N / \log \log N$.

This classical mathematical analysis is compelling, but it is important to note that it completely depends on ASSUMPTION J. If the hash function is not uniform and independent, the search and insert cost could be proportional to N , no better than with sequential search. ASSUMPTION J is much stronger than the corresponding assumption for other probabilistic algorithms that we have seen, and much more difficult to verify. With hashing, we are assuming that each and every key, no matter how complex, is equally likely to be hashed to one of M indices. We cannot afford to run experiments to test every possible key, so we would have to do more sophisticated experiments involving random sampling from the set of possible keys used in an application, followed by statistical analysis. Better still, we can use the algorithm itself as part of the test, to validate both ASSUMPTION J and the mathematical results that we derive from it.

Property L. In a separate-chaining hash table with M lists and N keys, the number of compares (equality tests) for search miss and insert is $\sim N/M$.

Evidence: Countless programmers since the 1950s have seen the speedups for separate-chaining hash tables predicted by PROPOSITION K, even for hash functions that clearly do not satisfy ASSUMPTION J. For example, the diagram on page 468 shows that list length distribution for our FrequencyCounter example (using our hash() implementation based on the hashCode() from Java's String data type) precisely matches the theoretical model. One exception that has been documented on numerous occasions is poor performance due to hash functions not taking all of the bits of the keys into account. Otherwise, the preponderance of the evidence from the experience of practical programmers puts us on solid ground in stating that hashing with separate chaining using an array of size M speeds up search and insert in a symbol table by a factor of M .

Table size. In a separate-chaining implementation, our goal is to choose the table size M to be sufficiently small that we do not waste a huge area of contiguous memory with empty chains but sufficiently large that we do not waste time searching through long chains. One of the virtues of separate chaining is that this decision is not critical: if more keys arrive than expected, then searches will take a little longer than if we had chosen a bigger table size ahead of time; if fewer keys are in the table, then we have extra-fast search with some wasted space. When space is not a critical resource, M can be chosen sufficiently large that search time is constant; when space is a critical resource, we still can get a factor of M improvement in performance by choosing M to be as

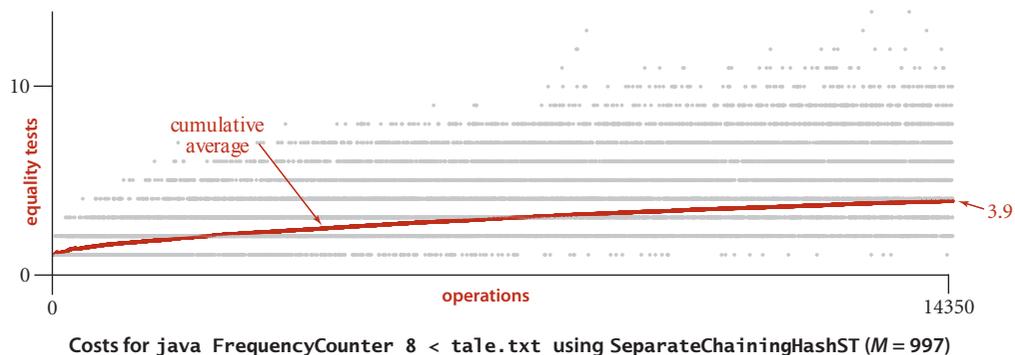


large as we can afford. For our example FrequencyCounter study, we see in the figure below a reduction in the average cost from thousands of compares per operation for SequentialSearchST to a small constant for SeparateChainingHashST, as expected. Another option is to use array resizing to keep the lists short (see EXERCISE 3.4.18).

Deletion. To delete a key-value pair, simply hash to find the SequentialSearchST containing the key, then invoke the delete() method for that table (see EXERCISE 3.1.5). Reusing code in this way is preferable to reimplementing this basic operation on linked lists.

Ordered operations. The whole point of hashing is to uniformly disperse the keys, so any order in the keys is lost when hashing. If you need to quickly find the maximum or minimum key, find keys in a given range, or implement any of the other operations in the ordered symbol-table API (page 366), then hashing is not appropriate, since these operations will all take linear time.

HASHING WITH SEPARATE CHAINING is easy to implement and probably the fastest (and most widely used) symbol-table implementation for applications where key order is not important. When your keys are built-in Java types or your own type with well-tested implementations of hashCode(), ALGORITHM 3.5 provides a quick and easy path to fast search and insert. Next, we consider an alternative scheme for collision resolution that is also effective.



Hashing with linear probing Another approach to implementing hashing is to store N key-value pairs in a hash table of size $M > N$, relying on empty entries in the table to help with collision resolution. Such methods are called open-addressing hashing methods.

The simplest open-addressing method is called linear probing: when there is a collision (when we hash to a table index that is already occupied with a key different from the search key), then we just check the next entry in the table (by incrementing the index). Linear probing is characterized by identifying three possible outcomes:

- Key equal to search key: search hit
- Empty position (null key at indexed position): search miss
- Key not equal to search key: try next entry

We hash the key to a table index, check whether the search key matches the key there, and continue (incrementing the index, wrapping back to the beginning of the table if we reach the end) until finding either the search key or an empty table entry. It is customary to refer to the operation of determining whether or not a given table entry

key	hash	value	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
S	6	0							S									
E	10	1							S				E					
A	4	2			A				S				E					
R	14	3			A				S				E				R	
C	5	4			A	C			S				E				R	
H	4	5			A	C	S	H					E				R	
E	10	6			A	C	S	H					E				R	
X	15	7			A	C	S	H					E				R	X
A	4	8			A	C	S	H					E				R	X
M	1	9	M		A	C	S	H					E				R	X
P	14	10	P	M		A	C	S	H				E				R	X
L	6	11	P	M		A	C	S	H	L			E				R	X
E	10	12	P	M		A	C	S	H	L			E				R	X

entries in red are new
 entries in gray are untouched
 keys in black are probes
 probe sequence wraps to 0
 keys []
 vals []

Trace of linear-probing ST implementation for standard indexing client

ALGORITHM 3.6 Hashing with linear probing

```

public class LinearProbingHashST<Key, Value>
{
    private int N;           // number of key-value pairs in the table
    private int M = 16;     // size of linear-probing table
    private Key[] keys;     // the keys
    private Value[] vals;   // the values

    public LinearProbingHashST()
    {
        keys = (Key[]) new Object[M];
        vals = (Value[]) new Object[M];
    }

    private int hash(Key key)
    { return (key.hashCode() & 0x7fffffff) % M; }

    private void resize() // See page 474.

    public void put(Key key, Value val)
    {
        if (N >= M/2) resize(2*M); // double M (see text)

        int i;
        for (i = hash(key); keys[i] != null; i = (i + 1) % M)
            if (keys[i].equals(key)) { vals[i] = val; return; }
        keys[i] = key;
        vals[i] = val;
        N++;
    }

    public Value get(Key key)
    {
        for (int i = hash(key); keys[i] != null; i = (i + 1) % M)
            if (keys[i].equals(key))
                return vals[i];
        return null;
    }
}

```

This symbol-table implementation keeps keys and values in parallel arrays (as in `BinarySearchST`) but uses empty spaces (marked by `null`) to terminate clusters of keys. If a new key hashes to an empty entry, it is stored there; if not, we scan sequentially to find an empty position. To search for a key, we scan sequentially starting at its hash index until finding `null` (search miss) or the key (search hit). Implementation of `keys()` is left as EXERCISE 3.4.19.

holds an item whose key is equal to the search key as a probe. We use the term interchangeably with the term compare that we have been using, even though some probes are tests for `null`.

The essential idea behind hashing with open addressing is this: rather than using memory space for references in linked lists, we use it for the empty entries in the hash table, which mark the ends of probe sequences. As you can see from `LinearProbingHashST` (ALGORITHM 3.6), applying this idea to implement the symbol-table API is quite straightforward. We implement the table with parallel arrays, one for the keys and one for the values, and use the hash function as an index to access the data as just discussed.

Deletion. How do we delete a key-value pair from a linear-probing table? If you think about the situation for a moment, you will see that setting the key's table position to `null` will not work, because that might prematurely terminate the search for a key that was inserted into the table later. As an example, suppose that we try to delete C in this way in our trace example, then search for H. The hash value for H is 4, but it sits at the end of the cluster, in position 7. If we set position 5 to `null`, then `get()` will not find H. As a consequence, we need to reinsert into the table all of the keys in the cluster to the right of the deleted key. This process is trickier than it might seem, so you are encouraged to trace through the code at right (see EXERCISE 3.4.17).

AS WITH SEPARATE CHAINING, the performance of hashing with open addressing depends on the ratio $\alpha = N/M$, but we interpret it differently. We refer to α as the load factor of a hash table. For separate chaining, α is the average number of keys per list and is often larger than 1; for linear probing, α is the percentage of table entries that are occupied; it cannot be greater than 1. In fact, we cannot let the load factor reach 1 (completely full table) in `LinearProbingHashST` because a search miss would go into an infinite loop in a full table. Indeed, for the sake of good performance, we use array resizing to guarantee that the load factor is between one-eighth and one-half. This strategy is validated by mathematical analysis, which we consider before we discuss implementation details.

```
public void delete(Key key)
{
    if (!contains(key)) return;
    int i = hash(key);
    while (!key.equals(keys[i]))
        i = (i + 1) % M;
    keys[i] = null;
    vals[i] = null;
    i = (i + 1) % M;
    while (keys[i] != null)
    {
        Key    keyToRedo = keys[i];
        Value  valToRedo = vals[i];
        keys[i] = null;
        vals[i] = null;
        N--;
        put(keyToRedo, valToRedo);
        i = (i + 1) % M;
    }
    N--;
    if (N > 0 && N == M/8)
        resize(M/2);
}
```

Deletion for linear probing

Clustering. The average cost of linear probing depends on the way in which the entries clump together into contiguous groups of occupied table entries, called clusters, when they are inserted. For example, when the key C is inserted in our example, the result is a cluster (A C S) of length 3, which means that four probes are needed to insert H because H hashes to the first position in the cluster. Short clusters are certainly a requirement for efficient performance. This requirement can be problematic as the table fills, because long clusters are common. Moreover, since all table positions are equally likely to be the hash value of the next key to be inserted (under the uniform hashing assumption), long clusters are more likely to increase in length than short ones, because a new key hashing to any entry in the cluster will cause the cluster to increase in length by 1 (and possibly much more, if there is just one table entry separating the cluster from the next one). Next, we turn to the challenge of quantifying the effect of clustering to predict performance in linear probing, and using that knowledge to set design parameters in our implementations.

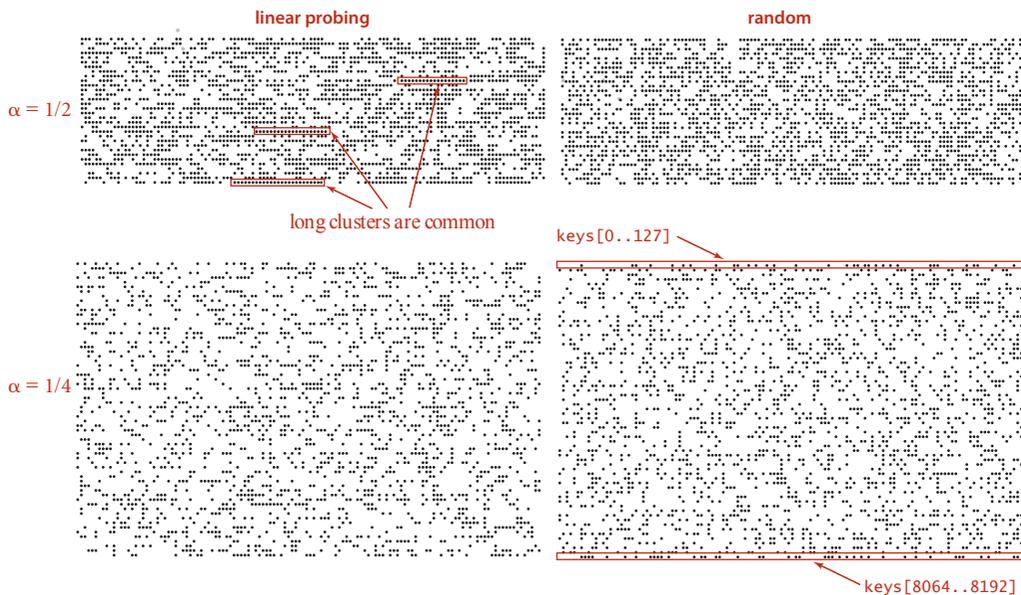
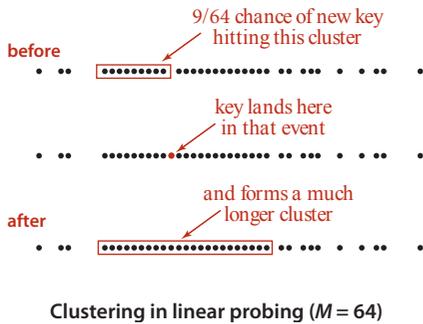


Table occupancy patterns (2,048 keys, tables laid out in 128-position rows)

Analysis of linear probing. Despite the relatively simple form of the results, precise analysis of linear probing is a very challenging task. Knuth's derivation of the following formulas in 1962 was a landmark in the analysis of algorithms:

Proposition M. In a linear-probing hash table of size M and $N = \alpha M$ keys, the average number of probes (under ASSUMPTION J) required is

$$\sim \frac{1}{2} \left(1 + \frac{1}{1 - \alpha} \right) \quad \text{and} \quad \sim \frac{1}{2} \left(1 + \frac{1}{(1 - \alpha)^2} \right)$$

for search hits and search misses (or inserts), respectively. In particular, when α is about $1/2$, the average number of probes for a search hit is about $3/2$ and for a search miss is about $5/2$. These estimates lose a bit of precision as α approaches 1, but we do not need them for that case, because we will only use linear probing for α less than one-half.

Discussion: We compute the average by computing the cost of a search miss starting at each position in the table, then dividing the total by M . All search misses take at least 1 probe, so we count the number of probes after the first. Consider the following two extremes in a linear-probing table that is half full ($M = 2N$): In the best case, table positions with even indices could be empty, and table positions with odd indices could be occupied. In the worst case, the first half of the table positions could be empty, and the second half occupied. The average length of the clusters in both cases is $N/(2N) = 1/2$, but the average number of probes for a search miss is 1 (all searches take at least 1 probe) plus $(0 + 1 + 0 + 1 + \dots)/(2N) = 1/2$ in the best case, and is 1 plus $(N + (N - 1) + \dots)/(2N) \sim N/4$ in the worst case. This argument generalizes to show that the average number of probes for a search miss is proportional to the *squares* of the lengths of the clusters: If a cluster is of length t , then the expression $(t + (t - 1) + \dots + 2 + 1) / M = t(t + 1)/(2M)$ counts the contribution of that cluster to the grand total. The sum of the cluster lengths is N , so, adding this cost for all entries in the table, we find that the total average cost for a search miss is $1 + N/(2M)$ plus the sum of the squares of the lengths of the clusters, divided by $2M$. Thus, given a table, we can quickly compute the average cost of a search miss in that table (see EXERCISE 3.4.21). In general, the clusters are formed by a complicated dynamic process (the linear-probing algorithm) that is difficult to characterize analytically, and quite beyond the scope of this book.

PROPOSITION M tells us (under our usual ASSUMPTION J) that we can expect a search to require a huge number of probes in a nearly full table (as α approaches 1 the values of the formulas describing the number of probes grow very large) but that the expected number of probes is between 1.5 and 2.5 if we can ensure that the load factor α is less than $1/2$. Next, we consider the use of array resizing for this purpose.

Array resizing We can use our standard array-resizing technique from CHAPTER 1 to ensure that the load factor never exceeds one-half. First, we need a new constructor for `LinearProbingHashST` that takes a fixed capacity as argument (add a line to

```
private void resize(int cap)
{
    LinearProbingHashST<Key, Value> t;
    t = new LinearProbingHashST<Key, Value>(cap);
    for (int i = 0; i < M; i++)
        if (keys[i] != null)
            t.put(keys[i], vals[i]);
    keys = t.keys;
    vals = t.vals;
    M = t.M;
}
```

Resizing a linear-probing hash table

the constructor in ALGORITHM 3.6 that sets M to the given value before creating the arrays). Next, we need the `resize()` method given at left, which creates a new `LinearProbingHashST` of the given size and puts all the key-value pairs from the old table into the new one by rehashing all the keys. These additions allow us to implement array doubling. The call to `resize()` in the first statement in `put()` ensures that the table is at

most one-half full. This code builds a hash table twice the size with the same keys, thus halving the value of α . As in other applications of array resizing, we also need to add

```
if (N > 0 && N <= M/8) resize(M/2);
```

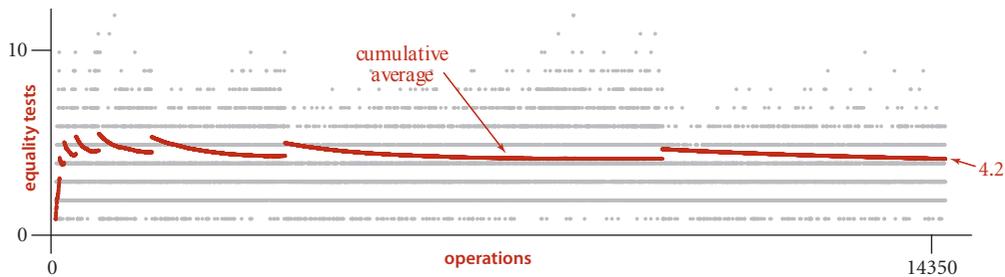
as the last statement in `delete()` to ensure that the table is at least one-eighth full. This ensures that the amount of memory used is always within a constant factor of the number of key-value pairs in the table. With array resizing, we are assured that $\alpha \leq 1/2$.

Separate chaining. The same method works to keep lists short (of average length between 2 and 8) in separate chaining: replace `LinearProbingHashST` by `SeparateChainingHashST` in `resize()`, call `resize(2*M)` when $(N \geq M/2)$ in `put()`, and call `resize(M/2)` when $(N > 0 \ \&\& \ N \leq M/8)$ in `delete()`. For separate chaining, array resizing is optional and not worth your trouble if you have a decent estimate of the client's N : just pick a table size M based on the knowledge that search times are proportional to $1 + N/M$. For linear probing, array resizing is necessary. A client that inserts more key-value pairs than you expect will encounter not just excessively long search times, but an infinite loop when the table fills.

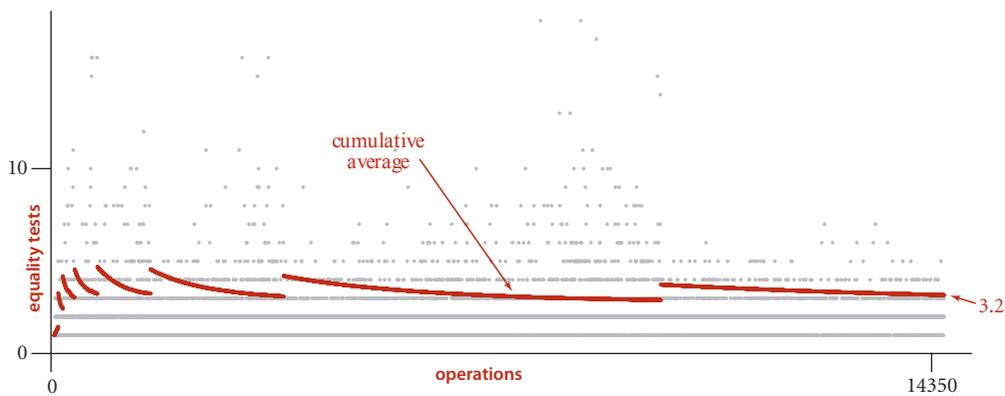
Amortized analysis. From a theoretical standpoint, when we use array resizing, we must settle for an amortized bound, since we know that those insertions that cause the table to double will require a large number of probes.

Proposition N. Suppose a hash table is built with array resizing, starting with an empty table. Under ASSUMPTION J, any sequence of t search, insert, and delete symbol-table operations is executed in expected time proportional to t and with memory usage always within a constant factor of the number of keys in the table.

Proof: For both separate chaining and linear probing, this fact follows from a simple restatement of the amortized analysis for array growth that we first discussed in CHAPTER 1, coupled with PROPOSITION K and PROPOSITION M.



Costs for java FrequencyCounter 8 < tale.txt using SeparateChainingHashST (with doubling)



Costs for java FrequencyCounter 8 < tale.txt using LinearProbingHashST (with doubling)

The plots of the cumulative averages for our `FrequencyCounter` example (shown at the bottom of the previous page) nicely illustrate the dynamic behavior of array resizing in hashing. Each time the array doubles, the cumulative average increases by about 1, because each key in the table needs to be rehashed; then it decreases because about half as many keys hash to each table position, with the rate of decrease slowing as the table fills again.

Memory As we have indicated, understanding memory usage is an important factor if we want to tune hashing algorithms for optimum performance. While such tuning is for experts, it is a worthwhile exercise to calculate a rough estimate of the amount of memory required, by estimating the number of references used, as follows: Not counting the memory for keys and values, our implementation `SeparateChainingHashST` uses memory for M references to `SequentialSearchST` objects plus M `SequentialSearchST` objects. Each `SequentialSearchST` object has the usual 16 bytes of object overhead plus one 8-byte reference (`first`), and there are a total of N `Node` objects, each with 24 bytes of object overhead plus 3 references (`key`, `value`, and `next`). This compares with an extra reference per node for binary search trees. With array resizing to ensure that the table is between one-eighth and one-half full, linear probing uses between $4N$ and $16N$ references. Thus, choosing hashing on the basis of memory usage is not normally justified. The calculation is a bit different for primitive types (see EXERCISE 3.4.24)

method	space usage for N items (reference types)
<i>separate chaining</i>	$\sim 48N + 32M$
<i>linear probing</i>	between $\sim 32N$ and $\sim 128N$
<i>BSTs</i>	$\sim 56N$

Space usage in symbol tables

SINCE THE EARLIEST DAYS OF COMPUTING, researchers have studied (and are studying) hashing and have found many ways to improve the basic algorithms that we have discussed. You can find a huge literature on the subject. Most of the improvements push down the space-time curve: you can get the same running time for searches using less space or get faster searches using the same amount of space. Other improvements involve better guarantees, on the expected worst-case cost of a search. Others involve improved hash-function designs. Some of these methods are addressed in the exercises.

Detailed comparison of separate chaining and linear probing depends on myriad implementation details and on client space and time requirements. It is not normally justified to choose separate chaining over linear probing on the basis of performance (see EXERCISE 3.5.31). In practice, the primary performance difference between the two methods has to do with the fact that separate chaining uses a small block of memory for each key-value pair, while linear probing uses two large arrays for the whole table. For huge tables, these needs place quite different burdens on the memory management system. In modern systems, this sort of tradeoff is best addressed by experts in extreme performance-critical situations.

With hashing, under generous assumptions, it is not unreasonable to expect to support the search and insert symbol-table operations in constant time, independent of the size of the table. This expectation is the theoretical optimum performance for any symbol-table implementation. Still, hashing is not a panacea, for several reasons, including:

- A good hash function for each type of key is required.
- The performance guarantee depends on the quality of the hash function.
- Hash functions can be difficult and expensive to compute.
- Ordered symbol-table operations are not easily supported.

Beyond these basic considerations, we defer the comparison of hashing with the other symbol-table methods that we have studied to the beginning of SECTION 3.5.

Q&A

Q. How does Java implement `hashCode()` for `Integer`, `Double`, and `Long`?

A. For `Integer` it just returns the 32-bit value. For `Double` and `Long` it returns the exclusive or of the first 32 bits with the second 32 bits of the standard machine representation of the number. These choices may not seem to be very random, but they do serve the purpose of spreading out the values.

Q. When using array resizing, the size M of the table is always a power of 2. Isn't that a potential problem, because it only uses the least significant bits of `hashCode()`?

A. Yes, particularly with the default implementations. One way to address this problem is to first distribute the key values using a prime larger than M , as in the following example:

```
private int hash(Key x)
{
    int t = x.hashCode() & 0x7fffffff;
    if (lgM < 26) t = t % primes[lgM+5];
    return t % M;
}
```

This code assumes that we maintain an instance variable `lgM` that is equal to $\lg M$ (by initializing to the appropriate value, incrementing when doubling, and decrementing when halving) and an array `primes[]` of the largest prime less than each power of 2 (see the table at right). The constant 5 is an arbitrary choice—we expect the first `%` to distribute the values equally among the values less than the prime and the second to map about 2^5 of those values to each value less than M . Note that the point is moot for large M .

k	δ_k	$\text{primes}[k]$ ($2^k - \delta_k$)
5	1	31
6	3	61
7	1	127
8	5	251
9	3	509
10	3	1021
11	9	2039
12	3	4093
13	1	8191
14	3	16381
15	19	32749
16	15	65521
17	1	131071
18	5	262139
19	1	524287
20	3	1048573
21	9	2097143
22	3	4194301
23	15	8388593
24	3	16777213
25	39	33554393
26	5	67108859
27	39	134217689
28	57	268435399
29	3	536870909
30	35	1073741789
31	1	2147483647

Primes for hash table sizes

Q. I've forgotten. Why don't we implement `hash(x)` by returning `x.hashCode() % M`?

A. We need a result between 0 and $M-1$, but in Java, the `%` function may be negative.

Q. So, why not implement `hash(x)` by returning `Math.abs(x.hashCode()) % M`?

A. Nice try. Unfortunately, `Math.abs()` returns a negative result for the largest negative number. For many typical calculations, this overflow presents no real problem, but for hashing it would leave you with a program that is likely to crash after a few billion inserts, an unsettling possibility. For example, `s.hashCode()` is -2^{31} for the Java `String` value "polygenelubricants". Finding other strings that hash to this value (and to 0) has turned into an amusing algorithm-puzzle pastime.

Q. Do Java library hash functions satisfy ASSUMPTION J?

A. No. For example, the `hashCode()` implementation in the `String` data type is not only deterministic but it is specified in the API.

Q. Why not use `BinarySearchST` or `RedBlackBST` instead of `SequentialSearchST` in ALGORITHM 3.5?

A. Generally, we set parameters so as to make the number of keys hashing to each value small, and elementary symbol tables are generally better for the small tables. In certain situations, slight performance gains may be achieved with such hybrid methods, but such tuning is best left for experts.

Q. Is hashing faster than searching in red-black BSTs?

A. It depends on the type of the key, which determines the cost of computing `hashCode()` versus the cost of `compareTo()`. For typical key types and for Java default implementations, these costs are similar, so hashing will be significantly faster, since it uses only a constant number of operations. But it is important to remember that this question is moot if you need ordered operations, which are not efficiently supported in hash tables. See SECTION 3.5 for further discussion.

Q. Why not let the linear probing table get, say, three-quarters full?

A. No particular reason. You can choose any value of α , using PROPOSITION M to estimate search costs. For $\alpha = 3/4$, the average cost of search hits is 2.5 and search misses is 8.5, but if you let α grow to $7/8$, the average cost of a search miss is 32.5, perhaps more than you want to pay. As α gets close to 1, the estimate in PROPOSITION M becomes invalid, but you don't want your table to get that close to being full.

EXERCISES

3.4.1 Insert the keys E A S Y Q U T I O N in that order into an initially empty table of $M = 5$ lists, using separate chaining. Use the hash function $11 \cdot k \% M$ to transform the k th letter of the alphabet into a table index.

3.4.2 Develop an alternate implementation of `SeparateChainingHashST` that directly uses the linked-list code from `SequentialSearchST`.

3.4.3 Modify your implementation of the previous exercise to include an integer field for each key-value pair that is set to the number of entries in the table at the time that pair is inserted. Then implement a method that deletes all keys (and associated values) for which the field is greater than a given integer k . Note: This extra functionality is useful in implementing the symbol table for a compiler.

3.4.4 Write a program to find values of a and M , with M as small as possible, such that the hash function $(a \cdot k) \% M$ for transforming the k th letter of the alphabet into a table index produces distinct values (no collisions) for the keys S E A R C H X M P L. The result is known as a *perfect hash function*.

3.4.5 Is the following implementation of `hashCode()` legal?

```
public int hashCode()
{ return 17; }
```

If so, describe the effect of using it. If not, explain why.

3.4.6 Suppose that keys are t -bit integers. For a modular hash function with prime M , prove that each key bit has the property that there exist two keys differing only in that bit that have different hash values.

3.4.7 Consider the idea of implementing modular hashing for integer keys with the code $(a \cdot k) \% M$, where a is an arbitrary fixed prime. Does this change mix up the bits sufficiently well that you can use nonprime M ?

3.4.8 How many empty lists do you expect to see when you insert N keys into a hash table with `SeparateChainingHashST`, for $N=10, 10^2, 10^3, 10^4, 10^5$, and 10^6 ? *Hint:* See EXERCISE 2.5.31.

3.4.9 Implement an eager `delete()` method for `SeparateChainingHashST`.

3.4.10 Insert the keys E A S Y Q U T I O N in that order into an initially empty table

of size $M=16$ using linear probing. Use the hash function $11 \cdot k \% M$ to transform the k th letter of the alphabet into a table index. Redo this exercise for $M=10$.

3.4.11 Give the contents of a linear-probing hash table that results when you insert the keys E A S Y Q U T I O N in that order into an initially empty table of initial size $M=4$ that is expanded with doubling whenever half full. Use the hash function $11 \cdot k \% M$ to transform the k th letter of the alphabet into a table index.

3.4.12 Suppose that the keys A through G, with the hash values given below, are inserted in some order into an initially empty table of size 7 using a linear-probing table (with no resizing for this problem).

key	A	B	C	D	E	F	G
hash ($M=7$)	2	0	0	4	4	4	2

Which of the following could not possibly result from inserting these keys?

- E F G A C B D
- C E B G F D A
- B D F A C E G
- C G B A D E F
- F G B D A C E
- G E C A D B F

Give the minimum and the maximum number of probes that could be required to build a table of size 7 with these keys, and an insertion order that justifies your answer.

3.4.13 Which of the following scenarios leads to expected linear running time for a random search hit in a linear-probing hash table?

- All keys hash to the same index.
- All keys hash to different indices.
- All keys hash to an even-numbered index.
- All keys hash to different even-numbered indices.

3.4.14 Answer the previous question for search miss, assuming the search key is equally likely to hash to each table position.

3.4.15 How many compares could it take, in the worst case, to insert N keys into an initially empty table, using linear probing with array resizing?

EXERCISES *(continued)*

3.4.16 Suppose that a linear-probing table of size 10^6 is half full, with occupied positions chosen at random. Estimate the probability that all positions with indices divisible by 100 are occupied.

3.4.17 Show the result of using the `delete()` method on page 471 to delete C from the table resulting from using `LinearProbingHashST` with our standard indexing client (shown on page 469).

3.4.18 Add a constructor to `SeparateChainingHashST` that gives the client the ability to specify the average number of probes to be tolerated for searches. Use array resizing to keep the average list size less than the specified value, and use the technique described on page 478 to ensure that the modulus for `hash()` is prime.

3.4.19 Implement `keys()` for `SeparateChainingHashST` and `LinearProbingHashST`.

3.4.20 Add a method to `LinearProbingHashST` that computes the average cost of a search hit in the table, assuming that each key in the table is equally likely to be sought.

3.4.21 Add a method to `LinearProbingHashST` that computes the average cost of a search miss in the table, assuming a random hash function. Note: You do not have to compute any hash functions to solve this problem.

3.4.22 Implement `hashCode()` for various types: `Point2D`, `Interval`, `Interval2D`, and `Date`.

3.4.23 Consider modular hashing for string keys with $R = 256$ and $M = 255$. Show that this is a bad choice because any permutation of letters within a string hashes to the same value.

3.4.24 Analyze the space usage of separate chaining, linear probing, and BSTs for double keys. Present your results in a table like the one on page 476.

CREATIVE PROBLEMS

3.4.25 *Hash cache.* Modify `Transaction` on page 462 to maintain an instance variable `hash`, so that `hashCode()` can save the hash value the first time it is called for each object and does not have to recompute it on subsequent calls. Note: This idea works only for immutable types.

3.4.26 *Lazy delete for linear probing.* Add to `LinearProbingHashST` a `delete()` method that deletes a key-value pair by setting the value to `null` (but not removing the key) and later removing the pair from the table in `resize()`. Your primary challenge is to decide when to call `resize()`. Note: You should overwrite the `null` value if a subsequent `put()` operation associates a new value with the key. Make sure that your program takes into account the number of such tombstone items, as well as the number of empty positions, in making the decision whether to expand or contract the table.

3.4.27 *Double probing.* Modify `SeparateChainingHashST` to use a second hash function and pick the shorter of the two lists. Give a trace of the process of inserting the keys E A S Y Q U T I O N in that order into an initially empty table of size $M=3$ using the function $11k \% M$ (for the k th letter) as the first hash function and the function $17k \% M$ (for the k th letter) as the second hash function. Give the average number of probes for random search hit and search miss in this table.

3.4.28 *Double hashing.* Modify `LinearProbingHashST` to use a second hash function to define the probe sequence. Specifically, replace $(i + 1) \% M$ (both occurrences) by $(i + k) \% M$ where k is a nonzero key-dependent integer that is relatively prime to M . Note: You may meet the last condition by assuming that M is prime. Give a trace of the process of inserting the keys E A S Y Q U T I O N in that order into an initially empty table of size $M=11$, using the hash functions described in the previous exercise. Give the average number of probes for random search hit and search miss in this table.

3.4.29 *Deletion.* Implement an eager `delete()` method for the methods described in each of the previous two exercises.

3.4.30 *Chi-square statistic.* Add a method to `SeparateChainingHashST` to compute the χ^2 statistic for the hash table. With N keys and table size M , this number is defined by the equation

$$\chi^2 = (M/N) \left((f_0 - N/M)^2 + (f_1 - N/M)^2 + \dots + (f_{M-1} - N/M)^2 \right)$$

CREATIVE PROBLEMS *(continued)*

where f_i is the number of keys with hash value i . This statistic is one way of checking our assumption that the hash function produces random values. If so, this statistic, for $N > cM$, should be between $M - \sqrt{M}$ and $M + \sqrt{M}$ with probability $1 - 1/c$.

3.4.31 *Cuckoo hashing.* Develop a symbol-table implementation that maintains two hash tables and two hash functions. Any given key is in one of the tables, but not both. When inserting a new key, hash to one of the tables; if the table position is occupied, replace that key with the new key and hash the old key into the other table (again kicking out a key that might reside there). If this process cycles, restart. Keep the tables less than half full. This method uses a constant number of equality tests in the worst case for search (trivial) and amortized constant time for insert.

3.4.32 *Hash attack.* Find 2^N strings, each of length 2^N , that have the same `hashCode()` value, supposing that the `hashCode()` implementation for `String` is the following:

```
public int hashCode()
{
    int hash = 0;
    for (int i = 0; i < length(); i++)
        hash = (hash * 31) + charAt(i);
    return hash;
}
```

Strong hint: Aa and BB have the same value.

3.4.33 *Bad hash function.* Consider the following `hashCode()` implementation for `String`, which was used in early versions of Java:

```
public int hashCode()
{
    int hash = 0;
    int skip = Math.max(1, length()/8);
    for (int i = 0; i < length(); i += skip)
        hash = (hash * 37) + charAt(i);
    return hash;
}
```

Explain why you think the designers chose this implementation and then why you think it was abandoned in favor of the one in the previous exercise.

EXPERIMENTS

3.4.34 *Hash cost.* Determine empirically the ratio of the time required for `hash()` to the time required for `compareTo()`, for as many commonly-used types of keys for which you can get meaningful results.

3.4.35 *Chi-square test.* Use your solution from EXERCISE 3.4.30 to check the assumption that the hash functions for commonly-used key types produce random values.

3.4.36 *List length range.* Write a program that inserts N random `int` keys into a table of size $N/100$ using separate chaining, then finds the length of the shortest and longest lists, for $N = 10^3, 10^4, 10^5, 10^6$.

3.4.37 *Hybrid.* Run experimental studies to determine the effect of using `RedBlackBST` instead of `SequentialSearchST` to handle collisions in `SeparateChainingHashST`. This solution carries the advantage of guaranteeing logarithmic performance even for a bad hash function and the disadvantage of necessitating maintenance of two different symbol-table implementations. What are the practical effects?

3.4.38 *Separate-chaining distribution.* Write a program that inserts 10^5 random non-negative integers less than 10^6 into a table of size 10^5 using separate chaining, and that plots the total cost for each 10^3 consecutive insertions. Discuss the extent to which your results validate PROPOSITION K.

3.4.39 *Linear-probing distribution.* Write a program that inserts $N/2$ random `int` keys into a table of size N using linear probing, then computes the average cost of a search miss in the resulting table from the cluster lengths, for $N = 10^3, 10^4, 10^5, 10^6$. Discuss the extent to which your results validate PROPOSITION M.

3.4.40 *Plots.* Instrument `LinearProbingHashST` and `SeparateChainingHashST` to produce plots like the ones shown in the text.

3.4.41 *Double probing.* Run experimental studies to evaluate the effectiveness of double probing (see EXERCISE 3.4.27).

3.4.42 *Double hashing.* Run experimental studies to evaluate the effectiveness of double hashing (see EXERCISE 3.4.28).

3.4.43 *Parking problem.* (D. Knuth) Run experimental studies to validate the hypothesis that the number of compares needed to insert M random keys into a linear-probing hash table of size M is $\sim cM^{3/2}$, where $c = \sqrt{\pi}/2$.



3.5 APPLICATIONS

FROM THE EARLY DAYS OF COMPUTING, when symbol tables allowed programmers to progress from using numeric addresses in machine language to using symbolic names in assembly language, to modern applications of the new millennium, when symbolic names have meaning across worldwide computer networks, fast search algorithms have played and continue to play an essential role in computation. Modern applications for symbol tables include organization of scientific data, from searching for markers or patterns in genomic data to mapping the universe; organization of knowledge on the web, from searching in online commerce to putting libraries online; and implementing the internet infrastructure, from routing packets among machines on the web to shared file systems and video streaming. Efficient search algorithms have enabled these and countless other important applications. We will consider several representative examples in this section:

- A dictionary client and an indexing client that enable fast and flexible access to information in comma-separated-value files (and similar formats), which are widely used to store data on the web
- An indexing client for building an inverted index of a set of files
- A sparse-matrix data type that uses a symbol table to address problem sizes far beyond what is possible with the standard implementation

In CHAPTER 6, we consider a symbol table that is appropriate for tables such as databases and file systems that contain a vast number of keys, as large as can be reasonably contemplated.

Symbol tables also play a critical role in algorithms that we consider throughout the rest of the book. For example, we use symbol tables to represent graphs (CHAPTER 4) and to process strings (CHAPTER 5).

As we have seen throughout this chapter, developing symbol-table implementations that can guarantee fast performance for all operations is certainly a challenging task. On the other hand, the implementations that we have considered are well-studied, widely used, and available in many software environments (including Java libraries). From this point forward, you certainly should consider the symbol-table abstraction to be a key component in your programmer's toolbox.

Which symbol-table implementation should I use? The table at the bottom of this page summarizes the performance characteristics of the algorithms that we have considered in propositions and properties in this chapter (with the exception of the worst-case results for hashing, which are from the research literature and unlikely to be experienced in practice). It is clear from the table that, for typical applications, your decision comes down to a choice between hash tables and binary search trees.

The advantages of hashing over BST implementations are that the code is simpler and search times are optimal (constant), if the keys are of a standard type or are sufficiently simple that we can be confident of developing an efficient hash function for them that (approximately) satisfies the uniform hashing assumption. The advantages of BSTs over hashing are that they are based on a simpler abstract interface (no hash function need be designed); red-black BSTs can provide guaranteed worst-case performance; and they support a wider range of operations (such as rank, select, sort, and range search). As a rule of thumb, most programmers will use hashing except when one or more of these factors is important, when red-black BSTs are called for. In CHAPTER 5, we will study one exception to this rule of thumb: when keys are long strings, we can build data structures that are even more flexible than red-black BSTs and even faster than hashing.

algorithm (data structure)	worst-case cost (after N inserts)		average-case cost (after N random inserts)		key interface	memory (bytes)
	search	insert	search hit	insert		
<i>sequential search (unordered list)</i>	N	N	$N/2$	N	<code>equals()</code>	$48 N$
<i>binary search (ordered array)</i>	$\lg N$	N	$\lg N$	$N/2$	<code>compareTo()</code>	$16 N$
<i>binary tree search (BST)</i>	N	N	$1.39 \lg N$	$1.39 \lg N$	<code>compareTo()</code>	$64 N$
<i>2-3 tree search (red-black BST)</i>	$2 \lg N$	$2 \lg N$	$1.00 \lg N$	$1.00 \lg N$	<code>compareTo()</code>	$64 N$
<i>separate chaining[†] (array of lists)</i>	$< \lg N$	$< \lg N$	$N/(2M)$	N/M	<code>equals()</code> <code>hashCode()</code>	$48 N + 32 M$
<i>linear probing[†] (parallel arrays)</i>	$c \lg N$	$c \lg N$	< 1.50	< 2.50	<code>equals()</code> <code>hashCode()</code>	between $32 N$ and $128 N$

[†] under uniform hashing assumption

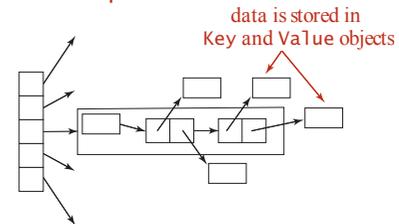
Asymptotic cost summary for symbol-table implementations

Our symbol-table implementations are useful for a wide range of applications, but our algorithms are easily adapted to support several other options that are widely used and worth considering.

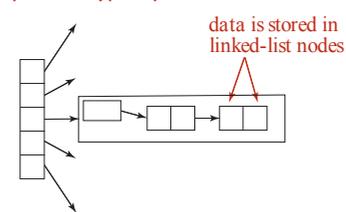
Primitive types. Suppose that we have a symbol table with integer keys and associated floating-point numbers. When we use our standard setup, the keys and values are stored as `Integer` and `Double` wrapper-type values, so we need two extra memory references to access each key-value pair. These references may be no problem in an application that involves thousands of searches on thousands of keys but may represent excessive cost in an application that involves billions of searches on millions of keys. Using a primitive type instead of `Key` would save one reference per key-value pair. When the associated value is also primitive, we can eliminate another reference. The situation is diagrammed at right for separate chaining; the same tradeoffs hold for other implementations. For performance-critical applications, it is worthwhile and not difficult to develop versions of our implementations along these lines (see EXERCISE 3.5.4).

Duplicate keys. The possibility of duplicate keys sometimes needs special consideration in symbol-table implementations. In many applications, it is desirable to associate multiple values with the same key. For example, in a transaction-processing system, numerous transactions may have the same customer key value. Our convention to disallow duplicate keys amounts to leaving duplicate-key management to the client. We will consider an example of such a client later in this section. In many of our implementations, we could consider the alternative of leaving key-value pairs with duplicate keys in the primary search data structure and to return any value with the given key for a search. We might also add methods to return all values with the given key. Our BST and hashing implementations are not difficult to adapt to keep duplicate keys within the data structure; doing so for red-black BSTs is just slightly more challenging (see EXERCISE 3.5.9 and EXERCISE 3.5.10). Such implementations are common in the literature (including earlier editions of this book).

standard implementation



primitive-type implementation



Memory usage for separate chaining

Java libraries. Java’s `java.util.TreeMap` and `java.util.HashMap` libraries are symbol-table implementations based on red-black BSTs and hashing with separate chaining respectively. `TreeMap` does not directly support `rank()`, `select()`, and other operations in our ordered symbol-table API, but it does support operations that enable efficient implementation of these. `HashMap` is roughly equivalent to our `SeparateChainingHashST` implementation—it uses array resizing to enforce a load factor of about 75 percent. Java’s `java.util.IdentityHashMap` library is a symbol-table implementation that uses reference-equality in place of object-equality; it is roughly equivalent to our `LinearProbingHashST` with a load factor of 2/3.

TO BE CONSISTENT AND SPECIFIC, we use in this book the symbol-table implementation based on red-black BSTs from SECTION 3.3 or the one based on linear-probing hashing from SECTION 3.4. For economy and to emphasize client independence from specific implementations, we use the name `ST` as shorthand for `RedBlackBST` for ordered symbol tables in client code and the name `HashST` as shorthand for `LinearProbingHashST` when order is not important and hash functions are available. We adopt these conventions with full knowledge that specific applications might have demands that could call for some variation or extension of one of these algorithms and data structures. Which symbol table should you use? Whatever you decide, test your choice to be sure that it is delivering the performance that you expect.

Set APIs Some symbol-table clients do not need the values, just the ability to insert keys into a table and to test whether a key is in the table. Because we disallow duplicate keys, these operations correspond to the following API where we are just interested in the set of keys in the table, not any associated values:

```
public class SET<Key>


---


    SET()                create an empty set
    void add(Key key)    add key into the set
    void delete(Key key) remove key from the set
    boolean contains(Key key) is key in the set?
    boolean isEmpty()    is the set empty?
    int size()           number of keys in the set
    String toString()    string representation of the set
```

API for a basic set data type

You can turn any symbol-table implementation into a SET implementation by ignoring values or by using a simple wrapper class (see EXERCISES 3.5.1 through 3.5.3).

Extending SET to include union, intersection, complement, and other common mathematical set operations requires a more sophisticated API (for example, the complement operation requires some mechanism for specifying a universe of all possible keys) and provides a number of interesting algorithmic challenges, as discussed in EXERCISE 3.5.17.

As with ST, we have unordered and ordered versions of SET. If keys are Comparable, we can include `min()`, `max()`, `floor()`, `ceiling()`, `deleteMin()`, `deleteMax()`, `rank()`, `select()`, and the two-argument versions of `size()` and `get()` to define a full API for ordered keys. To match our ST conventions, we use the name SET in client code for ordered sets and the name HashSET when order is not important.

To illustrate uses of SET, we consider *filter* clients that read a sequence of strings from standard input and write some of them to standard output. Such clients have their origin in early systems where main memory was far too small to hold all the data, and they are still relevant today, when we write programs that take their input from the web. As example input, we use `tinyTale.txt` (see page 371). For readability, we preserve newlines from the input to the output in examples, even though the code does not do so.

Dedup. The prototypical filter example is a SET or HashSET client that removes duplicates in the input stream. It is customary to refer to this operation as dedup. We maintain a set of the string keys seen so far. If the next key is in the set, ignore it; if it is not in the set, add it to the set and print it. The keys appear on standard output in the order they appear on standard input, with duplicates removed. This process takes space proportional to the number of distinct keys in the input stream (which is typically far smaller than the total number of keys).

```
public class DeDup
{
    public static void main(String[] args)
    {
        HashSet<String> set;
        set = new HashSet<String>();
        while (!StdIn.isEmpty())
        {
            String key = StdIn.readString();
            if (!set.contains(key))
            {
                set.add(key);
                StdOut.println(key);
            }
        }
    }
}
```

Dedup filter

```
% java DeDup < tinyTale.txt
it was the best of times worst
age wisdom foolishness
epoch belief incredulity
season light darkness
spring hope winter despair
```

Whitelist and blacklist. Another classic filter uses keys in a separate file to decide which keys from the input stream are passed to the output stream. This general process has many natural applications. The simplest example is a *whitelist*, where any key that is in the file is identified as “good.” The client might choose to pass through to standard output any key that is not in the whitelist and to ignore any key that is in the whitelist (as in the example considered in our first program in CHAPTER 1); another client might choose to pass through to standard output any key that is in the whitelist and to ignore any key that is not in the whitelist (as shown in the HashSET client `WhiteFilter` at right). For example, your email application might use such a filter to allow you to specify the addresses of your friends and to direct it to consider emails from anyone else as spam. We build a HashSET of the keys in the specified list, then read the keys from standard input. If the next key is in the set, print it; if it is not in the set, ignore it. A blacklist is the opposite, where any key that is in the file is identified as “bad.” Again, there are two natural filters for clients using a blacklist. In our email example, you might specify the addresses of known spammers and direct the email application to let through all mail not from one of those addresses. We can implement a HashSET client `BlackFilter` that implements this filter by negating the filter test in `WhiteFilter`. Typical practical situations such as a credit card company using a blacklist to filter out stolen card numbers or an internet router using a whitelist to implement a firewall are likely to involve huge lists, unbounded input streams, and strict response requirements. The sorts of symbol-table implementations that we have considered enable such challenges to easily be met.

```
public class WhiteFilter
{
    public static void main(String[] args)
    {
        HashSet<String> set;
        set = new HashSet<String>();
        In in = new In(args[0]);
        while (!in.isEmpty())
            set.add(in.readString());
        while (!StdIn.isEmpty())
        {
            String word = StdIn.readString();
            if (set.contains(word))
                StdOut.println(word);
        }
    }
}
```

Whitelist filter

```
% more list.txt
was it the of

% java WhiteFilter list.txt < tinyTale.txt
it was the of it was the of

% java BlackFilter list.txt < tinyTale.txt
best times worst times
age wisdom age foolishness
epoch belief epoch incredulity
season light season darkness
spring hope winter despair
```

Dictionary clients The most basic kind of symbol-table client builds a symbol table with successive put operations in order to support get requests. Many applications also take advantage of the idea that a symbol table is a dynamic dictionary, where it is easy to look up information and to update the information in the table. The following list of familiar examples illustrates the utility of this approach:

- *Phone book.* When keys are people’s names and values are their phone numbers, a symbol table models a phone book. A very significant difference from a printed phone book is that we can add new names or change existing phone numbers. We could also use the phone number as the key and the name as the value—if you have never done so, try typing your phone number (with area code) into the search field in your browser.
- *Dictionary.* Associating a word with its definition is a familiar concept that gives us the name “dictionary.” For centuries people kept printed dictionaries in their homes and offices in order to check the definitions and spellings (values) of words (keys). Now, because of good symbol-table implementations, people expect built-in spell checkers and immediate access to word definitions on their computers.
- *Account information.* People who own stock now regularly check the current price on the web. Several services on the web associate a ticker symbol (key) with the current price (value), usually along with a great deal of other information. Commercial applications of this sort abound, including financial institutions associating account information with a name or account number or educational institutions associating grades with a student name or identification number.
- *Genomics.* Symbols play a central role in modern genomics. The simplest example is the use of the letters A, C, T, and G to represent the nucleotides found in the DNA of living organisms. The next simplest is the correspondence between codons (nucleotide triplets) and amino acids (TTA corresponds to leucine, TCT to serine, and so forth), then the correspondence between sequences of amino acids and proteins, and so forth. Researchers in genomics routinely use various types of symbol tables to organize this knowledge.
- *Experimental data.* From astrophysics to zoology, modern scientists are awash in experimental data, and organizing and efficiently accessing this data are vital to understanding what it means. Symbol tables are a critical starting point, and advanced data structures and algorithms that are based on symbol tables are now an important part of scientific research.
- *Compilers.* One of the earliest uses of symbol tables was to organize information for programming. At first, programs were simply sequences of numbers, but programmers very quickly found that using symbolic names for operations and

memory locations (variable names) was far more convenient. Associating the names with the numbers requires a symbol table. As the size of programs grew, the cost of the symbol-table operations became a bottleneck in program development time, which led to the development of data structures and algorithms like the ones we consider in this chapter.

- *File systems.* We use symbol tables regularly to organize data on computer systems. Perhaps the most prominent example is the *file system*, where we associate a file name (key) with the location of its contents (value). Your music player uses the same system to associate song titles (keys) with the location of the music itself (value).
- *Internet DNS.* The domain name system (DNS) that is the basis for organizing information on the internet associates URLs (keys) that humans understand (such as `www.princeton.edu` or `www.wikipedia.org`) with IP addresses (values) that computer network routers understand (such as `208.216.181.15` or `207.142.131.206`). This system is the next-generation “phone book.” Thus, humans can use names that are easy to remember and machines can efficiently process the numbers. The number of symbol-table lookups done each second for this purpose on internet routers around the world is huge, so performance is of obvious importance. Millions of new computers and other devices are put onto the internet each year, so these symbol tables on internet routers need to be dynamic.

Despite its scope, this list is still just a representative sample, intended to give you a flavor of the scope of applicability of the symbol-table abstraction. Whenever you specify something by name, there is a symbol table at work. Your computer’s file system or the web might do the work for you, but there is still a symbol table there somewhere.

As a specific example, we consider a symbol-table client that you can use to look up information that is kept in a table on a file or a web page using the comma-separated-value (.csv) file format. This simple format achieves the (admittedly modest) goal of keeping tabular data in a form that anyone can read (and is likely to be able to read in the future) without needing to use a particular application: the data is in text form, one row per line, with entries separated by commas. You can find on the booksite

domain	key	value
<i>phone book</i>	name	phone number
<i>dictionary</i>	word	definition
<i>account</i>	account number	balance
<i>genomics</i>	codon	amino acid
<i>data</i>	data/time	results
<i>compiler</i>	variable name	memory location
<i>file share</i>	song name	machine
<i>internet</i>	website	IP address

Typical dictionary applications

```

% more amino.csv
TTT,Phe,F,Phenylalanine
TTC,Phe,F,Phenylalanine
TTA,Leu,L,Leucine
TTG,Leu,L,Leucine
TCT,Ser,S,Serine
TCC,Ser,S,Serine
...
GAA,Gly,G,Glutamic Acid
GAG,Gly,G,Glutamic Acid
GGT,Gly,G,Glycine
GGC,Gly,G,Glycine
GGA,Gly,G,Glycine
GGG,Gly,G,Glycine

% more DJIA.csv
...
20-Oct-87,1738.74,608099968,1841.01
19-Oct-87,2164.16,604300032,1738.74
16-Oct-87,2355.09,338500000,2246.73
15-Oct-87,2412.70,263200000,2355.09
...
30-Oct-29,230.98,10730000,258.47
29-Oct-29,252.38,16410000,230.07
28-Oct-29,295.18,9210000,260.64
25-Oct-29,299.47,5920000,301.22
...

% more ip.csv
...
www.ebay.com,66.135.192.87
www.princeton.edu,128.112.128.15
www.cs.princeton.edu,128.112.136.35
www.harvard.edu,128.103.60.24
www.yale.edu,130.132.51.8
www.cnn.com,64.236.16.20
www.google.com,216.239.41.99
www.nytimes.com,199.239.136.200
www.apple.com,17.112.152.32
www.slashdot.org,66.35.250.151
www.espn.com,199.181.135.201
www.weather.com,63.111.66.11
www.yahoo.com,216.109.118.65
...

% more UPC.csv
...
0002058102040,,"1 1/4"" STANDARD STORM DOOR"
0002058102057,,"1 1/4"" STANDARD STORM DOOR"
0002058102125,,"DELUXE STORM DOOR UNIT"
0002082012728,"100/ per box","12 gauge shells"
0002083110812,"Classical CD","Bits and Pieces"
002083142882,CD,"Garth Brooks - Ropin' The Wind"
0002094000003,LB,"PATE PARISIEN"
0002098000009,LB,"PATE TRUFFLE COGNAC-M&H 8Z RW"
0002100001086,"16 oz","Kraft Parmesan"
0002100002090,"15 pieces","Wrigley's Gum"
0002100002434,"One pint","Trader Joe's milk"
...

```

Typical comma-separated-value (.csv) files

numerous .csv files that are related to various applications that we have described, including amino.csv (codon-to-amino-acid encodings), DJIA.csv (opening price, volume, and closing price of the Dow Jones Industrial Average, for every day in its history), ip.csv (a selection of entries from the DNS database), and upc.csv (the Uniform Product Code bar codes that are widely used to identify consumer products). Spreadsheet and other data-processing applications programs can read and write .csv files, and our example illustrates that you can also write a Java program to process the data any way that you would like.

LookupCSV (on the facing page) builds a set of key-value pairs from a file of comma-separated values as specified on the command line and then prints out values corresponding to keys read from standard input. The command-line arguments are the file name and two integers, one specifying the field to serve as the key and the other specifying the field to serve as the value.

The purpose of this example is to illustrate the utility and flexibility of the symbol-table abstraction. What website has IP address 128.112.136.35? (www.cs.princeton.edu) What amino acid corresponds to the codon TCC? (Serine) What was the DJIA on October 29, 1929? (230.07) What product has UPC 0002100001086? (Kraft Parmesan) You can easily look up the answers to questions like these with LookupCSV and the appropriate .csv files.

Performance is not much of an issue when handling interactive queries (since your computer can look through millions

Dictionary lookup

```
public class LookupCSV
{
    public static void main(String[] args)
    {
        In in = new In(args[0]);
        int keyField = Integer.parseInt(args[1]);
        int valField = Integer.parseInt(args[2]);

        ST<String, String> st = new ST<String, String>();

        while (in.hasNextLine())
        {
            String line = in.readLine();
            String[] tokens = line.split(",");
            String key = tokens[keyField];
            String val = tokens[valField];
            st.put(key, val);
        }

        while (!StdIn.isEmpty())
        {
            String query = StdIn.readString();
            if (st.contains(query))
                StdOut.println(st.get(query));
        }
    }
}
```

This data-driven symbol-table client reads key-value pairs from a file, then prints the values corresponding to the keys found on standard input. Both keys and values are strings. The fields to serve as the key and value are taken as command-line arguments.

```
% java LookupCSV ip.csv 1 0
128.112.136.35
www.cs.princeton.edu
```

```
% java LookupCSV amino.csv 0 3
TCC
Serine
```

```
% java LookupCSV DJIA.csv 0 3
29-Oct-29
230.07
```

```
% java LookupCSV UPC.csv 0 2
0002100001086
Kraft Parmesan
```

of things in the time it takes to type a query), so fast implementations of ST are not noticeable when you use LookupCSV. However, when a program is doing the lookups (and a huge number of them), performance matters. For example, an internet router might need to look up millions of IP addresses per second. In this book, we have already seen the need for good performance with FrequencyCounter, and we will see several other examples in this section.

Examples of similar but more sophisticated test clients for .csv files are described in the exercises. For instance, we could make the dictionary dynamic by also allowing standard-input commands to change the value associated with a key, or we could allow range searching, or we could build multiple dictionaries for the same file.

Indexing clients Dictionaries are characterized by the idea that there is one value associated with each key, so the direct use of our ST data type, which is based on the associative-array abstraction that assigns one value to each key, is appropriate. Each account number uniquely identifies a customer, each UPC uniquely identifies a product, and so forth. In general, of course, there may be multiple values associated with a given key. For example, in our amino.csv example, each codon identifies one amino acid, but each amino acid is associated with a list of codons, as in the example aminoI.csv at right, where each line contains an amino acid and the list of codons associated with it. We use the term *index* to describe symbol tables that associate multiple values with each key. Here are some more examples:

- **Commercial transactions.** One way for a company that maintains customer accounts to keep track of a day's transactions is to keep an index of the day's transactions. The key is the account number; the value is the list of occurrences of that account number in the transaction list.
- **Web search.** When you type a keyword and get a list of websites containing that keyword, you are using an index created by your web search engine. There is one value (the set of pages) associated with each key (the query), although the reality is a bit more complicated because we often specify multiple keys.

aminoI.csv

```

Alanine,AAT,AAC,GCT,GCC,GCA,GCG
Arginine,CGT,CGC,CGA,CGG,AGA,AGG
Aspartic Acid,GAT,GAC
Cysteine,TGT,TGC
Glutamic Acid,GAA,GAG
Glutamine,CAA,CAG
Glycine,GGT,GGC,GGG,"," separator
Histidine,CAT,CAC
Isoleucine,ATT,ATC,ATA
Leucine,TTA,TTG,CTT,CTC,CTA,CTG
Lysine,AAA,AAG
Methionine,ATG
Phenylalanine,TTT,TTC
Proline,CCT,CCC,CCA,CCG
Serine,TCT,TCA,TCG,AGT,AGC
Stop,TAA,TAG,TGA
Threonine,ACT,ACC,ACA,ACG
Tyrosine,TAT,TAC
Tryptophan,TGG
Valine,GTT,GTC,GTA,GTG

```

↑ key

↓ values

A small index file (20 lines)

- *Movies and performers.* The file `movies.txt` on the booksite (excerpted below) is taken from the *Internet Movie Database* (IMDB). Each line has a movie name (the key), followed by a list of performers in that movie (the value), separated by slashes.

We can easily build an index by putting the values to be associated with each key into a single data structure (a Queue, say) and then associating that key with that data structure as value. Extending `LookupCSV` along these lines is straightforward, but we leave that as an exercise (see EXERCISE 3.5.12) and consider instead `LookupIndex` on page 499, which uses a symbol table to build an index from files like `aminoI.txt` and `movies.txt` (where the separator character need not be a comma, as in a `.csv` file, but can be specified on the command line). After building the index, `LookupIndex` then takes key queries and prints the values associated with each key. More interesting, `LookupIndex` also builds an inverted index associated with each file, where values and keys switch roles. In the amino acid example, this gives the same functionality as `Lookup` (find the amino acid associated with a given codon); in the movie-performer example it adds the ability to find the movies associated with any given performer, which is implicit in the data but would be difficult to produce without a symbol table. *Study this example carefully*, as it provides good insight into the essential nature of symbol tables.

domain	key	value
<i>genomics</i>	amino acid	list of codons
<i>commercial</i>	account number	list of transactions
<i>web search</i>	search key	list of web pages
<i>IMDB</i>	movie	list of performers

Typical indexing applications

```

movies.txt
...
Tin Men (1987)/DeBoy, David/Blumenfeld, Alan/...
Tirez sur le pianiste (1960)/Heymann, Claude/...
Titanic (1997)/Mazin, Stan/...DiCaprio, Leonardo/...
Titus (1999)/Weisskopf, Hermann/Rhys, Matthew/...
To Be or Not to Be (1942)/Verebes, Ernö (I)/...
To Be or Not to Be (1983)/.../Brooks, Mel (I)/...
To Catch a Thief (1955)/Paris, Manuel/...
To Die For (1995)/Smith, Kurtwood/.../Kidman, Nicole/...
...

```

↑ key

"/" separator

values

Small portion of a large index file (250,000+ lines)

Inverted index. The term inverted index is normally applied to a situation where values are used to locate keys. We have a large amount of data and want to know where certain keys of interest occur. This application is another prototypical example of a symbol-table client that uses an intermixed sequence of calls to `get()` and `put()`. Again, we associate each key with a SET of locations, where the occurrences of the key can be found. The nature and use of the location depend on the application: in a book, a location might be a page number; in a program, a location might be a line number; in genomics, a location might be a position in a genetic sequence; and so forth:

- *Internet Movie DataBase (IMDB).* In the example just considered, the input is an index that associates each movie with a list of performers. The inverted index associates each performer with a list of movies.
- *Book index.* Every textbook has an index where you look up a term and get the page numbers containing that term. While creating a good index generally involves work by the book author to eliminate common and irrelevant words, a document preparation system will certainly use a symbol table to help automate the process. An interesting special case is known as a concordance, which associates each word in a text with the set of positions in the text where that word occurs (see EXERCISE 3.5.20).

domain	key	value
<i>IMDB</i>	performer	set of movies
<i>book</i>	term	set of pages
<i>compiler</i>	identifier	set of places used
<i>file search</i>	search term	set of files
<i>genomics</i>	subsequence	set of locations

Typical inverted indices

- *Compiler.* In a large program that uses a large number of symbols, it is useful to know where each name is used. Historically, an explicit printed symbol table was one of the most important tools used by programmers to keep track of where symbols are used in their programs. In modern systems, symbol tables are the basis of software tools that programmers use to manage names.
- *File search.* Modern operating systems provide you with the ability to type a term and to learn the names of files containing that term. The key is the term; the value is the set of files containing that term.
- *Genomics.* In a typical (if oversimplified) scenario in genomics research, a scientist wants to know the positions of a given genetic sequence in an existing genome or set of genomes. Existence or proximity of certain sequences may be of scientific significance. The starting point for such research is an index like a concordance, but modified to take into account the fact that genomes are not separated into words (see EXERCISE 3.5.15).

Index (and inverted index) lookup

```

public class LookupIndex
{
    public static void main(String[] args)
    {
        In in = new In(args[0]);    // index database
        String sp = args[1];        // separator

        ST<String, Queue<String>> st = new ST<String, Queue<String>>();
        ST<String, Queue<String>> ts = new ST<String, Queue<String>>();

        while (in.hasNextLine())
        {
            String[] a = in.readLine().split(sp);
            String key = a[0];
            for (int i = 1; i < a.length; i++)
            {
                String val = a[i];
                if (!st.contains(key)) st.put(key, new Queue<String>());
                if (!ts.contains(val)) ts.put(val, new Queue<String>());
                st.get(key).enqueue(val);
                ts.get(val).enqueue(key);
            }
        }

        while (!StdIn.isEmpty())
        {
            String query = StdIn.readLine();
            if (st.contains(query))
                for (String s : st.get(query))
                    StdOut.println(" " + s);

            if (ts.contains(query))
                for (String s : ts.get(query))
                    StdOut.println(" " + s);
        }
    }
}

```

This data-driven symbol-table client reads key-value pairs from a file, then prints the values corresponding to the keys found on standard input. Keys are strings; values are lists of strings. The separating delimiter is taken as a command-line argument.

```

% java LookupIndex aminoI.csv ","
Serine
TCT
TCA
TCG
AGT
AGC
TCG
Serine
% java LookupIndex movies.txt "/"
Bacon, Kevin
Animal House (1978)
Apollo 13 (1995)
Beauty Shop (2005)
Diner (1982)
...
Tin Men (1987)
DeBoy, David
Blumenfeld, Alan
...

```

`FileIndex` (on the facing page) takes file names from the command line and uses a symbol table to build an inverted index associating every word in any of the files with a SET of file names where the word can be found, then takes keyword queries from standard input, and produces its associated list of files. This process is similar to that used by familiar software tools for searching the web or for searching for information on your computer; you type a keyword to get a list of places where that keyword occurs. Developers of such tools typically embellish the process by paying careful attention to

- The form of the query
- The set of files/pages that are indexed
- The order in which files are listed in the response

For example, you are certainly used to typing queries that contain multiple keywords to a web search engine (which is based on indexing a large fraction of the pages on the web) that provides answers in order of relevance or importance (to you or to an advertiser). The exercises at the end of this section address some of these embellishments. We will consider various algorithmic issues related to web search later, but the symbol table is certainly at the heart of the process.

As with `LookupIndex`, you are certainly encouraged to download `FileIndex` from the booksite and use it to index some text files on your computer or some websites of interest, to gain further appreciation for the utility of symbol tables. If you do so, you will find that it can build large indices for huge files with little delay, because each put operation and get request is taken care of immediately. Providing this immediate response for huge dynamic tables is one of the classic triumphs of algorithmic technology.

File indexing

```

import java.io.File;
public class FileIndex
{
    public static void main(String[] args)
    {
        ST<String, SET<File>> st = new ST<String, SET<File>>();
        for (String filename : args)
        {
            File file = new File(filename);
            In in = new In(file);
            while (!in.isEmpty())
            {
                String word = in.readString();
                if (!st.contains(word)) st.put(word, new SET<File>());
                SET<File> set = st.get(word);
                set.add(file);
            }
        }
        while (!StdIn.isEmpty())
        {
            String query = StdIn.readString();
            if (st.contains(query))
                for (File file : st.get(query))
                    StdOut.println(" " + file.getName());
        }
    }
}

```

This symbol-table client indexes a set of files. We search for each word in each file in a symbol table, maintaining a SET of file names that contain the word. Names for In can also refer to web pages, so this code can also be used to build an inverted index of web pages.

```

% more ex1.txt
it was the best of times

% more ex2.txt
it was the worst of times

% more ex3.txt
it was the age of wisdom

% more ex4.txt
it was the age of foolishness

```

```

% java FileIndex ex*.txt
age
  ex3.txt
  ex4.txt
best
  ex1.txt
was
  ex1.txt
  ex2.txt
  ex3.txt
  ex4.txt

```

Sparse vectors Our next example illustrates the importance of symbol tables in scientific and mathematical calculations. We describe a fundamental and familiar calculation that becomes a bottleneck in typical practical applications, then show how using a symbol table can remove the bottleneck and enable solution of vastly larger problems. Indeed, this particular calculation was at the core of the PageRank algorithm that was developed by S. Brin and L. Page and led to the emergence of Google in the early 2000s (and is a well-known mathematical abstraction that is useful in many other contexts).

$$\begin{array}{c}
 \mathbf{a}[][] \\
 \left[\begin{array}{ccccc}
 0 & .90 & 0 & 0 & 0 \\
 0 & 0 & .36 & .36 & .18 \\
 0 & 0 & 0 & .90 & 0 \\
 .90 & 0 & 0 & 0 & 0 \\
 .47 & 0 & .47 & 0 & 0
 \end{array} \right]
 \end{array}
 \begin{array}{c}
 \mathbf{x}[] \\
 \left[\begin{array}{c}
 .05 \\
 .04 \\
 .36 \\
 .37 \\
 .19
 \end{array} \right]
 \end{array}
 =
 \begin{array}{c}
 \mathbf{b}[] \\
 \left[\begin{array}{c}
 .036 \\
 .297 \\
 .333 \\
 .045 \\
 .1927
 \end{array} \right]
 \end{array}$$

Matrix-vector multiplication

The basic calculation that we consider is matrix-vector multiplication: given a matrix and a vector, compute a result vector whose i th entry is the dot product of the given vector and the i th row of the matrix. For simplicity, we consider the case when the matrix is square with N rows and N columns and the vectors are of size N . This operation is elementary to code in Java, requiring time proportional to N^2 , for the N multiplications to compute each of the N entries in the

result vector, which also matches the space proportional to N^2 that is required to store the matrix.

In practice, it is very often the case that N is huge. For example, in the Google application cited above, N is the number of pages on the web. At the time PageRank was developed, that was in the tens or hundreds of billions and it has skyrocketed since, so the value of N^2 would be far more than 10^{20} . No one can afford that much time or space, so a better algorithm is needed.

Fortunately, it is also often the case that the matrix is sparse: a huge number of its entries are 0. Indeed, for the Google application, the average number of nonzero entries per row is a small constant: virtually all web pages have links to only a few others (not all the pages on the web). Accordingly, we can represent the matrix as an array of sparse vectors, using a `SparseVector` implementation like the `HashST` client on the facing page. Instead of using the

```

...
double[][] a = new double[N][N];
double[] x = new double[N];
double[] b = new double[N];
...
// Initialize a[][] and x[]
...
for (int i = 0; i < N; i++)
{
    sum = 0.0;
    for (int j = 0; j < N; j++)
        sum += a[i][j]*x[j];
    b[i] = sum;
}

```

Standard implementation of matrix-vector multiplication

Sparse vector with dot product

```
public class SparseVector
{
    private HashST<Integer, Double> st;

    public SparseVector()
    { st = new HashST<Integer, Double>(); }

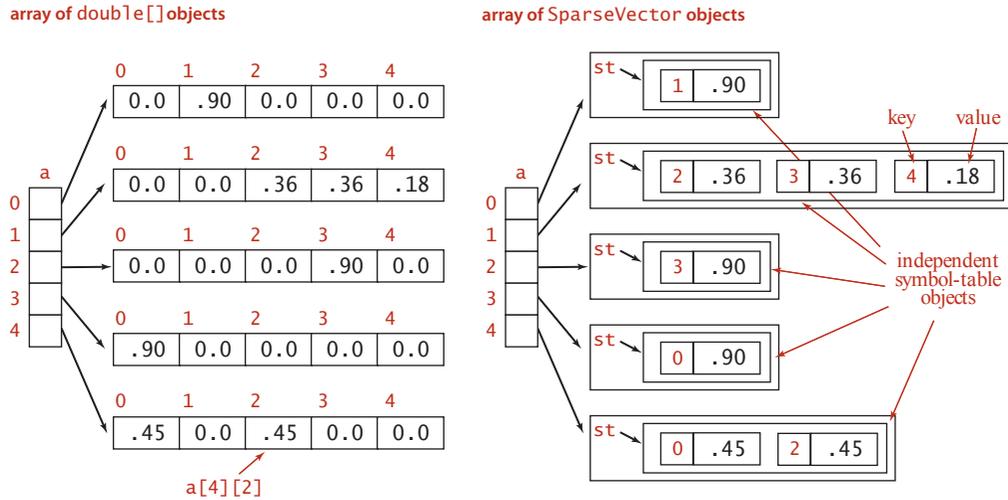
    public int size()
    { return st.size(); }

    public void put(int i, double x)
    { st.put(i, x); }

    public double get(int i)
    {
        if (!st.contains(i)) return 0.0;
        else return st.get(i);
    }

    public double dot(double[] that)
    {
        double sum = 0.0;
        for (int i : st.keys())
            sum += that[i]*this.get(i);
        return sum;
    }
}
```

This symbol-table client is a bare-bones sparse vector implementation that illustrates an efficient dot product for sparse vectors. We multiply each entry by its counterpart in the other operand and add the result to a running sum. The number of multiplications required is equal to the number of nonzero entries in the sparse vector.



Sparse matrix representations

code `a[i][j]` to refer to the element in row `i` and column `j`, we use `a[i].put(j, val)` to set a value in the matrix and `a[i].get(j)` to retrieve a value. As you can see from the code below, matrix-vector multiplication using this class is even simpler than with the array representation (and it more clearly describes the computation). More important, it only requires time proportional to N plus the number of nonzero elements in the matrix.

For small matrices or matrices that are not sparse, the overhead for maintaining symbol tables can be substantial, but it is worth your while to be sure to understand the ramifications of using symbol tables for huge sparse matrices. To fix ideas, consider a huge application (like the one faced by Brin and Page) where N is 10 billion or 100 billion, but the average number of nonzero elements per row is less than 10. For such an application, using symbol tables speeds up matrix-vector multiplication by a factor of a billion or more. The elementary nature of this application should not detract from its importance: programmers who do not take advantage of the potential to save time and space in this way severely limit their potential to solve practical problems, while programmers who do

```
..
SparseVector[] a;
a = new SparseVector[N];
double[] x = new double[N];
double[] b = new double[N];
...
// Initialize a[] and x[]
...
for (int i = 0; i < N; i++)
    b[i] = a[i].dot(x);
```

Sparse matrix-vector multiplication

take factor-of-a-billion speedups when they are available are likely to be able to address problems that could not otherwise be contemplated.

Building the matrix for the Google application is a graph-processing application (and a symbol-table client!), albeit for a huge sparse matrix. Given the matrix, the Page-Rank calculation is nothing more than doing a matrix-vector multiplication, replacing the source vector with the result vector, and iterating the process until it converges (as guaranteed by fundamental theorems in probability theory). Thus, the use of a class like `SparseVector` can improve the time and space usage for this application by a factor of 10 billion or 100 billion or more.

Similar savings are possible in many scientific calculations, so sparse vectors and matrices are widely used and typically incorporated into specialized systems for scientific computing. When working with huge vectors and matrices, it is wise to run simple performance tests to be sure that the kinds of performance gains that we have illustrated here are not being missed. On the other hand, array processing for primitive types of data is built into most programming languages, so using arrays for vectors that are not sparse, as we did in this example, may offer further speedups. Developing a good understanding of the underlying costs and making the appropriate implementation decisions is certainly worthwhile for such applications.

SYMBOL TABLES ARE A PRIMARY CONTRIBUTION OF ALGORITHMIC TECHNOLOGY to the development of our modern computational infrastructure because of their ability to deliver savings on a huge scale in a vast array of practical applications, making the difference between providing solutions to a wide range of problems and not being able to address them at all. Few fields of science or engineering involve studying the effects of an invention that improves costs by factors of 100 billion—symbol-table applications put us in just that position, as we have just seen in several examples, and these improvements have had profound effects. The data structures and algorithms that we have considered are certainly not the final word: they were all developed in just a few decades, and their properties are not fully understood. Because of their importance, symbol-table implementations continue to be studied intensely by researchers around the world, and we can look forward to new developments on many fronts as the scale and scope of the applications they address continue to expand.

Q&A

Q. Can a SET contain null?

A. No. As with symbol tables, keys are non-null objects.

Q. Can a SET be null?

A. No. A SET can be empty (contain no objects), but not null. As with any Java data type, a variable of type SET can have the value null, but that just indicates that it does not reference any SET. The result of using new to create a SET is always an object that is not null.

Q. If all my data is in memory, there is no real reason to use a filter, right?

A. Right. Filtering really shines in the case when you have no idea how much data to expect. Otherwise, it may be a useful way of thinking, but not a cure-all.

Q. I have data in a spreadsheet. Can I develop something like LookupCSV to search through it?

A. Your spreadsheet application probably has an option to export to a .csv file, so you can use LookupCSV directly.

Q. Why would I need FileIndex? Doesn't my operating system solve this problem?

A. If you are using an OS that meets your needs, continue to do so, by all means. As with many of our programs, FileIndex is intended to show you the basic underlying mechanisms of such applications and to suggest possibilities to you.

Q. Why not have the dot() method in SparseVector take a SparseVector object as argument and return a SparseVector object?

A. That is a fine alternate design and a nice programming exercise that requires code that is a bit more intricate than for our design (see EXERCISE 3.5.16). For general matrix processing, it might be worthwhile to also add a SparseMatrix type.

EXERCISES

- 3.5.1** Implement SET and HashSET as “wrapper class” clients of ST and HashST, respectively (provide dummy values and ignore them).
- 3.5.2** Develop a SET implementation `SequentialSearchSET` by starting with the code for `SequentialSearchST` and eliminating all of the code involving values.
- 3.5.3** Develop a SET implementation `BinarySearchSET` by starting with the code for `BinarySearchST` and eliminating all of the code involving values.
- 3.5.4** Develop classes `HashSTint` and `HashSTdouble` for maintaining sets of keys of primitive `int` and `double` types, respectively. (Convert generics to primitive types in the code of `LinearProbingHashST`.)
- 3.5.5** Develop classes `STint` and `STdouble` for maintaining ordered symbol tables where keys are primitive `int` and `double` types, respectively. (Convert generics to primitive types in the code of `RedBlackBST`.) Test your solution with a version of `SparseVector` as a client.
- 3.5.6** Develop classes `HashSETint` and `HashSETdouble` for maintaining sets of keys of primitive `int` and `double` types, respectively. (Eliminate code involving values in your solution to EXERCISE 3.5.4.)
- 3.5.7** Develop classes `SETint` and `SETdouble` for maintaining ordered sets of keys of primitive `int` and `double` types, respectively. (Eliminate code involving values in your solution to EXERCISE 3.5.5.)
- 3.5.8** Modify `LinearProbingHashST` to keep duplicate keys in the table. Return any value associated with the given key for `get()`, and remove all items in the table that have keys equal to the given key for `delete()`.
- 3.5.9** Modify `BST` to keep duplicate keys in the tree. Return any value associated with the given key for `get()`, and remove all nodes in the tree that have keys equal to the given key for `delete()`.
- 3.5.10** Modify `RedBlackBST` to keep duplicate keys in the tree. Return any value associated with the given key for `get()`, and remove all nodes in the tree that have keys equal to the given key for `delete()`.

EXERCISES *(continued)*

- 3.5.11** Develop a `Multiset` class that is like `SET`, but allows equal keys and thus implements a mathematical multiset.
- 3.5.12** Modify `LookupCSV` to associate with each key all values that appear in a key-value pair with that key in the input (not just the most recent, as in the associative-array abstraction).
- 3.5.13** Modify `LookupCSV` to make a program `RangeLookupCSV` that takes two key values from the standard input and prints all key-value pairs in the `.csv` file such that the key falls within the range specified.
- 3.5.14** Develop and test a static method `invert()` that takes as argument an `ST<String, Bag<String>>` and produces as return value the inverse of the given symbol table (a symbol table of the same type).
- 3.5.15** Write a program that takes a string on standard input and an integer k as command-line argument and puts on standard output a sorted list of the k -grams (substrings of length k) found in the string, each followed by its index in the string.
- 3.5.16** Add a method `sum()` to `SparseVector` that takes a `SparseVector` as argument and returns a `SparseVector` that is the term-by-term sum of this vector and the argument vector. Note: You need `delete()` (and special attention to precision) to handle the case where an entry becomes 0.

CREATIVE PROBLEMS

3.5.17 *Finite mathematical sets.* Your goal is to develop an implementation of the following API for processing finite mathematical sets:

<code>public class MathSET<Key></code>	
<code> MathSET(Key[] universe)</code>	<i>create the empty set (using given universe)</i>
<code> void add(Key key)</code>	<i>put key into the set</i>
<code> MathSET<Key> complement()</code>	<i>set of keys in the universe that are not in this set</i>
<code> void union(MathSET<Key> a)</code>	<i>put any keys from a into the set that are not already there</i>
<code> void intersection(MathSET<Key> a)</code>	<i>remove any keys from this set that are not in a</i>
<code> void delete(Key key)</code>	<i>remove key from the set</i>
<code> boolean contains(Key key)</code>	<i>is key in the set?</i>
<code> boolean isEmpty()</code>	<i>is the set empty?</i>
<code> int size()</code>	<i>number of keys in the set</i>

API for a basic finite set data type

3.5.18 *Multisets.* After referring to EXERCISES 3.5.2 and 3.5.3 and the previous exercise, develop APIs `MultiHashSet` and `MultiSET` for multisets (sets that can have equal keys) and implementations `SeparateChainingMultiSET` and `BinarySearchMultiSET` for multisets and ordered multisets, respectively.

3.5.19 *Equal keys in symbol tables.* Consider the API `MultiST` (unordered or ordered) to be the same as our symbol-table APIs defined on page 363 and page 366, but with equal keys allowed, so that the semantics of `get()` is to return any value associated with the given key, and we add a new method

```
Iterable<Value> getAll(Key key)
```

CREATIVE PROBLEMS *(continued)*

that returns all values associated with the given key. Using our code for `SeparateChainingHashST` and `BinarySearchST` as a starting point, develop implementations `BinarySearchMultiST` and `SeparateChainingMultiST` for these APIs.

3.5.20 *Concordance.* Write an ST client `Concordance` that puts on standard output a concordance of the strings in the standard input stream (see page 498).

3.5.21 *Inverted concordance.* Write a program `InvertedConcordance` that takes a concordance on standard input and puts the original string on standard output stream. Note: This computation is associated with a famous story having to do with the Dead Sea Scrolls. The team that discovered the original tablets enforced a secrecy rule that essentially resulted in their making public only a concordance. After a while, other researchers figured out how to invert the concordance, and the full text was eventually made public.

3.5.22 *Fully indexed CSV.* Implement an ST client `FullLookupCSV` that builds an array of ST objects (one for each field), with a test client that allows the user to specify the key and value fields in each query.

3.5.23 *Sparse matrices.* Develop an API and an implementation for sparse 2D matrices. Support matrix addition and matrix multiplication. Include constructors for row and column vectors.

3.5.24 *Non-overlapping interval search.* Given a list of non-overlapping intervals of items, write a function that takes an item as argument and determines in which, if any, interval that item lies. For example, if the items are integers and the intervals are 1643–2033, 5532–7643, 8999–10332, 5666653–5669321, then the query point 9122 lies in the third interval and 8122 lies in no interval.

3.5.25 *Registrar scheduling.* The registrar at a prominent northeastern University recently scheduled an instructor to teach two different classes at the same exact time. Help the registrar prevent future mistakes by describing a method to check for such conflicts. For simplicity, assume all classes run for 50 minutes starting at 9:00, 10:00, 11:00, 1:00, 2:00, or 3:00.

3.5.26 *LRU cache.* Create a data structure that supports the following operations: access and remove. The access operation inserts the item onto the data structure if it's not already present. The remove operation deletes and returns the item that was least

recently accessed. *Hint*: Maintain the items in order of access in a doubly linked list, along with pointers to the first and last nodes. Use a symbol table with keys = items, values = location in linked list. When you access an element, delete it from the linked list and reinsert it at the beginning. When you remove an element, delete it from the end and remove it from the symbol table.

3.5.27 *List*. Develop an implementation of the following API:

```
public class List<Item> implements Iterable<Item>
```

List()	<i>create a list</i>
void addFront(Item item)	<i>add item to the front</i>
void addBack(Item item)	<i>add item to the back</i>
Item deleteFront()	<i>remove from the front</i>
Item deleteBack()	<i>remove from the back</i>
void delete(Item item)	<i>remove item from the list</i>
void add(int i, Item item)	<i>add item as the ith in the list</i>
Item delete(int i)	<i>remove the ith item from the list</i>
boolean contains(Item item)	<i>is item in the list?</i>
boolean isEmpty()	<i>is the list empty?</i>
int size()	<i>number of items in the list</i>

API for a list data type

Hint: Use two symbol tables, one to find the *i*th item in the list efficiently, and the other to efficiently search by item. (Java's `java.util.List` interface contains methods like these but does not supply any implementation that efficiently supports all operations.)

3.5.28 *UniQueue*. Create a data type that is a queue, except that an element may only be inserted the queue once. Use an existence symbol table to keep track of all elements that have ever been inserted and ignore requests to re-insert such items.

CREATIVE PROBLEMS *(continued)*

3.5.29 *Symbol table with random access.* Create a data type that supports inserting a key-value pair, searching for a key and returning the associated value, and deleting and returning a random key. *Hint:* Combine a symbol table and a randomized queue (see EXERCISE 1.3.35).

EXPERIMENTS

3.5.30 *Duplicates (revisited)*. Redo EXERCISE 2.5.31 using the Dedup filter given on page 490. Compare the running times of the two approaches. Then use Dedup to run the experiments for $N = 10^7$, 10^8 , and 10^9 , repeat the experiments for random long values and discuss the results.

3.5.31 *Spell checker*. With the file dictionary.txt from the booksite as command-line argument, the BlackFilter client described on page 491 prints all misspelled words in a text file taken from standard input. Compare the performance of RedBlackBST, SeparateChainingHashST, and LinearProbingHashST for the file WarAndPeace.txt (available on the booksite) with this client and discuss the results.

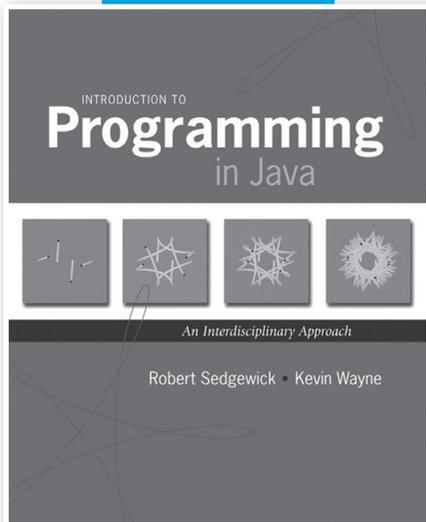
3.5.32 *Dictionary*. Study the performance of a client like LookupCSV in a scenario where performance matters. Specifically, design a query-generation scenario instead of taking commands from standard input, and run performance tests for large inputs and large numbers of queries.

3.5.33 *Indexing*. Study a client like LookupIndex in a scenario where performance matters. Specifically, design a query-generation scenario instead of taking commands from standard input, and run performance tests for large inputs and large numbers of queries.

3.5.34 *Sparse vector*. Run experiments to compare the performance of matrix-vector multiplication using SparseVector to the standard implementation using arrays.

3.5.35 *Primitive types*. Evaluate the utility of using primitive types for Integer and Double values, for LinearProbingHashST and RedBlackBST. How much space and time are saved, for large numbers of searches in large tables?

This page intentionally left blank



Introduction to Programming in Java: An Interdisciplinary Approach

Robert Sedgewick/Kevin Wayne
©2008 • 736 pp • ISBN: 0-321-49805-4

Introduction to Programming in Java takes an interdisciplinary approach to teaching programming with the Java programming language.

Features

This book thoroughly covers the field and is ideal for introductory programming courses. It can also be used for courses that integrate programming with mathematics, science, or engineering.

Students learn basic computer science concepts in the context of interesting applications in science, engineering, and commercial computing, leveraging familiar science and math while preparing students to use computers effectively in later courses. This serves to demonstrate that computation is not merely a tool, but an integral part of the modern world that pervades scientific inquiry and commercial development.

The book takes an “objects in the middle” approach where students learn basic control structures and functions, then how to use, create, and design classes.

A full programming model includes standard libraries for input, graphics, sound, and image processing that students can immediately put to use.

An integrated Companion Website features extensive Java coding examples, additional exercises, and Web links.

Instructors, contact your Pearson representative to receive an exam copy, or email PearsonEd.CS@Pearson.com.



REGISTER



THIS PRODUCT

informit.com/register

Register the Addison-Wesley, Exam Cram, Prentice Hall, Que, and Sams products you own to unlock great benefits.

To begin the registration process, simply go to **informit.com/register** to sign in or create an account.

You will then be prompted to enter the 10- or 13-digit ISBN that appears on the back cover of your product.

Registering your products can unlock the following benefits:

- Access to supplemental content, including bonus chapters, source code, or project files.
- A coupon to be used on your next purchase.

Registration benefits vary by product. Benefits will be listed on your Account page under Registered Products.

About InformIT — THE TRUSTED TECHNOLOGY LEARNING SOURCE

INFORMIT IS HOME TO THE LEADING TECHNOLOGY PUBLISHING IMPRINTS Addison-Wesley Professional, Cisco Press, Exam Cram, IBM Press, Prentice Hall Professional, Que, and Sams. Here you will gain access to quality and trusted content and resources from the authors, creators, innovators, and leaders of technology. Whether you're looking for a book on a new technology, a helpful article, timely newsletters, or access to the Safari Books Online digital library, InformIT has a solution for you.

informIT.com

THE TRUSTED TECHNOLOGY LEARNING SOURCE

Addison-Wesley | Cisco Press | Exam Cram
IBM Press | Que | Prentice Hall | Sams

SAFARI BOOKS ONLINE

PEARSON

InformIT is a brand of Pearson and the online presence for the world's leading technology publishers. It's your source for reliable and qualified content and knowledge, providing access to the top brands, authors, and contributors from the tech community.

↕ Addison-Wesley

Cisco Press

EXAM/CRAM

IBM Press

QUE

PRENTICE HALL

SAMS

Safari

LearnIT at InformIT

Looking for a book, eBook, or training video on a new technology? Seeking timely and relevant information and tutorials? Looking for expert opinions, advice, and tips? **InformIT has the solution.**

- Learn about new releases and special promotions by subscribing to a wide variety of newsletters. Visit informit.com/newsletters.
- Access FREE podcasts from experts at informit.com/podcasts.
- Read the latest author articles and sample chapters at informit.com/articles.
- Access thousands of books and videos in the Safari Books Online digital library at safari.informit.com.
- Get tips from expert blogs at informit.com/blogs.

Visit informit.com/learn to discover all the ways you can access the hottest technology content.

Are You Part of the IT Crowd?

Connect with Pearson authors and editors via RSS feeds, Facebook, Twitter, YouTube, and more! Visit informit.com/socialconnect.



Try Safari Books Online FREE

Get online access to 5,000+ Books and Videos



Safari[®]
Books Online

FREE TRIAL—GET STARTED TODAY!
www.informit.com/safaritrial



Find trusted answers, fast

Only Safari lets you search across thousands of best-selling books from the top technology publishers, including Addison-Wesley Professional, Cisco Press, O'Reilly, Prentice Hall, Que, and Sams.



Master the latest tools and techniques

In addition to gaining access to an incredible inventory of technical books, Safari's extensive collection of video tutorials lets you learn from the leading video training experts.

WAIT, THERE'S MORE!



Keep your competitive edge

With Rough Cuts, get access to the developing manuscript and be among the first to learn the newest technologies.



Stay current with emerging technologies

Short Cuts and Quick Reference Sheets are short, concise, focused content created to get you up-to-speed quickly on new and cutting-edge technologies.



Adobe Press



Cisco Press



IBM Press



Microsoft Press



O'REILLY



que



SAMS

