Métodos de pesquisa e ordenação

João Luiz Pereira Marciano, MSc.

Sumário

Métodos de pesquisa

A pesquisa computacional de dados (entendida aqui como a busca por valores armazenados em memória) apresenta várias facetas, em que se deve levar em conta os tipos de dados tratados, a forma de armazenamento e os recursos computacionais envolvidos.

Duas formas básicas de pesquisa são a linear ou seqüencial, e a binária. A pesquisa linear busca exaustivamente pelo elemento procurado, e pode ser representada da seguinte forma:

```
Função PesquisaSequencial(Li,Ls,Chave:Inteiro):Inteiro;
var i,j:inteiro;

    começo
      j:=-1;
      para i:=Li até Ls faça
          se a[i] = Chave então
                  começo
                    j := i;
                    pare;
                  fim;
      PesquisaSequencial:=j;
    fim;
```

Nesta função, assume-se os dados dispostos num arranjo (array) de inteiros, com limite inferior igual a Li e limite superior igual a Ls. Observe-se que Li e Ls não necessitam ser os limites inferior e superior de todo o array, mas apenas de um subarranjo onde se deseje pesquisar.

Já a pesquisa binária pode se apresentar da seguinte forma:

```
Função PesquisaBinaria(Li,Ls,Chave:Inteiro):Inteiro;
var j,m:inteiro;

    começo

      Enquanto Verdadeiro faça
            começo
              m := (Li + Ls) div 2;
              if Chave < a[m] então
                 Ls := m - 1
              senão se Chave > a[m] então
                 Li := m + 1
              senão
                 começo
                   PesquisaBinaria := m;
                   pare;
                 fim;
              se Li > Ls então
                 começo
                   PesquisaBinaria := -1;
                   pare;
                 fim;
            fim;
    fim;
```

A pesquisa binária subdivide o arranjo em duas partes, e restringe a busca em apenas uma destas, dependendo do valor da chave ser menor ou maior que o valor alocado na subdivisão. Este método é muito mais poderoso que a pesquisa linear: num conjunto de 1024 elementos, pode-se determinar se o elemento pesquisado pertence ou não ao conjunto em no máximo 10 comparações. Uma restrição se aplica: os dados devem estar ordenados. Isto leva à necessidade de métodos de ordenação de dados eficientes.

Existem várias formas e modalidades de ordenação (*sort*) de dados, que podem ser escolhidas conforme a necessidade. Se o número de itens a ordenar não é muito grande (e "grande" já se torna um conceito quase que subjetivo, sendo dependente da máquina e dos recursos computacionais disponíveis), será muito mais fácil empregar um método simples de ordenação.

Via de regra, os métodos elementares de ordenação aqui apresentados tomam N^2 passos para ordenar N itens arranjados de maneira aleatória. Se N é suficientemente pequeno, isto não será problema, mesmo porque, caso os itens estejam de alguma forma pré-ordenados, alguns dos métodos mais simples poderão ser até mais rápidos que os métodos mais sofisticados. Contudo, estes métodos básicos não devem ser tomados como modelos para aplicações de âmbito geral, uma vez que são mais custosos, computacionalmente falando.

Nos modelos apresentados, considerar-se-á que se deseja realizar a chamada ordenação interna, ou seja, a ordenação de dados em memória primária, notadamente em vetores, em contraposição à ordenação externa, que envolve dados em memória secundária. Assim, a quantidade de memória a ser utilizada pelo método de ordenação pode se tornar preponderante.

Uma outra característica dos métodos de ordenação que pode se mostrar desejável é a estabilidade: um método de ordenação é dito estável se preserva a ordem relativa das chaves iguais no arranjo inicial. Por exemplo, caso se queira ordenar uma lista alfabética de alunos pela nota, um método estável produzirá uma lista na qual os estudantes com a mesma nota estarão ordenados alfabeticamente, mas um método não estável produzirá uma lista sem vestígios da ordenação inicial. A maioria dos métodos simples é estável, mas muitos dos métodos mais sofisticados não o são.

O exemplo a seguir, que ordena três registros de dados (na realidade, três valores inteiros), mostra as convenções que serão utilizadas neste texto. Em particular, o programa realiza a ordenação de exatos $N = 3$ registros, caso muito peculiar, mas que servirá para ilustrar os nossos propósitos.

```
Programa OrdenaTres;
constante   maxN = 3;
var         a:vetor[1..maxN] de inteiro;
            N,i:inteiro;

Procedimento Sort3;
var       t:inteiro;
```

```
        começo
          se a[1] > a[2] então
             começo
               t:=a[1];
               a[1]:=a[2];
               a[2]:=t;
             fim;
          se a[1] > a[3] então
             começo
               t:=a[1];
               a[1]:=a[3];
               a[3]:=t;
             fim;
          se a[2] > a[3] então
             começo
               t:=a[2];
               a[2]:=a[3];
               a[3]:=t;
             fim;
fim;

{================================================================}
{              P r o g r a m a     p r i n c i p a l             }
{================================================================}

começo
     para i:=1 até maxN faça
          começo
            escreva('Forneca o valor ',i,': ');
            leia(a[i]);
          fim;

     Sort3;

     para i:=1 até maxN faça
         escreva(a[i]);

fim.
```

As três atribuições que se seguem a cada comando se implementam operações de "troca" ("*swap*"). Estas trocas devem, naturalmente, refletir o tipo de dado em tratamento, e devem ser escritas criteriosamente, de modo a otimizar a performance do algoritmo como um todo.

Estas instruções, neste exemplo, ilustram a idéia de "comparar dois registros e trocar suas posições, se necessário, de modo a fazer com que o menor deles assuma a menor posição". Isto se refletirá em todos os métodos a seguir.

Ordenação por seleção

Um dos algoritmos de ordenação mais simples é este: encontre o menor elemento do arranjo e troque sua posição com o primeiro elemento, então encontre o segundo menor elemento e coloque-o na segunda posição do arranjo, e assim por diante, até que todo o arranjo esteja ordenado. Este método é chamado método de seleção, porque atua pela repetida "seleção" do menor elemento restante. O programa

a seguir é uma implementação deste método. Para cada i de 1 a ***N***-1, ele troca o mínimo elemento em a[i..***N***] por a[i]:

```
Programa Ordena_por_Selecao;

Constante   N = 100;

var         a:vetor[1..N] de inteiro;
            i:inteiro;


Procedimento Selecao;
var i,j,min,t:inteiro;

    começo

      para i:=1 até N-1 faça
          começo

            min:=i;
            para j:=i+1 até N faça
                se a[j] < a[min] então min:=j;

            t:=a[min];
            a[min]:=a[i];
            a[i]:=t;

          fim;
fim;

{=================================================================}
{               P r o g r a m a     p r i n c i p a l             }
{=================================================================}

começo

     { inicializa o gerador de nos. aleatorios }

     randomize;

     { preenche o vetor com N numeros aleatorios }

     para i:=1 até N faça
            a[i] := random(N);

     Selecao;

     para i:=1 até N faça
         escreva(a[i]);

fim.
```

Enquanto o indexador i percorre o arranjo da esquerda para a direita, os elementos à esquerda do indexador são colocados em suas posições definitivas no arranjo, de tal forma que o arranjo estará totalmente ordenado ao final do percorrimento.

Ordenação por inserção

Um algoritmo quase tão simples quanto o da seleção, mas mais flexível, é o da inserção. Este método é o utilizado para ordenar um conjunto de cartas de baralho: considere os elementos, um de cada vez, inserindo cada um em seu devido lugar entre os já considerados (e mantendo-os ordenados). O elemento em consideração é inserido meramente movendo-se os elementos maiores que ele uma posição para a direita, e então inserindo-se o elemento desejado na posição vaga criada. Este processo é implementado pelo programa a seguir: para cada i de 2 até N, o subarranjo a[1..i] é ordenado colocando-se a[i] em sua respectiva posição entre os elementos no subarranjo à esquerda.

```
Programa Ordena_por_Insercao;

Constante   N = 100;

var         a:vetor[1..N] de inteiro;
            i:inteiro;

Procedimento Insercao;
var i,j,v:inteiro;

    começo

      para i:=2 até N faça
          começo

            v:=a[i];
            j:=i;
            enquanto (j > 1) e (a[j-1] > v) faça
                  começo
                    a[j]:=a[j-1];
                    j:=j-1;
                  fim;
            a[j]:=v;
          fim;
fim;

{=================================================================}
{               P r o g r a m a     p r i n c i p a l             }
{=================================================================}

começo

     { inicializa o gerador de nos. aleatorios }

     randomize;

     { preenche o vetor com N numeros aleatorios }

     para i:=1 até N faça
            a[i] := random(N);

     Insercao;

     para i:=1 até N faça
         escreva(a[i]);
fim.
```

Como no caso da ordenação por seleção, os elementos à esquerda do indexador i estão em ordem durante o percorrimento, mas ainda não em suas posições finais, e assim têm que ser movidos para abrir espaço para elementos menores encontrados após i. Contudo, o arranjo estará totalmente ordenado ao final do percorrimento.

Bubblesort

O sort da bolha tem um princípio de funcionamento bastante simples: percorra o arranjo, trocando elementos adjacentes de posição, se necessário. Quando não for necessária mais nenhuma troca, o arranjo estará ordenado. Uma implementação é dada a seguir:

```
Programa Bubble_Sort;

Constante   N = 100;
var         a:vetor[1..N] de inteiro;
            i:inteiro;

Procedimento Bolha;
var i,j,t:inteiro;

    começo

      para i:=N até 1 passo -1 faça
          para j:=2 até i faça
              se a[j-1] > a[j] então
                 começo
                   t:=a[j-1];
                   a[j-1]:=a[j];
                   a[j]:=t;
                 fim;
fim;

{=================================================================}
{               P r o g r a m a     p r i n c i p a l             }
{=================================================================}

começo

     { inicializa o gerador de nos. aleatorios }

     randomize;

     { preenche o vetor com N numeros aleatorios }

     para i:=1 até N faça
            a[i] := random(N);

     Bolha;

     para i:=1 até N faça
         escreva(a[i]);

fim.
```

Uma outra forma de visualizar o funcionamento deste algoritmo é a seguinte: observe que sempre que o maior elemento é encontrado no primeiro passo, ele é trocado com cada um dos elementos à sua direita, até atingir a posição final do arranjo. No segundo passo, o segundo maior elemento será levado, da mesma forma, até a sua posição, etc. O sort da bolha opera como um tipo de sort por seleção, embora realize mais trabalho para levar cada elemento à sua posição devida.

Algumas propriedades dos métodos elementares de ordenação

Em [Sedgewick 86], mostra-se as seguintes propriedades:

Prop1. O sort por seleção usa cerca de $N^2/2$ comparações e realiza N trocas.
Prop2. O sort por inserção usa cerca de $N^2/4$ comparações e realiza $N^2/8$ trocas no caso médio, o dobro no pior caso.
Prop3. O sort da bolha usa cerca de $N^2/2$ comparações e $N^2/2$ trocas no caso médio e no pior caso.
Prop4. O sort por inserção é linear para arranjos "quase ordenados".
Prop5. O sort por seleção é linear para arranjos com registros grandes e chaves pequenas.

Shellsort

O sort por inserção é lento porque troca apenas elementos adjacentes. Por exemplo, se o menor elemento estiver no final do arranjo, são necessários N passos para colocá-lo em seu devido lugar. O shellsort é uma extensão simples do sort por inserção, que ganha velocidade por permitir a troca de elementos distantes entre si.

A idéia é arranjar os dados de tal forma que, tomando-se seu h-ésimo elemento (a partir de qualquer posição), tem-se um arranjo ordenado, dito h-ordenado. Falando de outra forma, um arranjo h-ordenado é composto de h arranjos ordenados, tomados juntos. Através da h-ordenação para valores grandes de h, pode-se mover elementos dentro do arranjo por grandes distâncias, e assim ordenar mais facilmente para valores menores que h. Usando-se este procedimento para qualquer seqüência de valores que termine no tamanho 1 produzirá um arranjo totalmente ordenado.

Uma maneira de implementar o shellsort é, para cada h, usar o sort de inserção independentemente em cada um dos h subarranjos. A escolha dos valores de h pode ser feita pelo critério de Knuth [Niemann 97], como se segue:

Faça $h_1=1$, $h_{s+1} = 3h_s +1$, e pare com h_t quando $h_{t+2} \geq N$.

Assim, os valores de h são computados como se segue:

$h_1 = 1$
$h_2 = (3 * 1) + 1 = 4$
$h_3 = (3 * 4) + 1 = 13$
$h_4 = (3 * 13) + 1 = 40$

$h_5 = (3 * 40) + 1 = 121$
$h_6 = (3 * 121) + 1 = 364$, etc.

Uma propriedade muito importante [Sedgewick 86] é a seguinte:

Prop6. Shellsort nunca realiza mais do que $N^{3/2}$ comparações (para os incrementos 1, 4, 13, ...)

Uma implementação para o shellsort é a seguinte:

```
Programa ShellSort;
Constante   N = 100;

var         a:vetor[1..N] de inteiro;
            i:inteiro;

Procedimento Shell;
var i,j,h,v:inteiro;

    começo

      { determinacao do valor de h }

      h:=1;

      repita

        h := 3 * h + 1;

      até h > N;

      repita

        h:= h div 3;

        para i:=h+1 até N faça
            começo
              v:=a[i];
              j:=i;

              enquanto a[j-h] > v faça
                    começo

                      a[j] := a[j-h];
                      j:=j-h;

                      se j<=h então pare;

                    fim;

              a[j]:=v;
            fim;
      até h=1;
fim;

{=================================================================}
{              P r o g r a m a     p r i n c i p a l              }
{=================================================================}
```

```
começo

     { inicializa o gerador de nos. aleatorios }

     randomize;

     { preenche o vetor com numeros N aleatorios }

     para i:=1 até N faça
            a[i] := random(N);

     Shell;

     para i:=1 até N faça
         escreva(a[i]);

fim.
```

A instrução `pare` causa o encerramento imediato do loop em execução, levando a execução do programa para a instrução seguinte à que ocasionou o loop (no caso, após o fim do enquanto).

Quicksort

Quicksort é provavelmente o tipo de sort mais utilizado, desde a sua criação por C.A.R. Hoare, em 1960. Quicksort é de implementação simples e de uso geral, além de apresentar a seguinte propriedade:

Prop7. Quicksort requer N log N operações, em média, para ordenar N itens.

As limitações são que o quicksort é recursivo, o que dificulta sua implementação quando não se dispõe de recursão, e necessita de N^2 operações no pior caso.

Quicksort baseia-se no princípio de "dividir e conquistar": particiona-se o arranjo em duas partes, e então ordena-se as partes independentemente. Como se verá, a posição exata da partição pode variar, e assim o algoritmo toma a seguinte expressão geral:

```
Procedimento Quicksort(e,d:inteiro);
var i:inteiro;
começo
    if d > e então
        começo
            i:=particao(e,d);
            quicksort(e,i-1);
            quicksort(i+1,d);
        fim;
fim;
```

Os parâmetros e (esquerda) e d (direita) delimitam o subarranjo dentro do arranjo original a ser ordenado. A chamada quicksort(1,N) ordena o arranjo inteiro.

O ponto crucial do algoritmo se torna, assim, a partição, que deve manipular o arranjo de tal modo que as três condições seguintes se apliquem:

- o elemento a[i] está em na posição final do arranjo para algum i;
- todos os elementos em a[l], ..., a[i-1] são menores ou iguais a a[i];
- todos os elementos em a[i+1], ... , a[d] são maiores ou iguais a a[i].

Isto pode ser obtido procedendo-se da seguinte forma: primeiro, arbitrariamente escolha a[d] como sendo o elemento que irá para sua posição final. Depois, percorra da extremidade esquerda para a direita no arranjo até que um elemento maior que a[d] seja encontrado, e percorra da extremidade direita para a esquerda até que um elemento menor que a[d] seja encontrado. Os dois elementos que encerrarem estes percorrimentos estão obviamente fora de suas posições no arranjo, então troque suas posições. Continuando-se assim, assegura-se que todos os elementos no arranjo à esquerda do indexador da esquerda são menores que a[d], e que todos os elementos no arranjo à direita do indexador da direita são maiores que a[d]. Quando os indexadores se cruzam, o processo de particionamento está quase completo: tudo o que resta é trocar a[d] com o elemento mais à esquerda do subarranjo à direita (o elemento apontado pelo indexador da esquerda).

Eis uma implementação completa do quicksort:

```
Programa QuickSort;
Constante   N = 100;

var         a:vetor[1..N] de inteiro;
            i:inteiro;

Procedimento Quick(e,d:inteiro);
var i,j,v,t:inteiro;

    começo

      se d > e então
         começo

           v:=a[d];
           i:=e-1;
           j:=d;

           repita

             repita

               i:=i+1;

             até a[i] >=v;

             repita

               j:=j-1;
```

```
            até a[j] <= v;

            t:=a[i];
            a[i]:=a[j];
            a[j]:=t;

          até j <= i;

          a[j]:=a[i];
          a[i]:=a[d];
          a[d]:=t;

          quick(e,i-1);
          quick(i+1,d);

        fim;

fim;

{=================================================================}
{              P r o g r a m a     p r i n c i p a l              }
{=================================================================}

começo

     { inicializa o gerador de nos. aleatorios }

     randomize;

     { preenche o vetor com N numeros aleatorios }

     para i:=1 até N faça
            a[i] := random(N);

     Quick(1,N);

     para i:=1 até N faça
         escreva(a[i]);

fim.
```

Estruturas de árvore

É bastante comum o emprego da recursão para proceder-se à definição de tipos de dados. Desta forma, pode-se definir uma lista, por exemplo, como se segue:

Uma lista com tipo básico T corresponde a uma das seguintes situações:

1. A lista vazia.
2. A concatenação (cadeia) de um elemento de tipo T com uma seqüência já existente, cujo tipo básico também seja T.

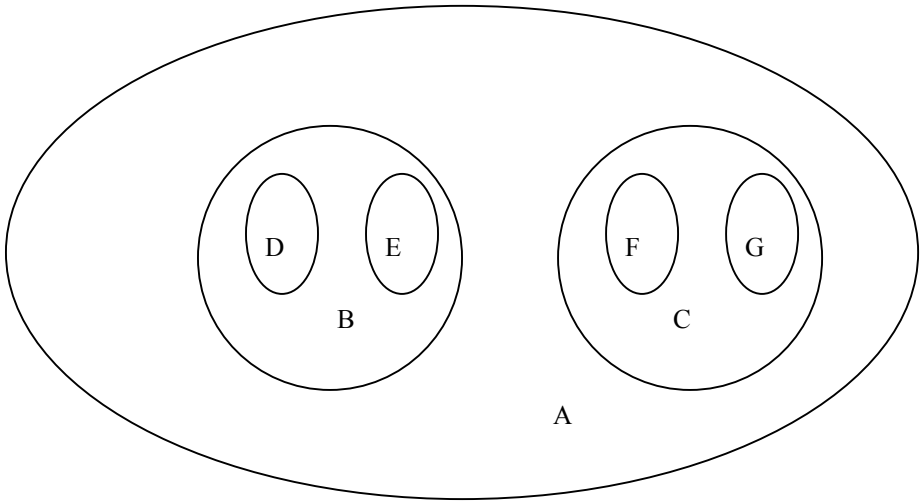
A recursão pode ainda ser utilizada na definição de estruturas muito mais sofisticadas, das quais a estrutura de árvore é um exemplo bastante conhecido. Assim, pode-se definir uma estrutura de árvore da seguinte forma:

Uma estrutura de árvore com tipo básico T pode ser definida recursivamente conforme uma das duas situações abaixo:

1. A estrutura vazia.
2. Um nó de tipo T, associado a um número finito de estruturas disjuntas de árvore, de mesmo tipo base T, denominadas *sub-árvores*.

Observando-se a similaridade das definições recursivas de listas e estruturas de árvore, torna-se evidente que a lista pode ser considerada como uma estrutura de árvore na qual cada nó tem no máximo uma única sub-árvore. Por esta razão, a lista é muitas vezes também denominada *árvore degenerada*.

Existem várias maneiras de representar uma estrutura de árvore. Por exemplo, uma estrutura de árvore cujo tipo básico T tem como domínio o conjunto de letras é mostrada em várias notações na figura 1. Todas estas representações mostram a e mesma estrutura, sendo, portanto, equivalentes. É a representação segundo um grafo que ilustra explicitamente as relações de ramificação exibidas pela estrutura. Por razões óbvias, esta representação gráfica motivou historicamente a adoção, para esta estrutura, do nome *árvore*. É também usual a representação de árvores "de cabeça para baixo", com a raiz na parte inferior.

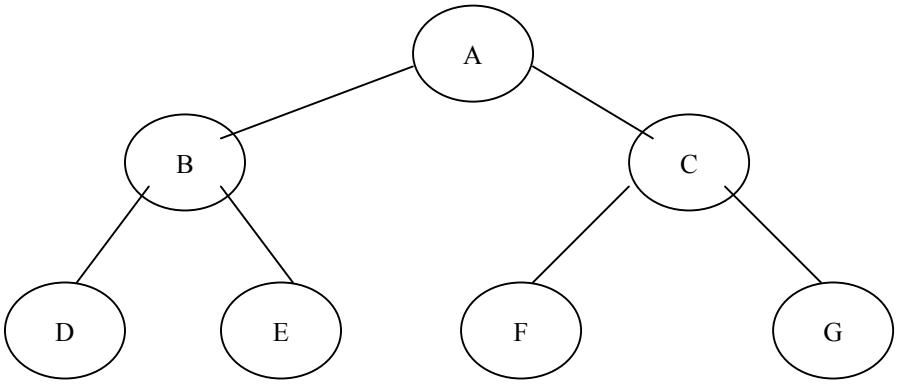


(a)

(A (B (D, E)), (C (F, G))) (b)

A
B
D
E
C
F
G

(c)



(d)

Figura 1 – Representação de uma estrutura de árvore: (a) conjuntos aninhados; (b) lista com parênteses aninhados; (c) denteação ou indentamento; (d) grafo.

Algumas definições:

Def 1. Uma *árvore ordenada* é uma árvore na qual os ramos de cada nó são ordenados. Portanto, as duas árvores ordenadas mostradas na Figura 2 são distintas, formando objetos diferentes.

Def 2. Um nó y que está diretamente abaixo de um nó x é chamado *descendente* (direto) de x; estando x em um nível i, então y é considerado pertencente ao nível i+1.

Def 3. Reciprocamente, o nó x é denominado *ancestral* (direto) de y. A raiz de uma árvore é definida como pertencente ao nível 0.

Def 4. O nível máximo a que pertence algum elemento qualquer de uma árvore determina a *profundidade* ou *altura* da árvore.

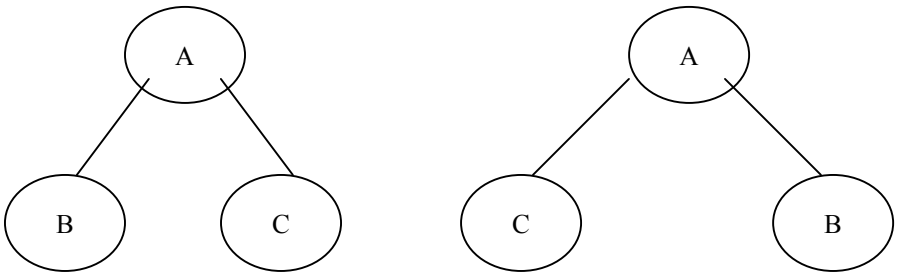


Figura 2: Duas árvores binárias distintas

Def 5. Se um elemento não apresenta descendentes, é denominado nó terminal ou folha da árvore, ao passo que todo elemento que não for terminal é denominado nó interior.

Def 6. O número de descendentes (diretos) de um nó interior é denominado grau deste nó. Assim, o grau máximo atingido pelos nós da árvore determina o grau desta árvore.

Def 7. O número de ramos ou arestas que devem ser percorridos para que, partindo-se da raiz, seja atingido o nó x é denominado comprimento de trajeto do nó x. A raiz possui comprimento de trajeto igual a zero, seus descendentes diretos possuem comprimento de trajeto igual a 1, e assim por diante. Em geral, um nó pertencente ao nível i possui o comprimento de trajeto igual a i.

Def 8. As árvores ordenadas de grau 2, denominadas *árvores binárias*, possuem uma importância particular. Define-se uma árvore binária como sendo um conjunto finito de elementos (nós) que podem ou ser vazios ou consistir em uma raiz (nó) com duas árvores binárias disjuntas associadas, denominadas, respectivamente, sub-árvore esquerda e sub-árvore direita da raiz.

Exemplos de árvores binárias são: árvore genealógica (invertida); chaves de um torneio de tênis; uma expressão aritmética com operadores diádicos, na qual cada operador faz o papel de um nó da árvore, representando-se seus operadores como sub-árvores.

Def 9. O comprimento da trajetória de uma árvore é definido como sendo a soma dos comprimentos dos trajetos de todos os seus componentes, e é também denominado *comprimento interno de trajeto,* ou *comprimento da trajetória interna*. O comprimento interno de trajeto da árvore mostrada na Figura 1, por exemplo, é igual a 10. O comprimento interno de trajeto de uma árvore qualquer pode ser calculado pela expressão:

$Cit = \sum n_i * i$, onde n_i é o número de nós pertencentes ao nível i.

O número máximo de nós em uma árvore de altura *h* e grau *d* é atingido quando todos os nós possuem d sub-árvores, exceto os de nível h. Para uma árvore de grau *d*, o nível 0 contém então um nó (a raiz), o nível 1 contém seus *d* descendentes, o nível 2 contém os d^2 descendentes dos *d* nós do nível 1, etc. Disto resulta

$$N_d(h) = \sum_{i=0}^{h} d^i$$

como o número máximo de nós para uma árvore de altura *h* e grau *d*. Para *d* = 2, obtém-se

$$N_2(h) = 2^{h+1} - 1$$

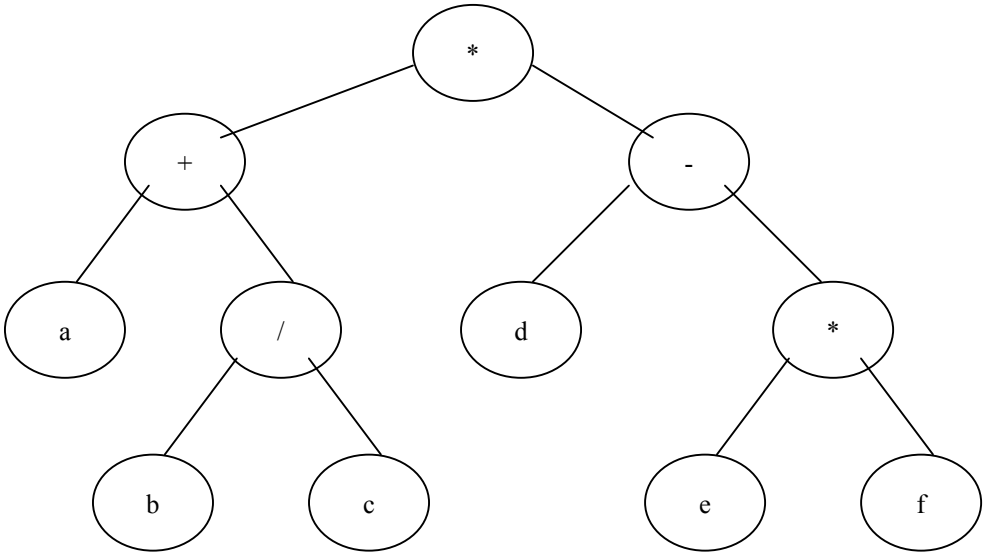


Figura 3: representação em árvore da expressão (a + b/c) * (d – e*f)

Para a implementação computacional de tais estruturas, é natural que sejam usados os apontadores providos pelas linguagens. Desta forma, define-se os nós como sendo variáveis com uma estrutura fixa, na qual o grau da árvore determina o número de apontadores que referenciam as sub-árvores dos nós. A referência à árvore vazia é denotada por NULO. Assim, a árvore da Figura 3 consiste em componentes de um tipo definido conforme se segue:

```
Tipo Ptr = ponteiro para Nó;
Tipo Nó = Registro
              Op:caracter;
              Esquerda, direita: Ptr;
          Fim;
```

Pretende-se distribuir uniformemente os nós pela árvore, de forma tal que a árvore resultante tenha altura mínima.

A regra de distribuição uniforme para um número n conhecido de nós pode ser expressa através de uma formulação recursiva:

1. Usar um nó para raiz;
2. Gerar a sub-árvore esquerda com nl = n DIV 2 nós de acordo com estas mesmas regras;
3. Gerar a sub-árvore direita com nr = n – nl nós de acordo com estas regras.

Estas regras podem ser expressas por meio de um procedimento recursivo, como parte do programa demonstrado a seguir, que lê o arquivo de entrada e constrói uma árvore perfeitamente balanceada.

Def 10. Uma árvore é *perfeitamente balanceada* se, para cada nó, os números de sub-árvores das suas sub-árvores esquerda e direita diferem, no máximo, de uma unidade.

```
Programa ConstroiArvore;

Tipo Ptr = ponteiro para Nó;

Tipo Nó = registro
                Chave:caracter;
                Esquerda,direita:Ptr;
            End;

Var N:Inteiro;
    Raiz:Ptr;

Função Árvore (n:inteiro):Ptr;
Var novonó: Ptr;
    x:caracter;
    nesq,ndir:inteiro;

começo
  se n = 0
  então novonó = NULO
  senão começo
            nesq := n DIV 2;
            ndir := n – nesq – 1;
            leia(x);
            aloca(novonó,tamanho(Nó));
            novonó->chave := x;
            novonó->esquerda := arvore(nesq);
            novonó->direita := arvore(ndir);
        fim:
  árvore := novonó;
fim;
```

```
Procedimento ImprimeArvore (t:ptr; h:inteiro);
Var i:inteiro;
Começo
    Se t <> NULO
   Então
     começo
          ImprimeArvore(t->esquerda,h+1);
          para i:= 1 até h faça escreva("  ");
          escreva(t->chave);escreva(nl);
          ImprimeArvore(t->direita,h+1);
     fim;
fim;

{ programa principal }

começo
  leia(n);
  raiz := Árvore(n);
  ImprimeArvore(raiz,0);
 Fim;
```

Por exemplo, os seguintes dados deveriam ser fornecidos para que se obtenha a árvore exibida na Figura 1 (d):

7 A B D E C F G

Operações básicas sobre árvores binárias

Há muitas tarefas que se podem executar sobre uma estrutura de árvore: uma bastante comum consiste em aplicar uma dada operação P sobre cada um dos elementos da árvore. A operação P pode ser vista como parâmetro de uma tarefa mais geral, a de percorrer (visitar) todos os nós da árvore, operação esta usualmente chamada *varredura da árvore*. Se a tarefa for encarada como um processo seqüencial único, então os nós individuais serão visitados em alguma ordem específica, podendo ser considerados como se estivessem dispostos em um arranjo linear. De fato, a descrição de muitos algoritmos é bastante facilitada se for possível raciocinar com base em uma ordenação predefinida, o que permite estabelecer o conceito de "próximo elemento" para cada um dos elementos da árvore. Há três tipos principais de ordem que surgem naturalmente do conceito da estrutura da árvore. Como a própria estrutura, estas ordens são convenientemente expressas em termos recursivos. Em relação à árvore da Figura 4, em que R denota a raiz e E e D respectivamente as sub-árvores esquerda e direita, os três tipos de ordenações são:

1. Pré-ordem: R, E,D (visitar a raiz antes da sub-árvores)
2. In-ordem: E, R, D (visitar a raiz entre as visitas à sub-árvores)
3. Pós-ordem: E, D, R (visitar a raiz após visitar as sub-árvores)

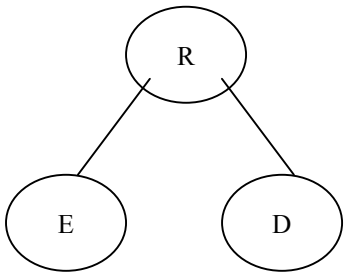


Figura 4. Árvore binária

Percorrendo-se a árvore da Figura 3 e registrando-se as seqüência de saída dos caracteres encontrados nos nós visitados, são obtidas as seguintes ordenações:

1. Pré-ordem: * + a / b c – d * e f
2. In-ordem: a + b / c * d – e * f
3. Pós-ordem: a b c / + d e f * - *

Da varredura em pré-ordem da árvore da expressão resulta a notação pré-fixa ou pré-fixada; a pós-ordem gera a notação pós-fixa ou pós-fixada, e a varredura in-ordem gera a notação in-fixa ou in-fixada.

Estes meios de varredura são demonstrados pelos três algoritmos a seguir, nos quais o parâmetro t denota a árvore a ser tratada, e o procedimento P denota a operação a ser realizada em cada nó. Sejam as seguintes definições:

```
Tipo Ptr = Ponteiro para Nó;
Tipo Nó = Registro ...
              esquerda, direita: Ptr;
          fim;
```

Define-se, então, os seguintes procedimentos:

```
Procedimento preordem(t:Ptr);
Começo
   se t <> NULO então
      começo
         P(t);
         preordem(t->esquerda);
         preordem(t->direita);
      fim;
fim;

Procedimento inordem(t:Ptr);
Começo
   se t <> NULO então
      começo
         inordem(t->esquerda);
         P(t);
```

```
            inordem(t->direita);
        fim;
fim;

Procedimento posordem(t:Ptr);
Começo
    se t <> NULO então
        começo
            posordem(t->esquerda);
            posordem(t->direita);
            P(t);
        fim;
fim;
```

Comentário: As árvores binárias freqüentemente são utilizadas para representar um conjunto de dados cujos elementos devam ser posteriormente acessíveis por intermédio de uma chave única. Se a árvore estiver organizada de tal forma que, para cada nó t_i, todas as chaves da sub-árvore esquerda de t_i sejam menores que a chave de t_i, enquanto as pertencentes à sub-árvore direita sejam maiores que a chave de t_i, então esta árvore pode ser denominada árvore de busca. Em uma árvore de busca, é possível localizar uma sub-árvore arbitrária partindo-se da raiz e prosseguindo-se ao longo de uma trajetória de busca, decidindo-se pelo desvio para a sub-árvore da esquerda ou da direita com base apenas no valor da chave do nó. n elementos podem ser organizados em uma árvore binária de altura igual a log n. Portanto, uma busca entre n elementos pode ser realizada com apenas log n comparações, se a árvore estiver perfeitamente balanceada. Assim, a árvore é uma forma de organização muito mais adequada do que uma lista linear. Quanto à busca, como ela segue uma trajetória única a partir da raiz até o nó desejado, ela pode ser programada pela seguinte iteração

```
Procedimento localiza(x:inteiro;t:Ptr):Ptr;
começo
    enquanto (t <> NULO) e (t->chave <> x) faça
        Se t->chave < x
        então t := t->direita
        senão t:= t->esquerda;
    retorne t;
fim;
```

Buscas e inserções em árvores

Freqüentemente, pode ser necessário realizar inserções e/ou remoções nos dados em tratamento. Uma vez mais, a estrutura de árvore se mostra bastante aplicável, principalmente se comparada com os vetores.

Inicialmente, será considerado o caso das inserções, sem a ocorrência de remoções. Um exemplo típico é a contagem das palavras em um texto. O problema consiste em, dada uma seqüência de números, determinar o número de ocorrências de

cada um deles. Isto significa que, iniciando-se com a árvore vazia, será efetuada uma busca de cada número na árvore. Se o número for encontrado, o seu contador de ocorrências será incrementado; caso contrário, o número será inserido e caracterizado como um novo número (com seu contador de ocorrências igual a 1). A esta tarefa dá-se o nome de *busca em árvore com inserção*. Seja a seguinte definição de dados:

```
Tipo NoPtr = Ponteiro para Nó;
     No =  registro
                    valor: inteiro;
                    contador:inteiro;
                    esquerda, direita: NoPtr;
                fim;
```

Admitindo-se, além disso um programa responsável pelo fornecimento de chaves e uma variável que indique a raiz da árvore onde deve ser feita a busca, pode-se elaborar um programa como segue:

```
Leia(x);
Enquanto (x <> "") faça
     Pesquisa(x,raiz);
     Leia(x);
Fim;
```

Uma vez mais, para esta situação é possível a obtenção de uma trajetória que leve ao elemento desejado. Porém, se a busca conduzir ao final de um ramo da árvore (ou seja, a sub-árvore vazia, designada por um apontador com valor NULO), então o número desejado deverá ser inserido na árvore na posição antes ocupada pela sub-árvore vazia.

A operação completa é demonstrada no pseudo-código a seguir. O procedimento de busca é formulado como um procedimento recursivo. Observe-se que o ponteiro p é um parâmetro passado por referência, e não por valor. Isto é essencial, já que, no caso da inserção, um novo valor do apontador deverá ser atribuído à variável que antes continha o valor NULO.

Fazendo-se uso da seguinte seqüência de entrada:

7 15 10 20 3 12 22 18

o programa produzirá a árvore binária de busca mostrada na figura 5.

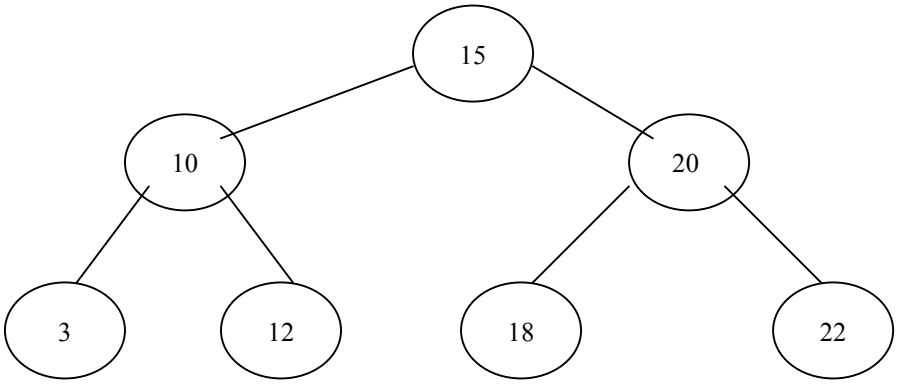


Figura 5: árvore binária de busca

```
Tipo NoPtr = Ponteiro para No;
     No =  registro
                  valor: inteiro;
                  contador:inteiro;
                  esquerda, direita: NoPtr;
               fim;

Var raiz:NoPtr;
i,N:inteiro;
Valor:inteiro;

Procedimento ImprimeArvore(t:NoPtr, h:inteiro);
Var i:inteiro;
Começo
   Se t <> NULO então
    Começo
       ImprimeArvore(t->esquerda,h+1);
       Para i:=1 até h faça escreva(“ ”);
        Escreva(t->valor,t->contador);escreva(nl);
       ImprimeArvore(t->direita,h+1);
    Fim;
Fim;


Procedimento Pesquisa(x:inteiro, var p:NoPtr);
Começo
    Se p = NULO então
     Começo
         Aloca(p, tamanho(No));
         p->valor := x;
         p->contador := 1;
        p->esquerda := NULO;
        p->direita := NULO;
     fim
  senão se x < p->valor
```

```
            então pesquisa(x,p->esquerda)
            senão se x > p->valor
                  então pesquisa(x,p->direita)
                  senão p->contador := p->contador + 1;
  fim;


 { Programa principal }

Começo
    Raiz := NULO;
    Leia (n);                { o primeiro número indica a quantidade de nós }
    Para i:=1 até N faça
       Começo
          Escreva("Forneça o no",i);
          Leia(valor);
          Pesquisa(valor,raiz);
        Fim;
    ImprimeArvore(raiz,0);
Fim;
```

Apesar do propósito deste algoritmo ser o de efetuar buscas, ele pode ser facilmente adaptado para executar operações de ordenação. De fato, ele se assemelha muito ao algoritmo de ordenação pelo método de inserção e, devido ao uso da estrutura de árvore em vez de vetor, torna-se desnecessária a relocação dos componentes para cima do ponto de inserção. A ordenação da árvore pode ser programada de tal modo que se torne tão eficiente quanto o melhor dos métodos conhecidos para ordenação de vetores. Algumas precauções, no entanto, devem ser tomadas. No caso de replicação, após a localização do elemento desejado, o novo elemento deverá ser também inserido. Se o caso x = p->chave for tratado de maneira idêntica ao caso x > p->chave, então o algoritmo representa um método estável de ordenação, ou seja, um método em que elementos de chaves idênticas são localizados, ao se percorrer a árvore, na mesma ordem em que se deu sua inserção.

Remoções em árvores

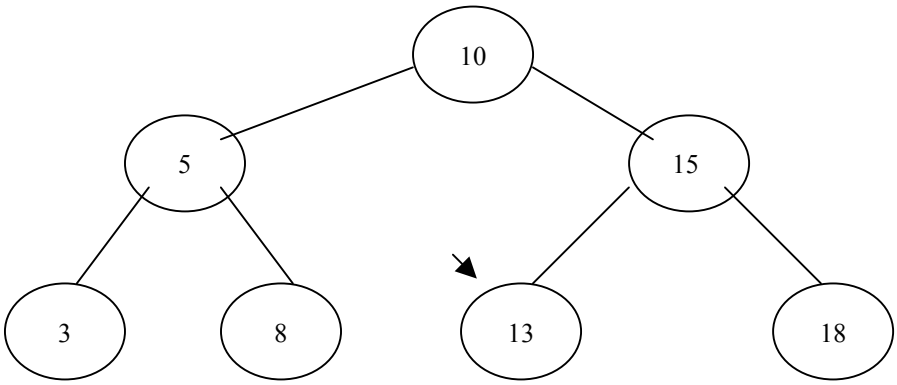
A tarefa de remoção consiste em definir um algoritmo para a remoção, isto é, para a eliminação de um nó, com chave x, pertencente a uma árvore com chaves ordenadas. Infelizmente, a remoção de um elemento não é tão simples quanto a operação de inserção – é trivial se o elemento a ser removido for um nó terminal ou tiver um único descendente. A dificuldade maior reside na remoção de um elemento com dois descendentes. Nesta situação, o elemento removido deve ser substituído ou pelo elemento mais à direita de sua sub-árvore esquerda ou pelo nó mais à esquerda de sua sub-árvore direita. Ambos possuem no máximo um só descendente. O procedimento de remoção discrimina três casos distintos:

Não existe nenhum elemento com valor igual a x;
O elemento com valor igual a x possui no máximo um só descendente;
O elemento com valor igual a x possui dois descendentes.

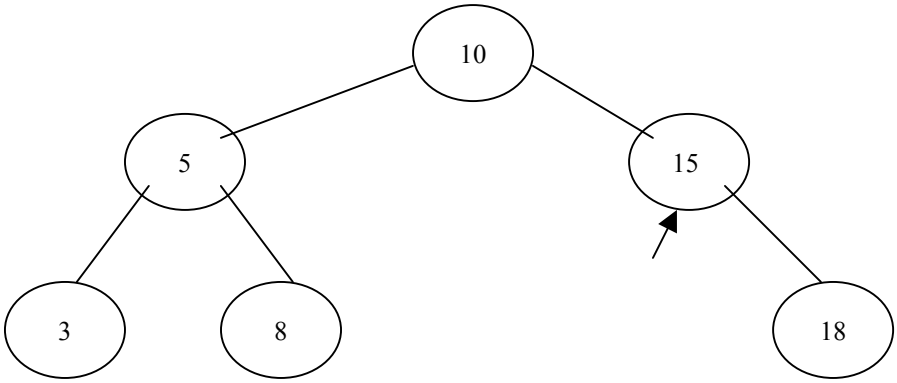
```
Procedimento delete(x:vetor[1..10] de caracter, var p:NoPtr);
Var q: NoPtr;

Procedimento Del(var r:NoPtr);
Começo
  Se r->direita <> NULO
  Então Del(r->direita)
  Senão
     Começo
        q->valor := r->valor;
        q->contador := r->contador;
        q := r;
        r := r->esquerda;
      fim;
fim; { Del }

começo { delete }
    se p = NULO
    então escreva("o no não se encontra na arvore")
    senão se x < p->valor
            então delete(x,p->esquerda)
            senão se x > p->valor
                    então delete(x,p->direita)
                    senão
                       começo { remover p }
                          q := p;
                          se q->direita = NULO
                          então p := q->esquerda
                          senão se q->esquerda = NULO
                                  então p := q->direita
                                  senão Del(q->esquerda);
                          { desaloca q }
                        fim;
fim { delete };
```

O procedimento auxiliar Del é ativado somente no terceiro caso. Este procedimento percorre um caminho na árvore, descendo ao longo do ramo mais à direita da sub-árvore à esquerda do elemento q a ser removido, e então substitui a informação (valor e contador) em q pelos valores correspondentes do componente mais à direita r daquela sub-árvore esquerda. Após esta operação, q pode ser descartado.

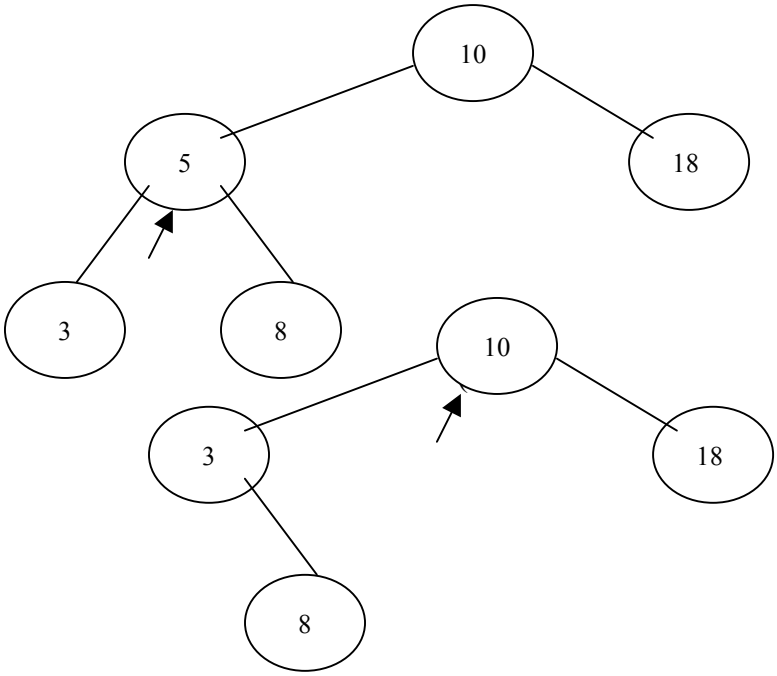
Para mostrar o funcionamento dos procedimentos acima, observe-se a figura 6. Seja inicialmente a árvore (a); removem-se, então, sucessivamente, os nós com chaves 13, 15, 5 e 10. As árvores resultantes são mostradas na figura 6 (b-e).



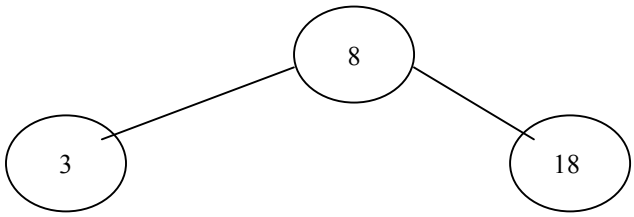
(a)



(b)



(d)



(e)

Figura 6 – Árvores resultantes da remoção de nós indicados

Árvores balanceadas

Pelo que se viu anteriormente, nota-se que procedimentos de inserção que sempre mantenham o balanceamento da árvore , conforme a definição vista, podem ser pouco vantajosos, em alguns casos. Algumas melhorias possíveis dependem da formulação de definições de balanceamento com base em condições menos rigorosas. Tal critério "imperfeito" de balanceamento conduziria a procedimentos mais simples de reorganização da arvore, acarretando apenas uma pequena deterioração no desempenho médio do processo de busca. Uma destas definições de balanceamento foi apresentada por Adelson-Velskii e Landis (AVL), e seu critério é o seguinte:

Def. AVL: Uma árvore é dita balanceada se e somente se para qualquer nó, as alturas de suas duas sub-árvores diferem de no máximo uma unidade.

As árvores que satisfazem esta condição serão doravante denominadas árvores balanceadas, e deve-se notar que árvores perfeitamente balanceadas são também balanceadas.

Esta definição conduz a um processo de rebalanceamento relativamente simples, com comprimento médio da trajetória de busca praticamente idêntico ao das árvores perfeitamente balanceadas. As seguintes operações podem ser realizadas em árvores balanceadas com processamento da ordem de log n unidades de tempo, ou seja, O(log n):

1. localizar um nó, dada uma chave;
2. inserir um nó, dada uma chave;
3. remover o nó cuja chave seja igual a uma chave dada.

Estas afirmações são conseqüências diretas de um teorema provado por Adelson-Velskii e Landis.

Inserção em árvores balanceadas

Será investigado o efeito da inserção de um novo nó em uma árvore balanceada. Dada uma raiz r com sub-árvores esquerda (E) e direita (D), três casos

podem ser identificados. Suponha-se que o novo nó seja inserido em E, acarretando o aumento de uma unidade em sua altura, podem ocorrer as seguintes situações antes e depois da inserção:

1. se $h_E = h_D$: E e D adquirem alturas diferentes, mas o critério de balanceamento não é violado.
2. se $h_E < h_D$: E e D assumem alturas iguais, isto é, houve mesmo uma melhoria no balanceamento da árvore.
3. se $h_E > h_D$: o critério de balanceamento foi violado, e a árvore deve ser reconstruída.

Considere-se a árvore da figura 7. Nós com chaves 9 ou 11 podem ser inseridos sem rebalanceamento. A árvore com raiz 10 ficará com ramos apenas em um dos lados (caso 1), e a árvore com raiz 8 terá seu balanceamento melhorado (caso 2). A inserção dos nós 1,3,5 ou 7, entretanto, exigirá um rebalanceamento subseqüente.

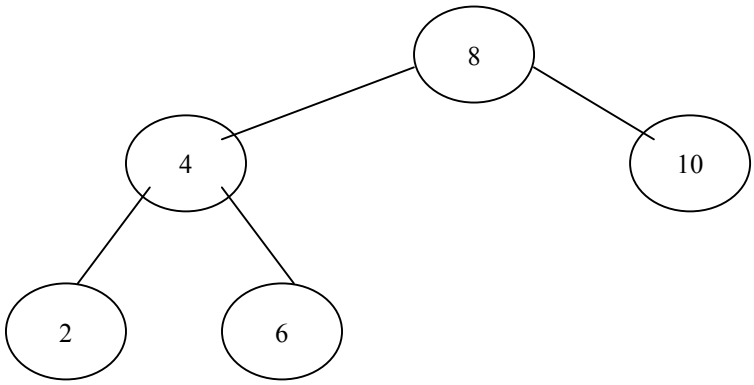


Figura 7 – Árvore balanceada

Um algoritmo para o rebalanceamento é extremamente dependente do modo como são registradas as informações acerca do balanceamento da árvore. Uma solução de caráter extremo consiste em manter tais informações completamente implícitas na própria estrutura da árvore. Neste caso, porém, o fator de balanceamento de um nó deverá ser recalculado toda vez que a árvore for submetida a uma operação de inserção, o que resultaria em um esforço computacional excessivo. Outra solução extrema utiliza um fator de balanceamento explicitamente armazenado para cada nó. A definição deste tipo de dado é então estendida para

```
Tipo NoPtr = Ponteiro para Nó;

   Nó =  registro
                valor: inteiro;
                contador:inteiro;
                esquerda, direita: NoPtr;
                bal:inteiro;
             fim;
```

O fator de balanceamento de um nó será aqui interpretado como sendo a diferença entre a altura das suas sub-árvores direita e esquerda. O processo de inserção de nó consiste essencialmente nas três partes consecutivas seguintes:

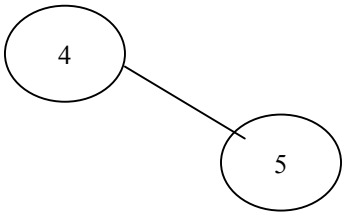
1. seguir a trajetória de busca até que seja constatado que a chave ainda não se encontra na árvore;
2. neste caso, inserir o nó e determinar o fator de balanceamento resultante desta inserção;
3. refazer o tratamento ao longo de todo o trajeto, testando o fator de balanceamento de cada nó, rebalanceando a árvore se for necessário.

Embora este método envolva alguns testes redundantes (uma vez atingido o balanceamento, não é necessário testar os ancestrais do nó), este esquema será adotado inicialmente por ser facilmente implementável através de uma extensão do procedimento de busca e inserção já analisado. Este procedimento descreve as operações necessárias em cada nó isolado, sendo, por sua formulação recursiva, facilmente adaptado para incorporar uma operação adicional a ser executada no caminho de volta ao longo do trajeto de busca. A cada passo, é necessário descobrir e informar ao procedimento se a altura da sub-árvore em que foi feita a inserção aumentou ou não. Para isto, na lista de parâmetros do procedimento será incluída a variável booleana h, que indicará que a sub-árvore cresceu em altura. Naturalmente, este parâmetro será passado por referência, uma vez que a variável a que se refere é utilizada para informar um resultado.

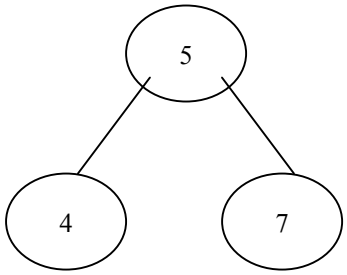
Suponha-se, agora, que o processo retorne, partindo do ramo esquerdo a um dado nó apontado por p, com a indicação de que houve um incremento em sua altura. Deve ser feita a distinção entre as três condições relacionadas com a altura da sub-árvore antes da inserção:

1. se $h_E < h_D$, p->bal = 1, o desbalanceamento em p foi equilibrado;
2. se $h_E = h_D$,p->bal = 0, e agora “o peso” da árvore é maior à esquerda;
3. se $h_E > h_D$, p->bal = –1, é necessário fazer o rebalanceamento da árvore.

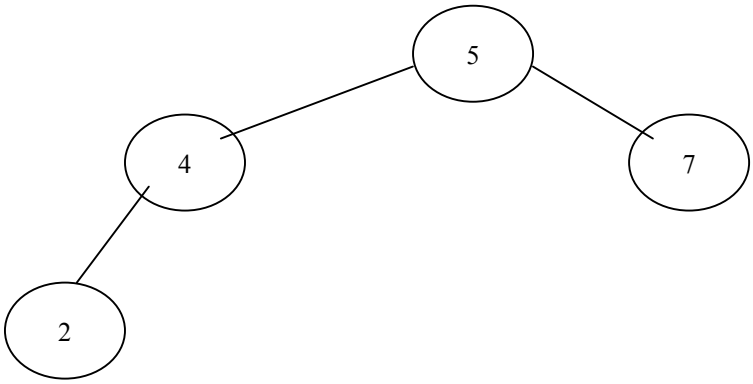
As operações necessárias de rebalanceamento são inteiramente expressas como uma seqüência de atribuições de valores aos apontadores. O princípio de funcionamento deste método está ilustrado na figura 8. Considere-se a árvore binária (a) que consiste apenas de dois nós. A inserção da chave 7 produz em primeiro lugar uma árvore desbalanceada, no caso, uma lista linear, que é balanceada gerando a árvore balanceada (b). As inserções dos valores 2 e 1 resultam em um desbalanceamento da árvore de raiz 4, que, ao ser balanceada, resulta na árvore (d). A inserção subseqüente da chave 3 viola o critério de balanceamento da árvore de raiz 5. O balanceamento requer uma operação mais complexa, que gera a árvore (e). Por fim, a inserção do nó 6 provoca a ocorrência de um novo rebalanceamento. A árvore final é (f).



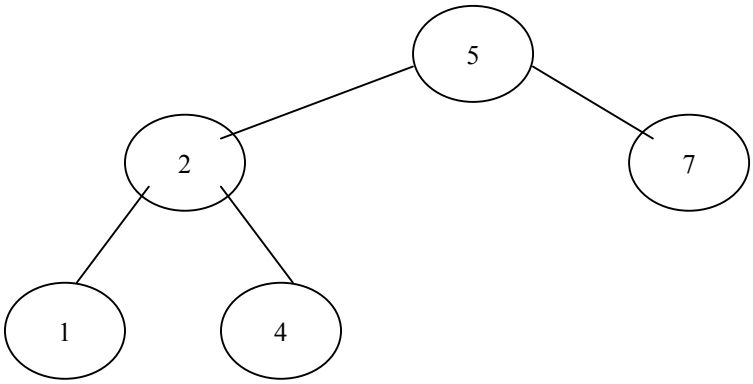
(a)



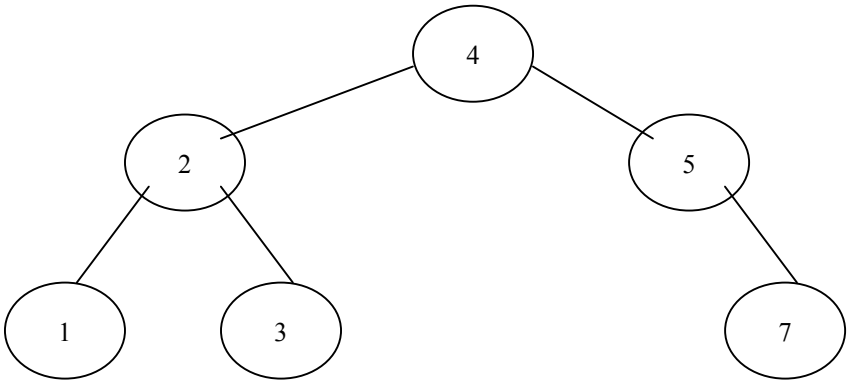
(b)



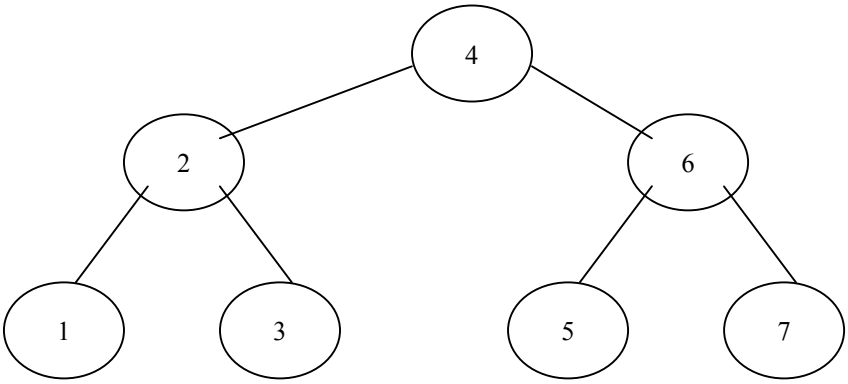
(c)



(d)



(e)



(f)

Figura 8.

```
Procedimento Pesquisa(x:inteiro,var p:Noptr,var h:booleano);
Var p1,p2:NoPtr;

Começo
   Se p = NULO então { inserir }
      começo
         aloca(p,tamanho(no));
         h:= VERDADEIRO;
         p->valor := x;
         p->contador := 1;
         esquerda:=NULO;
         direita:=NULO;
         bal:=0;
      fim
   senão se x < p->valor então
      começo
        pesquisa(x,p->esquerda,h);
        se h então { o ramo esquerdo cresceu }
              caso (p->bal) de
```

```
                1: começo
                     p->bal = 0;
                     h := FALSO;
                   fim;
                0: começo
                     p->bal := -1;
                   fim;
               -1: começo       { rebalanceamento }
                     p1 := p->esquerda;
                     se p1->bal = -1 então     { rotação simples à esquerda }
                       começo
                         p->esquerda := p1->direita;
                         p1->direita:=p;
                         p->bal := 0;
                         p->p1;
                        fim
                     senão começo  { rotação dupla esquerda-direita }
                         p2 := p1->direita;
                         p1->direita := p2->esquerda;
                         p2->esquerda := p1;
                         p->esquerda := p2->direita;
                         p2->direita := p;
                         se p2->bal = -1
                           então p->bal := 1
                           senão p->bal :=0;
                         se p2->bal = 1
                            então p1->bal := -1
                            senão p1->bal := 0;
                         p := p2;
                     fim;
                     p->bal := 0;
                     h := FALSO;
                   fim;
    fim;
senão se x > p->valor então
    começo
        pesquisa(x,p->direita,h);
        se h então   { o ramo direito cresceu }
              caso p->bal de
                -1: começo
                      p->bal := 0;
                      h :=FALSO;
                    fim;
                 0: começo
                      p->bal := 1;
                    fim;
                 1:começo   { rebalanceamento }
                      p1 := p->direita;
                      se p1->bal = 1 então  { rotação simples à direita }
                        começo
```

```
                        p->direita := p1->esquerda;
                        p1->esquerda := p;
                        p->bal:=0;
                        p := p1;
                     fim
                  senão { rotação dupla direita-esquerda }
                     começo
                        p2 := p1->esquerda;
                        p1->esquerda := p2->direita;
                        p2->direita := p1;
                        p->direita := p2->esquerda;
                        p2->esquerda := p;
                        se p2->bal = 1
                           então p->bal = -1
                           senão p->bal := 0;
                        se p2->bal = -1
                           então p1->bal := 1
                           senão p1->bal := 0;
                        p := p2;
                     fim;
                  p->bal := 0;
                  h := FALSO;
               fim;
      fim;
  senão p->contador := p->contador + 1;
fim;
```

Duas perguntas relativas ao desempenho do algoritmo de inserção em árvores balanceadas são as seguintes:

1. se todas as n! permutações de n chaves ocorrerem com igual probabilidade, qual é a altura esperada de uma árvore balanceada construída com este algoritmo?
2. qual é a probabilidade de que uma inserção venha a requerer um rebalanceamento da árvore?

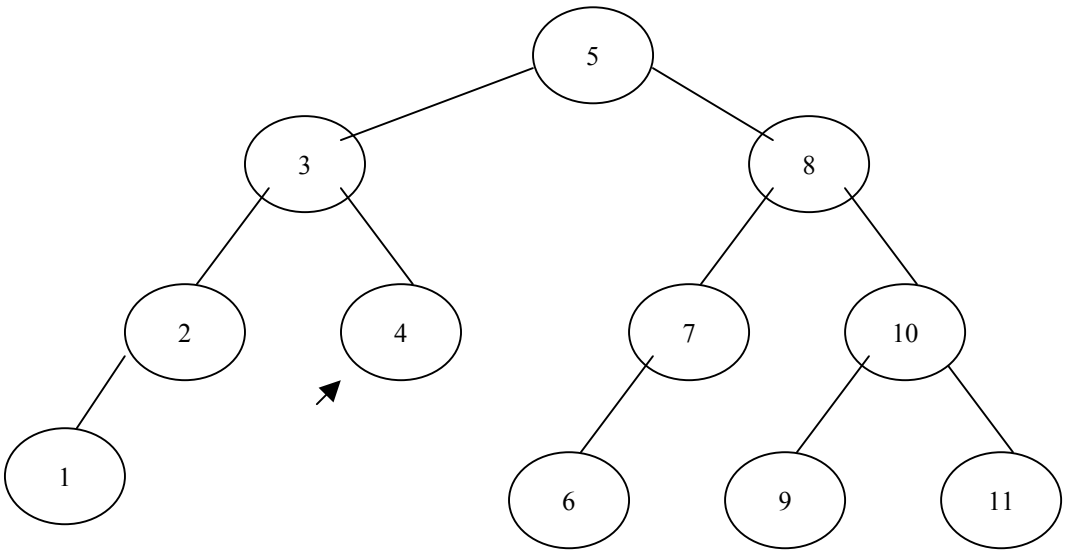
O problema da análise matemática deste algoritmo ainda está em aberto. Os testes empíricos sustentam a conjetura de que a altura esperada de uma árvore gerada por ele seja h = log(n) + 0,25. Evidências empíricas sugerem ainda que o rebalanceamento é necessário, em média, a cada duas inserções. No caso médio, as rotações simples e duplas são igualmente prováveis.

A complexidade das operações de rebalanceamento sugere que as árvores balanceadas devam ser utilizadas somente nos casos em que as operações de consulta às informações forem consideravelmente mais freqüentes do que as de inserção.

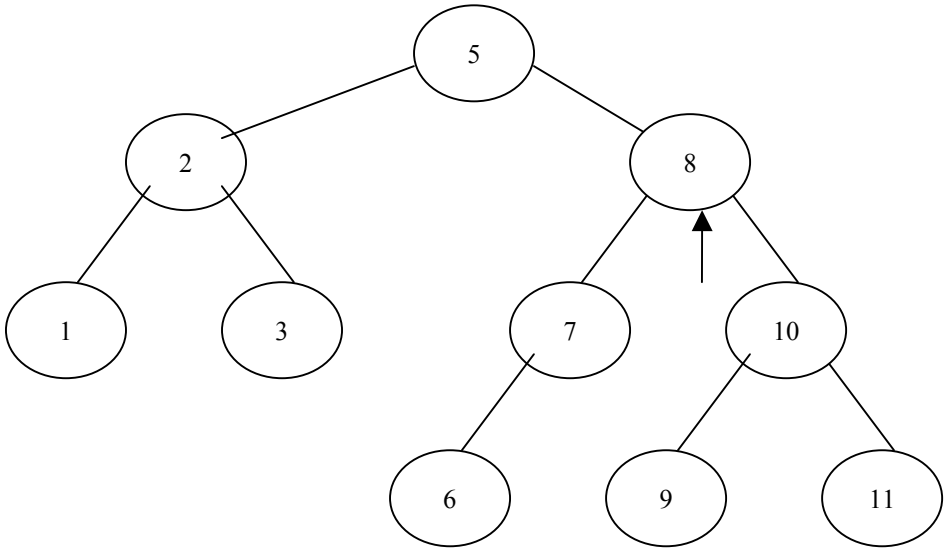
Remoções em árvores balanceadas

O algoritmo básico de remoção em árvore balanceada é mostrado a seguir. Os casos mais simples são os nós terminais e os nós com apenas um descendente. Se o nó a ser removido tiver duas sub-árvores, deve-se substituí-lo pelo nó mais à direita de sua sub-árvore esquerda. De maneira similar ao que ocorre na inserção, é acrescentado um parâmetro h, passado por referência e de tipo booleano, indicando que a altura da sub-árvore foi reduzida. O rebalanceamento deve ser efetuado somente quando o valor de h for verdadeiro, o que ocorre quando um nó é localizado e removido, ou se o próprio rebalanceamento reduzir a altura de alguma sub-árvore. O algoritmo exibido aplica os procedimentos balanceE e balanceD quando os ramos esquerdo e direito, respectivamente, tiverem sido reduzidos em sua altura.

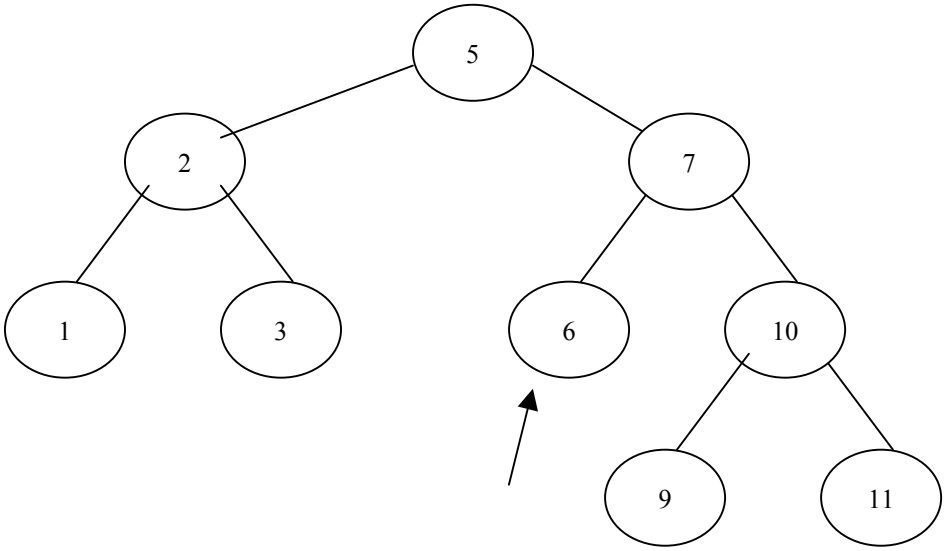
A figura 9 mostra a operação do algoritmo. Dada a árvore balanceada (a), as remoções sucessivas dos nós com chaves 4, 8, 6, 5, 2, 1 e 7 resultam nas árvores (b) a (h). A remoção do nó 4 é, em si mesma, simples, por se tratar de nó terminal. Mas isto acarreta o desbalanceamento do nó 3. A operação de rebalanceamento envolve uma rotação simples à esquerda. Novo rebalanceamento é necessário quando da remoção do nó 6. Desta vez, a sub-árvore direita da raiz 7 é rebalanceada por meio de uma rotação simples à direita. A remoção do nó 2, embora trivial em si por possuir apenas um descendente, exige uma complexa operação de rotação dupla direita-esquerda do nó 7.



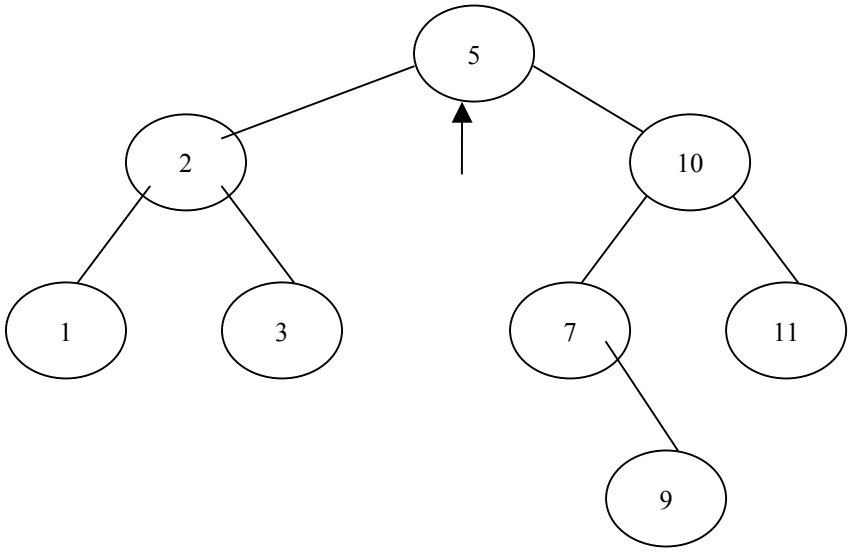
(a)



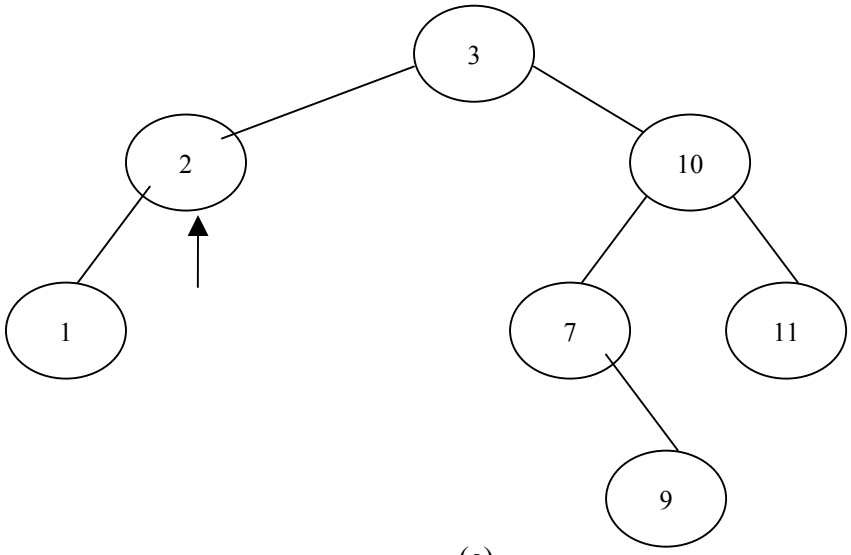
(b)



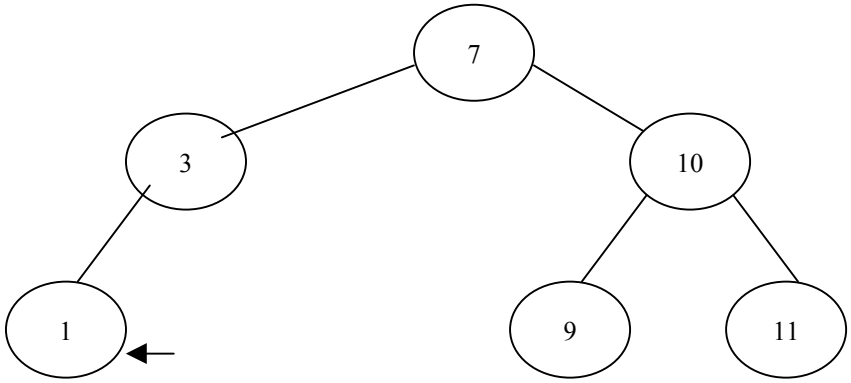
(c)



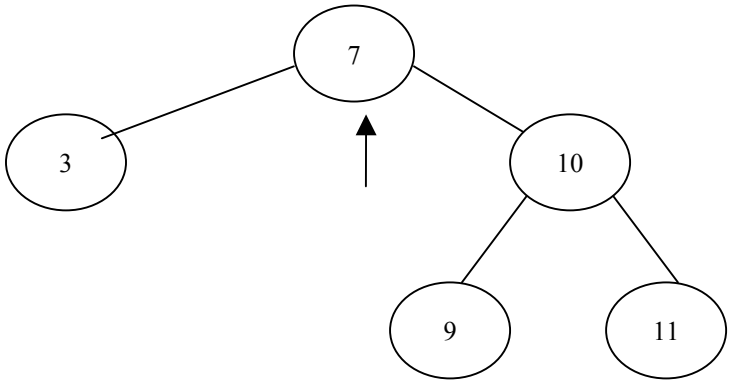
(d)



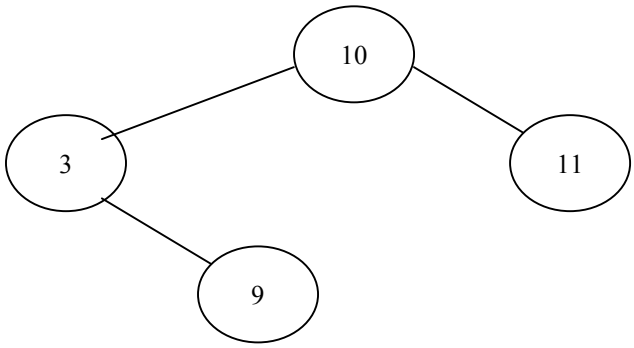
(e)



(f)



(g)



(h)

```
Procedimento balanceE(var p:NoPtr;var h:booleano);
Var p1,p2:Noptr;
B1,b2:inteiro;

Começo { o ramo esquerdo diminuiu }
   Caso p->bal de
       -1: começo
              p->bal := 0;
           fim;
        0: começo
              p->bal := 1;
              h := FALSO;
           fim;
        1: começo { rebalanceamento }
              p1 := p->direita;
              b1 := p1->bal;
              se b1 >= 0 então { rotação simples à direita }
                  começo
                      p->direita := p1->esquerda;
                      p1->esquerda := p;
```

```
                    se b1 = 0 então
                       começo
                          p->bal := 1;
                          p1->bal := -1;
                          h := FALSO;
                       fim;
                    senão
                       começo
                          p->bal := 0;
                          p1->bal := 0;
                       fim;
                    p = p1;
                 fim;
              senão { rotação dupla direita-esquerda }
                 começo
                    p2 := p1->esquerda;
                    b2 := p2->bal;
                    p1->esquerda := p2->direito;
                    p2->direita := p1;
                    p->direita := p2->esquerda;
                    p2->esquerda := p;
                   se b2 = 1
                      então p->bal := -1
                      senão p->bal := 0;
                   se b2 = -1
                      então p1->bal := 1
                      senão p1->bal := 0;
                   p := p2;
                   p2->bal := 0;
                 fim;
           fim; { rebalanceamento }
fim; { balanceE}


Procedimento balanceD(var p:Noptr;var h:booleano);
Var p1,p2:Noptr;
B1,b2:inteiro;

Começo { o ramo direito diminuiu }
   Caso p->bal de
         1: começo
               p->bal := 0;
            fim;
         0: começo
               p->bal := -1;
               h := FALSO;
            fim;
        -1: começo { rebalanceamento }
               p1 := p->esquerda;
               b1 := p1->bal;
```

```
                se b1 <= 0 então { rotação simples à esquerda }
                   começo
                      p->esquerda := p1->direita;
                      p1->direita := p;
                      se b1 = 0 então
                          começo
                             p->bal := -1;
                             p1->bal := 1;
                             h := FALSO;
                          fim;
                      senão
                          começo
                              p->bal := 0;
                              p1->bal := 0;
                          fim;
                      p := p1;
                   fim;
               senão { rotação dupla esquerda-direita }
                  começo
                     p2 := p1->direita;
                     b2 := p2->bal;
                     p1->direita := p2->esquerda;
                     p2->esquerda := p1;
                     p->esquerda := p2->direita;
                     p2->direita := p;
                     se b2 = -1
                        então p->bal := 1
                        senão p->bal := 0;
                     se b2 = 1
                        então p1->bal := -1
                        senão p1->bal := 0;
                     p := p2;
                     p2->bal := 0;
                  fim;
            fim;
fim; { balanceD }


Procedure Delete (x:inteiro;var p:NoPtr;var h:booleano);
Var q:NoPtr;

Procedure del(var r:NoPtr;var h:booleano);
Começo
   Se r->direita <> NULO
      Então começo
                Del(r->direita,h);
                Se h então balanceD(r,h);
             Fim;
      Senão começo
                q->valor = r->valor;
```

```
                    q->contador := r->contador;
                    q := r;
                     r := r->esquerda;
                    h := VERDADEIRO;
                fim;
fim;

Começo
   Se p = NULO
      Então { a chave não se encontra na árvore }
      Senão se x < p->valor
            Então começo
                     delete(x,p->esquerda,h);
                     se h então balanceE(p,h);
                  fim;
            senão se x > p->valor
                  então começo
                           delete(x,p->direita,h);
                           se h então balanceD(p,h);
                        fim;
                  senão { remover p }
                    começo
                      q := p;
                      se q->direita = NULO
                         então começo
                                  p := q->esquerda;
                                  h := VERDADEIRO;
                               fim;
                          senão se q->esquerda = NULO
                                  então começo
                                           q := q->direita;
                                           h := VERDADEIRO;
                                        fim;
                                  senão começo
                                           Del(q->esquerda,h);
                                           Se h então balanceE(p,h);
                                        fim;
                       fim;
{ desaloca q }
fim { delete };
```

A remoção de um elemento em uma árvore balanceada também pode ser feita, no pior caso, em O(log n) operações. Mas há uma grande diferença entre os procedimentos de inserção e remoção: enquanto a inserção de uma única chave pode resultar em uma rotação (de dois ou três nós no máximo), a remoção pode exigir uma rotação em cada um dos nós ao longo da trajetória de busca.

Hash

A principal questão discutida anteriormente, quanto se tratava de árvores, é a seguinte: dado um conjunto de elementos, caracterizados por uma chave (sobre a qual é definida uma relação de ordem entre os elementos), como este conjunto deve ser organizado de tal maneira que o acesso a um elemento, cuja chave é conhecida, seja efetuado da maneira mais econômica possível? Claramente, na memória de um computador, cada elemento é acessível, em última instância, através da especificação do seu endereço de memória. Assim, o problema colocado é, essencialmente, o de se determinar uma função H de mapeamento de chaves (C) em endereços (E):

$$H : C \rightarrow E$$

No caso das árvores e outras estruturas de dados, como listas, este mapeamento é implementado na forma de diversos algoritmos de pesquisa, baseados nas diferentes formas de organização dos dados. Apresenta-se agora mais um método, simples e em muitos casos bastante eficiente, mas do qual também são apresentadas algumas desvantagens.

A organização de dados utilizada nesta técnica é a estrutura de vetor. *H* é, portanto, um mapeamento que converte chaves em índices de vetor, razão pela qual a expressão "transformação de chaves" é utilizada para descrever a técnica a ser analisada.

A dificuldade fundamental do uso de transformação de chaves, de modo prático, consiste no fato de que o conjunto dos possíveis valores das chaves é muito maior que o conjunto de índices do vetor, ou de endereços de memória disponíveis. Considere-se o exemplo de nomes que consistem de até 16 letras, utilizados para identificar indivíduos em um conjunto de milhares de pessoas. Neste caso, há 26^{16} possíveis chaves, que devem ser mapeadas para a ordem de 10^3 possíveis índices. Observa-se, portanto, que a função *H* é claramente uma função que mapeia diversos elementos em um único endereço. Dada uma chave *c*, o primeiro passo em uma operação de acesso (busca) consiste em calcular seu índice associado *h = H(c)*. O segundo passo, evidentemente necessário, consiste em verificar se o elemento cuja chave é *c* seria de fato identificado por *h* no vetor (tabela) *T*, ou seja, verificar se *T[H(c)]*.chave = *c*. Neste ponto, duas questões surgem:

1. que tipo de função *H* deve ser utilizada?
2. como considerar a situação em que *H* não fornece o endereço do elemento desejado (ocorre colisão, ou seja, dois ou mais elementos são mapeados para o mesmo endereço)?

A resposta à segunda questão é que deve ser utilizado algum método para fornecer um endereço alternativo, e se este ainda não for o endereço desejado, um terceiro índice, e assim sucessivamente. A isto se chama *tratamento de colisões*.

Escolha de uma função de mapeamento

Um pré-requisito essencial para uma boa função de transformação é que ela distribua as chaves tão uniformemente quanto possível dentro do intervalo de valores de índices. Exceto quanto a esta exigência, a distribuição não é vinculada a nenhum padrão particular, sendo altamente desejável que esta distribuição seja totalmente aleatória. Tal propriedade deu a este método uma conotação, através do nome hashing (mapeamento), que significa pulverizar o argumento e espalhá-lo desordenadamente. A função *H* é chamada, em inglês, de *hash function* (função de mapeamento). *H* deverá ser passível de ser calculada de modo eficiente, a fim de minimizar o processamento do método.

Suponha-se que se dispõe de uma função de transferência ORD(c), que denota o número de ordem da chave c no conjunto de todas as possíveis chaves. Admita-se, ainda, que os índices do vetor pertençam ao intervalo 0 .. N-1, onde N é a capacidade do vetor. Então, uma escolha óbvia será

$$H(c) = ORD(c) \bmod N$$

onde *mod* denota o resto da divisão inteira. Esta função possui a propriedade de fornecer para as chaves valores que se espalham uniformemente no intervalo de validade do índice, sendo utilizada em muitas aplicações práticas. Pode, ainda ser calculada de modo muito eficiente, principalmente se *N* for uma potência de 2. Entretanto, este é exatamente um caso que deve ser evitado se as chaves forem alfanuméricas. A hipótese de que todas as chaves sejam igualmente prováveis é, neste caso, falsa, pois os termos que diferem de poucos caracteres serão mapeados, com alta probabilidade, para um mesmo índice, gerando uma distribuição nada uniforme. Torna-se recomendável, então, que *N* seja um número primo, pois assim os restos se distribuem no intervalo 0..*N*-1.

A escolha da função é de extrema importância para a eficiência e utilização do método, principalmente nos casos de criptografia.

Tratamento de colisões

Se uma célula da tabela, correspondente a uma dada chave, não contiver o elemento desejado e sim outro, esta situação caracteriza uma colisão. Em outras palavras, dois elementos possuem, nesta situação, chaves que são mapeadas no mesmo índice. Neste caso, será necessária uma segunda tentativa de localização, com base num índice obtido a partir da chave. Existem inúmeros métodos para a geração de índices secundários. Um método óbvio consiste em ligar entre si, através de uma lista encadeada, todos os índices *H(c)* primários idênticos. Este método é chamado *encadeamento direto*. Os elementos desta lista podem ou não estar armazenados na tabela primária. Caso estejam fora dela, a área de memória utilizada para este armazenamento é chamada *área de overflow*. Este método tem a desvantagem de exigir a manutenção das listas secundárias e de se dispor de espaço, em cada célula da

tabela, para o armazenamento de um apontador para a eventual lista de colisões a ela correspondente.

Uma alternativa consiste em se eliminar os apontadores, optando-se, em vez disso, por testar outras posições na mesma tabela, até que o elemento desejado seja encontrado ou até que se encontre uma posição ainda não preenchida, caso em que se conclui que a chave procurada não se encontra na tabela. Este método é denominado *método de endereçamento aberto*. Naturalmente, a seqüência de índices utilizada nos testes secundários deverá ser sempre a mesma para uma dada chave. O algoritmo de pesquisa em uma tabela, segundo este método, pode ser esboçado conforme o esquema a seguir:

```
h := H(c);
i:= 0;
repita
   se T[h].chave = c
       então elemento encontrado
       senão se T[h].chave = LIVRE
                 então elemento não se encontra na tabela
                 senão { colisão }
                     i := i + 1;
                     h := H (c)  + G(i);
                  fim;
até encontrado ou não na tabela ou tabela preenchida;
```

Várias funções de resolução de colisões já foram propostas. O método mais simples é tentar localizar o elemento procurado na próxima posição da tabela – considerada como circular – sucessivamente, até que seja encontrado o elemento correspondente à chave especificada, ou uma posição vazia. Conseqüentemente, G(i) = i; os índices h_i usados neste caso são:

$h_0 = H(c)$
$h_i = (h_0 + i)$ MOD N, i = 1 ... N-1

Este método é denominado *teste linear* e possui a desvantagem de as entradas tenderem a se agrupar em torno das chaves primárias. Naturalmente, pode-se escolher G de forma tal a distribuir-se uniformemente as chaves no conjunto restante das posições – processo que, se mal escolhido, pode se mostrar oneroso.

Análise da transformação de chaves

O número esperado *E* de testes para se buscar ou inserir em uma tabela alguma chave aleatoriamente escolhida é mostrado na tabela seguinte, em função do fator de carga *a* da tabela. Mesmo se uma tabela estiver com 90% de preenchimento, serão necessários em média 2,56 testes para localizar a chave ou para encontrar uma posição vazia para sua inserção. Note-se que esta cifra não depende do número de chaves, mas somente do fator de carga.

A (fator de carga = Percentual de ocupação)	E (quantidade esperada de testes)
0,1	1,05
0,25	1,15
0,5	1,39
0,75	1,85
0,9	2,56
0,95	3,15
0,99	4,66

A análise acima foi baseada com o uso de um método para tratamento de colisões que distribui uniformemente as chaves nas posições vazias remanescentes. Na prática, uma análise detalhada do método de teste linear apresenta um número esperado de testes igual a

$$E = \frac{1 - \frac{a}{2}}{1 - a}$$

Esta expressão produz os seguintes valores:

a (fator de carga = Percentual de ocupação)	E (quantidade esperada de testes)
0,1	1,06
0,25	1,17
0,5	1,50
0,75	2,50
0,9	5,50
0,95	10,50

Estes números tendem a apresentar a transformação de chaves como uma espécie de panacéia universal. De fato, seu desempenho é superior mesmo em relação à mais sofisticada das organizações de árvores, pelo menos em relação ao número de passos de comparação necessários à execução de buscas e inserções. Porém, é importante citar algumas das desvantagens deste método.

Certamente, a desvantagem mais importante, em relação aos métodos de alocação dinâmica de memória, é que a dimensão da tabela deve ser fixa, não podendo ser ajustada dinamicamente às necessidades de cada caso. É muito útil que se disponha de algum método que forneça uma estimativa a priori, razoavelmente boa para o número de elementos a serem classificados, evitando-se a sub-utilização de memória, baixo desempenho ou mesmo o estouro da tabela.

A segunda deficiência mais importante ocorre quando se deve tratar, além das inserções e pesquisas, a remoção de elementos da tabela. Esta remoção será tão complexa quanto for a modalidade escolhida para tratar as chaves secundárias

De modo geral, pode-se dizer que as organizações em árvores são preferidas quando o volume de dados for desconhecido ou fortemente variável, mas é evidente que uma análise caso a caso se torna necessária para uma escolha realmente acurada.

Algoritmo para transformação de chaves

Uma possível implementação do uso da transformação de chaves é a seguinte:

```
Programa Hash;

Constante P = 97;  { primo, dimensão da tabela }
                TamPalavra = 16;
                Livre = NULO;

Type ItemPtr = ponteiro para Item;
         Palavra = registro
                   Chave : vetor [0..TamPalavra-1] de caracter;
                   Primeiro, ultimo: ItemPtr;
         Fim;

         Item = registro
                   nroLinha: inteiro;
                   Próximo:ItemPtr;
         Fim;

Var linha:inteiro;
       trab:vetor[0..TamPalavra]de caracter;
      T:vetor [0..P-1] de Palavra;  { tabela de mapeamento  }

Procedimento ImprimeTabela;
Var i, k,m:inteiro;
Item:ItemPtr;
Começo
    Para k:= 0 até P-1 faça
       começo
           se T[K].chave <> livre então
                começo
                    escreva(T[k].chave);
              Item := T[k].primeiro;
              m := 0;
              repita
                   se m = 8 então
                        começo
                            escreva("\n");
                            m := 0;
                            para i:=1 até TamPalavra faça
                              escreva(" ");
                               fim;
                         m:= m + 1;
                   escreva(item->nroLinha);
                   item := item->próximo;
              até item = NULO;
```

```
              escreva("\n");
                 fim;
       fim;
fim;

Procedimento pesquisa(trab:vetor[0.TamPalavra-1] de caracter);
Var i, h, d:inteiro;
      Encontrou:booleano;
      Ch:caracter;
       x:ItemPtr;

começo
 { calcula o índice de mapeamento h para a palavra contida em trab }
     i:=0;
     h:=0;
     enquanto trab[i] > '\0' faça
          começo
              h := (256*h + ORD(trab[i])) MOD P;
              i := i + 1;
          fim;
     aloca(x,tamanho(Item));
     x->NroLinha := linha;
     x->proximo := NULO;
     d := 1;
     encontrou := FALSO;
     repita
        se T[h].chave = trab
            então começo { coincidência }
                        encontrou := VERDADEIRO;
                        T[h].ultimo->próximo := x;
                        T[h].ultimo := x;
                   fim;
            senão se T[h].chave = livre
                      então começo { entrada nova }
                                   T[h].chave := trab;
                                   T[h].primeiro := x;
                                   T[h].ultimo := x;
                                   Encontrou := VERDADEIRO;
                              fim;
                       senão começo { colisão }
                                    h := h + d;
                                    d := d + 2;
                                    se h > = P então h := h – P;
                                    se d = P
                                   então começo
                                          escreva("Excedida        a
capacidade da tabela");
                                              encerra;
                                         fim;
                                fim;

    até encontrou;
fim;

{ Programa principal }
var i:inteiro;
começo
     linha := 0;
     para i := 0 até P-1 faça T[i].chave = livre;
     repita
         leia(trab);
```

```
          linha = linha + 1;
          pesquisa(trab);
      até trab = "";
      ImprimeTabela;
Fim;
```

Referências

[Sedgewick 86] Sedgewick, Robert. *Algorithms*. Addison Wesley, 1986.

[Niemann 97] Niemann, Thomas. *Sorting and searching algorithms: a cookbook*. http://www.geocities.com/SoHo/2167/book.html.

Wirth, Niklaus. Algoritmos e Estruturas de Dados. Prentice-Hall do Brasil. 1990.