Universidade Estadual do Oeste do Paraná - UNIOESTE
Centro de Ciências Exatas e Tecnológicas - CCET
Curso de Ciência da Computação

ESTRUTURAS DE DADOS

CASCAVEL - PR
2019

SUMÁRIO

UNIDADE 1 – INTRODUÇÃO ÀS ESTRUTURAS DE DADOS 1

1.1 INFORMAÇÕES GERAIS 1
1.2 TIPOS PRIMITIVOS DE DADOS 1
1.3 MECANISMOS PARA CONSTRUÇÃO DE TIPOS 2
1.4 PROCEDIMENTOS E FUNÇÕES 7
1.4.1 PASSAGEM DE PARÂMETROS 8
1.4.2 PASSAGEM DE PARÂMETROS POR VALOR 9
1.4.3 PASSAGEM DE PARÂMETROS POR REFERÊNCIA 10

UNIDADE 2 – MATRIZES 11

2.1 INTRODUÇÃO 11
2.2 MATRIZES: CASO GERAL 11
2.3 MATRIZES ESPECIAIS 12
2.3.1 MATRIZES DIAGONAIS 13
2.3.2 MATRIZES TRIANGULARES 13
2.4 MATRIZES ESPARSAS 15

UNIDADE 3 – LISTAS, FILAS E PILHAS 20

3.1 LISTAS LINEARES 20
3.1.1 FUNDAMENTOS 20
3.1.2 REPRESENTAÇÕES 21
3.1.3 REPRESENTAÇÃO DE LISTAS POR ALOCAÇÃO SEQÜENCIAL 21
3.1.4 ACESSAR O K-ÉSIMO NÓ DE UMA LISTA 22
3.1.5 ALTERAR O VALOR DO K-ÉSIMO NÓ DE UMA LISTA 23
3.1.6 INSERIR UM NOVO NÓ ANTES DO K-ÉSIMO NÓ DE UMA LISTA 23
3.1.7 REMOVER O K-ÉSIMO NÓ DE UMA LISTA 23
3.1.8 REPRESENTAÇÃO DE LISTAS POR ENCADEAMENTO DOS NÓS 24
3.1.9 ALOCAÇÃO DINÂMICA DE MEMÓRIA 25
3.1.10 ENTENDENDO LISTAS REPRESENTADAS ATRAVÉS DE ALOCAÇÃO ENCADEADA DOS NÓS 26
3.1.11 ROTINAS BÁSICAS DE TRATAMENTO DE LISTAS 26
3.1.12 LISTAS COM DESCRITOR 31
3.1.13 LISTAS DUPLAMENTE ENCADEADAS 34
3.2 PILHAS 36
3.2.1 INIT, ISEMPTY E ISFULL 38
3.2.2 UM PRIMEIRO EXEMPLO DO USO DE PILHAS 39
3.2.3 IMPLEMENTAÇÃO SEQÜENCIAL DE PILHAS 40
3.2.4 ALGORITMOS PARA MANIPULAÇÃO DE PILHAS 41
3.3 FILAS 43
3.3.1 IMPLEMENTAÇÃO SEQÜENCIAL DE FILAS 44
3.3.2 PROBLEMAS NA IMPLEMENTAÇÃO SEQÜENCIAL DE FILAS 46
3.3.3 SOLUCIONANDO OS PROBLEMAS DA IMPLEMENTAÇÃO SEQÜENCIAL 47
3.3.4 IMPLEMENTAÇÃO CIRCULAR PARA FILAS 48
3.4 RECURSIVIDADE 49
3.4.1 INTRODUÇÃO 49
3.4.2 USO DE RECURSÃO NA SOLUÇÃO DE PROBLEMAS 51
3.4.3 QUANDO APLICAR RECURSÃO? 52
3.4.4 ELIMINANDO A RECURSÃO DE CAUDA 52
3.4.5 PILHAS E ROTINAS RECURSIVAS 53

UNIDADE 4 - ÁRVORES 55

4.1 ÁRVORES BINÁRIAS 55

4.1.1 ÁRVORES DE BUSCA BINÁRIA 56
4.1.2 OPERAÇÕES BÁSICAS EM ÁRVORES DE BUSCA BINÁRIA 57
4.1.3 ATRAVESSAMENTO EM ÁRVORES BINÁRIAS 61
4.2 ÁRVORES BALANCEADAS 65
4.3 ÁRVORES-AVL 65
4.3.1 INCLUSÃO EM ÁRVORES-AVL 65
4.3.2 IMPLEMENTAÇÃO DA INCLUSÃO 69
4.3.3 REMOÇÕES EM ÁRVORES-AVL 73
4.4 ÁRVORES HEAP E HEAPSORT 77
4.2.1 HEAPSORT 78
4.5 ÁRVORES B 80
4.4.1 ÁRVORES B MULTIDIRECIONAIS 82
4.6 OUTROS TIPOS DE ÁRVORES E SUAS REPRESENTAÇÕES 95
UNIDADE 5 – PESQUISA DE DADOS 96
5.1 MÉTODOS DE BUSCA 96
5.1.1 BUSCA LINEAR 96
5.1.2 BUSCA BINÁRIA 97
5.2 PROCESSAMENTO EM CADEIAS 98
5.2.1 INTRODUÇÃO 98
5.2.2 O PROBLEMA DO CASAMENTO DE CADEIAS 98
5.2.3 O ALGORITMO DA FORÇA BRUTA 99
5.2.4 O ALGORITMO DE KNUTH, MORRIS E PRATT 100
5.3 ESPALHAMENTOS 103
5.3.1 FUNDAMENTOS 103
5.3.2 APLICABILIDADE DO ESPALHAMENTO 105
5.3.3 TABELAS DE ESPALHAMENTO 106
5.3.4 FUNÇÕES DE ESPALHAMENTO 107
5.3.5 O MÉTODO DA DIVISÃO 108
5.3.6 TRANSFORMAÇÃO DE CHAVES ALFANUMÉRICAS 109
5.3.7 OPERAÇÕES BÁSICAS SOBRE TABELA DE ESPALHAMENTO 112

UNIDADE 1 – INTRODUÇÃO ÀS ESTRUTURAS DE DADOS

1.1 INFORMAÇÕES GERAIS

Niklaus Wirth afirma que programas de computador podem ser divididos em dois componentes: lógica e dados.

A lógica trata como as operações serão encadeadas de maneira a chegar ao resultado esperado. Este componente foi discutido na disciplina de algoritmos.

O segundo componente - dados - são os elementos a serem manipulados no programa. Neste ponto torna-se importante o estudo dos dados, principalmente na sua forma de estruturação, armazenamento e manipulação.

Este é o objetivo da disciplina de Estrutura de Dados. Estudar como os dados são estruturados, como são armazenados e, principalmente, como estes dados podem ser manipulados. Lembrando que a manipulação está condicionada à estrutura de dados empregada.

1.2 TIPOS PRIMITIVOS DE DADOS

Serão considerados disponíveis quatro tipos primitivos de dados:

Tipo	Abreviação	Conteúdo
Inteiro	int	-5, -2, -1, 0, 1, 3, 100
Real	real	-120.32, -50.12, 0, 1, 1.32
Lógico	log	V ou F
Caractere	car	A, a, B, b, C, c, !, ?, /

Para cada um desses tipos supomos que é utilizada uma representação adequada. Não vamos nos preocupar com as restrições que seriam impostas por um particular computador ou sistema; tais restrições envolvem, por exemplo, valores máximos dos inteiros, precisão dos reais, etc.

a) **Tipo inteiro**:

Os valores possíveis para um objeto do tipo ***int*** são os números inteiros (negativos, zero ou positivos). As operações permissíveis são:

Operação	Símbolo
soma	+
subtração	-
multiplicação	*
divisão inteira	div
resto da divisão	mod

Cada uma dessas operações recebe como argumentos um par de inteiros e fornece um resultado inteiro. Em particular, *div* e *mod* são, respectivamente, o quociente inteiro e o resto da divisão entre dois inteiros; por exemplo, 5 *div* 2 é 2, e 5 *mod* 2 é 1.

Além disso, podemos comparar dois inteiros para testar se são iguais (=), diferentes ($\neq$), ou segundo a ordem (<, $\leq$, >, $\geq$).

b) **Tipo real**:

Os objetos do tipo ***real*** são os números racionais, isto é, números normalmente representados por uma parte inteira e uma parte fracionária.

As operações do tipo *real* são:

Operação	Símbolo
soma	+
subtração	-
multiplicação	*
divisão	/

Cada uma das quatro operações acima recebe um par de números do tipo *real* e fornece um resultado do mesmo tipo. Além disso, como nos inteiros, podemos comparar dois elementos do tipo real conforme =, ≠, <, >, etc.

c) **Tipo lógico**:

Este tipo consiste de exatamente dois valores: verdadeiro e falso, sendo as constantes correspondentes V e F.

As operações sobre os valores lógicos são

Operação	Símbolo
e (conjunção)	e ou &
ou (disjunção)	ou ou |
não (negação)	not, ~ ou !

d) **Tipo caractere**:

Os objetos deste tipo são os chamados “caracteres alfanuméricos”: os dígitos decimais (0 - 9), as letras (A - Z) e alguns sinais especiais (espaço em branco, sinais de pontuação, etc.).

No tipo *caractere* podemos realizar comparações do tipo =, ≠, >, < e ainda a operação de adição (+) que concatena caracteres.

1.3 MECANISMOS PARA CONSTRUÇÃO DE TIPOS

Veremos, a seguir, alguns mecanismos que permitem a construção de outro tipo a partir dos tipos primitivos. O formato geral para a definição de um tipo é o seguinte:

tipo nome_do_tipo_definido = definição_do_tipo

Poderemos construir os seguintes tipos:

- vetor;
- matriz;
- registro;
- referência (ponteiro);
- enumeração.

a) **Vetor**:

O vetor permite a construção de um tipo cujos valores são agregados homogêneos de um tamanho definido, isto é, suas componentes são todas de um

mesmo tipo.

O formato de definição para vetor é o seguinte:

tipo nome_do_vetor = vetor [limite_inferior.. limite_superior] de tipo;

onde limite_inferior e limite_superior são constantes do tipo *inteiro* e *tipo* é o tipo das componentes.

Por exemplo:
tipo VET_NOME_ALUNOS = vetor [1..10] de caractere;

Criamos um tipo cujos elementos são compostos por 10 nomes. Um elemento deste tipo pode conter o nome dos 10 alunos de uma classe.

Supondo que uma variável ***TURMA_A*** do tipo VET_NOME_ALUNOS possua os seguintes valores:

Ana	Pedro	Paulo	Carla	José	João	Maria	Cláudia	Mário	Iara
1	2	3	4	5	6	7	8	9	10

Cada elemento num vetor possui um índice que indica sua posição dentro do vetor.

Caso queiramos referenciar Carla usaremos a seguinte notação:

TURMA_A[4]

O conteúdo deste elemento é Carla.

b) **Matriz**:

A matriz, como o vetor, permite a construção de um tipo cujos valores são agregados homogêneos de um tamanho definido, isto é, suas componentes são todas de um mesmo tipo.

O formato de definição para vetor é o seguinte:

tipo nome_da_matriz = matriz [lim_inf_1..lim_sup_1; lim_inf_2..lim_sup_2] de tipo;

onde lim_inf_1, lim_inf_2, lim_sup_1 e lim_sup_2 são constantes do tipo *inteiro* e *tipo* é o tipo das componentes.

Por exemplo:
tipo MAT_CLASSES_ALUNOS = matriz [1..3; 1..10] de caractere;

Criamos um tipo cujos elementos são compostos por 30 nomes. Um elemento deste tipo pode conter o nome dos 10 alunos de cada uma das 3 classes de uma escola.

Supondo que uma variável ***TURMAS_QUINTA_SERIE*** do tipo MAT_CLASSES_ALUNOS possua os seguintes valores:

1	Ana	Pedro	Paulo	Carla	José	João	Maria	Cláudia	Mário	Iara
2	Juca	Roger	Ivo	Cris	Joel	Márcio	Márcia	Sara	Denise	Carlos
3	Jucy	Darci	Iloir	Osmar	Daniel	Diogo	Ana	Cláudia	Josi	Julia
	1	2	3	4	5	6	7	8	9	10

Cada elemento numa matriz é referencial por dois índices, que indicam sua posição dentro da matriz.

Caso queiramos referenciar Márcia usaremos a seguinte notação:

TURMAS_QUINTA_SERIE [2, 7]

O conteúdo deste elemento é Márcia.

c) **Registro**:

Às vezes temos necessidade de agregados heterogêneos, isto é, cujas componentes não são necessariamente de um mesmo tipo.

O formato de definição para um registro é o seguinte:

```
tipo nome_do_registro = registro
                            campo1, campo2, ..., campon : tipo1;
                            campo1, campo2, ..., campon : tipo2;
                            ...
                            campo1, campo2, ..., campon : tipon;
                        fim registro;
```

Por exemplo, se quisermos construir um registro para armazenar o nome da disciplina e a turma à qual será ofertada esta disciplina, teremos:

```
tipo TURMAS_DISCIPLINA = registro
                           DISCIPLINA : caractere;
                           TURMA : inteiro;
                         fim registro;
```

Supondo uma variável ***GRADE_DISCIPLINAS*** do tipo TURMAS_DISCIPLINAS para a qual queiramos inserir a seguinte informação:

A disciplina de Cálculo Diferencial será ofertada para a turma número 3.

A atribuição desta informação para a variável GRADE_DISCIPLINAS se dará da seguinte maneira:

GRADE_DISCIPLINAS.DISCIPLINA ← ‘Cálculo Diferencial’;
GRADE_DISCIPLINAS.TURMA ← 3;

Noutro exemplo, o tipo fração pode ser assim constituído:

```
tipo FRACAO = registro
                NUMERADOR, DENOMINADOR : inteiro;
              fim registro;
```

A fração 2/7 poderia ser armazenada numa variável ***NUM_FRAC*** do tipo FRACAO, da seguinte forma:

NUM_FRAC.NUMERADOR ← 2;
NUM_FRAC.DENOMINADOR ← 7;

d) **Referência (Ponteiro)**:

Até agora não nos importamos muito com a maneira pela qual um objeto de um determinado tipo é armazenado na memória de um computador. Supomos que a

cada variável corresponde um espaço suficiente para guardar cada um de seus possíveis valores. Dizemos que a cada variável é alocado um espaço - que chamaremos de célula (do tipo da variável). Vamos supor que esta alocação é determinada pela simples ocorrência da declaração da variável. Assim, a declaração das variáveis *x* e *y* como de tipo inteiro ocasiona a alocação de uma célula inteira para cada uma delas.

O mecanismo de ***referência ou ponteiro*** permitirá uma modalidade dinâmica de alocação. A definição de tipo pelo mecanismo de referência obedece ao seguinte formato:

tipo nome_do_ponteiro = ^tipo;

Por exemplo:

tipo MATRICULA = ^inteiro;

Ao declararmos uma variável ***GEOGRAFIA*** do tipo MATRICULA estaremos criando um ponteiro para uma célula de memória. Nesta célula é possível armazenar um número inteiro, entretanto esta célula ainda não existe; existe somente um ponteiro que poderá apontar para esta célula.

A alocação do espaço para a variável geografia compreende duas partes:

1 - Parte de valor: para armazenar um valor do tipo inteiro;

2 - Parte de posição: para armazenar uma indicação da localização da parte de valor.

A simples declaração da variável ***GEOGRAFIA*** como sendo do tipo MATRICULA ocasiona a alocação de espaço para a parte de posição, mas não ainda para a parte de valor. Esta última é ocasionada pelo comando:

aloque (GEOGRAFIA);

que, além disso, faz com que na parte de posição seja colocada uma indicação da localização onde está alocada a parte de valor.

- 1 - Declaração de geografia:

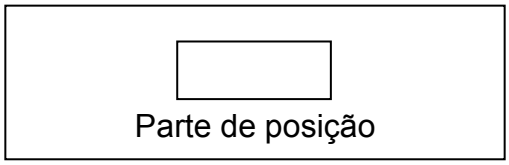


Figura 1.1 – Situação após ser declarada a variável “GEOGRAFIA”

- 2 - Após o comando ***aloque*** (GEOGRAFIA):

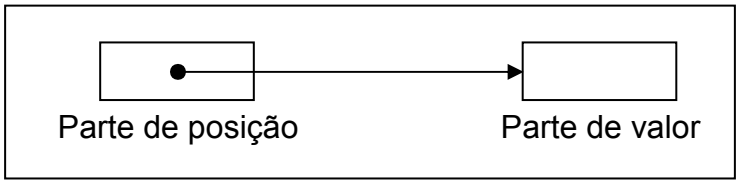


Figura 1.2 – Situação após ser executado o comando “aloque”

O espaço alocado pelo comando aloque pode ser liberado pela execução de:

desaloque (GEOGRAFIA);

cujo efeito é fazer a situação da Figura 1.2 retornar à situação da Figura 1.1.

Já que a variável GEOGRAFIA tem, por assim dizer, duas partes, vamos usar a seguinte notação:

- GEOGRAFIA designa a parte de posição (que é criada pela declaração da variável);
- GEOGRAFIA^ designa a parte de valor (que é apontada pela parte de posição).

A indicação da localização contida na parte de posição não é um objeto que possa ser manipulado por operações. Podemos apenas testar sua igualdade ou não por meio de = ou ≠. Um caso especial é a constante ***nil***; se o conteúdo de GEOGRAFIA é ***nil*** então não aponta para nenhuma parte de valor.

Os mecanismos de agregação (vetor, matriz e registro) geralmente pressupõem que suas componentes sejam armazenadas de maneira contígua em memória. O mecanismo de referência permite armazenamento disperso dos seus componentes.

e) **Enumeração**:

A enumeração permite definir tipos de dados por meio dos valores que os dados daquele tipo podem assumir. A definição é feita indicando-se um conjunto ordenado de identificadores que denotam os valores que os dados daquele tipo devem assumir.

O formato geral de uma enumeração é:

tipo nome_do_conjunto = (valor1, valor2, valor3, ..., valorn);

Como exemplo, suponha a definição do tipo mês, que é feita por enumeração dos valores válidos que uma variável deste tipo pode ter:

tipo MES = (jan, fev, mar, abr, maio, jun, jul, ago, set, out, nov, dez);

Assim, seria válida a seguinte construção:

```
var
  MES_NASC : MES;

se MES_NASC = dez então ...
```

Note que, pelo fato de os valores de definição do tipo formar um conjunto ordenado, existe uma relação de ordem entre eles. Portanto, é válido o uso de operadores de relação entre eles, isto é, é válido afirmarmos que jan < fev < ... < nov < dez, para o tipo antes definido.

1.4 PROCEDIMENTOS E FUNÇÕES

- **PROCEDIMENTOS**

Os procedimentos são utilizados quando um conjunto de comandos repete-se ao longo do algoritmo. Então, para não escrevermos várias vezes o mesmo bloco de comandos, usamos os procedimentos.

Sintaxe:

```
procedimento IDENTIFICADOR (parâmetros);
   Comandos;
fim procedimento IDENTIFICADOR;
```

Onde:
- IDENTIFICADOR é o nome de referência do procedimento;
- parâmetros é a lista de variáveis que serão passadas ao procedimento para serem manipuladas no seu interior. Na definição dos parâmetros também devem ser declarados seus tipos. Nem todo procedimento utiliza-se de parâmetros, portanto é um item opcional.

Exemplo:

```
procedimento LINHA (COMPRIMENTO : inteiro);
var
   I : inteiro;

   para I ← 1 até COMPRIMENTO faça
      escreva "-";
   fim para;

fim procedimento LINHA;
```

- **FUNÇÕES**

As funções constituem um tipo especial de sub-rotina, bastante semelhante ao procedimento, que tem a característica especial de retornar ao algoritmo chamador um valor associado ao nome da função. Esta característica permite uma analogia com o conceito de função na matemática.

A utilização de outras funções no algoritmo como por exemplo, seno, tangente ou qualquer outra função "especial", pode ser feito declarando-se um procedimento função.

A declaração de uma função é semelhante à de um procedimento. Difere somente na especificação do tipo da mesma, ou seja, do tipo de valor que será retornado. Apesar de terem sido citadas apenas funções numéricas, elas podem ser lógicas ou literais.

Sintaxe:

```
função IDENTIFICADOR (parâmetros):tipo de retorno;
   comandos;
fim função IDENTIFICADOR;
```

Onde:
- IDENTIFICADOR é o nome de referência da função;
- parâmetros é a lista de variáveis que serão passadas à função para serem manipuladas no seu interior. Na definição dos parâmetros também devem ser declarados seus tipos. Nem toda função utiliza-se de parâmetros, portanto é um item opcional.
- tipo de retorno é o tipo de dado que a função retornará (inteiro, real, lógico, caractere);

Exemplo:

```
função E_PAR (N: inteiro): lógico;
   se (N mod 2 = 0) então
      E_PAR ← Verdadeiro;
   senão
      E_PAR ← Falso;
   fim se;
fim função E_PAR;
```

1.4.1 PASSAGEM DE PARÂMETROS

A transferência de informações de e para sub-rotinas utilizando-se variáveis globais não constitui uma boa disciplina de programação.

Estas transferências precisam ser mais formalizadas e documentadas a bem da legitimidade, documentação e organização do programa elaborado.

Em algoritmos, a transferência de informações de e para sub-rotinas pode ser feita com a utilização de parâmetros.

Esta utilização formaliza a comunicação entre módulos. Além disso, permite que um módulo seja utilizado com operandos diferentes, dependendo do que se deseja do mesmo.

Parâmetros de definição são objetos utilizados dentro das sub-rotinas que em cada ativação representam objetos de nível mais externo. A forma de se utilizar parâmetros em sub-rotinas foi apresentada anteriormente.

A chamada de uma sub-rotina aparece numa expressão e tem a seguinte forma:

```
NOME DA SUBROTINA (Lista de parâmetros de chamada);
```

Exemplo:

```
início
   var
      A, B, C : real;
   procedimento EXEMPLO (VALOR_1, VALOR_2, VALOR_3 : real);
      MAIOR_VALOR : real;

      MAIOR_VALOR ← VALOR_1;
```

```
      se (VALOR_2 > MAIOR_VALOR) então
         MAIOR_VALOR ← VALOR_2;
      fim se;

      se (VALOR_3 > MAIOR_VALOR) então
         MAIOR_VALOR ← VALOR_3;
      fim se;

      escreva "O maior valor é: ", MAIOR_VALOR;

   fim procedimento EXEMPLO;

{corpo do programa principal}
   leia "Digite 3 números:"; A, B, C;
   EXEMPLO (A, B, C);

fim.
```

1.4.2 PASSAGEM DE PARÂMETROS POR VALOR

Na passagem de parâmetros por valor (ou por cópia) o parâmetro real é calculado e uma cópia de seu valor é fornecida ao parâmetro formal, no ato da invocação da sub-rotina. A execução da sub-rotina prossegue normalmente e todas as modificações feitas no parâmetro formal não afetam o parâmetro real, pois se trabalha apenas com uma cópia do mesmo.

Exemplo:

```
início
   var
      X : inteiro;

   procedimento PROC (Y : inteiro);

      Y ← Y + 1;
      escreva "Durante = ", Y;

   fim procedimento PROC;
   X ← 1;
   escreva "Antes = ", X;
   PROC (X);
   escreva "Depois = ", X;

fim.
```

O algoritmo anterior fornece o seguinte resultado:
Antes = 1;
Durante = 2;
Depois = 1;

Este tipo de ação é possível porque, neste mecanismo de passagem de parâmetros, é feita uma reserva de espaço em memória para os parâmetros formais,

para que neles seja armazenada uma cópia dos parâmetros reais.

1.4.3 PASSAGEM DE PARÂMETROS POR REFERÊNCIA

Neste mecanismo de passagem de parâmetros não é feita uma reserva de espaço em memória para os parâmetros formais. Quando uma sub-rotina com parâmetros passados por referência é chamada, o espaço de memória ocupado pelos parâmetros reais é compartilhado pelos parâmetros formais correspondentes. Assim, as eventuais modificações feitas nos parâmetros formais também afetam os parâmetros reais correspondentes.

Uma mesma sub-rotina pode utilizar diferentes mecanismos de passagem de parâmetros, para parâmetros distintos. Para diferenciar uns dos outros, convencionou-se colocar o prefixo **var** antes da definição dos parâmetros formais passados por referência. Se por exemplo uma sub-rotina tem o seguinte cabeçalho:

```
procedimento PROC (X, Y : inteiro; var Z : real; J : real)
```

Então:
- X e Y são parâmetros formais do tipo inteiro e são passados por valor;
- Z é um parâmetro formal real passado por referência;
- J é um parâmetro formal real passado por valor.

O exemplo do item anterior, alterado para que o parâmetro Y do procedimento seja passado por referência, torna-se:

```
início
   var
      X : inteiro;

   procedimento PROC (var Y : inteiro);

      Y ← Y + 1;
      escreva "Durante = ", Y;

   fim procedimento PROC;
   X ← 1;
   escreva "Antes = ", X;
   PROC (X);
   escreva "Depois = ", X;

fim.
```

O resultado do algoritmo modificado é:
Antes = 1;
Durante = 2;
Depois = 2;

UNIDADE 2 – MATRIZES

2.1 INTRODUÇÃO

Já estamos bastante familiarizados com a idéia de matriz. Em matemática, é usual se trabalhar com matrizes de elementos reais como:

$$M = \begin{bmatrix} 1,0 & -2,0 & 5,7 \\ -1,3 & 0,0 & 0,9 \end{bmatrix}$$

que é uma matriz real 2x3, isto é, com 2 linhas e 3 colunas.

Estamos acostumados a somar e multiplicar matrizes como esta acima. Porém, do ponto de vista de estrutura de dados, estamos mais interessados na maneira como temos acesso à informação armazenada em uma matriz. Isto é feito pela indicação de uma linha e uma coluna, o que identifica a posição onde que elas se "cruzam". Assim, através dos índices 1 e 3, identificamos a posição [1, 3], que no caso do exemplo acima contém o valor 5,7. Isto costuma ser indicado por M[1, 3] = 5,7.

Para localizarmos um elemento particular precisamos fornecer dois índices: sua linha e sua coluna. Por esta razão, elas são chamadas matrizes bidimensionais. Quando apenas um índice é suficiente, temos uma matriz unidimensional, que costuma ser chamada de vetor coluna, ou de vetor linha, conforme a representamos:

Vetor Coluna: $\begin{bmatrix} 7,0 \\ 0,1 \\ 2,3 \end{bmatrix}$

Vetor Linha: $\begin{bmatrix} 7,0 & 0,1 & 2,3 \end{bmatrix}$

É comum também o caso de matrizes tridimensionais. Por exemplo, uma matriz cujo elemento [i, j, k] dá a temperatura em um ponto de um cubo sólido.

2.2 MATRIZES: CASO GERAL

Os lugares de um teatro costumam ser identificados através da fila e da coluna de cada um. O serviço de reservas mantém um mapa que indica os lugares ocupados e os ainda vagos. Para fixar idéias, vamos considerar um teatro com 15 filas, numeradas de 1 a 15, cada fila com 20 cadeiras; 10 cadeiras à esquerda e 10 cadeiras à direita.

Vamos imaginar o mapa de reservadas como uma matriz bidimensional RES, da maneira seguinte:

<table>
<tr><td></td><td>-10</td><td>…</td><td>-1</td><td>0</td><td>1</td><td>…</td><td>100</td></tr>
<tr><td>1</td><td></td><td></td><td></td><td rowspan="4"></td><td></td><td></td><td></td></tr>
<tr><td>2</td><td colspan="3" rowspan="2">Lado
Esquerdo</td><td colspan="3" rowspan="2">Lado
Direito</td></tr>
<tr><td>⋮</td></tr>
<tr><td>15</td><td></td><td></td><td></td><td></td><td></td><td></td></tr>
</table>

Um par de índices como [7, -3] identifica a poltrona número 3 do lado esquerdo da fila 7; caso esteja reservada, teremos RES [7, -3] = V. Analogamente, RES [2, +5] = F quer dizer que a 5ª poltrona do lado direito da 2ª fila está livre. Pares com j = 0 (RES [i, 0]) indicam posições no corredor e são casos especiais.

A matriz RES poderia ser declarada da seguinte forma:

```
tipo TEATRO = matriz [1..15; -10..10] de lógico;
var
  RES : TEATRO;
```

O procedimento usual da reserva de um lugar pode então ser descrito através de operações de consulta e de atribuição à matriz RES:

- desejando-se a poltrona i da fila j, faz-se uma consulta.
  - caso RES [i, j] = V, deve-se escolher outro lugar;
  - caso RES [i, j] = F, a reserva é concedida e o mapa alterado através da atribuição RES [i, j] ← V.

Supondo que haja dois espetáculos por dia, é fácil imaginar os dois mapas de reserva correspondentes como uma matriz RESERVA tridimensional, na qual o novo índice k é 1 ou 2, conforme a sessão desejada. Agora o dado necessário para se fazer uma reserva é uma tupla [i, j, k] e as operações passam a ser:

- consulta: RESERVA [i, j, k];
- atribuição: RESERVA [i, j, k] ← Valor;

A matriz RESERVA poderia ser declarada da seguinte forma:

```
tipo THEATER = matriz [1..3; 1..15; -10..10] de lógico;
var
  RESERVA : THEATER;
```

Exercícios:

1) Escreva um procedimento para realizar a reserva de um local no teatro. Considere o caso em que há somente um espetáculo ao dia. O procedimento será executado até que o usuário deseja sair do procedimento.

2) Escreva um procedimento para atender o pedido de cancelamento de reservas.

3) Escreva um procedimento otimizado para realizar a transposição de uma matriz de ordem m x m.

2.3 MATRIZES ESPECIAIS

A representação vista até agora guarda todos os elementos da matriz. Freqüentemente ocorrem matrizes de números apresentando uma boa parte de elementos nulos. Deixando de guardar estes, podemos fazer uma economia razoável de memória. Se os elementos nulos estão concentrados em uma região da matriz,

como por exemplo acima da diagonal principal, então podemos lançar mão de representações especiais, simples e compactas.

2.3.1 MATRIZES DIAGONAIS

Consideremos a matriz M [3, 4], de reais abaixo:

$$M = \begin{bmatrix} 3{,}5 & 0{,}0 & 0{,}0 & 0{,}0 \\ 0{,}0 & -1{,}6 & 0{,}0 & 0{,}0 \\ 0{,}0 & 0{,}0 & 2{,}5 & 0{,}0 \end{bmatrix}$$

Matrizes como M, em que todos os elementos com i ≠ j são nulos, são chamadas de matrizes diagonais. Uma vez que somente os elementos da diagonal principal são diferentes de zero, podemos utilizar um vetor para armazenar este tipo de matriz.

Uma matriz diagonal com **m** linhas e **n** colunas contém **m.n** elementos. O vetor para armazenar esta matriz será composto por **p** elementos, sendo **p** o menor valor entre **m** e **n**.

Assim, para a matriz M acima, podemos utilizar um vetor com 3 posições, uma vez que ela possui **m** = 3 linhas e **n** = 4 colunas, implicando em **p** = 3

```
tipo MATDIAG = vetor [1..3] de real;
var
  M_DIAG : MATDIAG;
```

A consulta a uma matriz diagonal M_DIAG é bastante simples, como vemos a seguir (Implementar a consulta através de uma função):

```
função CONSULTA (M: MATDIAG; I, J : inteiro):real;

   se (i = j) então
      CONSULTA ← M[I]
   senão
      CONSULTA ← 0,0;
   fim se;

fim função CONSULTA;
```

A atribuição de valores numa matriz diagonal somente é possível para elementos em que i = j. Qualquer valor, diferente de zero, atribuído a uma posição com i ≠ j implica na matriz deixar de ser diagonal.

2.3.2 MATRIZES TRIANGULARES

Outro caso em que se pode economizar memória por meio de uma representação especial é o das matrizes triangulares. Por exemplo, as matrizes 5x4 abaixo são triangulares, sendo M triangular superior, e N triangular inferior.

$$M = \begin{bmatrix} 7,1 & 0,3 & 5,1 & 4,3 \\ 0,0 & -1,0 & 2,1 & 7,7 \\ 0,0 & 0,0 & 3,5 & 2,4 \\ 0,0 & 0,0 & 0,0 & 4,2 \\ 0,0 & 0,0 & 0,0 & 0,0 \end{bmatrix} \quad N = \begin{bmatrix} 8,3 & 0,0 & 0,0 & 0,0 \\ 7,1 & 2,4 & 0,0 & 0,0 \\ 3,5 & 4,2 & 4,7 & 0,0 \\ 4,5 & 6,7 & -0,1 & 2,3 \\ 3,1 & -4,0 & 5,1 & 5,1 \end{bmatrix}$$

Dos 5 x 4 = 20 elementos da matriz M, apenas os 10 elementos não abaixo da diagonal principal são diferentes de 0,0. Assim, poderíamos representar M por:

M = [7,1 0,3 5,1 4,3 -1,0 2,1 7,7 3,5 2,4 4,2]

Já no caso de N, bastaria guardar os 14 elementos não acima da diagonal principal.

Vamos nos restringir a matrizes quadradas. Uma matriz é dita triangular inferior quando P[i, j] = 0 sempre que i < j. Então, na 1ª linha de P, todos os elementos são nulos, exceto talvez P[1, 1]. Na 2ª linha, só P[2, 1] e P[2, 2] é que podem ser não nulos. Na linha i, só podem ser diferentes de 0 todos os elementos P[i, 1], P[i, 2], ..., P[i, i]. Assim, o número de elementos não nulos em P pode ser obtido por:

$$m = \frac{n \cdot (n+1)}{2}$$

onde:

- m = número de elementos não nulos em P;
- n = número de linhas e colunas em P (matriz quadrada).

Veja o exemplo:

$$MAT = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 3 & 5 & 0 & 0 & 0 \\ 1 & 4 & 7 & 0 & 0 \\ 8 & 3 & 2 & 1 & 0 \\ 7 & 4 & 5 & 2 & 3 \end{bmatrix} \quad m = \frac{5 \cdot (5+1)}{2} = 15$$

Poderíamos, então, armazenar esta matriz num vetor com m = 15 posições, da seguinte maneira:

MAT_V = [1, 3, 5, 1, 4, 7, 8, 3, 2, 1, 7, 4, 5, 2, 3]

Mas como vamos saber que o elemento MAT[4, 3] = MAT_V[9] = 2?

Para acessar um elemento da matriz, através de suas coordenadas i e j, numa matriz triangular inferior, podemos utilizar a seguinte regra:

MAT[i, j] = MAT_V[j + (i . (i - 1)) div 2]

Assim:

MAT[4, 3] = MAT_V[3 + (4 . (4 - 1)) div 2] = MAT_V[3 + 6] = MAT_V[9] = 2

Exercícios:

2) Escreva um algoritmo para inserir elementos numa matriz triangular inferior de ordem 5.

```
tipo MATTI = vetor [1..15] de real;

procedimento INSERE_MAT_DIAG_INF (var MATRIZ : MATTI);
var
   I, J : inteiro;

   para I = 1 até 5 faça
      para J = 1 até 5 faça

         se (I >= J) então
            escreva ´Digite o elemento ´, I, J;
            leia (MATRIZ[J + (I * (I - 1)) div 2]);
         fim se;

      fim para
   fim para

fim procedimento INSERE_MAT_DIAG_INF;
```

3) Escreva um algoritmo para inserir elementos numa matriz triangular superior de ordem 5.

$[i, j] \rightarrow k \Rightarrow k = \frac{2n(i-1)+i(1-i)}{2} + j$, onde n = ordem da matriz

2.4 MATRIZES ESPARSAS

Matrizes esparsas são aquelas matrizes onde há uma quantidade elevada de elementos nulos. Por exemplo:

$$ME = \begin{bmatrix} 6 & 0 & 3 & 0 & 0 & 0 \\ 0 & 0 & 0 & 2 & 0 & 3 \\ 0 & 1 & 0 & 0 & 0 & 0 \end{bmatrix}$$

Podemos otimizar o uso de memória armazenando somente os elementos não nulos desta matriz num vetor. Mas ao armazenarmos somente o valor perdemos a posição do elemento na matriz. Portanto, devemos armazenar, junto ao valor, seu respectivo índice.

VETME = [(1, 1, 6) (1, 3, 3) (2, 4, 2) (2, 6, 3) (3, 2, 1)]

Note que cada elemento deste vetor é constituído por 3 valores: o primeiro valor é a linha do elemento, o segundo a coluna e o terceiro o valor armazenado na matriz.

VETME trata-se de um vetor onde cada elemento é uma tupla e a construção desta tupla pode ser realizada através de um registro. Assim, teremos:

```
tipo ELEMENTO = registro
                   LINHA : inteiro;
                   COLUNA : inteiro;
                   VALOR : inteiro;
                fim registro;

tipo VETOR = vetor [1..5] de ELEMENTO;

var
   VETME : VETOR;
```

Na estrutura acima é possível armazenar apenas cinco elementos, pois o vetor é estático. O que acontecerá que inserirmos mais um elemento não nulo na matriz? Será impossível fazermos isso.

Pensando nestes casos, devemos declarar o vetor com um número maior de elementos, por exemplo 8 elementos:

```
tipo VETOR = vetor [1..8] de ELEMENTO;
```

Desta maneira teremos 3 posições de folga no vetor que representa a matriz esparsa, sendo possível inserir até três elementos novos na mesma.

A consulta numa matriz esparsa se dá pela busca de um elemento que contenha o índice procurado. Caso este índice não exista significa que aquele elemento possui valor 0.

Exercícios:

5) Escreva o algoritmo de uma função para buscar um elemento numa matriz esparsa.

```
início
   CONST NUM = 8;
   tipo ELEMENTO = registro
                      LINHA : inteiro;
                      COLUNA : inteiro;
                      VALOR : inteiro;
                   fim registro;

   tipo MATESP = vetor [1..NUM] de ELEMENTO;

   função BUSCA (I, J : inteiro; MATRIZ : MATESP): inteiro;
   var
      D : inteiro;

      para D = 1 até NUM faça

         se ((MATRIZ[D].linha = I) e
             (MATRIZ[D].coluna = J)) então
            BUSCA ← MATRIZ[D].VALOR;
            exit;
```

```
            senão
               BUSCA ← 0;
            fim se;
        fim para
    fim função BUSCA;
fim.
```

Quando inserirmos um novo elemento numa matriz esparsa devemos manter os elementos ordenados pelos seus índices, de forma que o processo de busca seja o mais rápido possível. Da mesma maneira, ao atribuirmos o valor nulo para um elemento, devemos removê-lo da matriz, pois a mesma somente armazena os elementos não nulos.

Na inserção devemos nos preocupar em verificar três fatos:

- Nulidade do novo valor;
- Existência de espaço no vetor, que representa a matriz, para o novo elemento;
- Ordem dos elementos.

Quando um novo valor será inserido na matriz, e este valor for nulo, simplesmente não será inserido no vetor.

O exemplo a seguir mostra a inserção de um elemento não nulo na matriz esparsa.

A matriz ME que contém os valores

$$ME = \begin{bmatrix} 6 & 0 & 3 & 0 & 0 & 0 \\ 0 & 0 & 0 & 2 & 0 & 3 \\ 0 & 1 & 0 & 0 & 0 & 0 \end{bmatrix}$$

pode ser representada por um vetor

VETME = [(1, 1, 6) (1, 3, 3) (2, 4, 2) (2, 6, 3) (3, 2, 1) (, ,) (, ,) (, ,)]

- Inserção de um novo elemento:

Na inserção de um novo elemento devemos, primeiro, verificar se é mesmo um novo elemento, ou seja, se não há um elemento não nulo na matriz com o mesmo índice.

Supondo que temos de inserir o novo elemento ME[3, 4] = 5. Este índice não existe no vetor VETME, portanto trata-se de um novo elemento. Sua inserção será feita anexando-se o elemento na primeira posição livre de VETME, pois há espaço no vetor e o elemento a ser inserido tem índice que o posiciona após o último elemento: ME[3, 2] = 1.

VETME = [(1, 1, 6) (1, 3, 3) (2, 4, 2) (2, 6, 3) (3, 2, 1) (3, 4, 5) (, ,) (, ,)]

Analisemos, agora, a inserção de um novo elemento ME[2, 5] = 4.

O índice já existe em VETME? Não, então se trata de um elemento novo. Há espaço livre no vetor? Sim, portanto o elemento poderá ser inserido. Mas onde? O índice do elemento [2, 5] mostra que a inserção deve ser feita entre os elementos ME[2, 4] = 2 e ME[2, 6] = 3.

Mas ali não há posição livre; e nem deveria haver. Então, para que a inserção seja possível, devemos deslocar os elementos subseqüentes a ME[2, 4] = 2 uma posição para a direita, a fim de criar uma posição livre onde será inserido o elemento ME[2, 5] = 4.

VETME = [(1, 1, 6) (1, 3, 3) (2, 4, 2) (, ,) (2, 6, 3) (3, 2, 1) (3, 4, 5) (, ,)]

VETME = [(1, 1, 6) (1, 3, 3) (2, 4, 2) (2, 5, 4) (2, 6, 3) (3, 2, 1) (3, 4, 5) (, ,)]

- Alteração de um elemento:

Analisemos, agora, a inserção do elemento ME[1, 3] = 0.

O índice já existe em VETME? Caso a resposta seja sim, então se trata de uma alteração. Assim, temos de nos preocupar com dois possíveis casos:

- O elemento será alterado para um valor nulo;
- O elemento será alterado para um valor não nulo.

O novo valor ser nulo equivale a remover o elemento de VETME. Assim, todos os elementos não nulos à direita do elemento devem ser movidos uma posição para a esquerda, sendo que o último deverá receber valores nulos.

VETME = [(1, 1, 6) (1, 3, 3) (2, 4, 2) (2, 5, 4) (2, 6, 3) (3, 2, 1) (3, 4, 5) (, ,)]

VETME = [(1, 1, 6) (2, 4, 2) (2, 5, 4) (2, 6, 3) (3, 2, 1) (3, 4, 5) (3, 4, 5) (, ,)]

VETME = [(1, 1, 6) (2, 4, 2) (2, 5, 4) (2, 6, 3) (3, 2, 1) (3, 4, 5) (, ,) (, ,)]

Se o elemento será alterado para um valor não nulo, basta localizá-lo em VETME e alterar o terceiro elemento da tupla. Por exemplo, o elemento VETME[2, 5] = 8.

VETME = [(1, 1, 6) (2, 4, 2) (2, 5, 8) (2, 6, 3) (3, 2, 1) (3, 4, 5) (3, 4, 5) (, ,)]

Exercícios:

6) Escreva o algoritmo de uma função para inserir/alterar elementos numa matriz esparsa.

```
início
   CONST NUM = 8;
   tipo ELEMENTO = registro
                     LINHA : inteiro;
                     COLUNA : inteiro;
                     VALOR : inteiro;
                   fim registro;
```

```
   tipo MATESP = vetor [1..NUM] de ELEMENTO;

   função INSERCAO (I, J, VALOR : inteiro; MATRIZ : MATESP) : lógico;
   var
      K : inteiro;

      se (BUSCA(I, J, K, MATRIZ) = 0) então      {posição está vazia}
         {K retornará a posição onde está ou será inserido o elemento}
         se (VALOR <> 0) então                   {Há o que incluir!}
            se (HAESPACO(MATRIZ)) então          {É possível incluir!}
               se (EHULTIMO(I, J, MATRIZ)) então {Última posição}
                  MATRIZ[K] ← VALOR;
                  INSERCAO ← Verdadeiro;
               senão
                  DESLOCADIREITA (K, MATRIZ); {Abre espaço no vetor}
                  MATRIZ[K] ← VALOR;
                  INSERCAO ← Verdadeiro;
               fim se;
            senão
               INSERCAO ← Falso;
            fim se;
         senão
            INSERCAO ← Verdadeiro;
         fim se;
      senão                                  {posição está ocupada}
         se (VALOR <> 0) então               {alterar o valor da posição}
            MATRIZ[K] ← VALOR;
         senão
            DESLOCAESQUERDA (K, MATRIZ); {desloca para esquerda e}
                                         {excluiu o último da direita}
         fim se;
      fim se;
   fim função INSERCAO;
fim.
```

UNIDADE 3 – LISTAS, FILAS E PILHAS

3.1 LISTAS LINEARES

Freqüentemente nos deparamos, na solução de determinados problemas, com conjuntos de dados que se relacionam entre si de alguma forma, refletindo algumas propriedades que estes dados apresentam no problema real. Naturalmente, desejamos que este relacionamento seja preservado, com o objetivo de se poder fazer uso do mesmo, quando estes forem representados no computador.

Considere, por exemplo, o caso de um problema que envolva a manipulação de dados de precipitações pluviométricas diárias de um período de um mês. Se o problema consistir em obter-se, digamos, a precipitação pluviométrica média do período, não há nenhuma necessidade de se conhecer o relacionamento existente entre os dados diários. Se, entretanto, o problema consistir em determinar uma função que expresse o fenômeno (por exemplo, precipitação x tempo) no período, então é necessário conhecer-se a relação de ordem cronológica dos dados.

3.1.1 FUNDAMENTOS

Uma lista linear é uma coleção L: $[a_1, a_2, ..., a_n]$, $n \geq 0$, cuja propriedade estrutural baseia-se apenas na posição relativa dos elementos, que são dispostos linearmente. Se n = 0, dizemos que a lista L é **vazia**; caso contrário, são válidas as seguintes propriedades:

- a_1 é o primeiro elemento de L;
- a_n é o último elemento de L;
- a_k, com $1 < k < n$, é precedido pelo elemento a_{k-1} e seguido pelo elemento a_{k+1} em L.

Em outras palavras, a característica fundamental de uma lista linear é o sentido de ordem unidimensional dos elementos que a compõem. Uma ordem que nos permite dizer com precisão onde a coleção inicia-se e onde termina, sem possibilidade de dúvida.

Entre as diversas operações que podemos realizar sobre listas, temos:

- acessar um elemento qualquer da lista;
- inserir um elemento numa posição específica da lista;
- remover um elemento de uma posição específica da lista;
- combinar duas listas em uma única;
- particionar uma lista em duas;
- obter cópias de uma lista;
- determinar o total de elementos na lista;
- ordenar os elementos da lista;
- apagar uma lista
- outras ...

Considerando-se somente as operações de acesso, inserção e remoção, restritas aos extremos da lista, temos casos especiais que aparecem muito freqüentemente na modelagem de problemas a serem resolvidos por computador. Tais casos especiais recebem também nomes especiais:

- **Pilha**: lista linear onde todas as inserções, remoções e acessos são realizados em um único extremo da lista. Listas com esta característica são também denominadas listas **LIFO** (Last-In/First-Out, ou em português: último que entra/primeiro que sai);
- **Fila**: lista linear onde todas as inserções são feitas num certo extremo e todas as remoções e acessos são realizados no outro. Filas são também denominadas de listas **FIFO** (First-In/First-Out, ou em português: primeiro que entre/primeiro que sai);
- **Fila Dupla**: lista linear onde as inserções, remoções ou acessos são realizados em qualquer extremo. Filas duplas são também denominadas **DEQUE** (Double-Ended QUEue, ou em português: fila de extremidade dupla). Uma Fila Dupla pode ainda gerar dois casos especiais: Fila Dupla de Entrada Restrita (se a inserção for restrita a um único extremo) e Fila Dupla de Saída Restrita (se a remoção for restrita a um único extremo).

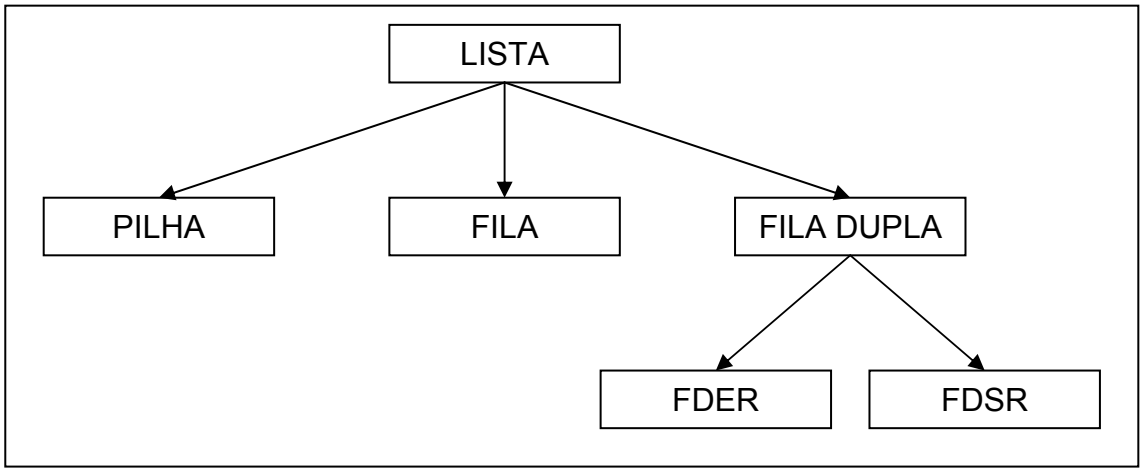


FIGURA 3.1 – Casos especiais de listas lineares

3.1.2 REPRESENTAÇÕES

Existem várias formas possíveis de se representar internamente uma lista linear. A escolha de uma destas formas dependerá da freqüência com que determinadas operações serão executadas sobre a lista, uma vez que algumas representações são favoráveis a algumas operações, enquanto que outras não o são, no sentido de exigir um maior esforço computacional para a sua execução.

A seguir, serão discutidas as duas formas mais freqüentes usadas para representar listas lineares: por *alocação seqüencial* e por *encadeamento dos nós*.

3.1.3 REPRESENTAÇÃO DE LISTAS POR ALOCAÇÃO SEQÜENCIAL

A representação por alocação seqüencial explora a seqüencialidade da memória do computador, de tal forma que os nós de uma lista sejam armazenados em *endereços* contíguos, ou igualmente distanciados um do outro. Neste caso, a relação de ordem entre os nós da lista é representada pelo fato de que se o endereço do nó x_i

é conhecido, então o endereço do nó x_{i+1} pode ser determinado.

Esquematicamente, a representação de uma lista linear por alocação seqüencial tem a forma mostrada na Figura 3.2, abaixo:

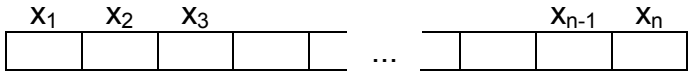


FIGURA 3.2 – Esquema de representação por alocação seqüencial

Esta estrutura é a mesma do agregado homogêneo (vetor). Assim uma lista de *N* nós x_1, x_2, ..., x_n é definida da seguinte forma:

tipo LISTA = vetor [1..N] de tipo;

onde tipo é o tipo de dado a ser representado em cada nó da lista.

Para, finalmente, declararmos uma lista do tipo acima definido, escrevemos:

var X : LISTA;

O comprimento de uma lista (quantidade de nós) pode se modificar durante a execução do programa; assim, entenderemos uma lista como sendo um vetor com N elementos dentro de um vetor com M elementos, onde $M \geq N$.

A seguir são apresentados os algoritmos para implementar algumas das operações mencionadas na seção 3.1.1, sobre listas lineares representadas por alocação seqüencial. Todas as operações serão baseadas em listas do tipo anteriormente definido.

3.1.4 ACESSAR O K-ÉSIMO NÓ DE UMA LISTA

Esta operação pode ser implementada através de uma única construção: X[K], que pode aparecer em uma expressão qualquer. Nesta operação pode ocorrer, entretanto, que K > FIM ou $K \leq 0$; isto é, uma tentativa de acessar um nó que não existe na lista. Para prevenir esta situação, podemos implementar a operação através de uma função que testa a validade de K e retorne um sinal que indique a ocorrência destas situações anômalas, da seguinte forma:

i) se o retorna da função é F então $K \leq 0$ ou K > FIM; (neste caso a operação não foi executada);
ii) se o retorna da função é V, então $0 < K \leq$ FIM; (neste caso a operação foi executada e VAL contém o dado do k-ésimo nó).

Esta convenção também será usada nas demais operações.

```
função ACESSAR(X: LISTA; K, FIM: inteiro; var VAL: tipo):lógico;
   se ((K <= 0) ou (K > FIM)) então
      ACESSAR ← Falso;
   senão
      VAL ← X[K];
      ACESSAR ← Verdadeiro;
   fim se;
fim função ACESSAR;
```

3.1.5 ALTERAR O VALOR DO K-ÉSIMO NÓ DE UMA LISTA

```
função ALTERAR (var X : LISTA; K, FIM :inteiro;
                VAL: tipo):lógico;

   se ((K <= 0) ou (K > FIM)) então
      ALTERAR ← Falso;
   senão
      X[K] ← VAL;
      ALTERAR ← Verdadeiro;
   fim se;

fim função ALTERAR;
```

3.1.6 INSERIR UM NOVO NÓ ANTES DO K-ÉSIMO NÓ DE UMA LISTA

Neste procedimento suporemos que no vetor X haja pelo menos um elemento disponível para acomodar o novo nó; ou seja, assumiremos que FIM < M, onde M é o comprimento do vetor X.

```
função INSERIR (var X : LISTA; K, var FIM :inteiro;
                VAL: tipo):lógico;
var
   I : inteiro;

   se ((K <= 0) ou (K > FIM)) então
      INSERIR ← Falso;
   senão
      para I = FIM até K faça {contador decrescente}
         X[I + 1] ← X[I];
      fim para;

      FIM ← FIM + 1;
      X[K] ← VAL;
      INSERIR ← Verdadeiro;
   fim se;

fim função INSERIR;
```

3.1.7 REMOVER O K-ÉSIMO NÓ DE UMA LISTA

```
função REMOVER (var X : LISTA; K, var FIM :inteiro):lógico;
var
   I : inteiro;

   se ((K <= 0) ou (K > FIM)) então
      REMOVER ← Falso;
   senão
      para I = K até (FIM - 1) faça
```

```
            X[I] ← X[I + 1];
        fim para;

        FIM ← FIM - 1;
        REMOVER ← Verdadeiro;
    fim se;

fim função REMOVER;
```

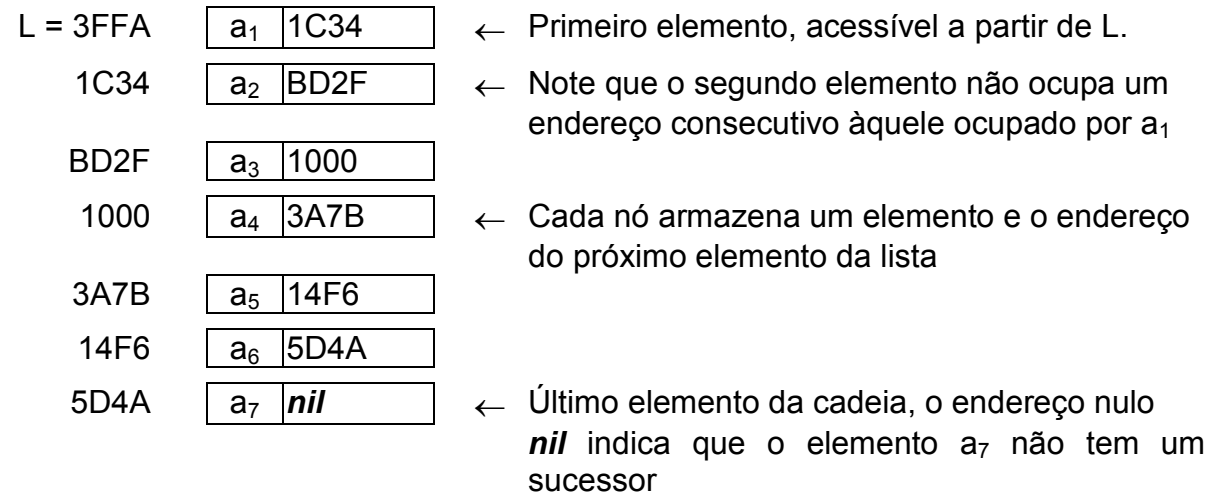
3.1.8 REPRESENTAÇÃO DE LISTAS POR ENCADEAMENTO DOS NÓS

Ao invés de manter os elementos agrupados numa área contínua de memória, isto é, ocupando células consecutivas, na alocação encadeada os elementos ocupam quaisquer células (não necessariamente consecutivas) e, para manter a relação de ordem linear, juntamente com cada elemento é armazenado o endereço do próximo elemento da lista.

Desta forma, na alocação encadeada, os elementos são armazenados em blocos de memória denominados ***nós***, sendo que cada nó é composto por dois campos: um para armazenar dados e outro para armazenar endereço..

Dois endereços especiais devem ser destacados:

- o endereço do primeiro elemento da lista (L);
- o endereço do elemento fictício que segue o último elemento da lista (***nil***).



A alocação apresenta como maior vantagem a facilidade de inserir ou remover elementos do meio da lista. Como os elementos não precisam estar armazenados em posições consecutivas de memória, nenhum dado precisa ser movimentado, bastando atualizar o campo de ligação do elemento que precede aquele inserido ou removido. Por exemplo, para remover o elemento a_2 da lista representada anteriormente, basta mudar o nó no endereço 3FFA de (a1, 1C34) para (a_1, BD2F). Como apenas o primeiro elemento é acessível diretamente através do endereço L, a grande desvantagem da alocação encadeada surge quando desejamos acessar uma posição específica dentro da lista. Neste caso, devemos partir do primeiro elemento e ir seguindo os campos de ligação, um a um, até atingir a posição desejada. Obviamente, para listas extensas, esta operação pode ter um alto custo em relação a tempo.

3.1.9 ALOCAÇÃO DINÂMICA DE MEMÓRIA

As estruturas de dados vistas até o momento são organizadas de maneira fixa. Isto é, criamos as variáveis e estas contam com um tamanho fixo em memória. Arquivos permitem uma estrutura com um número indeterminado de elementos, porém sempre arrumados na forma de uma seqüência.

O que devemos fazer quando tanto o número de elementos quanto sua forma de organização variam dinamicamente? Para resolver este problema temos necessidade de um mecanismo que permita:

- criar espaço para novas variáveis em tempo de execução;
- definir "ligações" entre estas variáveis, de uma forma dinâmica.

Variáveis dinâmicas não possuem nome próprio, portanto não são referenciadas por seus nomes. A referência a uma variável dinâmica é feita por ponteiros. Um ponteiro é uma variável cujo conteúdo é o endereço de uma posição de memória. Um ponteiro é declarado fornecendo-se o tipo de variável por ele apontada. Ou seja, "P" é um ponteiro para um tipo "T" se houver a declaração:

```
P : ^T;
```

Ao iniciar o programa, o valor de "P" estará indefinido. Existe uma constante predefinida, do tipo ponteiro, chamada ***nil*** que não aponta para objeto algum. Note que o significado de ***nil***: não apontar para objeto algum é diferente de indefinido que significa variável não inicializada.

```
P ← nil;
```

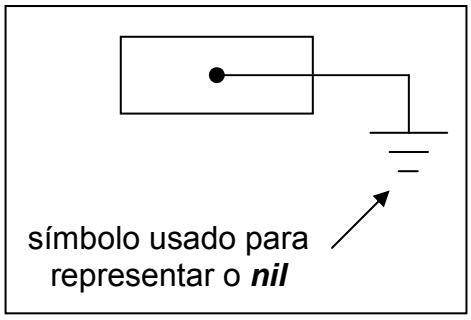


Figura 3.3 – O valor **nil**

A criação de uma variável dinâmica do tipo T é feita pelo operador ***aloque***. Assim, o procedimento padrão:

```
aloque (P)
```

cria uma variável do tipo T, sem nome, e coloca em "P" o endereço desta variável. Graficamente podemos indicar a geração de uma variável dinâmica da seguinte forma:

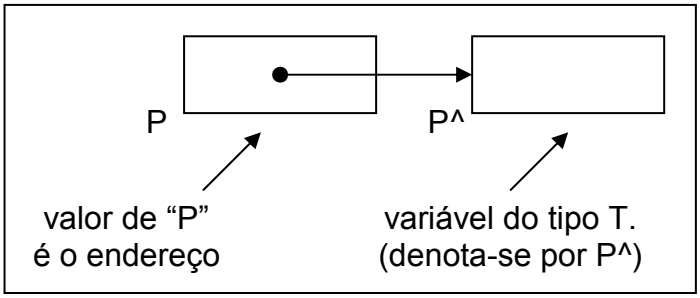


Figura 3.4 – Geração de uma variável dinâmica

Note que a variável dinâmica anterior é referenciada como P^, que significa variável apontada por “P”.

A remoção de uma variável criada dinamicamente, apontada por “P”, pode ser realizada através do seguinte comando:

```
desaloque (P);
```

3.1.10 ENTENDENDO LISTAS REPRESENTADAS ATRAVÉS DE ALOCAÇÃO ENCADEADA DOS NÓS

A estrutura de dados mais simples que pode ser obtida “ligando” elementos com ponteiros é a “lista”.

Na sua forma mais simples, uma lista é uma seqüência de elementos encadeados por ponteiros. Esta seqüência pode ter um número indeterminado de elementos, cujo primeiro está sendo apontado por uma variável “apontador do início da lista”.

Assim, podemos representar graficamente uma lista como:

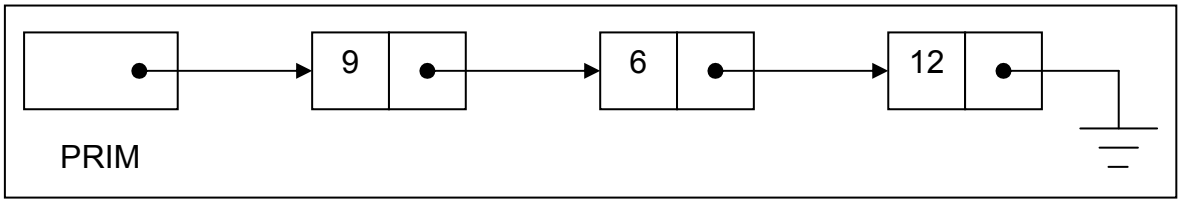


Figura 3.5 – Representação gráfica de uma lista

Neste exemplo podemos identificar:

- o “apontador para o início da lista” que é a variável “PRIM”, cujo conteúdo é o endereço do primeiro elemento;
- uma seqüência com 3 elementos: (9, 6, 12);
- cada elemento da lista tem dois campos: o primeiro é um inteiro e o segundo um apontador para o elemento seguinte na lista;
- o último elemento da lista não aponta para nenhum elemento e seu campo de apontador tem o valor ***nil***.

3.1.11 ROTINAS BÁSICAS DE TRATAMENTO DE LISTAS

Vamos supor uma lista como a mostrada no exemplo acima. Ela pode ser definida como:

```
tipo PONTEIRO = ^ELEMENTO;
tipo ELEMENTO = registro
                   CHAVE : inteiro;
                   PROX : PONTEIRO;
                fim registro;

var
P, PRIM : PONTEIRO;
```

Existem algumas operações que podem ser realizadas com listas.

Destacam-se a criação de uma lista, a procura de um elemento, a inserção de um elemento, a retirada de um elemento e a varredura na lista, processando seus elementos.

Adotaremos que nesta lista os elementos são inseridos na ordem inversa em que são obtidos.

Para melhor compreendermos o problema vamos analisar um exemplo. Suponha a lista:

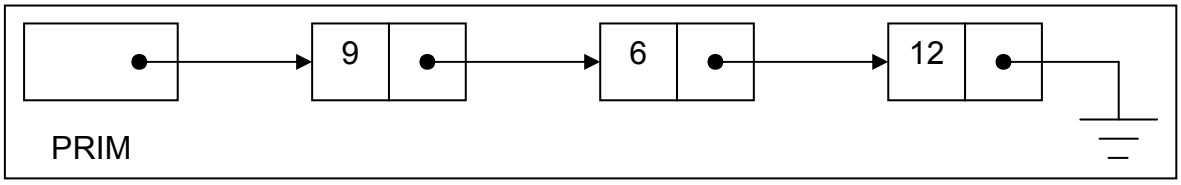


Figura 3.6 – Exemplo de uma lista

e desejamos acrescentar um elemento com chave = 5. Para isso devemos fazer:

1) criar um novo elemento apontado por “P”:

```
aloque (P)
```

2) atualizar o campo chave:

```
P^.CHAVE ← 5;
```

3) colocar o elemento no início da lista:

```
P^.PROX ← PRIM;
```

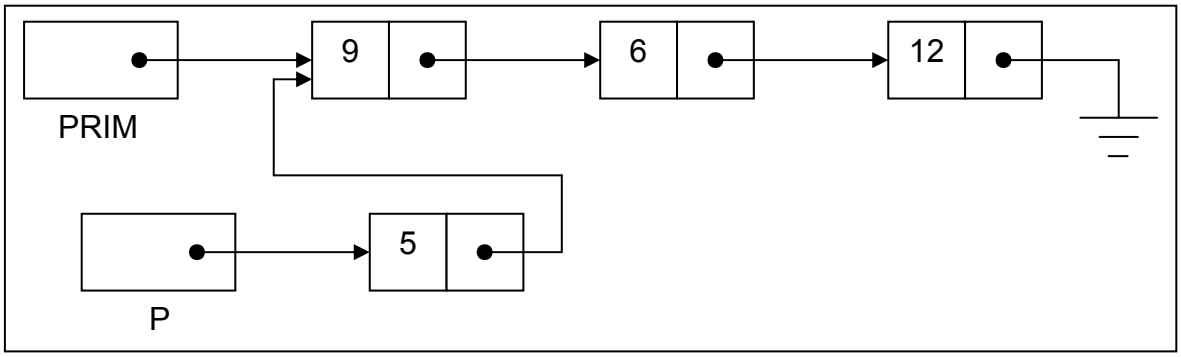


Figura 3.7 – Inserção de um novo elemento na lista

4) atualizar o início da lista:

```
PRIM ← P;
```

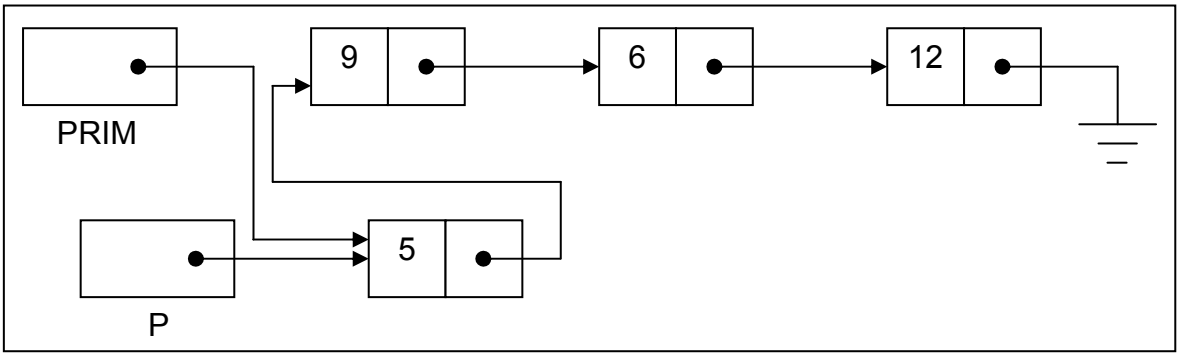


Figura 3.8 – Atualização do ponteiro PRIM completando a inserção do novo elemento

Assim uma rotina para gerar uma lista de inteiros a partir de um arquivo de entrada, que contém números inteiros, pode ser:

```
procedimento CRIALISTA (MEUARQUIVO : ARQINTEIROS;
                        var PRIM: PONTEIRO);
var
P : PONTEIRO;
NUMERO : inteiro;

   abra (MEUARQUIVO);
   PRIM ← nil;

   enquanto não fda(MEUARQUIVO) faça
      aloque (P);
      copie (MEUARQUIVO, NUMERO);
      P^.CHAVE ← NUMERO;
      P^.PROX ← PRIM;
      PRIM ← P;
      avance (MEUARQUIVO);
   fim enquanto;

fim procedimento CRIALISTA;
```

A função abaixo procura um elemento, cujo campo chave seja igual ao dado passado como parâmetro, retornando um ponteiro que aponta para aquele dado em memória.

```
função BUSCA (PRIM: PONTEIRO; DADO: inteiro): PONTEIRO;
P: PONTEIRO;
NAOACHOU: lógico;

   P ← PRIM;
   NAOACHOU ← V;

   enquanto (P <> nil) e (NAOACHOU) faça
      se P^.CHAVE = DADO então
         NAOACHOU ← F
      senão
         P ← P^.PROX;
      fim se;
   fim enquanto;

   BUSCA ← P;

fim função BUSCA;
```

Caso o elemento não exista na lista a função BUSCA retornará ***nil***.

Para inserirmos um elemento na lista devemos considerar dois casos:

a) a inserção deve ser feita após o elemento aponta por “P”;
b) a inserção deve ser feita antes do elemento apontado por “P”.

Caso a - Inserção feita após o elemento apontado por “P”:

Esta representação mostra a variável “P” apontando para o elemento cujo campo de informação vale 3 e a variável “AUX”, do mesmo tipo de “P”, apontando para um elemento a ser inserido. A inserção implica fazer o campo “PROX” da variável apontada por “AUX” apontar para o sucessor de “P^” e fazer “P^” apontar para o novo elemento. Ou seja:

```
AUX^.PROX ← P^.PROX;
P^.PROX ← AUX;
```

Antes:

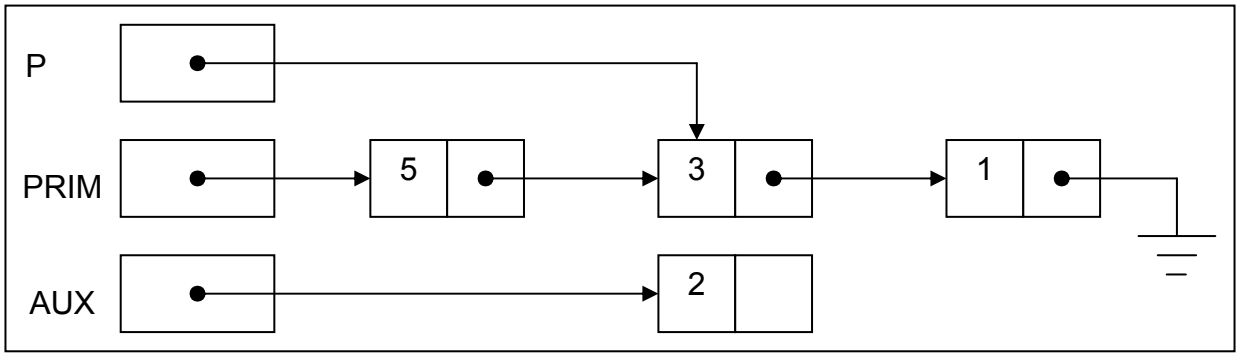


Figura 3.9 – Inserção feita após um elemento apontado por P (Antes)

Depois:

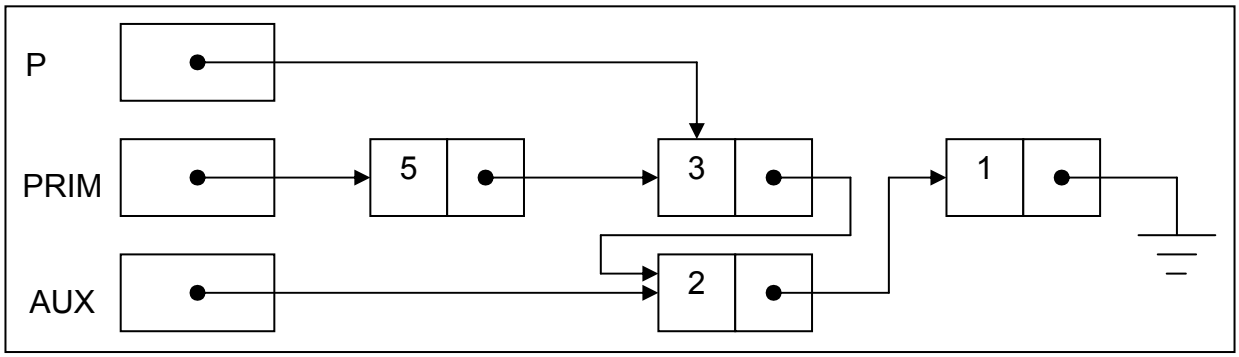


Figura 3.10 - Inserção feita após um elemento apontado por P (Depois)

Caso b - Inserção feita antes do elemento apontado por “P”:

Procedemos da seguinte maneira: criamos um novo elemento, apontado por AUX. Inserimos esse elemento após o apontado por “P”. Copiamos o campo de informação do elemento apontado por “P” para o novo elemento da lista. Colocamos a nova informação, que está na variável “DADO” no elemento apontado por “P”. Ou seja:

```
AUX^.PROX ← P^.PROX;
P^.PROX ← AUX;
AUX^.CHAVE ← P^.CHAVE;
P^.CHAVE ← DADO;
```

Antes:

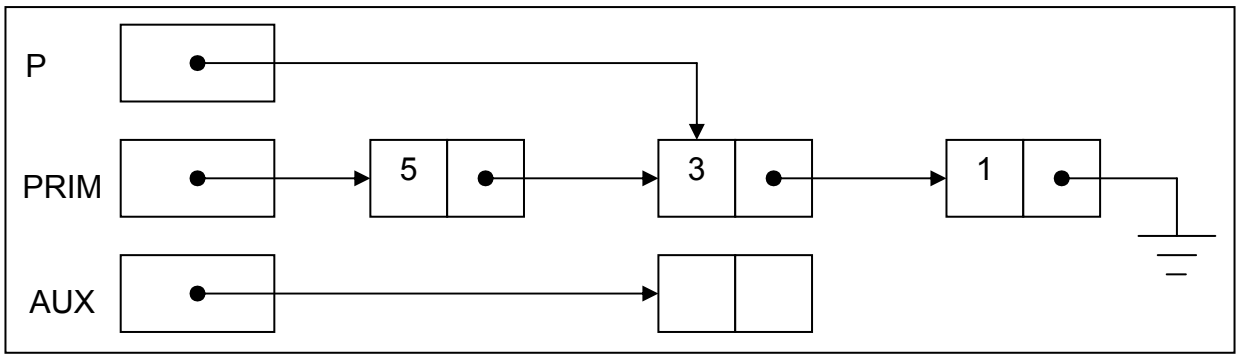


Figura 3.11 - Inserção feita antes de um elemento apontado por P (Antes)

Depois:

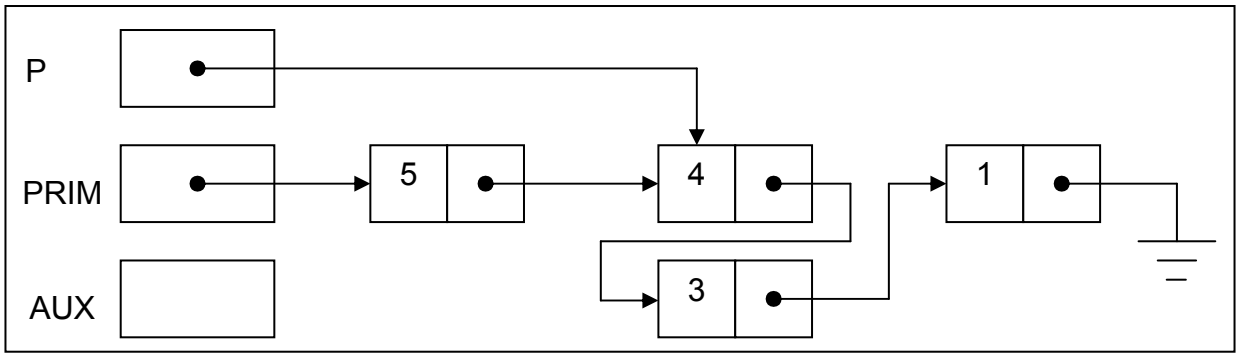


Figura 3.12 - Inserção feita antes de um elemento apontado por P (Depois)

Para remover um elemento apontado por “P” procedemos de maneira similar ao que foi feito anteriormente. Seja a lista:

Antes:

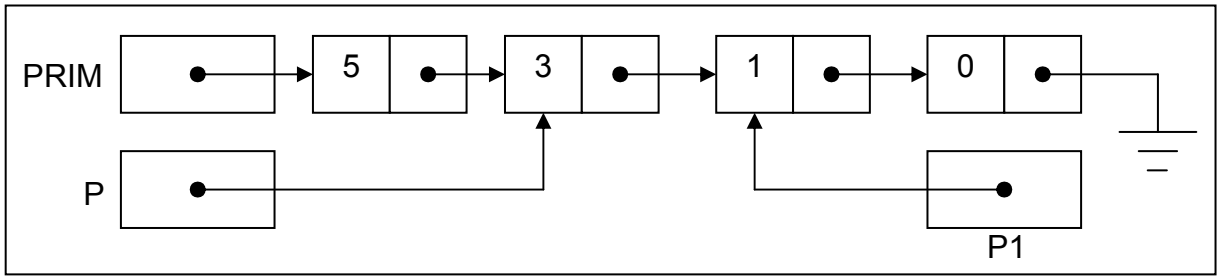


Figura 3.13 – Remoção de um elemento apontado por P (Antes)

Depois:

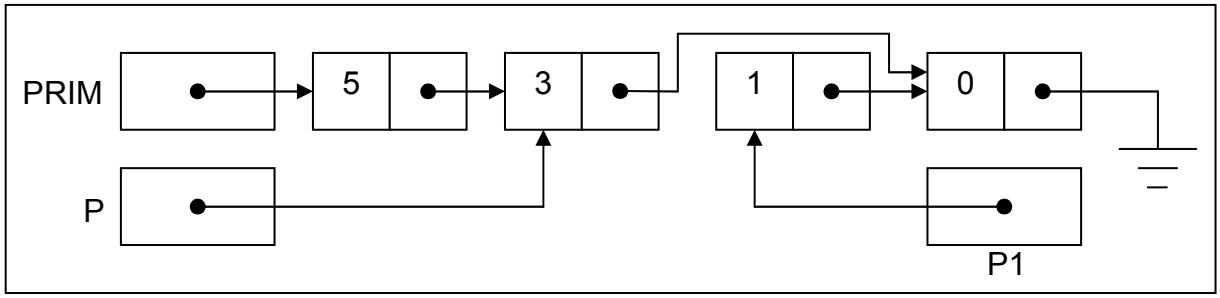


Figura 3.14 – Remoção de um elemento apontado por P (Depois)

Retirar o elemento cujo campo de informação vale 3 é o mesmo que copiar para ele o campo de informação do elemento seguinte e retirá-lo da lista. Usando-se uma variável “P1”, do mesmo tipo de “P”, temos o seguinte trecho de algoritmo:

```
P1 ← P^.PROX;
P^.CHAVE ← P1^.CHAVE;
P^.PROX ← P1^.PROX;
desaloque (P1);
```

Para percorrer a lista, processando os elementos, vamos considerar que o processamento de cada elemento é feito pelo procedimento “PROCESSA”, que recebe como parâmetro o campo de informações do elemento. Fazemos “P” apontar para o início da lista e enquanto houver elemento na lista, chama-se “PROCESSA” e atualiza-se o valor de “P”, que passa a apontar para o próximo elemento.

```
P ← PRIM;
enquanto P <> nil faça
   PROCESSA (P);
   P ← P^.PROX;
fim enquanto;
```

3.1.12 LISTAS COM DESCRITOR

Podemos simplificar a representação de uma lista se reunirmos, em um único elemento, as referências ao primeiro e último elemento da lista.

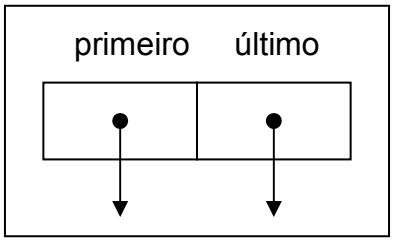


Figura 3.15 – Um nó descritor

A este elemento que reúne as referências ao início e ao fim da lista damos a denominação de *nó descritor*. O acesso aos elementos da lista será sempre efetuado através do seu descritor.

O nó descritor de uma lista pode conter outras informações sobre a lista, a critério do projetista, tais como: quantidade de nós na lista, descrição dos dados contidos nos nós, etc.

A figura, a seguir, mostra esquematicamente uma lista encadeada com nó descritor, no qual foi incluído um campo que indica a quantidade de nós existentes na lista. Nesta nova estrutura, a variável PRIM aponta para o nó descritor e não para o primeiro nó da lista.

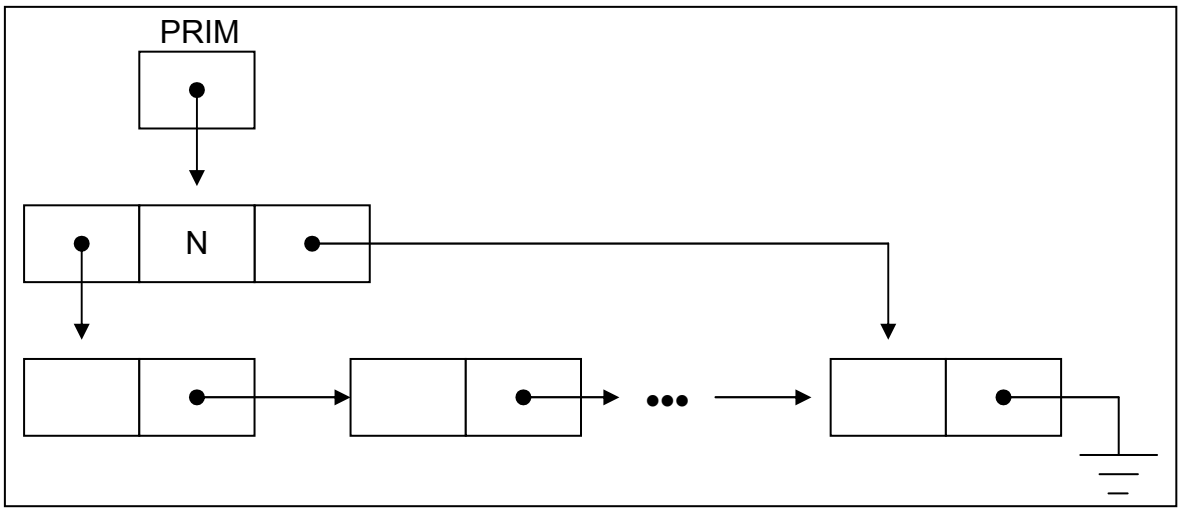


Figura 3.16 – Uma lista encadeada com nó-descritor

O nó descritor, neste caso, é um dado com a seguinte definição:

```
tipo DESCRITOR = registro
                    I : PONTEIRO;
                    N : inteiro;
                    F : PONTEIRO;
                 fim registro;
```

Uma lista vazia passa a ser representada, agora, da seguinte forma:

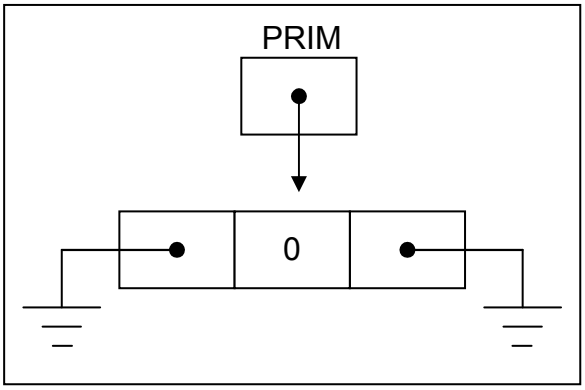


Figura 3.17 – Representação de uma lista vazia, implementada com nó descritor

Usando esta nova estrutura de lista encadeada, passa a ser necessária a existência de uma nova operação: criar uma lista vazia. Esta operação consiste em alocar um nó do tipo descritor (PRIM), tornar seus dois ponteiros nulos e atribuir o valor zero ao campo N, gerando-se desta forma a situação mostrada na Figura 3.17.

O procedimento criar, abaixo, implementa esta operação:

```
procedimento CRIAR (var PRIM : ^DESCRITOR)
   aloque (PRIM);
   PRIM^.I ← nil;
   PRIM^.N ← 0;
   PRIM^.F ← nil;
fim procedimento CRIAR;
```

A seguir são apresentados outros procedimentos para manipulação de listas encadeadas com descritor. O primeiro procedimento, INSERE_ESQ, implementa a operação de inserção de um nó com o dado VALOR à esquerda da lista cujo descritor é apontado pela variável PRIM.

```
Procedimento INSERE_ESQ (PRIM: ^DESCRITOR; VALOR : inteiro);
var
   P : ^ELEMENTO;
   aloque (P);
   P^.CHAVE ← VALOR;

   se PRIM^.N = 0 então {testa se a lista está vazia}
      PRIM^.I ← P;
      PRIM^.F ← P;
      PRIM^.N ← 1;
      P^.PROX ← nil;
   senão
      P^.PROX ← PRIM^.I;
      PRIM^.I ← P;
      PRIM^.N ← PRIM^.N + 1;
   fim se;

fim procedimento INSERE ESQ;
```

O procedimento seguinte, INSERE_DIR, implementa a operação de inserção de um nó com o dado VALOR à direita da lista cujo descritor é apontado por PRIM.

```
Procedimento INSERE_DIR (PRIM: ^DESCRITOR; VALOR : inteiro);
var
   P, Q : ^ELEMENTO;

   aloque (P);
   P^.CHAVE ← VALOR;
   P^.PROX ← nil;

   se PRIM^.N = 0 então {testa se a lista está vazia}
      PRIM^.I ← P;
      PRIM^.F ← P;
      PRIM^.N ← 1;
   senão
      Q ← PRIM^.F;
      PRIM^.F ← P;
      Q^.PROX ← P;
      PRIM^.N ← PRIM^.N + 1;
   fim se;

fim procedimento INSERE DIR;
```

Para remover o nó da esquerda de uma lista, podemos utilizar o procedimento REMOVE_ESQ, a seguir apresentado. Este procedimento remove o primeiro nó da lista, se houver, e retorna o dado que o nó removido continha através do

parâmetro VALOR.

```
Procedimento REMOVE_ESQ (PRIM: ^DESCRITOR; var VALOR : inteiro);
var
   P : ^ELEMENTO;

   se PRIM^.N = 0 então {testa se a lista está vazia}
      ERRO(3);          {chama rotina que trata o erro}
   senão
      P ← PRIM^.I;
      VALOR ← P^.CHAVE;
      PRIM^.I ← P^.PROX;
      PRIM^.N ← PRIM^.N – 1;
      desaloque (P);

      se PRIM^.N = 0 então {testa se a lista ficou vazia}
         PRIM^.F ← nil;
      fim se;
   fim se;

fim procedimento REMOVE_ESQ;
```

A operação de remoção do nó à direita da lista com descritor envolve o mesmo problema apresentado para o caso de lista sem descritor: a necessidade de percorrer todos os nós da lista, seqüencialmente, a partir do primeiro (da esquerda), até atingir o nó da direita. A fim de evitar a necessidade deste caminhamento, podemos estruturar uma lista linear encadeada de forma que o caminhamento sobre a mesma possa ser feito em ambos os sentidos. Uma organização que implementa este recurso é denominada de *lista linear duplamente encadeada*.

3.1.13 LISTAS DUPLAMENTE ENCADEADAS

Uma lista linear duplamente encadeada é aquela em que cada nó possui dois ponteiros, ao invés de um só. O primeiro é usado para indicar o nó predecessor, enquanto que o segundo aponta para o nó sucessor. A Figura 3.18 mostra esquematicamente uma lista linear duplamente encadeada com 3 nós.

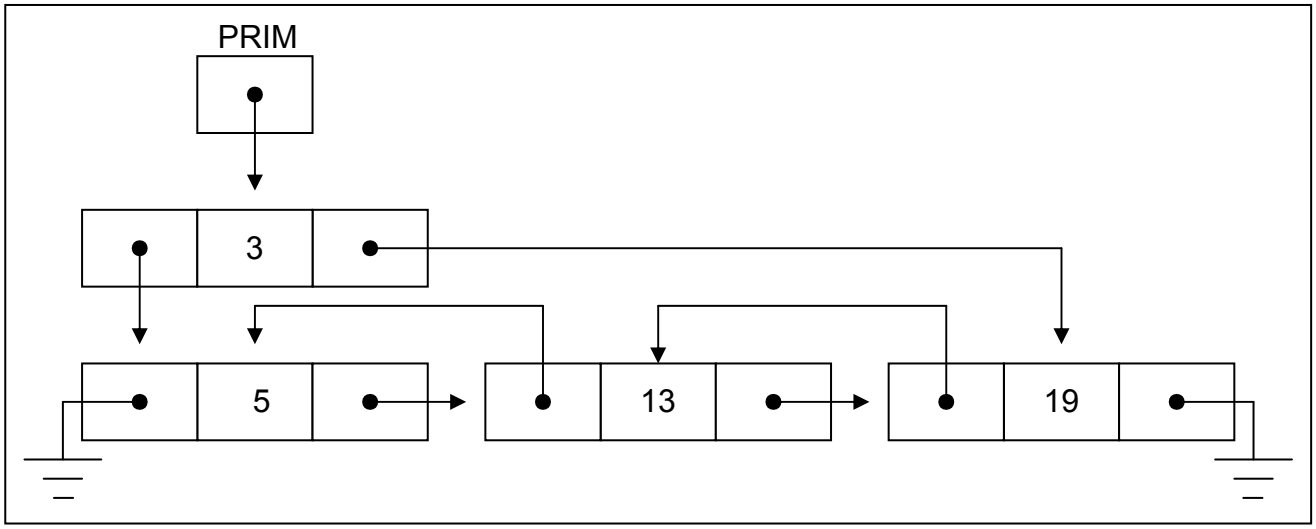


Figura 3.18 – Uma lista linear duplamente encadeada com 3 nós

A definição do tipo dos nós de uma lista duplamente encadeada é feita da seguinte forma:

```
tipo ELEMENTO = registro
                   ESQ : PONTEIRO;
                   CHAVE : inteiro;
                   DIR : PONTEIRO;
                fim registro;
```

Se unirmos as duas extremidades livres da lista, obteremos uma *lista circular* , conforme mostrado na Figura 3.19.

Em uma lista circular, cada nó satisfaz a seguinte condição:

(P^.DIR)^.ESQ = P = (P^.ESQ)^.DIR

onde P é a referência a um nó qualquer da lista.

Agora, com a nova organização proposta, a implementação da operação de remoção do nó da direita fica simplificada, no sentido de não ser mais necessário o caminhamento linear sobre a lista.

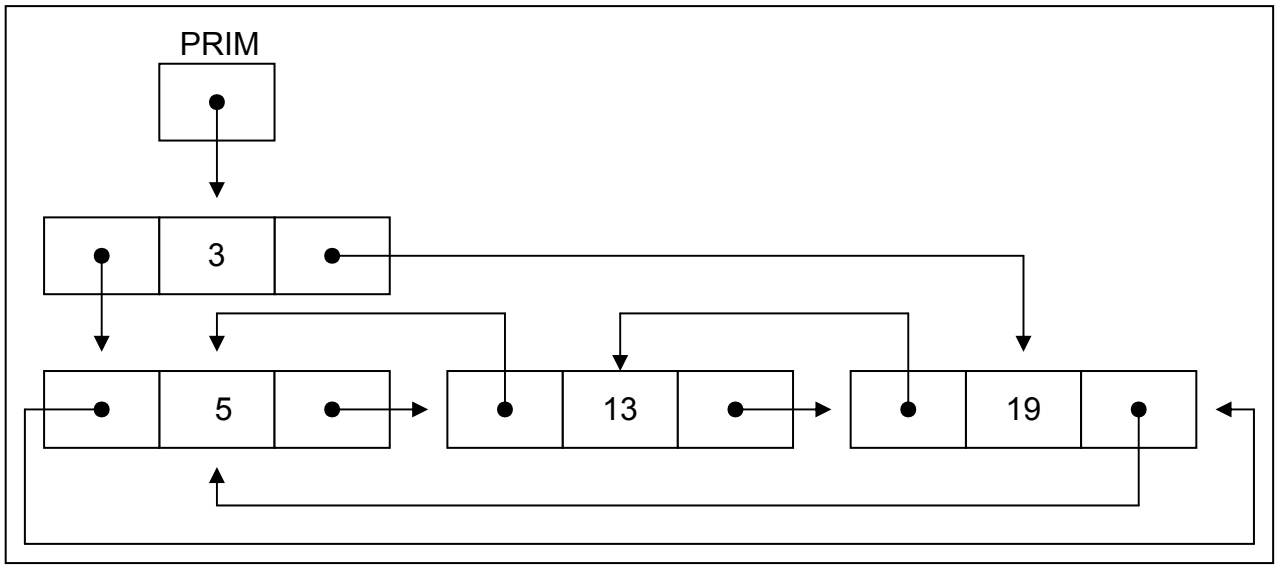


Figura 3.19 – Representação de uma lista circular com 3 nós

O procedimento REMOVE_DIR, apresentado a seguir, implementa esta operação, sobre uma lista circular.

```
procedimento REMOVE_DIR (PRIM: ^DESCRITOR; var VALOR : inteiro);
var
   P, Q, R : ^ELEMENTO;

   se PRIM^.N = 0 então {testa se a lista está vazia}
      ERRO(3);          {chama rotina que trata o erro}
   senão
      P ← PRIM^.F;     {se lista não vazia, remove}
      VALOR ← P^.CHAVE;
```

```
      se PRIM^.N = 1 então {testa se a lista tem só um nó}
         PRIM^.I ← nil;
         PRIM^.F ← nil;
      senão
         Q ← P^.ESQ;
         R ← PRIM^.I;
         Q^.DIR ← P^.DIR;
         R^.ESQ ← Q;
         PRIM^.F ← Q;
      fim se;

      PRIM^.N ← PRIM^.N - 1;
      desaloque (P);

   fim se;

fim procedimento REMOVE DIR;
```

3.2 PILHAS

A **pilha** é uma das estruturas de dados mais úteis em computação. Uma infinidade de problemas clássicos da área podem ser resolvidos com o uso delas.

Uma **pilha** é um tipo especial de lista linear em que todas as operações de inserção e remoção são realizadas numa mesma extremidade, denominada topo.

Cada vez que um novo elemento deve ser inserido na pilha, ele é colocado no seu topo; e em qualquer momento, apenas aquele posicionado no topo da pilha pode ser removido. Devido a esta disciplina de acesso, os elementos são sempre removidos numa ordem inversa àquela em que foram inseridos, de modo que o último elemento que entra é exatamente o primeiro que sai. Daí o fato de estas listas serem também denominadas **LIFO** (Last-In/First-Out).

O exemplo mais comum do quotidiano é uma pilha de pratos, onde o último prato colocado é o primeiro a ser usado (removido).

Uma pilha suporta três operações básicas, tradicionalmente denominadas como:

- **Top**: acessa o elemento posicionado no topo da pilha;
- **Push**: insere um novo elemento no topo da pilha;
- **Pop**: remove um elemento do topo da pilha.

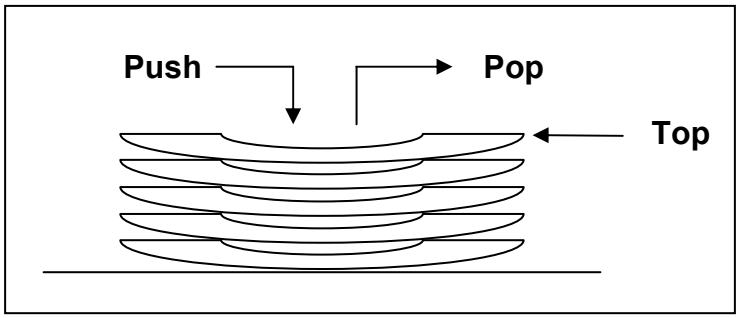


Figura 3.20 – Uma pilha de pratos

Sendo **P** uma pilha e **x** um elemento qualquer, a operação **Push (P, x)** aumenta o tamanho da pilha P, acrescentando o elemento **x** no seu topo. A operação **Pop(P)** faz com que a pilha diminua, removendo e retornando o elemento existente no seu topo. Das três operações básicas, a única que não altera o estado da pilha é **Top(P)**; ela simplesmente retorna uma cópia do elemento existente no topo da pilha, sem removê-lo.

Observe a seguir, como estas operações interagem para alterar o estado de uma pilha **P**, inicialmente vazia, cuja extremidade esquerda foi escolhida como topo.

Operação	Estado da Pilha	Resultado
—————————	P: [ ]	—————————
Push (P, a)	P: [a]	—————————
Push (P, b)	P: [b, a]	—————————
Push (P, c)	P: [c, b, a]	—————————
Pop (P)	P: [b, a]	c
Pop(P)	P: [a]	b
Push (P, d)	P: [d, a]	—————————
Push (P, e)	P: [e, d, a]	—————————
Top (P)	P: [e, d, a]	e
Pop (P)	P: [d, a]	e
Pop(P)	P: [a]	d

Para melhor visualização, ao invés de utilizar a notação de lista linear, geralmente as pilhas são representadas na forma de um gráfico, crescendo na vertical, de baixo para cima, conforme o esquema a seguir:

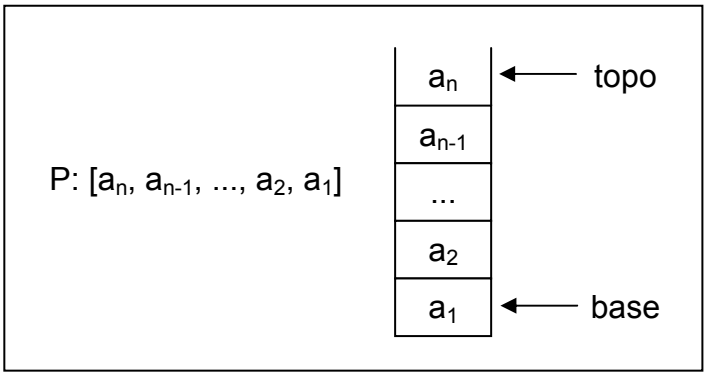


Figura 3.21 – Notação linear e gráfica de pilhas

Temos a seguir, representados na forma gráfica, os sucessivos estados que uma pilha assume quando novos elementos são nela colocados ou dela retirados.

Note que a representação de pilha na forma de lista linear sugere que a posição de topo seja fixa. Isto torna necessário empurrar os elementos para baixo sempre que um novo elemento for colocado na pilha. Ao remover o elemento do topo, os demais elementos devem ser puxados de volta para cima. Já na representação gráfica, fica implícito que, na verdade, é a posição de topo que se movimenta toda vez que a pilha cresce ou diminui.

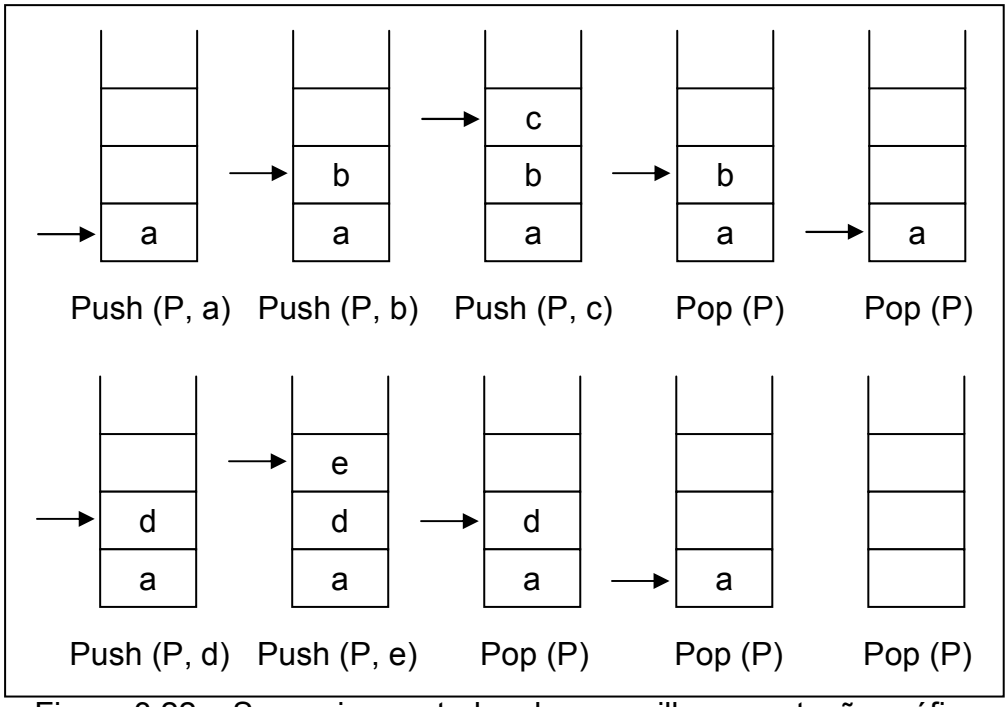


Figura 3.22 – Sucessivos estados de uma pilha na notação gráfica

3.2.1 INIT, ISEMPTY E ISFULL

Imagine uma pilha de pratos sobre uma mesa, dentro de uma sala. Seria possível colocar novos pratos indefinidamente sobre ela? Obviamente, não! Em algum momento o prato do topo da pilha tocaria o teto da sala. E o contrário: seria possível remover pratos do topo da pilha indefinidamente? Também não! Em algum momento a pilha tornar-se-ia vazia. A mesa e o teto são limites físicos que impedem que a pilha cresça ou diminua indefinidamente.

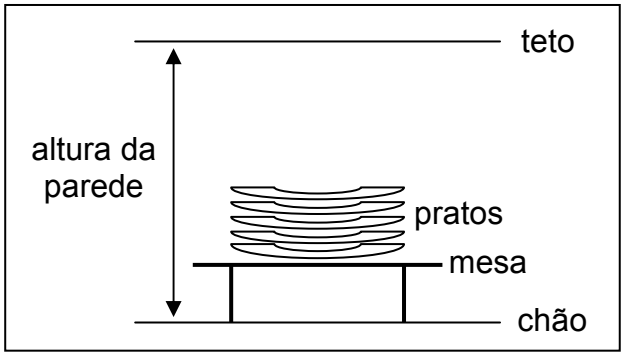


Figura 3.23 – Limitações físicas de uma pilha

No exemplo da Figura 3.23, antes de começarmos a empilhar os pratos sobre a mesa, devemos garantir que a mesa estará limpa, isto é, que não existem panelas, talheres ou qualquer outro objeto no local onde colocaremos os pratos. Para adicionar um prato à pilha, primeiro verificamos se ainda existe espaço entre o topo da pilha e o teto da sala. Finalmente, para remover, precisamos nos certificar de que ainda existem pratos sobre a mesa.

Assim, precisamos de mais três operações essenciais para manipular pilhas:

- **Init**: inicializa a pilha no estado “vazia”;
- **IsEmpty**: verifica se a pilha está vazia;
- **IsFull**: verifica se a pilha está cheia.

Sempre que uma variável é criada, ela permanece com conteúdo indefinido, até que um determinado valor seja a ela atribuído. Geralmente, a criação de uma variável se restringe apenas à alocação da área de memória necessária para representá-la; nenhum valor inicial á armazenado nesta área, até que uma instrução específica para esta finalidade seja executada. No caso de uma variável do tipo **pilha**, isto não é diferente.

A operação **Init(P)** tem como objetivo definir um estado inicial para a pilha **P**. Por uma questão de bom senso, uma pilha sempre é inicializada no estado “vazia”. Toda vez que criamos uma variável pilha, antes de qualquer coisa, devemos inicializá-la para garantir que não haverá nenhuma “sujeira” no local onde ela será “montada”!

Para verificarmos se uma pilha **P** está vazia, podemos usar a função lógica **IsEmpty(P)**, que toma como argumento a pilha em que estamos interessados e retorna verdadeiro somente se ela estiver vazia, sem nenhum elemento armazenado. A função **IsFull(P)** é usada para verificar se uma pilha está cheia, isto é, ela retorna verdadeiro somente quando não há mais espaço para armazenar elementos na pilha.

3.2.2 UM PRIMEIRO EXEMPLO DO USO DE PILHAS

Este primeiro exemplo objetiva mostrar como um programa completo, que utiliza o tipo **pilha**, pode ser escrito. O programa simplesmente pede ao usuário que digite um número inteiro positivo em decimal e, em seguida, mostra o número no sistema binário.

Para entender a lógica do programa, lembre-se de que para converter um número inteiro da base decimal para a binária, devemos dividi-lo sucessivamente por 2, até obtermos um quociente igual a 0. Neste momento, os restos obtidos nas divisões devem ser tomados em ordem inversa. Veja o exemplo:

```
 13 | 2      6 | 2      3 | 2      1 | 2
-12   6     -6   3     -2   1     -0   0
---          ---        ---        ---
 1            0          1          1
```

Assim, o valor 13 decimal fica 1101 em binário. No programa a seguir, a pilha é utilizada para armazenar os restos obtidos, de modo que depois eles possam ser recuperados no ordem inversa em que foram gerados.

```
program DEC_BIN;
uses PILHAS;
var
   P : Pilha;
   x, n : integer;

begin
   writeln ('Digite um inteiro decimal positivo: ');
   readln(n);
```

```
   Init(P);              {torna a pilha vazia}

   repeat
      x := n mod 2;      {calcula o resto}
      Push(P, x);        {empilha o resto}
      n := n div 2;      {calcula o quociente}
   until (n = 0);        {quociente 0, pára!}

   write ('Correspondente ao binário: ');

   while (not IsEmpty(P)) do begin     {pilha vazia, pára!}
      x := Pop(P);       {desempilha o resto}
      write (x);         {imprime o resto}
   end;
end.
```

Exercícios:

7) Escreva os algoritmos para compor uma biblioteca para manipulação de pilhas. Esta biblioteca deve conter os seguintes procedimentos e funções: Init, IsEmpty, IsFull, Top, Push e Pop.

3.2.3 IMPLEMENTAÇÃO SEQÜENCIAL DE PILHAS

Como os elementos da pilha são armazenados em seqüência, um sobre o outro, e a inclusão ou exclusão de elementos não requer movimentação de dados, o esquema de alocação seqüencial de memória mostra-se bastante apropriado para implementá-las. Neste esquema, a forma mais simples de se representar um a pilha na memória consiste em:

- um **vetor**, que serve para armazenar os elementos contidos na pilha;
- um **índice**, utilizado para indicar a posição corrente de topo da pilha.

Para agrupar estas duas partes e formar a estrutura coesa que é a pilha, usaremos o **registro**:

```
const MAX = 50;
tipo ELEM = caractere;
     PILHA = registro
             TOPO : inteiro;
             MEMO : vetor [1..MAX] de ELEM;
      fim registro;

var
   P : PILHA;
```

As duas primeiras linhas de código têm como objetivo tornar a implementação o mais independente possível dos tipos e quantidades de dados manipulados. Por exemplo, se desejássemos uma pilha com capacidade de armazenar até 200 valores reais, bastaria fazer duas pequenas alterações nas linhas 1 e 2, e nenhum algoritmo precisaria ser alterado:

```
const MAX = 200;
tipo ELEM = real;
```

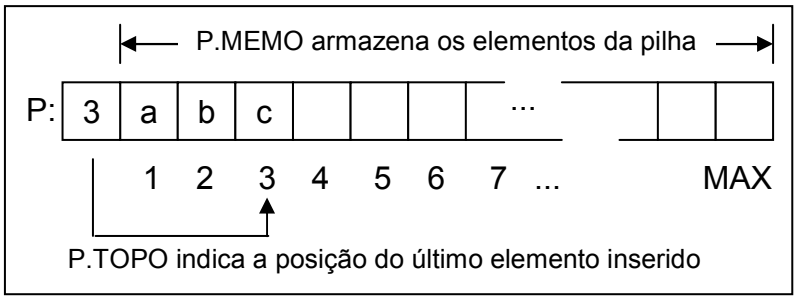


Figura 3.24 – Estrutura de armazenamento da Pilha P: [c, b, a]

Como uma pilha é um registro, podemos acessar seus componentes individuais, usando o operador ponto (.) e o nome do campo em que estamos interessados.

3.2.4 ALGORITMOS PARA MANIPULAÇÃO DE PILHAS

Como sabemos, implementar um tipo de dados não se resume apenas em especificar a estrutura sob a qual os dados serão mantidos na memória, mas requer também o desenvolvimento dos algoritmos que descrevem o funcionamento das operações básicas sobre aquela estrutura.

- **Inicializando a Pilha:**

Inicializar a pilha é atribuir 0 à TOPO, pois 0 é uma posição inexistente no vetor e, além disto, pode ser mudado facilmente para 1 quando o primeiro elemento for inserido na pilha; ou retornar a 0 quando o último elemento for dela removido.

```
procedimento INIT (var P: PILHA);

   P.TOPO ← 0;

fim procedimento INIT;
```

- **Verificando Limites:**

Na implementação seqüencial, a checagem de pilha vazia ou cheia é extremamente simples! Se acabamos de inicializar uma pilha **P**, usando a operação **Init(P)**, é claro que a operação que testa a pilha vazia, **IsEmpty(P)**, deverá retornar um valor lógico verdadeiro.

```
função ISEMPTY(P: PILHA): lógico;
   se (P.TOPO = 0) então
      ISEMPTY ← Verdadeiro;
   senão
      ISEMPTY ← Falso;
   fim se;
fim função ISEMPTY;
```

A operação que testa se a pilha está cheia ou não, também é bastante simples. Se o campo **TOPO** é usado para registrar a posição do elemento que ocupa o topo da pilha, é claro que a pilha estará cheia quando o elemento do topo estiver armazenado na última posição do vetor **MEMO**, representada pela constante **MAX**.

```
função ISFULL(P: PILHA): lógico;

   se (P.TOPO = MAX) então
      ISFULL ← Verdadeiro;
   senão
      ISFULL ← Falso;
   fim se;

fim função ISFULL;
```

- **Empilhando um Elemento:**

Para inserir um novo elemento na pilha, primeiro verificamos se ainda existe espaço. Existindo, temos que colocar o novo elemento no topo da pilha.

```
procedimento PUSH (var P: PILHA; X: ELEM);

   se (não ISFULL(P)) então
      P.TOPO ← P.TOPO + 1;
      P.MEMO[P.TOPO] ← X;
   senão
      escreva ´Stack Overflow!´
   fim se;

fim procedimento PUSH;
```

- **Desempilhando um Elemento:**

Para retirar um elemento de uma pilha, primeiro temos que nos certificar de que ela não se encontra vazia. Caso a pilha não esteja vazia, então o elemento que está no topo deverá ser retornado como resultado da operação e a variável **P.TOPO** deverá ser decrementada.

```
função POP (var P: PILHA): ELEM;

   se (não ISEMPTY(P)) então
      POP ← P.MEMO[P.TOPO];
      P.TOPO ← P.TOPO - 1;
   senão
      escreva ´Stack Underflow!´
   fim se;

fim função POP;
```

- **Obtendo o Valor do Elemento do Topo:**

Algumas vezes, temos a necessidade de apenas “observar” o elemento que se encontra no topo da pilha. A operação **TOP(P)** nos permite fazer tal acesso, sem alterar o estado da pilha.

```
função TOP (P: PILHA): ELEM;

   se (não ISEMPTY(P)) então
      TOP ← P.MEMO[P.TOPO];
   senão
      escreva ´Stack Underflow!´
   fim se;

fim função TOP;
```

3.3 FILAS

Uma fila é um tipo especial de lista linear em que as inserções são realizadas num extremo, ficando as remoções restritas ao outro. O extremo onde os elementos são inseridos é denominado **final** da fila, e aquele onde são removidos é denominado **começo** da fila. As filas são também denominadas listas **FIFO** (First-In/First-Out).

Um exemplo bastante comum de filas verifica-se num balcão de atendimento, onde pessoas formam uma fila para aguardar até serem atendidas. Naturalmente, devemos desconsiderar os casos de pessoas que “furam a fila” ou que desistem de aguardar! Diferentemente das filas no mundo real, o tipo de dados abstrato não suporta inserção nem remoção no meio da lista.

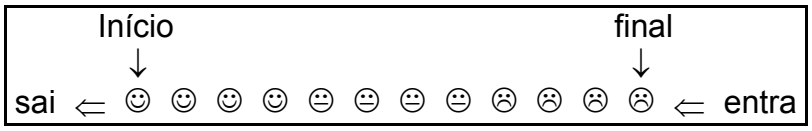


Figura 4.25 – Uma fila de caixa bancário

A palavra **queue**, da língua inglesa, significa fila. Por tradição, as duas operações básicas que uma fila suporta são denominadas como a seguir:

- **Enqueue**: insere um elemento no final da fila;
- **Dequeue**: remove um elemento do começo da fila.

Sendo **F** uma fila e **x** um elemento qualquer, a operação **Enqueue (F, x)** aumenta o tamanho da fila **F**, acrescentando o elemento **x** no seu final. A operação **Dequeue (F)** faz a fila diminuir, já que remove e retorna o elemento posicionado no seu começo.

Operação	Estado da Fila	Resultado
_____	F: [ ]	_____
Enqueue (F, a)	F: [a]	_____
Enqueue (F, b)	F: [a, b]	_____

Operação	Estado da Fila	Resultado
Enqueue (F, c)	F: [a, b, c]	_____
Enqueue (F, d)	F: [a, b, c, d]	_____
Dequeue (F)	F: [b, c, d]	a
Dequeue (F)	F: [c, d]	b
Enqueue (F, e)	F: [c, d, e]	_____
Enqueue (F, f)	F: [c, d, e, f]	_____
Enqueue (F, Dequeue (F))	F: [d, e, f]	c
	F: [d, e, f, c]	_____
Dequeue (F)	F: [e, f, c]	d
Dequeue (F)	F: [f, c]	e
Dequeue (F)	F: [c]	f

3.3.1 IMPLEMENTAÇÃO SEQÜENCIAL DE FILAS

Graficamente, representamos uma fila como uma coleção de objetos que cresce da esquerda para a direita, com dois extremos bem-definidos: começo e final:

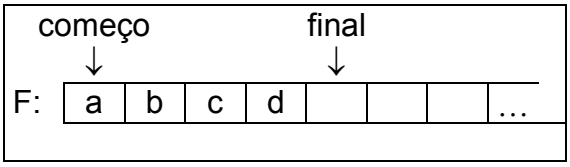


Figura 3.26 – Representação gráfica da fila F: [a, b, c, d]

Intuitivamente, a partir da representação gráfica, percebemos que é possível implementar uma fila tendo três recursos básicos:

- Espaço de memória para armazenar os elementos;
- Uma referência ao primeiro elemento da coleção;
- Uma referência à primeira posição livre, após o último elemento da fila.

O espaço de memória seqüencial pode ser alocado por meio de um vetor. Para referenciar o primeiro elemento e a primeira posição disponível no final da fila, podemos utilizar duas variáveis inteiras que sirvam como índice aos elementos do vetor.

```
const MAX = 50;
tipo ELEM = caractere;
     FILA = registro
              COMECO : inteiro;
              FINAL: inteiro;
              MEMO : vetor [1..MAX] de ELEM;
       fim registro;

var
   F : FILA;
```

Observe que somente na oitava linha é que uma variável do tipo Fila foi realmente criada. Veja, na Figura 3.27, como ficaria a estrutura de armazenamento da fila F: [a, b, c].

A primeira operação a ser definida, deve ser aquela que inicializa a fila, indicando que ela se encontra vazia. Considerando a forma escolhida para representar a fila (figura anterior), podemos verificar que, à medida que novos elementos vão sendo inseridos, o índice **F.FINAL** vai se deslocando para a direita. Analogamente, quando um elemento é removido, o índice **F.COMECO** "persegue" **F.FINAL**. A conseqüência disto é que, conforme os elementos vão entrando e saindo da fila, ela vai se movendo gradativamente para a direita. Particularmente, quando a fila não tiver mais nenhum elemento, o índice **F.COMECO** terá alcançado o índice **F.FINAL**, isto é, teremos **F.COMECO** igual a **F.FINAL**.

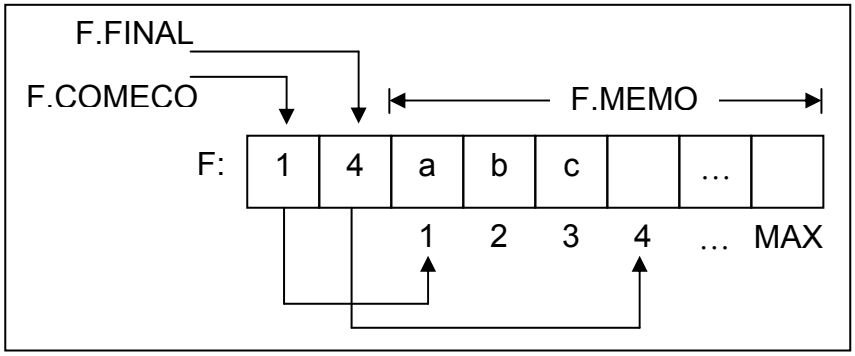


Figura 3.27 – Estrutura de armazenamento da fila F: [a, b, c]

Como desejamos uma fila inicialmente vazia e **F.COMECO** igual a **F.FINAL** indica esta situação, bastaria atribuir qualquer valor inteiro **k** aos dois índices, tornando-os iguais. Entretanto, convencionamos que **F.FINAL** apontaria a posição livre onde deveria ser armazenado um elemento que estivesse entrando na fila. Desta forma, a melhor escolha é **k = 1**. Indicamos que a fila está vazia e, ao mesmo tempo, que a posição disponível para inserção é a primeira posição do vetor **F.MEMO**.

```
procedimento QINIT (var F: FILA);

   F.COMECO ← 1;
   F.FINAL ← 1;

fim procedimento QINIT;
```

Note que a fila estará vazia sempre que **F.COMECO** for igual a **F.FINAL**, mesmo que o valor destes índices seja **k ≠ 1**. Adicionalmente, a fila estará cheia quando tivermos **F.FINAL > MAX**, isto é, quando não existir uma posição livre após o último elemento da fila.

```
função QISEMPTY(var F: FILA): lógico;

   QISEMPTY ← (F.COMECO = F.FINAL);

fim função QISEMPTY;


função QISFULL(var F: FILA): lógico;

   QISFULL ← (F.FINAL > MAX);

fim função QISFULL;
```

Para adicionar um elemento à fila, primeiramente precisamos verificar se existe espaço suficiente; isto é feito facilmente com a função **QISFULL()**. Caso exista, devemos lembrar que o índice **F.FINAL** aponta justamente a posição onde deverá entrar o novo elemento. Após o elemento ter sido armazenado no final da fila, o índice **F.FINAL** deve ser atualizado para que passe a indicar a próxima posição disponível. Como estamos utilizando uma área seqüencial para armazenar os elementos, basta incrementar o valor de **F.FINAL** para que ele aponte a próxima posição livre no vetor.

```
procedimento ENQUEUE (var F: FILA; X: ELEM);

   se (não QISFULL(F)) então
      F.MEMO[F.FINAL] ← X;
      F.FINAL ← F.FINAL + 1;
   senão
      escreva "Fila cheia";
   fim se;

fim procedimento ENQUEUE;
```

Para remover, lembramos que **F.COMECO** aponta o elemento que deve ser atendido primeiro, caso exista um. Após o elemento ter sido removido, o índice **F.COMECO** deve ser atualizado para apontar o próximo elemento a ocupar o início da fila.

```
função DEQUEUE (var F: FILA): ELEM;

   se (não QISEMPTY(F)) então
      DEQUEUE ← F.MEMO[F.COMECO];
      F.COMECO ← F.COMECO + 1;
   senão
      escreva "Fila vazia";
   fim se;

fim função DEQUEUE;
```

3.3.2 PROBLEMAS NA IMPLEMENTAÇÃO SEQÜENCIAL DE FILAS

Suponha uma fila F: [a, b, c, d, e]. De acordo com a nossa implementação, e admitindo que a fila pode armazenar no máximo 5 elementos (MAX = 5), podemos representar a fila F como a seguir:

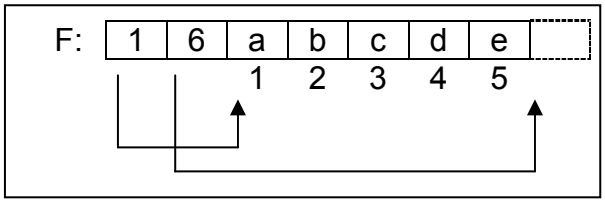


Figura 3.28 – Uma fila cheia

Vimos que cada vez que um elemento é removido, o índice que aponta o começo da fila desloca-se uma posição à direita. Se inicialmente ele vale 1, como

observamos na figura anterior, após a remoção de todos os elementos da fila F, teremos a situação esquematizada a seguir:

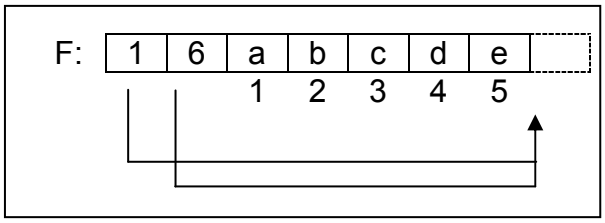


Figura 3.29 – Uma fila cheia ou vazia?

Se tentarmos inserir um novo elemento na fila, não conseguiremos, pois a função **QISFULL()** indica que não existe mais espaço disponível (**F.FINAL > MAX)**. Por outro lado, remover também não é possível, pois **QISEMPTY()** indica que a fila está vazia (**F.COMECO = F.FINAL)**.

Resumindo, chegamos a uma situação extremamente indesejável. Temos uma fila que está cheia e vazia ao mesmo tempo. Afinal, como isto é possível? Chegamos à conclusão de que esta nossa implementação não é muito eficiente, apresentando tanto desperdício de memória quanto problemas de lógica.

3.3.3 SOLUCIONANDO OS PROBLEMAS DA IMPLEMENTAÇÃO SEQÜENCIAL

Eliminar o erro lógico, que sinaliza fila vazia e cheia ao mesmo tempo, é bastante simples. Basta acrescentar uma variável contadora para indicar quantos elementos estão armazenados na fila. Esta variável deve estar inicialmente zerada. Quando um elemento for inserido, ela será incrementada; quando for removido, ela será decrementada. Desta forma, o impasse pode ser resolvido simplesmente consultando tal variável.

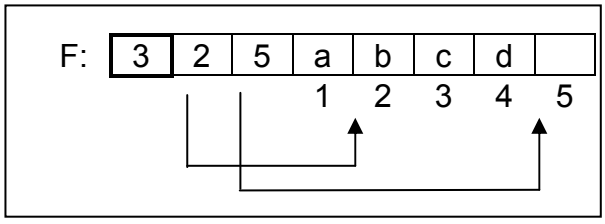


Figura 3.30 – A fila F: [b, c, d] com variável contadora

Para eliminar o desperdício de espaço, o ideal seria que cada posição liberada por um elemento removido se tornasse prontamente disponível para receber um novo elemento inserido. Para isto, teríamos de dispor de uma área seqüencial de memória tal que a posição 1 estivesse imediatamente após a posição **MAX**. Assim, **QISFULL()** somente indicaria fila cheia quando realmente todo o espaço de armazenamento estivesse esgotado.

Se fosse possível alocar uma área de memória como ilustra a Figura 3.31; então um índice **i**, com valor **MAX + 1**, estaria apontando para a primeira posição do vetor. Da mesma forma, um índice **j**, com valor **MAX + 2**, estaria apontando a segunda posição e assim por diante... Entretanto, as células de memória seguem uma organização linear e, para obter esta “circularidade”, deveremos lançar mão de um artifício: sempre que um índice for incrementado e seu valor ultrapassar a constante

MAX, restabelecemos seu valor a 1. A rotina **ADC()**, definida a seguir, serve para simular esta “circularidade”.

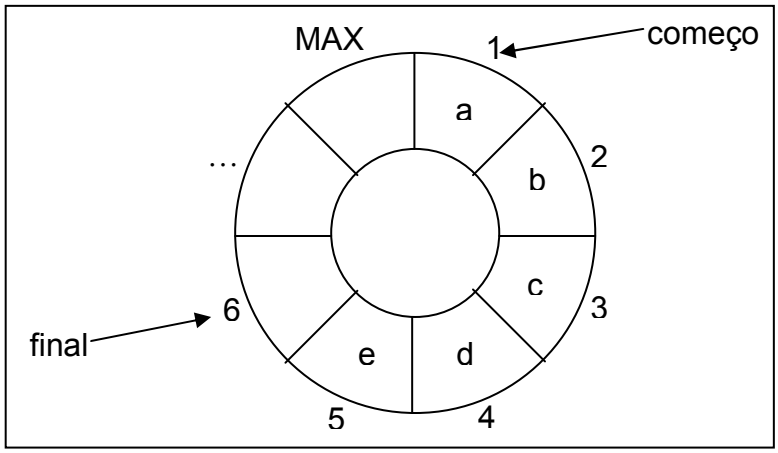


Figura 3.31 – Uma fila na representação circular

```
procedimento ADC (var I: inteiro);

   I ← I + 1;
   se I > MAX então I ← 1;

fim procedimento ADC;
```

3.3.4 IMPLEMENTAÇÃO CIRCULAR PARA FILAS

Temos, a seguir, a implementação circular completa para o tipo de dados Fila. As rotinas apresentadas são basicamente idênticas àquelas apresentadas na implementação seqüencial, exceto pelos detalhes discutidos na seção anterior.

```
const MAX = 50;

tipo ELEM = caractere;
     FILA = registro
              TOTAL : inteiro;
              COMECO : inteiro;
              FINAL: inteiro;
              MEMO : vetor [1..MAX] de ELEM;
       fim registro;

var
   F : FILA;

procedimento QINIT (var F: FILA);

   F.TOTAL ← 0;
   F.COMECO ← 1;
   F.FINAL ← 1;

fim procedimento QINIT;
```

```
função QISEMPTY(var F: FILA): lógico;

   QISEMPTY ← (F.TOTAL = 0);

fim função QISEMPTY;

função QISFULL(var F: FILA): lógico;

   QISFULL ← (F.TOTAL = MAX);

fim função QISFULL;

procedimento ADC (var I: inteiro);

   I ← I + 1;
   se I > MAX então I ← 1;

fim procedimento ADC;

procedimento ENQUEUE (var F: FILA; X: ELEM);

   se (não QISFULL(F)) então
      F.MEMO[F.FINAL] ← X;
      ADC(F.FINAL);
      F.TOTAL ← F.TOTAL + 1;
   senão
      escreva "Fila cheia";
   fim se;

fim procedimento ENQUEUE;

função DEQUEUE (var F: FILA): ELEM;

   se (não QISEMPTY(F)) então
      DEQUEUE ← F.MEMO[F.COMECO];
      ADC(F.COMECO);
      F.TOTAL ← F.TOTAL - 1;
   senão
      escreva "Fila vazia";
   fim se;

fim função DEQUEUE;
```

3.4 RECURSIVIDADE

3.4.1 INTRODUÇÃO

Um algoritmo que para resolver um problema divide-o em subproblemas mais simples, cujas soluções requerem a aplicação dele mesmo, é chamado **recursivo**.

Em termos de programação, uma rotina é recursiva quando ela chama a si mesma, seja de forma direta ou indireta. Em geral, uma rotina recursiva R pode ser expressa como uma composição formada por um conjunto de comandos C (que não contém chamadas a R) e uma chamada (recursiva) à rotina R:

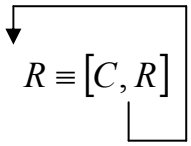


Figura 3.32 - Recursão Direta

Entretanto, pode-se ter também uma forma indireta de recursão, na qual as rotinas são conectadas através de uma cadeia de chamadas sucessivas que acaba retornando à primeira que foi chamada:

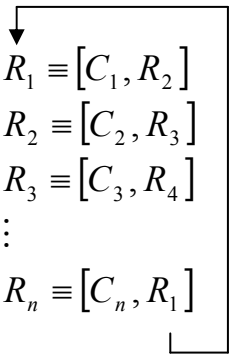


Figura 3.33 - Recursão Indireta

Assim, diz-se que uma rotina R é **indiretamente recursiva** se ela contém uma chamada a outra rotina S, que por sua vez contém uma chamada direta ou indireta a R. Note que a recursão nem sempre é explícita e, às vezes, pode ser difícil percebê-la através de uma simples leitura da rotina.

A recursão é uma técnica particularmente poderosa em definições matemáticas. Alguns exemplos familiares são: números naturais e certas funções:

- Números naturais:
  (a) 0 é um número natural;
  (b) o sucessor de um número natural é um número natural.

- A função fatorial n! (para inteiros não negativos):
  (a) 0! = 1
  (b) n > 0: n! = n * (n - 1)!

O poder da recursão deve-se, evidentemente, à possibilidade de se definir um conjunto infinito de objetos através de uma formulação finita. Do mesmo modo, um número infinito de cálculos pode ser definido por um programa recursivo finito, ainda que este não contenha repetições explícitas. Algoritmos recursivos, no entanto, são especialmente adequados quando o problema a ser resolvido, a função a ser calculada ou a estrutura de dados a ser processada já estejam definidos sob uma forma recursiva.

3.4.2 USO DE RECURSÃO NA SOLUÇÃO DE PROBLEMAS

Todo algoritmo deve ser executado em tempo finito, isto é, deve terminar após ter executado uma quantidade finita de passos. Para garantir que uma chamada recursiva não criará um "looping" que será executado infinitamente, é necessário que ela esteja condicionada a uma expressão lógica que, em algum instante, tornar-se-á falsa e permitirá que a recursão termine. Assim, uma rotina recursiva é melhor representada por $R \equiv [C, T \rightarrow R]$, onde $T \rightarrow R$ indica que a rotina R somente será chamada se o teste T for satisfeito.

A técnica básica para garantir o término da execução de um algoritmo recursivo consiste em:

- Exprimir T em termos de uma função *f(x)*, tal que *f(x)* ≤ 0 implica uma condição de parada;
- mostrar que *f(x)* decresce a cada passo de repetição, isto é, que temos a forma $R(x) \equiv [C, (f(x) > 0) \rightarrow R(x-1)]$, onde x decresce a cada chamada.

Na prática, ao definir uma rotina recursiva, dividimos o problema da seguinte maneira:

- **Solução Trivial:** dada por definição; isto é, não necessita da recursão para ser obtida. Esta parte do problema é resolvida pelo conjunto de comandos C.
- **Solução Geral:** parte do problema que em essência é igual ao problema original, sendo porém menor. A solução, neste caso, pode ser obtida por uma chamada recursiva R(x-1)

Para decidir se o problema terá solução trivial ou geral; isto é, se sua solução será obtida pela execução do conjunto de instruções C ou pela chamada recursiva R(x-1), usamos um teste. Por exemplo, vamos definir uma função recursiva para calcular o fatorial de um número natural:

- **Solução Trivial:** 0! = 1 {dada por definição}
- **Solução Geral:** n! = n * (n - 1)! {requer reaplicação da rotina para (n - 1) !}

Considerando *f*(n) = n, então n = 0 implica numa condição de parada do mecanismo recursivo, garantindo o término do algoritmo que calcula o fatorial:

```
função FAT (N: inteiro): inteiro;
   se (N = 0) então
      FAT ← 1
   senão
      FAT ← N * FAT (N - 1);
   fim se;
fim função FAT;
```

Em termos matemáticos, a recursão é uma técnica que, através de substituições sucessivas, reduz o problema a ser resolvido a um caso de solução trivial. Veja o cálculo de f(4) na figura a seguir:

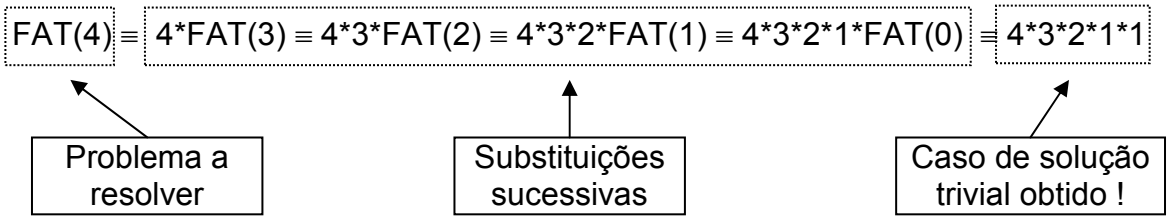


Figura 3.34 - Recursão vista como substituições sucessivas

3.4.3 QUANDO APLICAR RECURSÃO?

Embora a recursão seja uma ferramenta bastante interessante para resolução de problemas, nem sempre ela poderá ser empregada. Enquanto alguns problemas têm solução imediata com o uso de recursão, outros são praticamente impossíveis de se resolver de forma recursiva. É preciso analisar o problema e verificar se realmente vale a pena tentar encontrar uma solução recursiva.

A recursão, se bem utilizada, pode tornar um algoritmo muito elegante; isto é, claro, simples e conciso. Porém, na maioria das vezes, uma solução iterativa (não recursiva) será mais eficiente. Cada chamada recursiva implica em um custo tanto de tempo quanto de espaço, pois cada vez que a rotina é chamada, todas as variáveis locais são recriadas.

Assim, determinar precisamente se devemos usar recursão ou não, ao resolver um problema, é uma questão bastante difícil de ser respondida. Para isto teríamos que comparar as possíveis soluções recursiva e iterativa.

3.4.4 ELIMINANDO A RECURSÃO DE CAUDA

Dizemos que uma rotina apresenta recursão de cauda se a chamada recursiva está no final do seu código, tendo como única função criar um “looping” que será repetido até que a condição de parada seja satisfeita. Vejamos, por exemplo, a função para cálculo de fatorial:

```
função FAT (N: inteiro): inteiro;
   se (N = 0) então
      FAT ← 1
   senão
      FAT ← N * FAT(N - 1);
   fim se;
fim função FAT;
```

Looping!

Observe que a principal função da chamada FAT (N - 1) é criar um “looping” que se repete até que a condição de parada seja satisfeita. Infelizmente, este exemplo clássico de recursão (cálculo de fatorial) é um dos casos onde a recursão é menos eficiente que a iteração, justamente porque apresenta recursão de cauda.

Geralmente, se uma rotina R(x) tem como última instrução a chamada recursiva R(y), então podemos trocar R(y) pela atribuição x ← y, seguida de um desvio para o início do código de R. Isto funciona porque reexecutar R, para o novo valor de x,

tem o mesmo efeito de chamar R(y).

A recursão de cauda pode ser eliminada se empregarmos no seu lugar, uma estrutura de repetição que esteja condicionada à expressão de teste usada na versão recursiva. Veja como isto pode ser feito na função que calcula o fatorial:

```
função FAT_ITERATIVO (N: inteiro): inteiro;
var
   F : inteiro;

   F ← 1;
   enquanto (N > 0) faça
      F ← F * N;
      N ← N - 1;
   fim enquanto;

   FAT_ITERATIVO ← F;
fim função FAT;
```

Embora a função fatorial seja mais eficientemente implementada de forma iterativa, ela ainda é um excelente exemplo para se entender o que é recursão.

3.4.5 PILHAS E ROTINAS RECURSIVAS

O controle de chamadas e retornos de rotinas é feito através de uma pilha criada e mantida, automaticamente, pelo sistema. Na verdade, quando uma rotina é chamada, não apenas o endereço de retorno é empilhado, mas todas as suas variáveis locais são também recriadas na pilha. Por exemplo, ainda na versão recursiva da função fatorial, para a chamada FAT(4) teríamos:

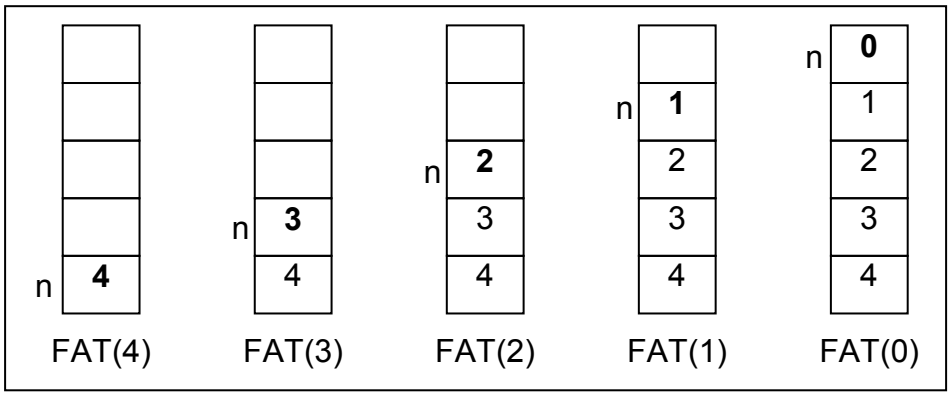


Figura 3.35 - Variável local n sendo recriada na pilha a cada chamada recursiva

Quando a rotina começa a ser executada, a variável n é criada no topo da pilha e inicializada com o valor que foi passado no momento da chamada; quando termina, a última cópia criada para a variável deixa de existir. A qualquer instante durante a execução, o valor assumido para a variável é aquele existente no topo da pilha.

Assim, utilizamos a memória para armazenar todos os argumentos intermediários e valores de retorno na pilha interna do sistema. Isto pode causar

problemas se há uma grande quantidade de dados, levando a estouro da pilha.

Basicamente, qualquer tipo de recursão pode ser eliminado se utilizarmos no seu lugar comandos de repetição e, eventualmente, pilhas. Normalmente, as rotinas assim modificadas serão mais rápidas que suas correspondentes em versão recursiva. Entretanto, se o uso de muitos comandos de repetição e várias pilhas for necessário para realizar a conversão da versão recursiva para a iterativa, talvez seja melhor permanecer com a rotina recursiva. Um algoritmo claro, simples e conciso vale mais que qualquer algoritmo “envenenado” que rode um pouquinho mais rápido.

A recursão geralmente é utilizada porque simplifica um problema conceitualmente, não porque é inerentemente mais eficiente.

UNIDADE 4 - ÁRVORES

Uma árvore é uma coleção finita de $n \geq 0$ nodos. Se $n = 0$, dizemos que a árvore é nula; caso contrário uma árvore apresenta as seguintes características:

- existe um nodo especial denominado **raiz**;
- os demais são particionados em T_1, T_2, ..., T_k estruturas disjuntas de árvores;
- as estruturas T_1, T_2, ..., T_k denominam-se **subárvores**.

A exigência de que as estruturas T_1, T_2, ..., T_k sejam coleções disjuntas, garante que um mesmo nodo não aparecerá em mais de uma subárvore ao mesmo tempo; ou seja, nunca teremos subárvores interligadas. Observe que cada uma das estruturas T_i é organizada na forma de árvore, isto caracteriza uma definição recursiva.

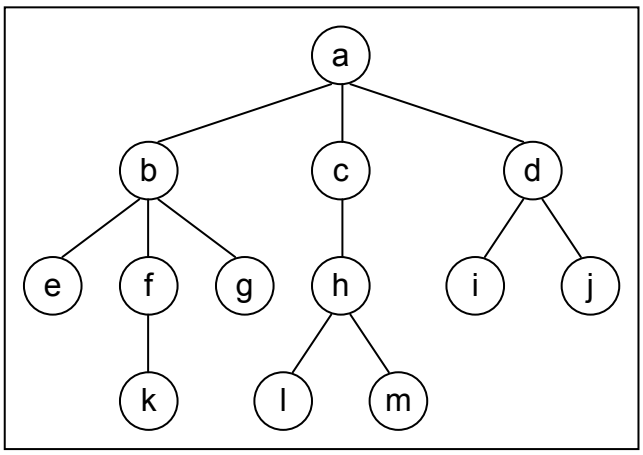


Figura 4.1 – A representação gráfica de uma árvore

O número de subárvores de um nodo denomina-se **grau**. Por exemplo, na Figura 4.1, o nodo ***a*** tem grau 3, o nodo ***d*** tem grau 2 e o nodo ***c*** tem grau 1. Um nodo que possui grau 0, ou seja, que não possui subárvores, denomina-se terminal ou **folha**; são folhas os nodos ***e***, ***k***, ***g***, ***l***, ***m***, ***i*** e ***j***. O grau de uma árvore (n-aridade), é definido como sendo igual ao máximo dos graus de todos os seus nodos. Na ilustração acima, podemos constatar que o grau da árvore é 3, pois nenhum nodo tem mais de 3 subárvores.

As raízes das subárvores de um nodo denominam-se **filhos** do nodo, que é o **pai** delas. Os filhos do nodo ***b*** são ***e***, ***f*** e ***g***; ***h*** é o pai de ***l*** e ***m***; os filhos do mesmo pai denominam-se **irmãos**. Intuitivamente, pode-se ainda definir termos tais como avô, neto, primo, descendentes, etc.

Por definição, dizemos que a raiz de uma árvore encontra-se no nível 1. Estando um nodo no nível n, seus filhos estarão no nível n + 1. A **altura** de uma árvore é definida como sendo o máximo dos níveis de todos os seus nodos. Uma árvore nula tem altura 0. A árvore da figura acima tem altura igual a 4. A subárvore com raiz ***d*** tem altura 2.

4.1 ÁRVORES BINÁRIAS

Uma **árvore binária** é uma árvore que pode ser nula, ou então tem as

seguintes características:

- existe um nodo especial denominado **raiz**;
- os demais nodos são particionados em T_1, T_2 estruturas disjuntas de árvores binárias;
- T_1 é denominada subárvore esquerda e T_2, subárvore direita da raiz.

Árvore binária é um caso especial de árvore em que nenhum nodo tem grau superior a 2, isto é, nenhum nodo tem mais que dois filhos. Adicionalmente, para árvores binárias existe um "senso de posição", ou seja, distingue-se entre uma subárvore esquerda e uma direita. Por exemplo, as árvores binárias representadas na figura abaixo são consideradas distintas: a primeira tem a subárvore direita nula e a segunda, a esquerda.

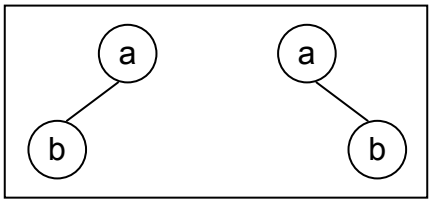


Figura 4.2 – Duas árvores binárias distintas

A terminologia básica utilizada para árvores binárias é a mesma definida para árvores de grau genérico. Podemos dizer agora filho esquerdo, filho direito, etc.

4.1.1 ÁRVORES DE BUSCA BINÁRIA

Uma árvore binária, cuja raiz armazena o elemento R, é denominada **árvore de busca binária** se:

- todo elemento armazenado na subárvore esquerda é menor que R;
- nenhum elemento armazenado na subárvore direita é menor que R;
- as subárvores esquerda e direita também são árvores de busca binária.

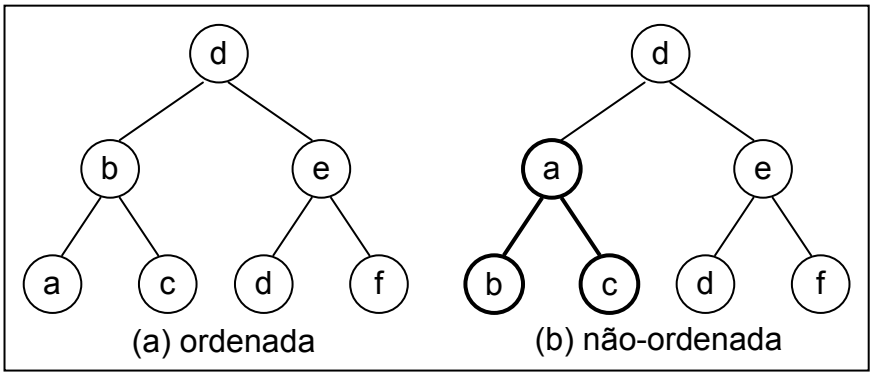


Figura 4.3 – Árvores binárias

Observe que a definição de árvore de busca binária também é recursiva. A árvore representada na figura (b) acima, tem o elemento ***d*** na raiz, todos os elementos na subárvore esquerda (***a***, ***b*** e ***c***) são menores que a raiz; nenhum elemento da subárvore direita (***d***, ***e*** e ***f***) é menor que a raiz e, no entanto, ela não é uma árvore de busca binária. Isto é devido ao fato de a subárvore esquerda não ser uma árvore de

busca binária. Note que a subárvore esquerda, cuja raiz é ***a***, tem uma subárvore esquerda não ser uma árvore de busca binária. Note que a subárvore esquerda, cuja raiz é ***a***, tem uma subárvore esquerda que armazena um elemento maior que a raiz. Logo, a árvore como um todo não está de acordo com a definição e não é considerada, portanto, uma árvore de busca binária!

4.1.2 OPERAÇÕES BÁSICAS EM ÁRVORES DE BUSCA BINÁRIA

Ao implementarmos árvores binárias, precisamos definir o formato dos nodos a serem empregados na organização dos dados. Como cada nodo precisa armazenar um elemento e referenciar duas subárvores, podemos defini-lo como a seguir:

```
tipo ELEM = char;
     ARVBIN = ^NODO;
     NODO = registro
               ESQ : ARVBIN;
               OBJ : ELEM;
               DIR : ARVBIN;
     fim registro;

var
   T : ARVBIN;
```

Como para as estruturas definidas anteriormente, uma árvore de busca binária vazia será representada por uma variável ponteiro nula. Veja a seguir, a estrutura de representação interna de uma árvore de busca binária na memória:

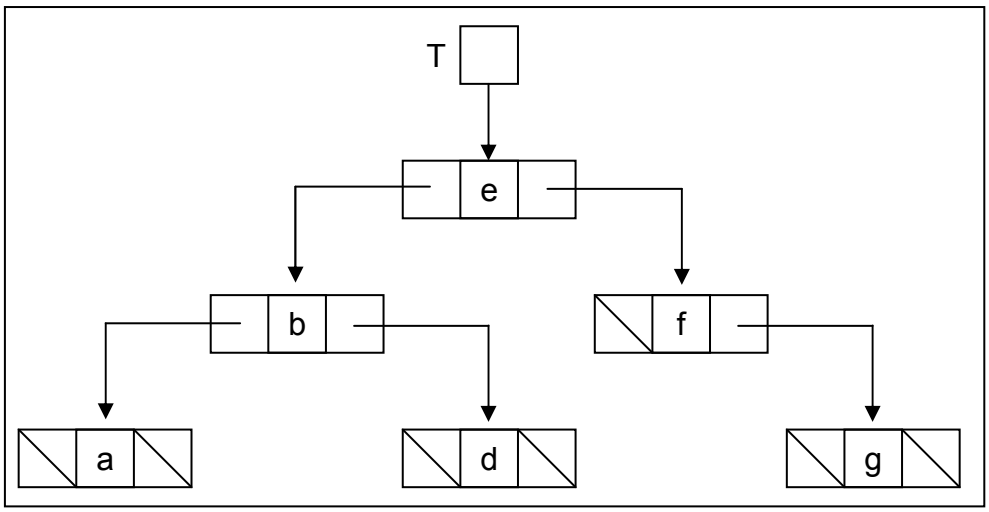


Figura 4.4 – Representação interna de uma árvore binária

A grande vantagem de uma lista encadeada é que nenhuma movimentação de dados é necessária quando um novo elemento precisa ser nela inserido; pois o ajuste de alguns ponteiros é suficiente para colocá-lo na posição adequada. A inserção numa árvore de busca binária é ainda mais eficiente, pois os novos elementos inseridos entram sempre na condição de folhas, isto é, um nodo não pode entrar numa árvore e já “assumir” filhos.

Para inserir o elemento ***c*** na árvore da figura anterior, começamos pelo nodo raiz apontado por T. Como ***c*** é menor que ***e***, tomamos a subárvore da esquerda. Comparando com a nova raiz, temos ***c*** > ***b*** e concluímos que o elemento ***c*** deverá ser armazenado na subárvore direita. O processo se repete até chegarmos a uma subárvore nula. Neste ponto, uma folha é alocada para armazenar o novo elemento entrando como raiz da subárvore nula. No nosso exemplo, o elemento ***c*** entraria como subárvore esquerda do nodo que armazena o elemento ***d***.

Inserir um elemento ***X*** numa árvore vazia é trivial. Se a árvore não se encontra vazia e o elemento a inserir é menor que a raiz, o inserimos na subárvore esquerda; caso contrário, devemos inseri-lo na subárvore direita:

```
procedimento TINS (var T: ARVBIN; X: ELEM);

   se (T = nil) então
      aloque (T);
      T^.OBJ ← X;
      T^.ESQ ← nil;
      T^.DIR ← nil;
   senão
      se (X < T^.OBJ) então
         TINS (T^.ESQ, X);
      senão
         TINS (T^.DIR, X);
      fim se;
   fim se;

fim procedimento TINS;
```

Uma vez que já podemos criar árvores de busca binária, vamos codificar uma rotina capaz de realizar pesquisas na árvore. Dada uma árvore de busca binária **T** e um elemento **X** a ser procurado entre seus nodos, temos quatro possibilidades a considerar:

- **T** é uma árvore nula: não há o que fazer;
- a raiz de **T** armazena o elemento **X**: a solução é trivial;
- o valor de **X** é menor que aquele armazenado na raiz de **T**: deve-se prosseguir com a busca na subárvore esquerda de **T**;
- o valor de **X** é maior ou igual àquele armazenado na raiz de **T**: deve-se prosseguir com a busca na subárvore direita de **T**.

Quando os elementos que a árvore armazena estão distribuídos de forma equilibrada em todas as subárvores, a busca numa árvore binária aproxima-se muito, em eficiência, da busca binária realizada em tabelas seqüenciais. A cada comparação, reduzimos aproximadamente para a metade os nodos que ainda terão que ser examinados, acelerando bastante o processo de pesquisa.

A função codificada a seguir, pesquisa uma árvore com o objetivo de encontrar um dado elemento. Se o elemento não está armazenado na árvore, um ponteiro nulo é retornado; caso contrário, a função retorna um ponteiro para a subárvore cuja raiz armazena o elemento procurado.

```
função TFND (T: ARVBIN; X: ELEM): ARVBIN;
   se (T = nil) então              {elemento não foi encontrado}
     TFND ← nil;
   senão
      se (X = T^.OBJ) então        {elemento encontrado na raiz}
         TFND ← T;
      senão
         se (X < T^.OBJ) então     {procura numa subárvore}
            TFND ← TFND(T^.ESQ, X);
         senão
            TFND ← TFND(T^.DIR, X);
         fim se;
      fim se;
   fim se;
fim função TFND;
```

O algoritmo para remoção em árvore de busca binária é talvez o mais trabalhoso que temos. Ainda assim, após analisarmos todos os possíveis casos, ele parecerá bastante simples. Para facilitar o entendimento, vamos admitir inicialmente que o elemento a ser removido encontra-se na raiz da árvore de busca binária **T**. Nestas condições, temos três possibilidades:

- a raiz não possui filhos: a solução é trivial; podemos removê-la e anular **T**;
- a raiz possui um único filho: podemos remover o nodo raiz, substituindo-o pelo seu nodo-filho;
- a raiz possui dois filhos: não é possível que os dois filhos assumam o lugar do pai; escolhemos então o nodo que armazena o maior elemento na subárvore esquerda de **T**, este nodo será removido e o elemento armazenado por ele entrará na raiz da árvore **T**.

O problema de remoção pode ser resolvido trivialmente no primeiro e no segundo caso. A dificuldade maior aparece quando o nodo a ser removido tem dois filhos, pois a sua remoção deixaria dois nodos “órfãos”.

Para entender a solução adotada para o terceiro caso, precisamos lembrar da definição de árvore de busca binária. Segundo ela, não podemos ter elementos maiores nem iguais à raiz numa subárvore esquerda, também não podemos ter elementos menores que a raiz numa subárvore direita. Ora, se tomamos o maior elemento da subárvore esquerda e o posicionarmos na raiz, então continua valendo a definição, isto é, a árvore continua ordenada após a alteração.

Sabemos que o maior elemento numa árvore binária ordenada encontra-se no nodo que ocupa a posição mais à direita possível. Este nodo certamente não terá um filho direito, pois se o tivesse não seria o maior da árvore; pode, entretanto, ter um filho esquerdo (menor).

Primeiro desenvolveremos uma função que recebe um ponteiro para uma árvore não-nula, procura o nodo que armazena o seu maior elemento, desliga-o da árvore e retorna o seu endereço:

```
função GETMAX (var T: ARVBIN): ARVBIN;

   se (T^.DIR = nil) então
      GETMAX ← T;
      T ← T^.ESQ;
   senão
      GETMAX ← GETMAX(T^.DIR);
   fim se;

fim função GETMAX;
```

De posse da função GETMAX(), podemos remover facilmente um nodo que tenha dois filhos:

```
P ← GETMAX (T^.ESQ); {desliga o nodo com o maior valor}
T^.OBJ ← P^.OBJ;     {armazena valor na raiz da árvore}
desaloque(P);          {libera o nodo desocupado}
```

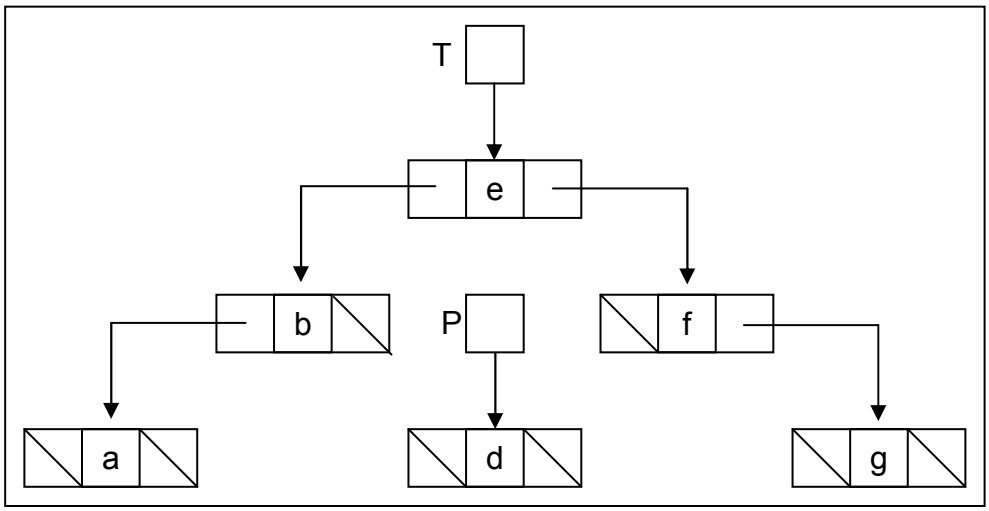


Figura 4.5 – Resultado da operação *P ← GETMAX (T^.ESQ)*

Veja, na figura anterior, o resultado da execução da operação GETMAX() sobre uma árvore de busca binária não-nula. Note que se o nodo removido ***d*** tivesse um filho esquerdo, este seria adotado como filho direito pelo nodo ***b***.

A simplificação que foi feita inicialmente, supondo que o elemento a ser removido estivesse na raiz da árvore, precisa agora ser desfeita. Na verdade, o elemento a ser removido pode estar armazenado em qualquer posição da árvore, ou mesmo nem existir. Desta forma, antes de remover um elemento de uma árvore, devemos encontrar o nodo que o armazena. Isto já foi visto quando definimos a operação TFND().

O algoritmo que vamos desenvolver agora é essencialmente igual àquele utilizado para pesquisa. A diferença é que, caso o elemento seja encontrado (na raiz de uma árvore), ele será removido:

```
procedimento TREM (var T: ARVBIN; X: ELEM);
var P : ARVBIN;

   se (T = nil) então exit;     {elemento não foi encontrado}
```

```
   se (X = T^.OBJ) então          {elemento encontrado na raiz}
      P ← T;
      se (T^.ESQ = nil) então
         T ← T^.DIR;             {a raiz não tem filho esquerdo}
      senão
         se (T^.DIR = nil) então
            T ← T^.ESQ;          {a raiz não tem filho direito}
         senão                    {a raiz tem ambos os filhos}
            P ← GETMAX(T^.ESQ);
            T^.OBJ ← P^.OBJ;
         fim se;
      fim se;

      desaloque(P);
   senão
      se (X < T^.OBJ) então
         TREM (T^.ESQ, X);       {procura na subárvore esquerda}
      senão
         TREM (T^.DIR, X);       {procura na subárvore direita}
      fim se;
   fim se;

fim procedimento TREM;
```

4.1.3 ATRAVESSAMENTO EM ÁRVORES BINÁRIAS

Atravessar uma árvore binária significa passar de forma sistemática, por cada um de seus nodos, realizando certo processamento. Existem quatro tipos básicos de atravessamento:

- em-ordem;
- pré-ordem;
- pós-ordem;
- em nível.

Os três primeiros tipos de atravessamento são facilmente compreendidos se fizermos uma analogia entre eles e as notações segundo as quais uma expressão aritmética pode ser escrita: vamos considerar, por exemplo, a expressão infixa A + B representada da seguinte forma:

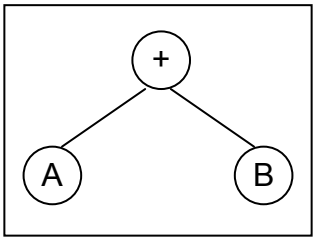


Figura 4.6 – A expressão A + B como árvore binária

Para reconstituir a forma infixa da expressão, deveríamos imprimir os nodos da seguinte maneira:

- imprimir a folha esquerda (E)
- imprimir a raiz (R)
- imprimir a folha direita (D)

Para obter a forma prefixa, fazemos:

- imprimir a raiz (R)
- imprimir a folha esquerda (E)
- imprimir a folha direita (D)

E, finalmente, para a forma pós-fixa teríamos:

- imprimir a folha esquerda (E)
- imprimir a folha direita (D)
- imprimir a raiz (R)

A seqüência de acesso ERD, necessária para reconstituir a forma infixa, é denominada atravessamento **em-ordem**, a seqüência RED, para forma pré-fixa, denomina atravessamento em **pré-ordem** e, finalmente, a seqüência EDR é denominada atravessamento **pós-ordem**.

Se o atravessamento em-ordem for aplicado sobre uma árvore de busca binária, então conseguimos acessar (processar) os nodos em ordem crescente. Isto é muito interessante, quando desejamos armazenar seqüências ordenadas através do uso de árvores binárias.

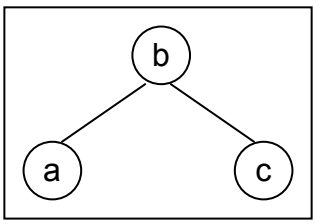


Figura 4.7 – Uma árvore de busca binária

Se as três formas de atravessamento são aplicadas à árvore da figura anterior, temos:

- em-ordem: ***a***, ***b***, ***c***
- pré-ordem: ***b***, ***a***, ***c***
- pós-ordem: ***a***, ***c***, ***b***

O exemplo da Figura 4.7 usa uma árvore muito simples; seja então uma árvore binária um pouco mais complexa (Figura 4.8), onde suas árvores não são apenas folhas.

Caso fosse aplicado o atravessamento em-ordem (ERD) para a árvore acima, os nodos seriam impressos na ordem: ***a***, ***b***, ***c***, ***d***, ***e***, ***f***. Para obter esta seqüência, é fácil perceber que primeiro a subárvore esquerda (da raiz ***d***) foi impressa em-ordem, produzindo a seqüência ***a***, ***b***, ***c***, depois, foi impressa a raiz ***d*** e, finalmente, a subárvore direita também foi impressa em-ordem.

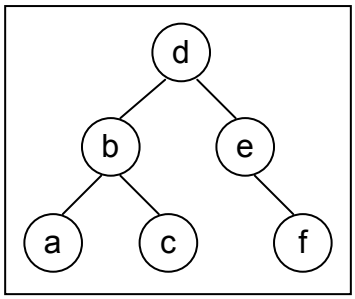


Figura 4.8 – Uma árvore binária com subárvores

Analisando melhor as figuras e a analogia feita a partir delas, concluímos que a seqüência básica de acesso ERD pode ser generalizada. Na verdade, cada subárvore não precisa se restringir a uma única folha:

- imprimir a subárvore esquerda (E)
- imprimir a raiz (R)
- imprimir a subárvore direita (D)

Observe que a seqüência ERD agora se tornou recursiva, pois ambas as subárvores devem ser impressas também em-ordem e a recursão pára quando chegamos a subárvores nulas. As seqüências pré-ordem e pós-ordem podem ser generalizadas segundo o mesmo raciocínio. Veja os algoritmos codificados a seguir:

```
procedimento EMORDEM (T: ARVBIN);

   se (T <> nil) então
      EMORDEM (T^.ESQ);
      escreva (T^.OBJ, ' ');
      EMORDEM (T^.DIR);
   fim se;

fim procedimento EMORDEM;


procedimento PREORDEM (T: ARVBIN);

   se (T <> nil) então
      escreva (T^.OBJ, ' ');
      PREORDEM (T^.ESQ);
      PREORDEM (T^.DIR);
   fim se;

fim procedimento PREORDEM;


procedimento POSORDEM (T: ARVBIN);

   se (T <> nil) então
      POSORDEM (T^.ESQ);
      POSORDEM (T^.DIR);
      escreva (T^.OBJ, ' ');
   fim se;

fim procedimento POSORDEM;
```

O quarto tipo de atravessamento, denominado **em-nível**, apesar de ser o mais facilmente compreendido, é aquele que apresenta maior dificuldade de programação. Supondo que ele fosse aplicado à árvore da figura anterior, obteríamos a seqüência: ***d***, ***b***, ***e***, ***a***, ***c***, ***f***. Observe que agora os nodos são acessados, por nível, da esquerda para a direita.

Para realizar um atravessamento em-nível, precisamos de uma fila contendo inicialmente apenas o nodo raiz. A partir daí, enquanto a fila não se tornar vazia, retiramos dela um nodo cujos filhos deverão ser colocados na fila, aí então, o nodo retirado da fila pode ser impresso.

```
procedimento EMNIVEL (T : ARVBIN);
var
   N : ARVBIN;
   F : FILA;

   se (T <> nil) então
      QINIT (F);
      ENQUEUE (F, T);

      enquanto (não QISEMPTY(F)) faça
         N ← DEQUEUE (F);
         se (N^.ESQ <> nil) então ENQUEUE (F, N^.ESQ);
         se (N^.DIR <> nil) então ENQUEUE (F, N^.DIR);
         escreva (N^.OBJ, ', ');
      fim enquanto;

   fim se;

fim procedimento EMNIVEL;
```

Começamos a falar de atravessamento, dizendo que "atravessar uma árvore é passar por todos os seus nodos realizando, para cada um deles, certo processamento". Nos quatro algoritmos que desenvolvemos, o processamento realizado foi uma simples impressão, entretanto qualquer outra tarefa poder ter sido executada. Por exemplo, o algoritmo TFND() emprega o atravessamento pré-ordem e, para cada nodo acessado, o processamento consiste numa comparação que verifica se o elemento armazenado pelo nodo é aquele procurado. Muitos outros algoritmos importantes no tratamento de árvores são baseados nestes tipos de atravessamento. Veja, por exemplo, o algoritmo TKILL(), para destruir árvores: ele usa um atravessamento do tipo pós-ordem, cujo processamento é a liberação do nodo acessado:

```
procedimento TKILL (var T: ARVBIN);

   se (T <> nil) então
      TKILL (T^.ESQ);       {libera subárvore esquerda}
      TKILL (T^.DIR);       {libera subárvore direita}
      desaloque (T);        {libera a raiz da árvore}
      T ← nil;
   fim se;

fim procedimento TKILL;
```

4.2 ÁRVORES BALANCEADAS

Para obtermos o máximo desempenho num busca binária, é necessário que a árvore de busca binária esteja perfeitamente balanceada. Uma árvore binária perfeitamente balanceada é aquela em que, para todo nodo da árvore, a diferença entre a altura da subárvore da esquerda e a altura da subárvore da direita seja nula.

Existem métodos que balanceiam perfeitamente uma árvore binária; entretanto estes métodos são caros computacionalmente e, após algumas inserções e exclusões de elementos, a árvore tornar-se-ia novamente desbalanceada.

Um critério menos rigoroso de balanceamento poderia ser adotado, conduzindo a procedimentos mais simples de reorganização da árvore, acarretando apenas uma pequena deterioração no desempenho médio do processo de busca.

4.3 ÁRVORES-AVL

Uma definição de balanceamento, menos rigorosa, foi apresentada por Adelson-Velskii e Landis. O critério de balanceamento é o seguinte:

"Uma árvore é dita balanceada se e somente se, para qualquer nodo, a altura de suas duas subárvores diferem de no máximo uma unidade".

As árvores que satisfazem esta condição são, em geral, denominadas árvores-AVL (em homenagem a seus inventores). Note que todas as árvores perfeitamente balanceadas são também AVL-balanceadas. A definição não apenas é simples como também conduz a um procedimento de rebalanceamento relativamente simples. Na figura seguinte são apresentados duas árvores: uma delas é árvore-AVL, a outra não.

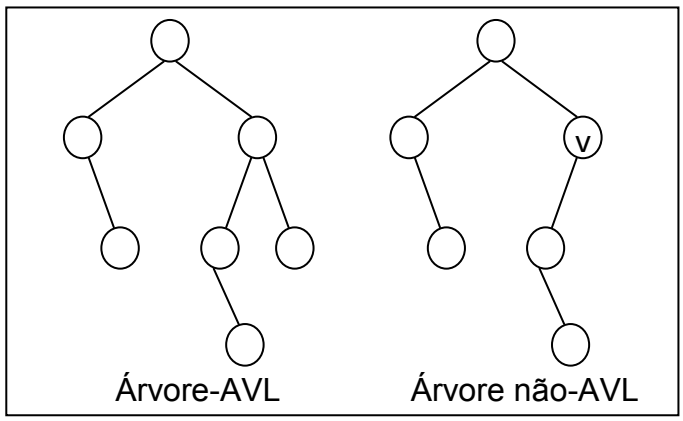


Figura 4.9 – Exemplos de árvore-AVL e de árvore não-AVL

Um nodo da árvore que apresenta diferença de altura entre suas subárvores esquerda e direita menor que 1, é chamado de nodo regulado, caso contrário é chamado de nodo desregulado. Uma árvore que contenha um nodo desregulado também é chamada de desregulada. Na figura acima o nodo ***v*** é um nodo desregulado pois a diferença de altura entre suas subárvores esquerda e direita é igual a 2 e, portanto, a árvore é não-AVL.

4.3.1 INCLUSÃO EM ÁRVORES-AVL

Seja ***T*** uma árvore AVL, na qual serão efetuadas inclusões de nodos. Para

que ***T*** se mantenha como árvore AVL após as inclusões é preciso efetuar operações de restabelecimento da regulagem de seus nodos, quando necessário.

A idéia consiste em verificar, após cada inclusão, se algum nodo ***p*** se encontra desregulado, isto é, se a diferença de altura entre as duas subárvores de ***p*** tornou-se maior do que um. Em caso positivo, aplicar transformações apropriadas para regulá-lo.

Serão utilizadas quatro transformações, indicadas na Figura 4.10, denominadas, respectivamente, **rotação direita**, **rotação esquerda**, **rotação dupla direita** e **rotação dupla esquerda**. As subárvores T_1, T_2, T_3 e T_4 que aparecem na figura podem ser vazias ou não. O nodo ***p*** é chamado ***raiz*** da transformação.

Por exemplo, a Figura 4.11(c) representa o efeito de uma rotação direita da raiz ***e***, efetuada na árvore da Figura 4.11(b). Observe que essas transformações preservam a natureza da árvore como sendo binária de busca. Isto é, se o valor da chave de um nodo ***v*** for inferior ao da correspondente ao nodo ***w***, então acontece uma dentre as seguintes possibilidades: ***v*** pertence à subárvore esquerda de ***w***, ou ***w*** pertence à subárvore direita de ***v***.

Uma análise da operação de inclusão é efetuada a seguir. No processo, tornar-se-á evidente como aplicar as transformações.

Suponha que o nodo ***q*** foi incluído em ***T***. Se após a inclusão todos os nodos mantiverem-se regulados, então a árvore manteve-se AVL e não há nada a efetuar. Caso contrário, seja ***p*** o nodo mais próximo às folhas de ***T*** que se tornou desregulado. Observe que não há ambigüidade na escolha de ***p***, pois qualquer subárvore de ***T*** que se tornou desregulada após a inclusão de ***q*** deve necessariamente conter ***p***. Então ***p*** se encontra no caminho de ***q*** à raiz de ***T***, e sua escolha é única.

Sejam $h_E(p)$ e $h_D(p)$ as alturas das subárvores esquerda e direita de ***p***, respectivamente. Naturalmente, $|h_E(p) - h_D(p)| > 1$. Então, conclui-se que $|h_E(p) - h_D(p)| = 2$, pois ***T*** era uma árvore-AVL e, além disso, a inclusão de um nodo não pode aumentar em mais de uma unidade a altura de qualquer subárvore. Podem-se identificar os seguintes casos de análise:

- **Caso 1**: $h_E(p) > h_D(p)$

Então, ***q*** pertence à subárvore esquerda de ***p***. Além disso, ***p*** possui o filho esquerdo $u \neq q$. Pois, caso contrário, ***p*** não estaria desregulado. Finalmente, por esse mesmo motivo, sabe-se que $h_E(u) \neq h_D(u)$. As duas seguintes possibilidades podem ocorrer:

- Caso 1.1: $h_E(u) > h_D(u)$

Essa situação corresponde à da Figura 4.10(a), sendo ***q*** um nodo pertencente a T_1. Observe que $h(T_1) - h(T_2) = 1$ e $h(T_2) = h(T_3)$. Conseqüentemente, a aplicação da rotação direita da raiz ***p*** transforma a subárvore considerada na da Figura 4.10(b), o que restabelece a regulagem de ***p***.

- Caso 1.2: $h_D(u) > h_E(u)$

Então ***u*** possui o filho direito ***v***, e a situação é ilustrada na Figura 4.10(e). Nesse caso, vale

$$|h(T_2) - h(T_3)| \leq 1 \text{ e } \max\{h(T_2), h(T_3)\} = h(T_1) = h(T_4)$$

Aplica-se então a rotação dupla direita da raiz ***p***. A nova configuração

aparece na Figura 4.10(f) e a regulagem de ***p*** é restabelecida.

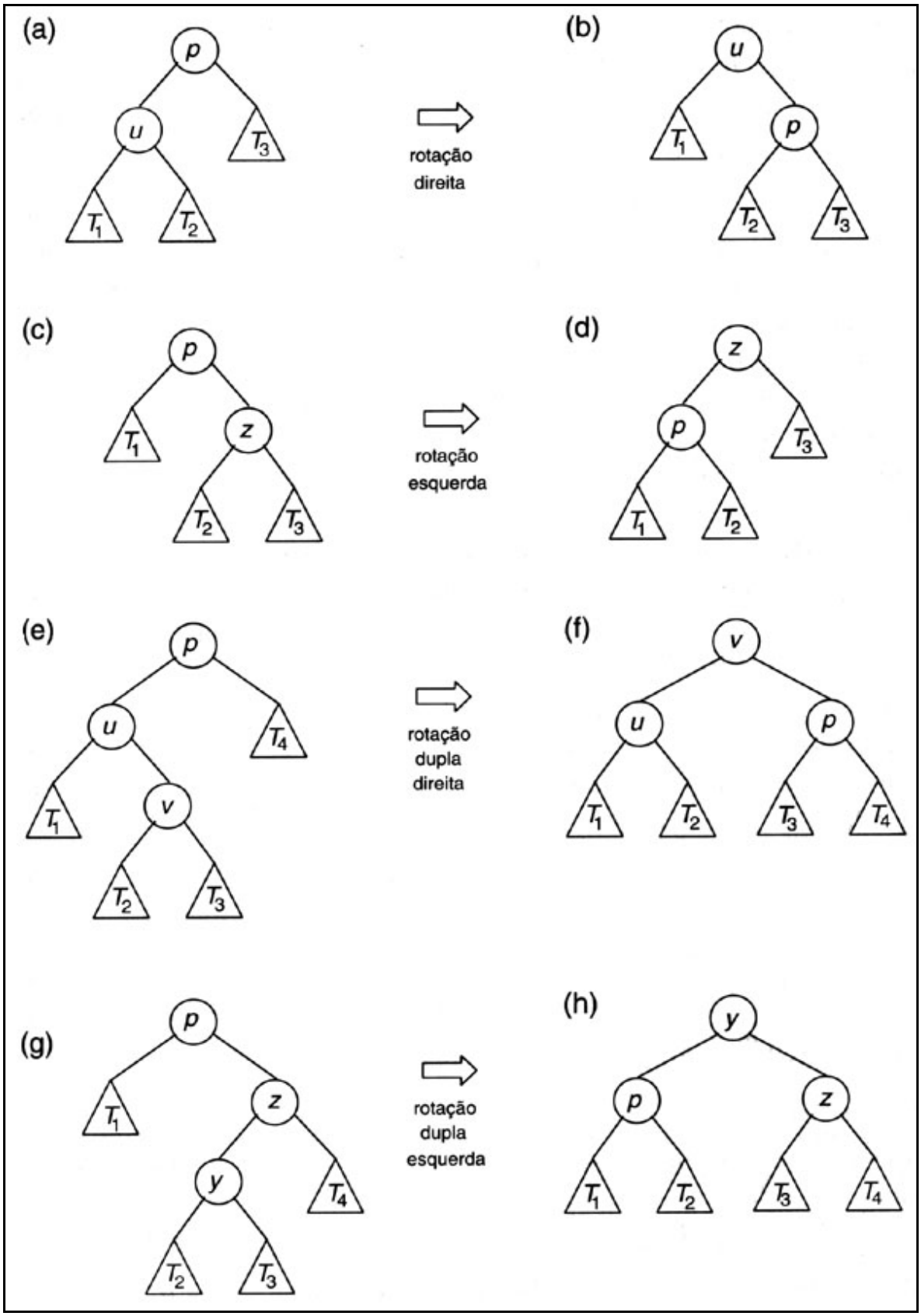


Figura 4.10 – Rotações para restabelecer a regulagem dos nodos em árvores-AVL

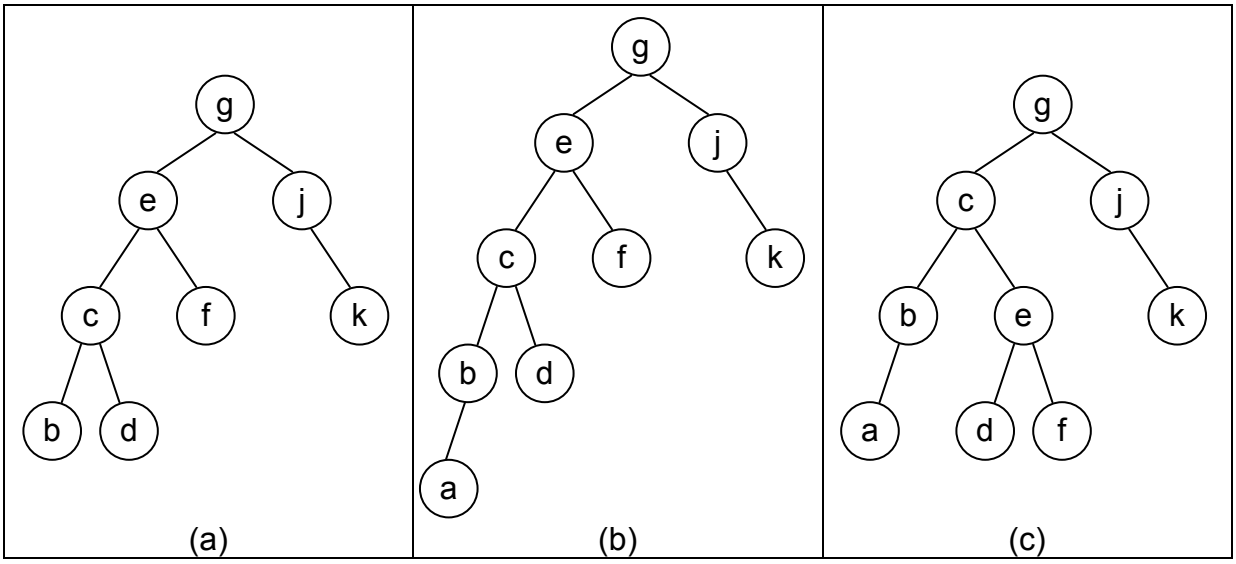


Figura 4.11 – Regulagem de uma árvore-AVL

- **Caso 2**: $h_D(p) > h_E(p)$

Dessa vez, ***p*** possui o filho direito $z \neq p$. Segue-se que $h_E(z) \neq h_D(z)$ e as demais situações são as seguintes:

- Caso 2.1: $h_D(z) > h_E(z)$

Esse caso corresponde à Figura 4.10(c), sendo ***q*** pertencente a T_3. As relações de altura são agora

$$h(T_3) - h(T_2) = 1 \text{ e } h(T_2) = h(T_1)$$

Isso significa que a rotação esquerda torna a árvore novamente AVL (Figura 4.10(d)).

- Caso 2.2: $h_E(z) > h_D(z)$

Então ***z*** possui o filho esquerdo ***y*** (Figura 4.10(g)). Observe que as alturas das subárvores T_1, T_2, T_3 e T_4 satisfazem as mesmas ralações do caso 1.2. Dessa forma, a aplicação da rotação dupla esquerda transforma a subárvore na indicada em 4.10(h). E ***p*** torna-se novamente regulado.

A argumentação apresentada mostra que a regulagem de ***p*** é restabelecida pela aplicação apropriada de uma das transformações estudadas. Além disso, é fácil concluir que o pai de ***p***, após a transformação, encontra-se também regulado, pois a transformação diminui a unidade de altura da subárvore de raiz ***p***. Isto assegura a regulagem de todos os nodos ancestrais de ***p***. Conseqüentemente, uma única transformação é suficiente para regular ***T***. Como exemplo, considere a árvore AVL da Figura 4.11(a), onde o valor de cada chave é o valor alfabético indicado. Suponha que o nodo ***a*** deva ser incluído na árvore. Após a inclusão, esta se transforma na árvore da Figura 4.11(b), que é não-AVL. O nodo ***e*** tornou-se desregulado, pois a altura de sua subárvore esquerda excede em duas unidades a da direita. Além disso, esse nodo é o mais próximo às folhas da árvore, dentre aqueles desregulados. Finalmente, o novo

nodo ***a*** pertence à subárvore esquerda do filho ***c*** de ***e***. Logo, a situação corresponde ao caso 1 da análise. A aplicação de uma rotação direita da raiz ***e*** torna esse nodo novamente regulado. O resultado da rotação está ilustrado na Figura 4.11(c). Observe, na Figura 51.11(b), que o nodo ***g*** também se encontra desregulado. A rotação utilizada também o regula.

4.3.2 IMPLEMENTAÇÃO DA INCLUSÃO

Nessa seção, é apresentada a implementação da operação de inclusão em árvores-AVL.

Seja ***T*** uma árvore-AVL e ***x*** a chave a ser incluída em algum novo nodo ***q***. O processo completo de inclusão pode ser dividido do seguinte modo. Inicialmente, efetua-se uma busca em ***T*** para verificar se ***x*** já se encontra em ***T***. Em caso positivo, o processo deve ser encerrado, pois se trata de uma chave duplicata. Caso contrário, a busca encontra o local correto do novo nodo. Segue-se, então, a inclusão propriamente dita. Deve-se verificar, em seguida, se essa operação tornou algum nodo desregulado. Em caso negativo, o processo termina, pois ***T*** permanece AVL. Caso contrário, a regulagem de ***T*** deve ser efetuada. De acordo com a última seção, essa regulagem consiste na execução de uma das operações de rotação da Figura 4.10. A última etapa, pois, consiste no reconhecimento da operação de rotação indicada ao caso e em sua posterior execução. Isto devolve a ***T*** a condição de árvore-AVL.

Em seguida, serão discutidos alguns aspectos do procedimento anterior. O primeiro problema ainda não detalhado é o de verificar se algum nodo ***v*** de ***T*** se tornou desregulado após a inclusão. Há uma solução direta bastante simples para resolver essa questão. Basta determinar as alturas de suas duas subárvores e subtrair uma da outra. Essa operação pode ser realizada sem dificuldades, percorrendo a subárvore de raiz ***v*** em ***T***. Mas isso consome tempo. Além disso, em princípio, qualquer nodo do caminho de ***q*** até a raiz da árvore pode ter se tornado desregulado. Então, a aplicação direta deste procedimento conduz a um algoritmo complexo, o que o torna fora de cogitação. Essa complexidade pode ser facilmente reduzida percorrendo-se ***T*** de baixo para cima e armazenando-se o valor da altura de cada subárvore. Naturalmente, $h(v) = 1 + \max\{h_E(v), h_D(v)\}$. Entretanto, para alcançar a meta de efetuar a inclusão num tempo menor, utiliza-se um processo diferente. Define-se o rótulo ***balanço(v)***, para cada nodo ***v*** de ***T*** como:

$$balanço(v) = h_D(v) - h_E(v)$$

Observa-se que ***v*** está regulado se e somente se $-1 \leq balanço(v) \leq 1$. O problema é como atualizar ***balanço(v)*** de forma eficiente, tendo em vista a inclusão de ***q***. A idéia consiste em determinar o efeito dessa inclusão nos valores de ***balanço(v)***. Se ***q*** pertencer à subárvore esquerda de ***v*** e essa inclusão ocasionar um aumento na altura dessa subárvore, então se subtrai uma unidade de ***balanço(v)***. Se esse valor decrescer para –2, indica que ***v*** se tornará desregulado. Analogamente, se ***q*** pertencer à subárvore direita de ***v*** e provocar um aumento de sua altura, adiciona-se uma unidade a ***balanço(v)***. O nodo ***v*** ficara desregulado, nesse caso, se ***balanço(v)*** aumentar para 2.

Para completar o processo, resta ainda determinar em que campos a inclusão de ***q*** provoca acréscimo na altura ***h(v)*** da subárvore T_v. Para resolver esta questão, é interessante que se obtenha um método simples e eficiente, isto é, que

possa ser resolvido em tempo constante. De início, observa-se que a inserção do nodo ***q*** acarreta obrigatoriamente uma alteração na altura da subárvore esquerda ou direita (de acordo com a nova chave) de seu nodo pai ***w***. A situação do campo ***balanço*** permite avaliar se essa alteração pode, ou não, se propagar aos outros nodos ***v*** do caminho de ***w*** até a raiz da árvore. Suponha que o nodo ***q*** é inserido na subárvore esquerda de ***v***. A análise se inicia com ***v = w*** e prossegue em seus nodos ancestrais, de forma recursiva. O processo se encerra quando da constatação de que a altura de T_v não foi modificada, ou de que ***v*** se tornou regulado. Três casos distintos podem ocorrer:

- **Caso 1**: ***balanço(v) = 1*** antes da inclusão.

Neste caso, ***balanço(v)*** se torna 0 e a altura da subárvore de raiz ***v*** não foi modificada. Conseqüentemente, as alturas dos nodos restantes do caminho até a raiz da árvore não se alteram.

- **Caso 2:** ***balanço(v) = 0*** antes da inclusão.

Neste caso, ***balanço(v)*** se torna –1 e a altura da subárvore de raiz ***v*** foi modificada. Conseqüentemente, os nodos restantes do caminho até a raiz também podem ter suas alturas modificadas e devem ser analisados. Se ***v*** é a raiz de ***T***, a análise encerra, pois nenhum nodo se tornou desregulado. Caso contrário, repetir o processo com ***v*** substituído por seu pai.

- **Caso 3:** ***balanço(v) = -1*** antes da inclusão.

Neste caso, ***balanço(v)*** se torna –2 e o nodo está desregulado. A rotação correta deve ser empregada. Qualquer rotação implica que a subárvore resultante tenha a mesma altura da subárvore antes da inclusão. As alturas dos ancestrais de ***v*** não mais necessitam de avaliação.

Para uma inserção na subárvore direita de ***v***, casos simétricos devem ser considerados.

Após a regulagem da árvore, completa-se o processo. O algoritmo visto a seguir busca e insere, se possível, uma nova chave ***x*** numa árvore-AVL. Observe que um só algoritmo foi considerado para as duas tarefas, ao contrário do que já foi visto anteriormente. Tal enfoque se deve à necessidade de conhecer todo o caminho da raiz até o nodo inserido ao se regular a árvore. O procedimento recursivo ***ins-AVL*** cumpre essa tarefa.

De início a posição de inserção é procurada determinando-se, na pilha de recursão, o caminho da raiz até o novo nodo. O procedimento ***início-nodo*** aloca o novo nodo. Após a inserção, o procedimento percorre o caminho inverso na árvore, acertando a campo ***bal*** que implementa o rótulo ***balanço***, já mencionado. Por meio deste pode-se conferir a regulagem de cada nodo, determinando-se então o nodo ***p*** apontado por ***pt***. A operação de rotação conveniente é efetuada imediatamente. O parâmetro lógico ***h*** retornar se houve ou não alteração na altura da subárvore do nodo em questão, induzindo então à atualização do campo ***bal***. A variável ***ptraiz*** é um ponteiro para a raiz da árvore. A chamada externa é ***ins-AVL (ptraiz, x, h)***.

```
tipo ELEM = char;
     ARVAVL = ^NODO;
     BALANCO = (-1, 0, +1);
     NODO = registro
               ESQ : ARVAVL;
               OBJ : ELEM;
               DIR : ARVAVL;
               BAL : BALANCO;
     fim registro;

var
   T : ARVAVL;

procedimento INÍCIO-NODO (var T: ARVAVL; X: ELEM);

   aloque (T);
   T^.ESQ ← nil;
   T^.DIR ← nil;
   T^.OBJ ← X;
   T^.BAL ← 0;

fim procedimento INÍCIO-NODO;

procedimento CASO1 (var T: ARVAVL; var H: lógico);
var
   TU, TV : ARVAVL;

   TU ← T^.ESQ;

   se (TU^.BAL = -1) então
      T^.ESQ ← TU^.DIR;
      TU^.DIR ← T;
      T^.BAL ← 0;
      T ← TU;
   senão
      TV ← TU^.DIR;
      TU^.DIR ← TV^.ESQ;
      TV^.ESQ ← TU;
      T^.ESQ ← TV^.DIR;
      TV^.DIR ← T;

      se (TV^.BAL = -1) então T^.BAL ← 1
      senão T^.BAL ← 0;
      fim se;

      se (TV^.BAL = 1) então TU^.BAL ← -1
      senão TU^.BAL ← 0;
      fim se;

      T ← TV;

   fim se;
```

```
   T^.BAL ← 0;
   H ← false;

fim procedimento CASO1;

procedimento CASO2 (var T: ARVAVL; var H: lógico);
var
   TU, TV : ARVAVL;

   TU ← T^.DIR;

   se (TU^.BAL = 1) então
      T^.DIR ← TU^.ESQ;
      TU^.ESQ ← T;
      T^.BAL ← 0;
      T ← TU;
   senão
      TV ← TU^.ESQ;
      TU^.ESQ ← TV^.DIR;
      TV^.DIR ← TU;
      T^.DIR ← TV^.ESQ;
      TV^.ESQ ← T;

      se (TV^.BAL = 1) então T^.BAL ← -1
      senão T^.BAL ← 0;
      fim se;

      se (TV^.BAL = -1) então TU^.BAL ← 1
      senão TU^.BAL ← 0;
      fim se;

      T ← TV;

   fim se;

   T^.BAL ← 0;
   H ← false;

fim procedimento CASO2;

procedimento INS-AVL (var T: ARVAVL; X: ELEM; var H: lógico);

   se (T = nil) então
      INÍCIO-NODO (T, X);
      H ← true
   senão
      se (X = T^.OBJ) então exit; {sai do procedimento}

      se (X < T^.OBJ) então
         INS-AVL (T^.ESQ, X, H);

         se (H) então
```

```
            caso T^.BAL
               1  : T^.BAL ← 0;
                    H ← false;
               0  : T^.BAL ← -1
               -1 : CASO1 (T, H); {rebalanceamento}
            fim caso;
         fim se

      senão
         INS-AVL (T^.DIR, X, H);

         se (H) então
            caso T^.BAL
               -1 : T^.BAL ← 0;
                    H ← false;
               0  : T^.BAL ← 1
               1  : CASO2 (T, H); {rebalanceamento}
            fim caso;
         fim se;

      fim se;
   fim se;
fim procedimento INS-AVL;
```

4.3.3 REMOÇÕES EM ÁRVORES-AVL

A remoção em árvores-AVL é muito mais complexa que a inserção. Isto é realmente verdade, embora a operação de rebalanceamento seja similar à realizada na inserção. Em particular, o rebalanceamento é efetuado, novamente, através de rotações simples ou duplas dos nodos.

Os casos mais simples são os nodos terminais e nodos com apenas um único descendente. Se o nodo a ser removido tiver duas subárvores, deve-se novamente substituí-lo pelo nodo mais à direita de sua subárvore esquerda. De maneira similar ao que ocorre no caso da inserção, é passado um parâmetro ***h***, passado por referência e de tipo lógico, indicando que a altura da árvore foi reduzida. O rebalanceamento deve ser efetuado somente quando o valor de ***h*** for verdadeiro. A variável ***h*** se torna verdadeira sempre que um nodo é localizado e removido, ou então se o próprio rebalanceamento reduzir a altura de alguma subárvore. No algoritmo, implementado a seguir, foram introduzidas as duas operações (simétricas) de balanceamento na forma de procedimentos, já que tais operações são necessárias em mais de um ponto do algoritmo de remoção. Note-se que os procedimentos ***balanceL*** e ***balanceR*** são aplicados quando os ramos esquerdo e direito, respectivamente, tiverem sido reduzidas em sua altura.

A operação do procedimento está ilustrada na Figura 4.12. Dada a árvore-AVL (a), as remoções sucessivas dos nodos com chaves 4, 8, 6, 5, 2, 1 e 7 resultam nas árvores (b)..(h). A remoção da chave 4 é, por si mesma, bastante simples, pois representa um nodo terminal. Entretanto, ela resulta em desbalanceamento do nodo 3. A correspondente operação de rebalanceamento envolve uma rotação simples à direita. O rebalanceamento se torna novamente necessário após a remoção do nodo 6. Desta vez a subárvore direita da raiz (7) é rebalanceada por meio de uma rotação

simples à esquerda. A remoção do nodo 2, embora trivial em si por possuir apenas um único descendente, exige uma complexa operação de rotação dupla à esquerda do nodo 3. O último caso, uma rotação simples à esquerda, é finalmente executada após a remoção do nodo 7, que foi de início substituído pelo elemento mais à direita de sua subárvore esquerda, isto é, pelo nodo de chave 3.

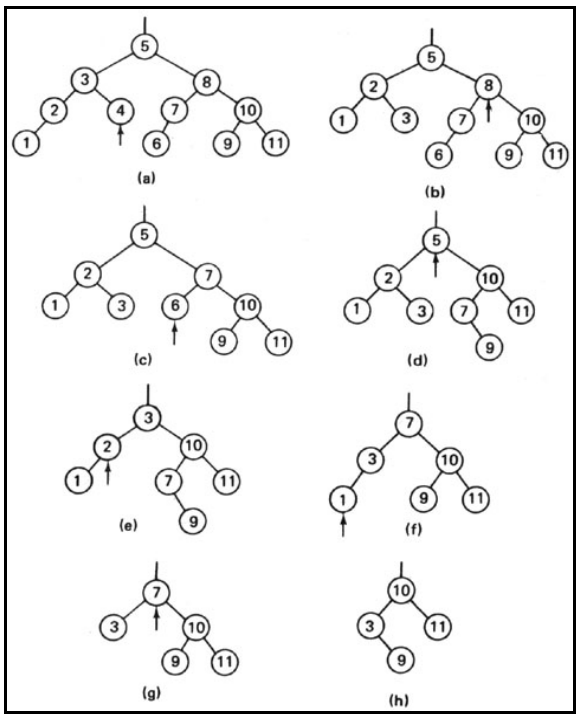


Figura 4.12 – Remoções em árvores-AVL

```
procedimento BALANCEL (var T: ARVAVL; var H: lógico);
var
   T1, T2 : ARVAVL;
   B1, B2 : BALANCO;

   caso T^.BAL
      -1 : T^.BAL ← 0;
      0  : T^.BAL ← 1;
           H ← false;
      1  : T1 ← T^.DIR;
           B1 ← T1^.BAL;

           se (B1 >= 0) então
              T^.DIR ← T1^.ESQ;
              T1^.ESQ ← T;

              se (B1 = 0) então
                 T^.BAL ← 1;
                 T1^.BAL ← -1;
                 H ← false;
              senão
                 T^.BAL ← 0;
                 T1^.BAL ← 0;
```

```
                fim se;

                T ← T1;

            senão
                T2 ← T1^.ESQ;
                B2 ← T2^.BAL;
                T1^.ESQ ← T2^.DIR;
                T2^.DIR ← T1;
                T^.DIR ← T2^.ESQ;
                T2^.ESQ ← T;

                se (B2 = 1) então T^.BAL ← -1;
                senão T^.BAL ← 0;
                fim se;

                se (B2 = -1) então T1^.BAL ← 1;
                senão T1^.BAL ← 0;
                fim se;

                T ← T2;
                T2^.BAL ← 0;

            fim se;
    fim caso;

fim procedimento BALANCEL;

procedimento BALANCER (var T: ARVAVL; var H: lógico);
var
    T1, T2 : ARVAVL;
    B1, B2 : BALANCO;

    caso T^.BAL
        1  : T^.BAL ← 0;
        0  : T^.BAL ← -1;
             H ← false;
        -1 : T1 ← T^.ESQ;
             B1 ← T1^.BAL;

             se (B1 <= 0) então
                T^.ESQ ← T1^.DIR;
                T1^.DIR ← T;

                se (B1 = 0) então
                   T^.BAL ← -1;
                   T1^.BAL ← 1;
                   H ← false;
                senão
                   T^.BAL ← 0;
                   T1^.BAL ← 0;
```

```
            fim se;

            T ← T1;

         senão
            T2 ← T1^.DIR;
            B2 ← T2^.BAL;
            T1^.DIR ← T2^.ESQ;
            T2^.ESQ ← T1;
            T^.ESQ ← T2^.DIR;
            T2^.DIR ← T;

            se (B2 = -1) então T^.BAL ← 1;
            senão T^.BAL ← 0;
            fim se;

            se (B2 = 1) então T1^.BAL ← -1;
            senão T1^.BAL ← 0;
            fim se;
            T ← T2;
            T2^.BAL ← 0;

         fim se;
   fim caso;

fim procedimento BALANCER;

procedimento DELETE (var T: ARVAVL; X: ELEM; var H: lógico);
var
   Q : ARVAVL;

   procedimento DEL (var R: ARVAVL; var H: lógico);

      se (R^.DIR <> nil) então
         DEL (R^.DIR, H);
         se (H) então BALANCER(R, H);
      senão
         Q^.OBJ ← R^.OBJ;
         Q ← R;
         R ← R^.ESQ;
         H ← true;
      fim se;

   fim procedimento DEL;

   se (T = nil) então exit; {chave não encontrada. Sair!}
   senão
      se (T^.OBJ > X) então
         DELETE (T^.ESQ, X, H);
         se (H) então BALANCEL (T, H);
      senão
         se (T^.OBJ < X) então
```

```
            DELETE(T^.DIR, X, H);
            se (H) então BALANCER (T, H);
         senão
            Q ← T;
            se (Q^.DIR = nil) então
               T ← Q^.ESQ;
               H ← true;
            senão
               se (Q^.ESQ = nil) então
                  T ← Q^.DIR;
                  H ← true;
               senão
                  DEL (Q^.ESQ, H);
                  se (H) então BALANCEL (T, H);
               fim se;
            fim se;
            desaloque (Q);
         fim se;
      fim se;
   fim se;
fim procedimento DELETE;
```

4.4 ÁRVORES HEAP E HEAPSORT

O heap é uma estrutura de dados eficiente que suporta as operações constrói, insere, retira, substitui e altera. A estrutura foi proposta por Williams (1964).

Um heap é definido como uma seqüência de itens com chaves

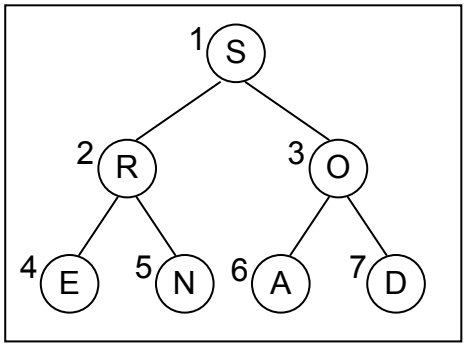
$$c[1], c[2], c[3], ..., c[n]$$

tal que

$$c[i] \geq c[2i]$$
$$c[i] \geq c[2i + 1]$$

para todo i = 1, 2, 3, ..., n/2

Observe a árvore seguinte:



Esta árvore binária representa um heap. As linhas que saem de uma chave

sempre levam a duas chaves menores no nível inferior. Conseqüentemente, a chave do nodo raiz é a maior chave do conjunto.

Uma árvore binária completa pode ser representada por um array, conforme ilustra a figura a seguir:

1	2	3	4	5	6	7
S	R	O	E	N	A	D

Esta representação é extremamente compacta e permite caminhar facilmente pelos nodos da árvore: os filhos de um nodo ***i*** estão nas posições ***2i*** e ***2i + 1*** (caso existam), e o pai de um nodo ***i*** está na posição ***i div 2***.

Um heap é uma árvore binária na qual cada nodo satisfaz a condição do heap apresentada acima. No caso da representação do heap por vetor, a maior chave está sempre na posição 1 do vetor. Os algoritmos para implementar as operações sobre o heap operam ao longo de um dos caminhos da árvore, a partir da raiz até o nível inferior da árvore.

4.2.1 HEAPSORT

O primeiro passo é construir um heap. Um método elegante foi apresentado por Floyd (1964). Dado um vetor A[1], A[2], ..., A[n], os itens A[n/2 + 1], A[n/2 + 2], ..., A[n] formam um heap pois não há dois índices ***i***, ***j*** tais que ***j = 2i*** ou ***j = 2i + 1***.

Considere o seguinte vetor:

	1	2	3	4	5	6	7
A:	O	R	D	E	N	A	S

Os itens A[4] a A[7] formam a parte inferior da árvore binária associada, onde nenhuma relação de ordem é necessária para formarem um heap. A seguir o heap é estendido para a esquerda (Esq = 3), englobando o item A[3], pais dos itens A[6] e A[7]. Neste momento a condição do heap é violada, e os itens D e S são trocados.

	1	2	3	4	5	6	7
A:	O	R	D	E	N	A	S
Esq = 3	O	R	S	E	N	A	D

A seguir o heap é novamente estendido para a esquerda (Esq = 2) incluindo o R, passo que não viola a condição do heap. Finalmente, o heap é estendido para a esquerda (Esq = 1), incluindo o item O, e os itens O e S são trocados, encerrando o processo.

	1	2	3	4	5	6	7
A:	O	R	S	E	N	A	D
Esq = 1	S	R	O	E	N	A	D

O procedimento a seguir mostra a implementação do algoritmo para construir o heap. O procedimento ***REFAZ*** reconstrói o heap conforme descrito anteriormente.

```
procedimento CONSTROI (var A: vetor; N: inteiro);
var
   Esq   : inteiro;

   procedimento REFAZ (ESQ, DIR: inteiro; var A: vetor);
   var
      I, J   : inteiro;
      X      : ELEM;

      I ← ESQ;
      J ← 2 * I;
      X ← A[I];

      enquanto (J <= DIR) faça
         se (J < DIR) então
            se (A[J] < A[J + 1]) então
               J ← J + 1;
            fim se;
         fim se;

         se (X >= A[J]) então
            break;
         fim se;

         A[I] ← A[J];
         I ← J;
         J ← 2 * i;
      fim enquanto;

      A[I] ← X;
   fim procedimento REFAZ;

   ESQ ← (N div 2) + 1;

   enquanto (ESQ > 1) faça
      ESQ ← ESQ - 1;
      REFAZ (ESQ, N, A);
   fim enquanto;

fim procedimento CONSTROI;
```

A partir do heap obtido pelo procedimento ***CONSTROI***, pega-se o item na posição 1 do vetor (raiz do heap) e troca-se com o item que está na posição ***n*** do vetor. A seguir, basta usar o procedimento ***REFAZ*** para reconstituir o heap para os itens A[1], A[2], ..., A[n – 1]. Repita estas duas operações com os ***n – 1*** itens restantes, depois com os ***n – 2*** itens, até que reste apenas 1 item.

Na figura abaixo é apresentado um exemplo do heapsort. Os valores assinalados representam os valores afetados pelo procedimento ***REFAZ***.

O programa na sequência mostra a implementação da rotina ***HEAPSORT***.

1	2	3	4	5	6	7
S	R	O	E	N	A	D
R	N	O	E	D	A	S
O	N	A	E	D	R	
N	E	A	D	O		
E	D	A	N			
D	A	E				
A	D					

```
procedimento HEAPSORT (var A: vetor; N: inteiro);
var
   ESQ, DIR : inteiro;
   X        : ELEM;

   {Aqui deve ser inserido o procedimento REFAZ}

   {Constrói o heap}
   ESQ ← (N div 2) + 1;
   DIR ← N;

   enquanto (ESQ > 1) faça
      ESQ ← ESQ - 1;
      REFAZ (ESQ, DIR, A);
   fim enquanto;

   {ordena o vetor}
   enquanto (DIR > 1) faça
      X ← A[1];
      A[1] ← A[DIR];
      A[DIR] ← X;
      DIR ← DIR - 1;
      REFAZ (ESQ, DIR, A);
   fim enquanto;

fim procedimento HEAPSORT;
```

4.5 ÁRVORES B

Até agora, restringiu-se a discussão aos casos de árvores em que cada nó possui no máximo dois descendentes, ou seja, estudaram-se apenas árvores binárias. Isto, evidentemente, é satisfatório se desejamos representar relações de parentesco de pessoas com o enfoque de árvore genealógica, ou seja, na forma de um relacionamento em que cada pessoa está associada a seus pais. Afinal de contas, ninguém possui mais do que um pai e uma única mãe. Como seria, entretanto, se fosse preferido o enfoque do ponto de vista dos descendentes? Será necessário considerar o fato de que algumas pessoas possuem mais do que dois filhos, e que suas árvores conterão nós com muitos ramos. Por falta de um termo melhor, tais árvores serão denominadas ***árvores multidirecionais***.

Naturalmente, não há nada de especial em tais estruturas, e já estudamos todos os recursos de programação e de definição de dados que permitem tratar tais

situações. Se, por exemplo, for imposto um limite superior absoluto para um número de filhos, então é possível representar os filhos através de um vetor que seja um dos componentes do registro que representa uma pessoa. Entretanto, se o número de filhos variar muito de uma pessoa para outra, isto poderá resultar em uma subutilização das áreas de memória. Neste caso, será mais apropriado representar a descendência na forma de uma lista linear, com um apontador para o descendente mais novo (ou mais velho) associado ao genitor. Uma possível definição de tipo para este caso é apresentada a seguir, e uma possível estrutura de dados está representada na Figura 4.13.

```
tipo PTR = ^PESSOA;
tipo PESSOA = registro
                 NOME : caractere;
                 IRMÃOS, DESCENDENTES : ^PESSOA;
              fim registro;
```

Entretanto, existe uma área muito prática, de interesse geral, para aplicação de árvores multidirecionais. É a construção e a manutenção de árvores de busca de grandes dimensões, nas quais as inserções e remoções são necessárias, mas para as quais a área da memória principal do computador não é suficientemente grande.

Admita-se, então, que os nós de uma árvore estejam armazenados em dispositivos de memória secundária, tais como um disco. As estruturas dinâmicas introduzidas neste capítulo são particularmente adequadas para incorporações de dispositivos de armazenamento secundário.

A principal inovação é simplesmente que os apontadores agora referem-se a áreas da memória de massa, em vez de representarem endereços de áreas da memória principal.

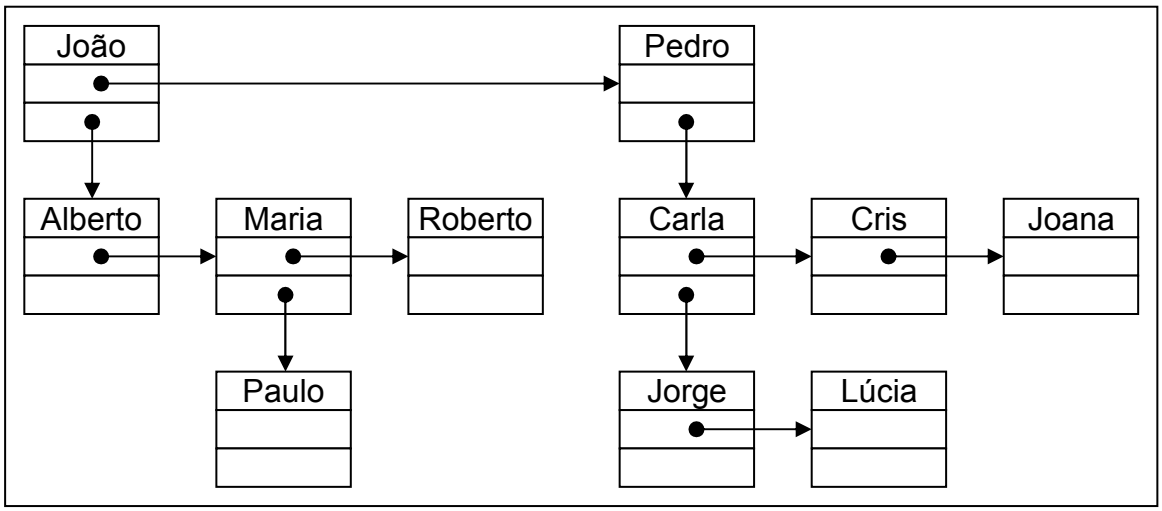


Figura 4.13 – Uma árvore multidirecional representada como árvore binária

Utilizar uma árvore binária para conjuntos de dados compostos de milhões de elementos requer, em média, aproximadamente, $\log_2 10^6$ (isto é, cerca de 20) passos de busca. Como agora, cada passo envolve um acesso de disco (com os inerentes atrasos de acesso), torna-se bastante desejável uma organização de memória capaz de permitir a redução do número de acessos. A árvore multidirecional é uma solução perfeita para este problema. Se for efetuado um acesso a um elemento localizado na memória secundária, é possível efetuar um acesso a todo um grupo

completo de elementos sem muito custo adicional. Isto sugere que a árvore seja subdividida em subárvores, e que as subárvores sejam representadas como unidades, do ponto de vista do acesso. Essas subárvores serão denominadas ***páginas***. A Figura 4.14 mostra uma árvores binária subdividida em páginas, cada qual formada por 7 nós.

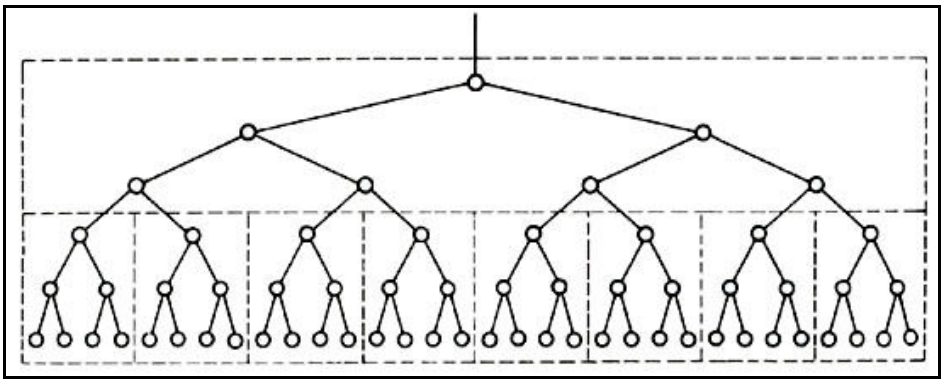


Figura 4.14 – Uma árvore binária subdividida em páginas

A redução do número de acessos a disco – cada acesso a uma página envolve agora um acesso ao disco – pode ser considerável. Suponha-se que se deseje alojar 100 nós em cada página, então a árvore de busca de milhões de elementos exigirá em média apenas $\log_{100} 10^6$ (isto é, cerca de 3) acessos à página, em vez de 20. Porém, naturalmente, se o crescimento da árvore for aleatório, então, no pior caso, pode-se ainda ter um número de acessos da ordem de 10^4. Portanto, um esquema de crescimento controlado é quase obrigatório no caso de árvores multidirecionais.

4.4.1 ÁRVORES B MULTIDIRECIONAIS

Caso se deseje definir um critério de crescimento controlado, então a exigência de um balanceamento perfeito é rapidamente eliminada, já que o custo computacional de balanceamento é muito alto. Para viabilizá-lo, é necessário relaxar as exigências de alguma forma. Um critério bastante conveniente foi proposto por R. Bayer e E. M. McCreight em 1970; todas as páginas, exceto uma, contém um número de nós entre ***n*** e ***2n***, para uma dada constante ***n***. Assim, em uma árvore com ***N*** elementos, e com páginas de, no máximo, ***2n*** nós, o pior caso exigirá $\log_n N$ acessos à página. É evidente que boa parte do custo computacional da pesquisa é devido exatamente a tais acessos. Além disso, o fator de utilização de memória, importante parâmetro de desempenho, é de pelo menos 50%, uma vez que as páginas estão sempre, no mínimo, preenchidas pela metade como decorrência da hipótese inicial. Com todas estas vantagens, o esquema envolve algoritmos comparativamente simples de busca, inserção e remoção. Estes algoritmos serão estudados a seguir em maiores detalhes.

As estruturas de dados sobre as quais os algoritmos deverão operar são denominadas ***árvores B***, e apresentam as seguintes características (***n*** é definido como sendo a ***ordem*** da árvore B):

- Cada página contém no máximo ***2n*** elementos (chaves);
- Cada página, exceto a que contém a raiz, apresenta no mínimo ***n*** elementos;
- Cada página é ou uma página terminal (folha), isto é, não possui descendentes, ou então possui ***m + 1*** descendentes, onde ***m*** é o número

de chaves contido nesta página;
- Todas as páginas terminais aparecem no mesmo nível.

A Figura 4.15 mostra uma árvore B de ordem 2 com 3 níveis. Todas as páginas contém 2, 3 ou 4 elementos; a exceção é a raiz, à qual é permitido apresentar somente um único elemento. Todas as páginas terminais aparecem no nível 3. As chaves ocorrem em ordem crescente da esquerda para a direita se a árvore B for condensada em um único nível através da inserção de descendentes entre as chaves da página de algum ancestral.

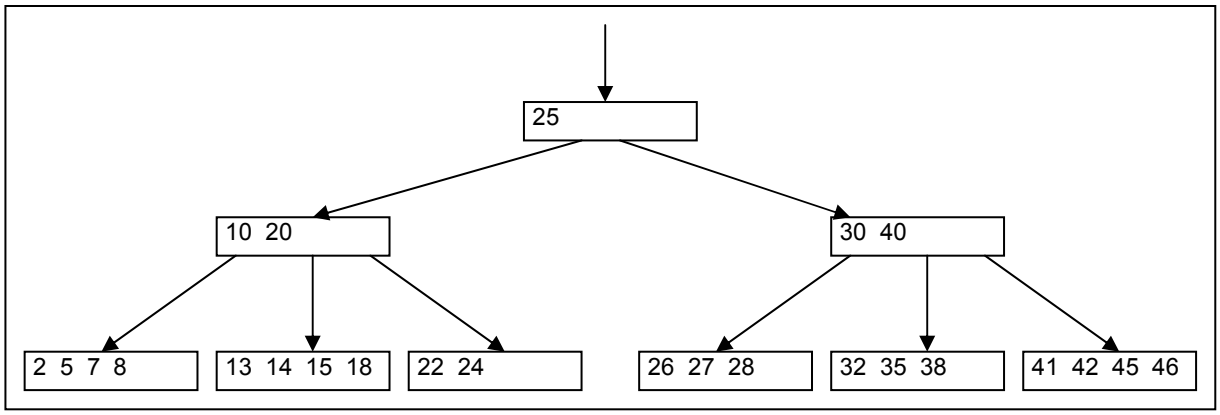


Figura 4.15 – Árvore B de ordem 2

Este arranjo representa uma extensão natural das árvores de busca binária, e determina o método de busca de um elemento, dada a respectiva chave. Considere-se uma página de forma igual à mostrada na Figura 4.16 e um dado argumento de busca ***x***.

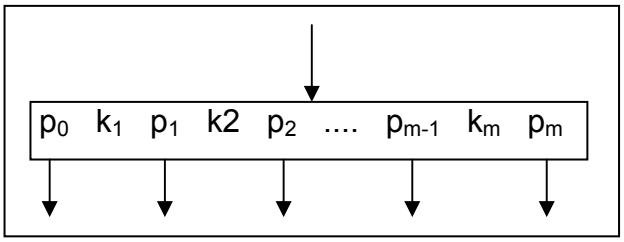


Figura 4.16 – Uma página de árvore B com ***m*** chaves

Admitindo-se que a página tenha sido movida para a memória principal, poder-se-á utilizar métodos convencionais de busca entre as chaves $k_1, ..., k_m$. Se ***m*** for suficientemente grande, pode-se aplicar a busca binária. Para ***m*** pequeno, no entanto, uma busca seqüencial ordinária pode ser utilizada. Note-se que o tempo exigido para a execução de uma operação completa de busca na memória principal é provavelmente desprezível quando comparado ao tempo necessário para se mover uma página da memória secundária para a principal. Se a busca não tiver sucesso, haverá três casos a considerar:

- $k_i < x < k_{i+1}$, para $1 \leq i < m$. A busca deve continuar na página p_i^;
- $k_m < x$. A busca deve continuar na página p_m^;
- $x < k_1$. A busca deve continuar na página p_0^;

Se, em algum caso, o apontador envolvido for ***nil***, ou seja, não houver nenhuma página de descendentes, então em toda a árvore não haverá nenhum

elemento com chave ***x***, e a busca se encerra sem sucesso.

Surpreendentemente, a operação de inserção em uma árvore B é, também, comparativamente simples. Se um elemento deve ser inserido em uma página com ***m*** < ***2n*** elementos, o processo de inserção deverá continuar operando sobre esta página. Somente as inserções em páginas completamente lotadas é que causam alterações na estrutura da árvore, podendo provocar a alocação de novas páginas. Para se compreender o que ocorre neste caso, será considerada a Figura 4.17, que ilustra a inserção da chave 22 em uma árvore B de ordem 2. Esta operação é realizada nos seguintes passos:

- Localiza-se, inicialmente, a página onde a chave 22 deverá ser inserida. Sua inserção na página ***C*** é impossível, uma vez que ***C*** já está lotada;
- A página ***C*** é decomposta em duas páginas. Para isto, uma nova página, ***D***, é alocada;
- As 2_{n+1} chaves são distribuídas igualmente entre as páginas ***C*** e ***D***, e a chave do elemento central é deslocada para um nível superior, na página dos ancestrais ***A***.

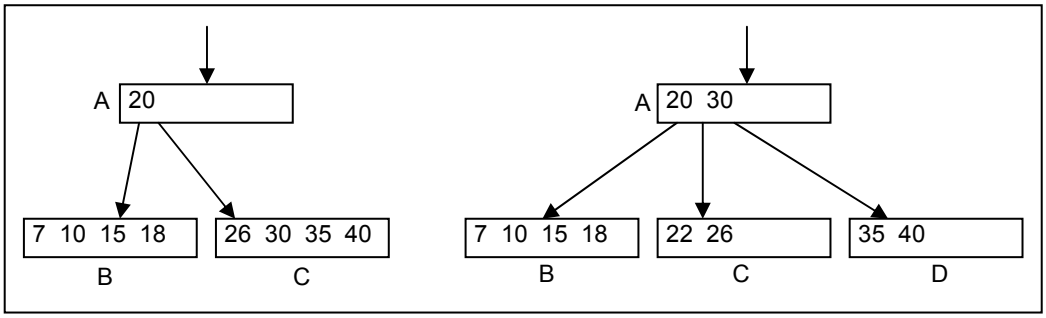


Figura 4.17 - Inserção da chave 22 na árvore B

Este esquema, bastante elegante, preserva todas as propriedades características das árvores B. Em particular, as páginas particionadas contêm exatamente ***n*** elementos cada uma. Naturalmente, a inserção de um elemento na página ancestral pode causar nova superlotação na página, provocando a propagação do particionamento. Em um caso extremo, o fenômeno poderá propagar-se até atingir a raiz. De fato, esta é a única maneira de uma árvore B crescer em profundidade. A árvore B possui, portanto, um estranho modo de crescimento: ela cresce a partir das folhas, em direção à raiz.

Agora será desenvolvido detalhadamente um algoritmo a partir destas especificações preliminares. Já é fácil intuir que uma definição recursiva será a mais conveniente, devido à propriedade, apresentada pelo processo de particionamento, de se propagar para trás ao longo da trajetória de busca. Assim sendo, a estrutura geral do algoritmo deverá ser similar à da inserção em árvores balanceadas, embora os detalhes sejam diferentes. Antes de tudo, deve-se formular uma definição da estrutura das páginas. Os elementos serão representados na forma de vetores.

```
tipo PPTR = ^PAGINA;
     INDICE = [0..2*N];
     ITEM = registro
               KEY : inteiro;
               P : PPTR;
               COUNT : inteiro;
            fim registro;
```

```
PAGINA = registro
            M : INDICE;
            P0 : PPTR;
            E : vetor [1..2*N] de ITEM;
         fim registro;
```

Novamente, o componente COUNT dos elementos da árvore incluirá todos os tipo de informações adicionais que possam estar associadas a cada elemento, porém sem participar de forma alguma do processo de busca propriamente dito. Note-se que, em cada página, há espaço para alojar ***2n*** elementos. O campo ***m*** indica o número efetivo de elementos existentes na página em cada instante. Como, exceto para a página que contém a raiz, vale a relação $m \geq n$, então pode-se garantir que, permanentemente, no mínimo 50% da área de memória estejam de fato utilizados.

O algoritmo de ***busca*** e ***inserção*** em árvore B foi escrito na forma de um procedimento denominado ***search*** (busca). Sua estrutura principal é trivial, e muito semelhante à utilizada no programa de busca em árvore binária balanceada, com a exceção de que as decisões tomadas durante o processamento não são meras escolhas binárias. Em vez disso, as buscas que são efetuadas internamente a uma dada página são implementadas na forma de uma busca binária no vetor "E" de elementos.

O algoritmo de **inserção** é construído na forma de um procedimento separado, apenas por questões de clareza. Este algoritmo é ativado após o algoritmo de busca ter detectado a necessidade de algum elemento ser mudado de nível na árvore, em direção à sua raiz. Esta condição é indicada por meio de um parâmetro booleano ***h***, que assume um papel semelhante ao do algoritmo de inserção em árvores balanceadas. Neste algoritmo, ***h*** indica que a subárvore cresceu. Se ***h*** for verdadeiro, um segundo parâmetro ***u*** representará o elemento cujo nível deve "subir" na árvore. O novo elemento é imediatamente passado adiante, através do parâmetro ***u***, para a página terminal, com a finalidade de promover a inserção propriamente dita. O esquema está esboçado a seguir.

```
procedimento SEARCH (X: inteiro; A: PPTR; var H: lógico;
                     var U: ITEM);

   se (A = nil) então   {X não se encontra na árvore, inserir}
      Atribui o valor de X ao elemento U;
      Atribui TRUE a H, indicando que um elemento U "subirá"
      na árvore;
   senão
      com (A^) faça
         busca binária de X no vetor E;

         se (ACHOU) então
            processar dados;
         senão
            SEARCH (X, descendente, H, U);

            se (H) então {um elemento "subiu" na árvore}
               se (número de elementos na página A^<2N) então
                  inserir U na página A^ e atribuir para H o
                  valor TRUE;
               senão
```

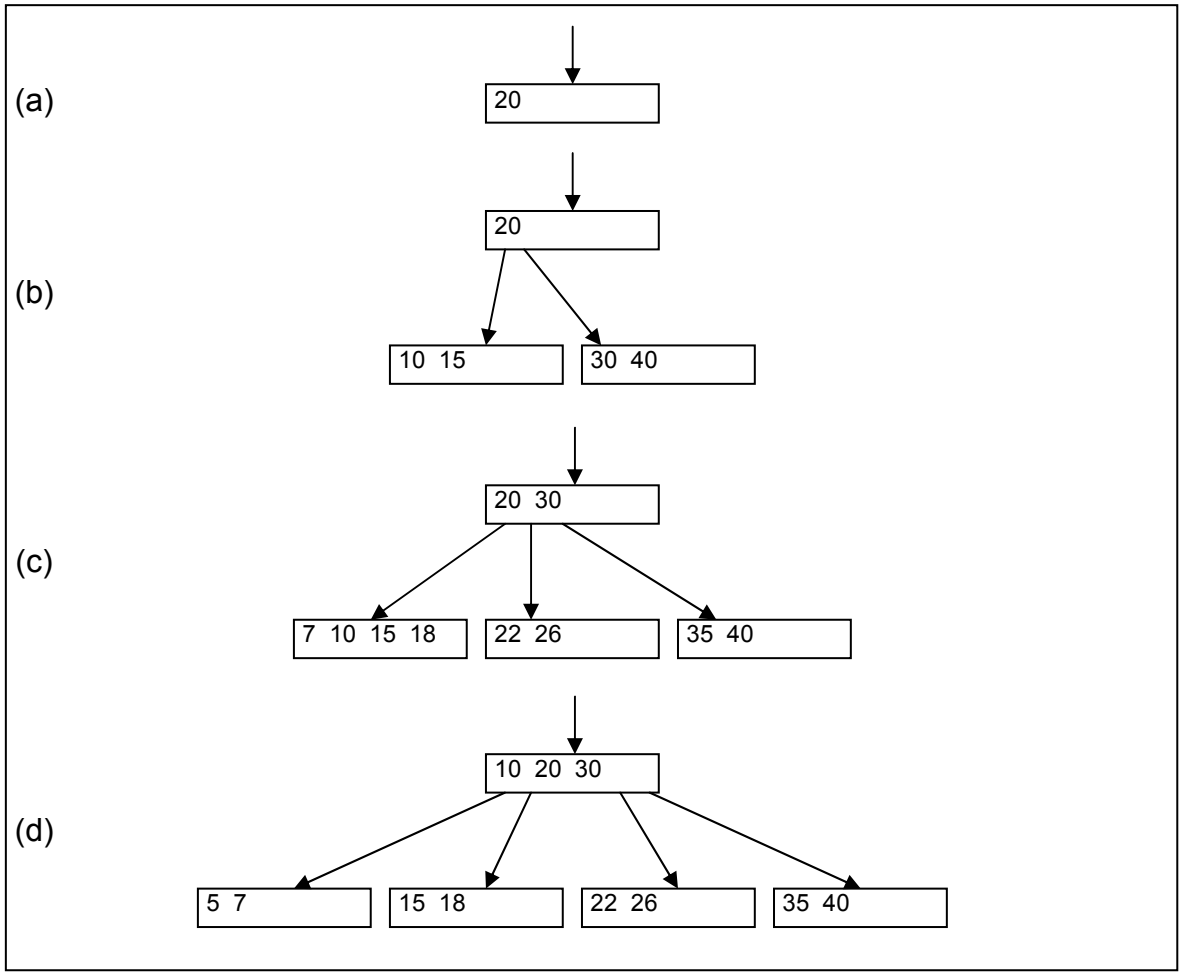
```
                    desmembra a página e faz o elemento central
                    "subir" na árvore;
                fim se;
            fim se;
        fim se;
    fim com;
fim se;
fim procedimento SEARCH;
```

Se o parâmetro ***h*** for verdadeiro após a chamada da rotina ***SEARCH*** no programa principal, é exigido um desmembramento da página que contém a raiz. Dado que tal página executa uma tarefa totalmente distinta da executada pelas demais, então o seu tratamento deverá ser programado em separado. Consiste apenas na alocação de uma nova página para a raiz e na inserção do elemento único, dado pelo parâmetro ***u***, nesta página. Como conseqüência, a nova página para raiz conterá apenas um único elemento.

A Figura 4.18 mostra o resultado do algoritmo anterior para construir uma árvore B a partir da seguinte seqüência de inserção de chaves:

20; 40 10 30 15; 35 7 26 18 22; 5; 42 13 46 27 8 32; 38 24 45 25;

Os "ponto e vírgulas" indicam as posições instantâneas, tomadas quando da alocação de cada página. A inserção da última chave provoca dois desmembramentos, bem como a alocação de três páginas adicionais.



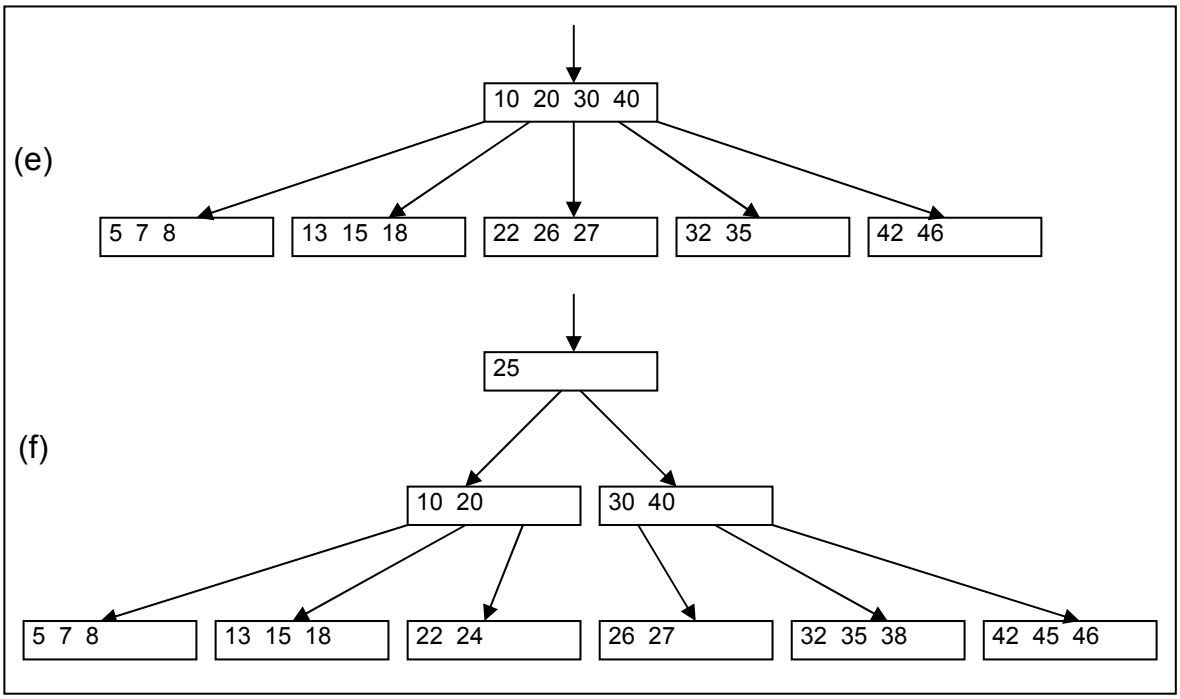


Figura 4.18 – Crescimento de uma árvore B de ordem 2

O comando ***com*** neste programa possui um significado especial. Em primeiro lugar, tal comando indica que os identificadores dos componentes da página se referem implicitamente à página A^ em todos os comandos que compõem o corpo do comando ***com*** associados. Se, de fato, as páginas residirem na memória secundária – como certamente seria necessário em grandes sistemas de banco de dados –, então o comando ***with*** poderá, adicionalmente, incorporar em sua interpretação a necessidade de uma transferência de página desejada para a memória principal. Dado que cada ativação da rotina ***SEARCH*** impõe a transferência de uma página para a memória principal, são necessárias, no máximo, $k = \log_n N$ chamadas recursivas para uma árvore que contenha ***N*** elementos. Portanto, o sistema deve ser capaz de acomodar simultaneamente ***k*** páginas na memória principal. Isto é um fator limitante para a extensão ***2n*** das páginas. Na realidade, necessita-se acomodar mais de ***k*** páginas, porque a operação de inserção pode causar novos desmembramentos de página. Uma conseqüência deste fato é que se torna conveniente alocar permanentemente a página que contém a raiz na memória principal, porque cada operação de busca certamente percorre, sem exceção, a página que contém a raiz da árvore.

Outro mérito da organização da árvore B é sua adequação e economia nos casos de pura atualização seqüencial de todo o banco de dados. Cada página e, neste caso, copiadas para a memória principal exatamente uma única vez.

A ***remoção*** de elementos da árvore B é bastante simples em essência. Seus detalhes, porém, são complexos. É possível distinguir-se duas diferentes circunstâncias:

- O elemento a ser removido é uma folha da árvore, ou seja, está armazenado em sua página terminal. Neste caso, o correspondente procedimento de remoção é bastante simples;
- O elemento não se encontra em uma página terminal, devendo ser substituído por um dos dois elementos a ele adjacentes na árvore, e que

se encontram em uma página terminal, permitindo deste modo uma fácil remoção dos mesmos.

No segundo caso, a localização da chave adjacente é análoga à operação anteriormente utilizada para a remoção em árvores binárias. Desce-se a árvore ao longo do seu apontador mais à direita, até encontrar a página terminal ***P***, e substituindo-se o elemento a ser removido pelo elemento mais à direita de ***P***, e reduzindo-se então de uma unidade o valor armazenado em ***P***. De qualquer maneira, a redução da dimensão deve ser imediatamente seguida de um teste do número de elementos ***m*** da página recém-reduzida, isto porque, se ***m < n***, então a principal característica das árvores B seria violada. Alguma operação adicional terá de ser executada. Esta condição de ***underflow*** é indicada pelo parâmetro lógico ***h***, passado por referência.

A única saída consiste em “emprestar” ou anexar um elemento de uma das páginas vizinhas, por exemplo, de ***Q***. Dado que isto envolve uma operação de cópia da página ***Q*** na memória principal, fica-se tentado a explorar ao máximo esta situação inconveniente e onerosa, executando-se a incorporação de mais de um elemento por vez. A estratégia usual é distribuir eqüitativamente entre as páginas ***P*** e ***Q*** os elementos nelas encontrados. Esta operação toma o nome de ***balanceamento de página***.

Naturalmente, poderá ocorrer que não haja nenhum elemento a ser incorporado até que ***Q*** tenha atingido sua dimensão mínima ***n***. Neste caso, o número total de elementos presentes nas páginas ***P*** e ***Q*** é ***2n – 1***; pode-se efetuar a fusão das duas páginas em uma única, incorporando-se antes o elemento central da página que contém os ancestrais de ***P*** e ***Q***, e, depois, descartar completamente a página ***Q***. O funcionamento do método pode ser visualizado considerando-se, por exemplo, a remoção da chave 22 na Figura 4.17. Novamente, a remoção da chave central da página que contém os ancestrais provoca a redução das dimensões da página para um nível abaixo do limite permitido ***n***, exigindo que seja tomada outra atitude especial (quer balanceamento, quer fusão) no nível seguinte. No caso extremo, a fusão de página pode até mesmo ser propagada por toda a extensão da trajetória, subindo em direção à raiz da árvore. Se a dimensão da raiz for reduzida a zero, ela própria será removida, causando uma redução na altura da árvore B. Esta é, de fato, a única situação em que uma árvore B logra reduzir-se em altura. A Figura 4.19 mostra a degeneração gradual da árvore B da Figura 4.18 como conseqüência da gradual remoção das chaves:

25 45 24; 38 32; 8 27 46 13 42; 5 22 18 26; 7 35 15;

Novamente os ponto e vírgulas marcam as posições em que foram tomados os instantâneos, e que corresponde às situações em que ocorrem eliminações de páginas. O algoritmo de remoção está incluído no algoritmo seguinte na forma de um procedimento. A semelhança de suas estruturas em relação às utilizadas no algoritmo de remoção de árvores balanceadas é particularmente notável.

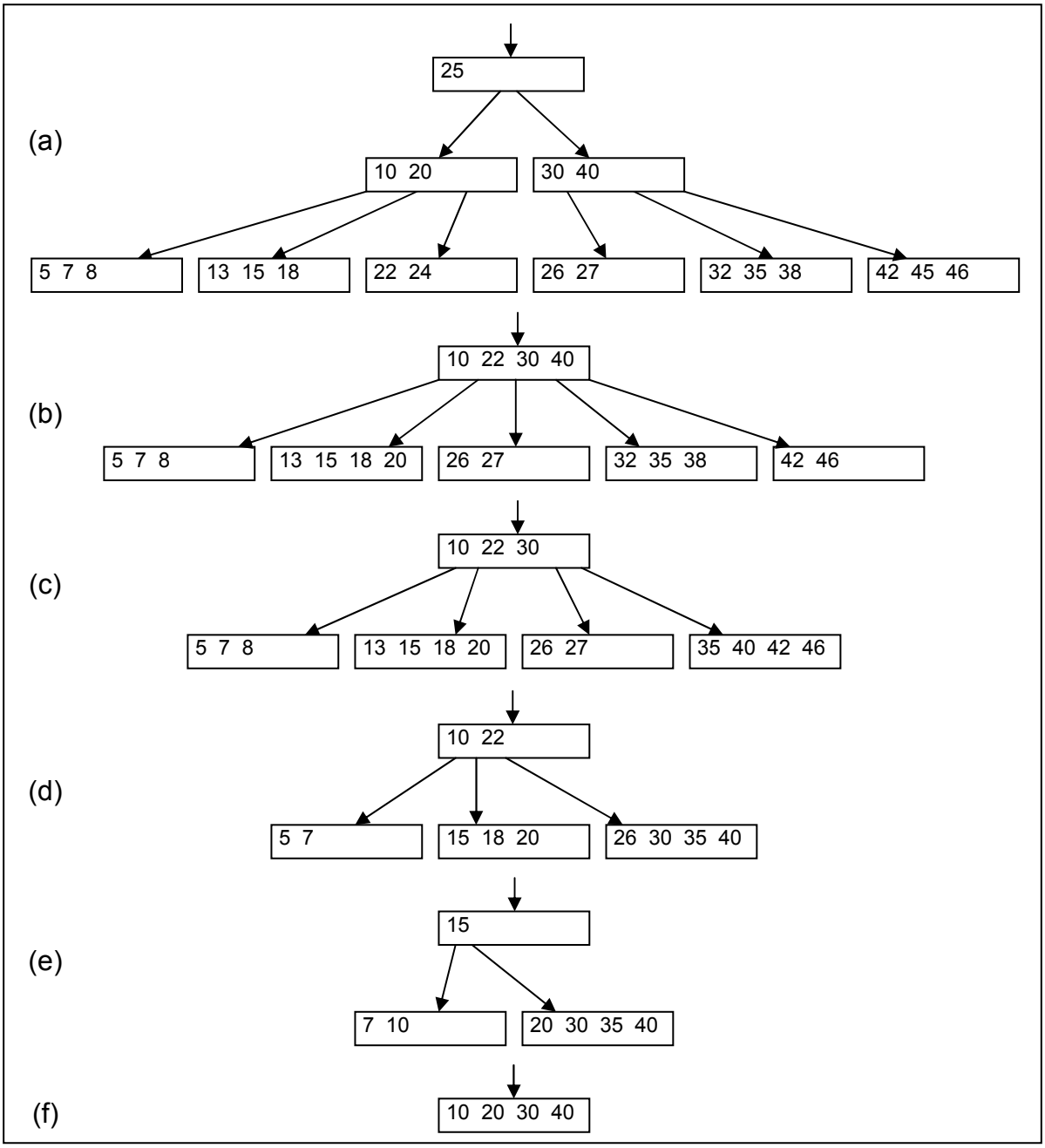


Figura 4.19 – Degeneração de uma árvore B de ordem 2

```
const N = 2;

tipo PPTR = ^PAGINA;
     INDICE = [0..2*N];
     ITEM = registro
               KEY : inteiro;
               P : PPTR;
               COUNT : inteiro;
            fim registro;
     PAGINA = registro
                 M : INDICE;
                 P0 : PPTR;
                 E : vetor [1..2*N] de ITEM;
```

```
            fim registro;

procedimento SEARCH (X: integer; A: PPTR; var H: lógico;
                     var V: ITEM);
{busca a chave x na árvore B com raiz A; se for encontrada,
incrementa o contador. Caso contrário, inserir novo elemento com
chave X. Se um elemento "sobe" na árvore, guardar em V o seu
valor. H = "a árvore ficou mais alta"}

var
   I, L, R : inteiro;
   B : PPTR;
   U : ITEM;

   se (A = nil) então
      H ← TRUE;

      com (V) faça
         KEY ← X;
         COUNT ← 1;
         P ← nil;
      fim com;
   senão
      com (A^) faça
         L ← 1;
         R ← M + 1; {busca binária}

         enquanto (L < R) faça
            I ← (L + R) div 2;
            se (E[I].KEY <= X) então
               L ← I + 1;
            senão
               R ← I;
            fim se;
         fim enquanto;

         R ← R - 1;
         se ((R > 0) e (E[R].KEY = X)) então
            INC(E[R].COUNT);
         senão {o elemento não se encontra nesta página}
            se (R = 0) então
               SEARCH (X, P0, H, U);
            senão
               SEARCH (X, E[R].P, H, U);
            fim se;

            se (H) então {insere U no lado direito de E[R]}
               se (M < 2*N) então
                  H ← FALSE;
                  M ← M + 1;

                  para I ← M até (R+2) faça {decrescente}
```

```
                        E[I] ← E[I-1];
                    fim para;

                    E[R+1] ← U;
                senão
                    aloque (B); {divide A em A, B e atribui ao}
                    se (R <= N) então {elemento central V}
                        se (R = N) então
                            V ← U;
                        senão
                            V ← E[N];

                            para I ← N até (R+2) faça {decresc.}
                                E[I] ← E[I-1];
                            fim para;

                            E[R+1] ← U;
                        fim se;

                        para I ← 1 até N faça
                            B^.E[I] ← A^.E[I+N];
                        fim para;

                    senão {insere na página direita}
                        R ← R - N;
                        V ← E[N+1];

                        para I ← 1 até (R-1) faça
                            B^.E[I] ← A^.E[I+N+1];
                        fim para;

                        B^.E[R] ← U;

                        para I ← (R+1) até N faça
                            B^.E[I] ← A^.E[I+N];
                        fim para;
                    fim se;

                    M ← N;
                    B^.M ← N;
                    B^.P0 ← V.P;
                    V.P ← B;
                fim se;
            fim se;
        fim se;
    fim com;
fim se;

fim procedimento SEARCH;
```

```
procedimento UNDERFLOW (C, A: PPTR; S : inteiro;
                        var H: lógico);
{A = página em que ocorreu overflow; C = página dos ancestrais,
S = índice do elemento eliminado em C}

var
   B: PPTR;
   I, K, MB, MC : inteiro;

   se (S < MC) então
      S ← S + 1;
      B ← C^.E[S].P;
      MB ← B^.M;
      K ← (MB - N + 1) div 2; (num. elementos na página B}
      A^.E[N] ← C^.E[S];
      A^.E[N].P ← B^.P0;

      se (K < 0) então {transfere K elementos de B para A}
         para I ← 1 até (K-1) faça
            A^.E[I+N] ← B^.E[I];
         fim para;

         C^.E[S] ← B^.E[K];
         C^.E[S].P ← B;
         B^.P0 ← B^.E[K].P;
         MB ← MB - K;

         para I ← 1 até MB faça
            B^.E[I] ← B^.E[I+K];
         fim para;

         B^.M ← MB;
         A^.M ← N-1+K;
         H ← FALSE;

      senão {fusão das páginas A e B}
         para I ← 1 até N faça
            A^.E[I+N] ← B^.E[I];
         fim para;

         para I ← S até (MC-1) faça
            C^.E[I] ← C^.E[I+1];
         fim para;

         A^.M ← 2*N;
         C^.M ← MC-1;
         H ← MC<=N;

         desaloque (B);
      fim se;
```

```
   senão {B = página à esquerda de A}
      se (S = 1) então
         B ← C^.P0;
      senão
         B ← C^.E[S-1].P;
      fim se;

      MB ← B^.M + 1;
      K ← (MB - N) div 2;

      se (K > 0) então {transfere K elementos da pág. B p/ A}
         para I ← N-1 até 1 faça {contador decrescente}
            A^.E[I+K] ← A^.E[I];
         fim para;

         A^.E[K] ← C^.E[S];
         A^.E[K].P ← A^.P0;
         MB ← MB - K;

         para I ← K-1 até 1 faça {contador decrescente}
            A^.E[I] ← B^.E[I+MB];
         fim para;

         A^.P0 ← B^.E[MB].P;
         C^.E[S] ← B^.E[MB];
         C^.E[S].P ← A;
         B^.M ← MB - 1;
         A^.M ← N - 1 + K;
         H ← FALSE;

      senão {fusão das páginas A e B}
         B^.E[MB] ← C^.E[S];
         B^.E[MB].P ← A^.P0;

         para I ← 1 até N-1 faça
            B^.E[I+MB] ← A^.E[I];
         fim para;

         B^.M ← 2*N;
         C^.M ← MC - 1;
         H ← MC<=N;

         desaloque (A);
      fim se;
   fim se;

fim procedimento UNDERFLOW;
```

```
procedimento DELETE (X: inteiro; A: PPTR; var H: lógico);
{busca e remove a chave X da árvore A; se ocorrer Underflow da
página, faz-se o balanceamento ou fusão com a página adjacente.}

var
   I, L, R : inteiro;
   Q : PPTR;

   procedimento DEL (P: PPTR; var H: lógico);
   var Q: PPTR;

      com (P^) faça
         Q ← E[M].P;

         se (Q <> nil) então
            DEL(Q, H);
            se (H) então
               UNDERFLOW(P, Q, M, H);
            fim se;
         senão
            P^.E[M].P ← A^.E[R].P;
            A^.E[R] ← P^.E[M];
            M ← M - 1;
            H ← M<N;
         fim se;

      fim com;
   fim procedimento DEL;

   se (A = nil) então {X não se encontra na árvore}
      H ← FALSE;
   senão
      com (A^) faça
         L ← 1;
         R ← M + 1;

         enquanto (L < R) faça
            I ← (L + R) div 2;

            se (E[I].KEY < X) então
               L ← I + 1;
            senão
               R ← I;
            fim se;

         fim enquanto;

         se (R = 1) então
            Q ← P0;
         senão
            Q ← E[R-1].P;
         fim se;
```

```
            se ((R <= M) e (E[R].KEY = X)) então
            {o elemento foi encontrado e agora retirado}
               se (Q = nil) então {A é uma página terminal}
                  M ← M - 1;
                  H ←  M<N;

                  para I ← R até M faça
                     E[I] ← E[I+1];
                  fim para;

               senão
                  DEL (Q, H);
                  se (H) então
                     UNDERFLOW (A, Q, R-1, H);
                  fim se;
               fim se;
            senão
               DELETE (X, Q, H);
               se (H) então
                  UNDERFLOW (A, Q, R-1, H);
               fim se;
            fim se;
         fim com;
      fim se;

fim procedimento DELETE;
```

4.6 OUTROS TIPOS DE ÁRVORES E SUAS REPRESENTAÇÕES

UNIDADE 5 – PESQUISA DE DADOS

5.1 MÉTODOS DE BUSCA

As buscas estão entre as tarefas mais freqüentes encontradas em programação de computadores. Há diversas variações básicas sobre o assunto e uma grande variedade de algoritmos foram desenvolvidos com esta finalidade. A hipótese básica no texto que segue é que a coleção de dados, entre os quais um determinado elemento deve ser procurado, é fixa. Admitiremos que este conjunto de ***N*** elementos seja representado através de um vetor, por exemplo como:

```
var
   A: vetor [1..N] de ITEM;
```

Tipicamente, o tipo ***item*** apresenta uma estrutura de registro, contendo um campo que atua como chave para a pesquisa. A tarefa consiste, então, em se encontrar um elemento de ***A*** cujo campo de chave seja igual a um argumento de busca ***X*** fornecido. O índice ***I*** resultante, que satisfaz à condição ***A[i].CHAVE = X*** permite então o acesso aos outros campos do elemento encontrado. Como o interesse básico deste estudo é exclusivamente o da busca, não sendo importantes os dados associados à chave, admitiremos que o tipo ***ITEM*** seja composto apenas do campo de chave, ou seja, o dado é a própria chave.

5.1.1 BUSCA LINEAR

Nos casos em que não se dispõe de informações adicionais sobre os dados a serem pesquisados, o procedimento mais óbvio é uma busca seqüencial por todo o vetor, aumentando-se desta maneira passo a passo o tamanho da região do vetor em que não figura o elemento desejado. Este procedimento é chamado ***busca linear***. Uma busca deste tipo termina quando for satisfeita uma das duas condições seguintes:

- O elemento é encontrado, isto é, $A_I = X$;
- Todo o vetor foi analisado, mas o elemento não foi encontrado.

Isto resulta no algoritmo abaixo:

```
I ← 1;

enquanto ((I <= N) e (A[I] <> X)) faça
   I ← I + 1;
fim enquanto;
```

Quando ***I*** for maior que ***N***, isto significa que não foi encontrado o elemento.

Evidentemente, o término das repetições é garantido pois, em cada passo, ***I*** é incrementado e, portanto, certamente alcançará o limite ***N*** após um número finito de passos, caso não exista o elemento desejado no vetor.

Obviamente, cada passo requer o incremento do índice e a avaliação de uma expressão booleana. O problema seguinte consiste em verificar se esta tarefa

pode ser simplificada, e portanto, a busca ser acelerada? A única possibilidade de se obter tal efeito consiste em se encontrar uma simplificação da expressão booleana, que depende certamente de dois fatores. Assim, é preciso estabelecer uma condição baseada em um único fator que implique nos dois fatores dos quais depende a expressão. Isto é possível desde que se possa garantir que tal elemento será encontrado. Para tanto introduz-se um elemento adicional, com valor ***X***, no final do vetor. A este elemento auxiliar dá-se o nome de ***sentinela***, porque ele evita que a busca avance o limite demarcado pelo índice. O vetor ***A*** será então declarado da seguinte forma:

```
var
   A: vetor [1..N+1] de ITEM;
```

e o algoritmo de busca linear com sentinela torna-se

```
A[N+1] ← X;
I ← 1;

enquanto (A[I] <> X) faça
   I ← I + 1;
fim enquanto;
```

Evidentemente, ***I = N+1*** implica que não foi encontrado o elemento desejado (exceto o correspondente à sentinela).

5.1.2 BUSCA BINÁRIA

É obvio que não existem meios de acelerar uma busca sem que se disponha de maiores informações acerca do elemento a ser localizado. Sabe-se também que uma busca pode ser mais eficiente se os dados estiverem ordenados. Suponha-se, por exemplo, uma lista telefônica em que os nomes não estejam listados em ordem alfabética. Uma coisa dessas seria completamente inútil. A seguir será apresentado, com base neste raciocínio, um algoritmo que explora o ato do vetor estar ordenado.

A idéia principal é a de testar um elemento sorteado aleatoriamente, por exemplo A_M, e compará-lo com um argumento de busca ***X***. Se tal elemento for igual a ***X***, a busca termina, se for menor do que ***X***, conclui-se que todos os elementos com índices menores ou iguais a ***M*** podem ser eliminados dos próximos testes, e se for maior que ***X***, todos aqueles que possuem índices maior ou igual a ***M*** podem ser também eliminados. Isto resulta no algoritmo abaixo, denominado ***busca binária***; ele utiliza duas variáveis-índices ***L*** e ***R***, que marcam, respectivamente, os limites Esquerdo e Direito da região de ***A*** em que um elemento ainda pode ser encontrado.

```
L ← 1;
R ← N;
FOUND ← Falso;

enquanto ((L <= R) e não(FOUND)) faça
   M ← (L + R) div 2;

   se (A[M] = X) então
```

```
      FOUND ← Verdadeiro;
   senão
      se (A[M] < X) então
         L ← M + 1;
      senão
         R ← M – 1;
      fim se;
   fim se;
fim enquanto;
```

Embora a escolha de M possa ser arbitrária no sentido de que o algoritmo funciona independentemente dele, o valor desta variável influencia na eficiência do algoritmo. Torna-se absolutamente claro que a meta é eliminar, em cada passo, o maior número possível de elementos em futuras buscas, sem levar em consideração o resultado da comparação. A solução ótima é escolher a mediana dos elementos, porque esta escolha elimina, em qualquer caso, metade dos elementos do vetor. Como conseqüência, o número máximo de passos será $\log_2 N$, arredondado para cima até o número inteiro mais próximo. Com isso, este algoritmo permite uma drástica melhora do desempenho em relação ao método de busca linear, onde o número esperado de comparações é $N/2$.

5.2 PROCESSAMENTO EM CADEIAS

5.2.1 INTRODUÇÃO

Entende-se por ***cadeia*** uma seqüência qualquer de elementos, denominados ***caracteres***. Os caracteres, por sua vez, são elementos escolhidos de um conjunto denominado ***alfabeto***. Por exemplo, 0110010 é uma cadeia com alfabeto {0, 1}.

As cadeias aparecem, em computação, no processamento de textos, palavras, mensagens, códigos, etc. com aplicações tão diversas quanto o tratamento computacional de dicionários, mensagens de correio eletrônico, seqüenciamento de DNA's em biologia computacional, criptografia de textos confidenciais e muitos outros. De fato, a importância do processamento de cadeias na computação vem crescendo consideravelmente, até mesmo para computadores de uso pessoal, nos quais a edição de textos tem sido a principal aplicação.

Nesta seção será abordado o problema do ***casamento de cadeias***. Dadas duas cadeias de caracteres, o problema consiste em verificar se a primeira delas contém a segunda, isto é, se os caracteres que compõem a segunda cadeia aparecem seqüencialmente também na primeira.

5.2.2 O PROBLEMA DO CASAMENTO DE CADEIAS

Já foi mencionado que uma cadeia é uma seqüência de caracteres escolhidos de um conjunto chamado alfabeto. O número de caracteres da cadeia é o seu ***comprimento***.

Sejam X, Y duas cadeias de caracteres com comprimentos n, m, respectivamente, $n \geq m$. Supõe-se que X e Y sejam representadas por vetores com

elementos x_i e y_j, $1 \leq i \leq n$ e $1 \leq j \leq m$. ***Y*** é uma subcadeia de ***X*** quando ***Y*** for uma subseqüência de elementos consecutivos de ***X***, isto é, quando existir algum inteiro $L \leq n - m$, tal que $x_{L+j} = y_j$, $1 \leq j \leq m$.

O problema do casamento de cadeias consiste em verificar se ***Y*** é subcadeia de ***X***. Em caso positivo, localizar ***Y*** em ***X***. Diz-se, então, que houve um casamento de ***Y*** com ***X*** na posição ***L + 1***.

Por exemplo, se ***X*** é a cadeia ***baaabea*** e ***Y*** é ***aabe***, então ***Y*** é subcadeia de ***X*** e a solução do problema do casamento de cadeias deve indicar que ***Y*** aparece em ***X*** a partir do terceiro caractere, como indica a Figura 5.8. Por outro lado, se ***Y*** é formada pela seqüência ***aeb***, então não é subcadeia de ***X***.

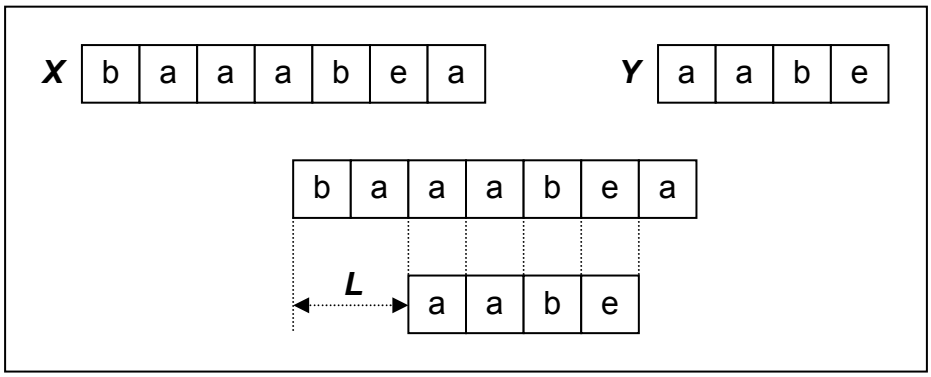


FIGURA 5.8 – O problema do casamento de cadeias

5.2.3 O ALGORITMO DA FORÇA BRUTA

Um método bastante simples para se verificar se ***Y*** é subcadeia de ***X*** consiste em examinar todas as possíveis situações. Se ***Y*** for subcadeia de ***X***, então ***Y*** se encontra em ***X***, deslocado de ***L*** posições à esquerda. A idéia consiste em comparar ***Y*** com a subcadeia de tamanho ***m*** de ***X***, que se inicia no caractere x_{L+1}, para todos os valores possíveis de ***L***. Ou seja, $L = 0, 1, 2, ..., n - m$. A Figura 5.9 ilustra a aplicação do algoritmo da força bruta para as mesma cadeias ***X*** e ***Y*** da Figura 5.8. O algoritmo, de início, compararia a subcadeia $X_1 = baaa$, iniciada em x_1, com ***Y***; em seguida, de início, a subcadeia $X_2 = aaab$, iniciada em x_2, com ***Y***. Nessas duas situações, o teste seria negativo. Na iteração seguinte, em que a subcadeia $X_3 = aabe$ é considerada, verifica-se o casamento.

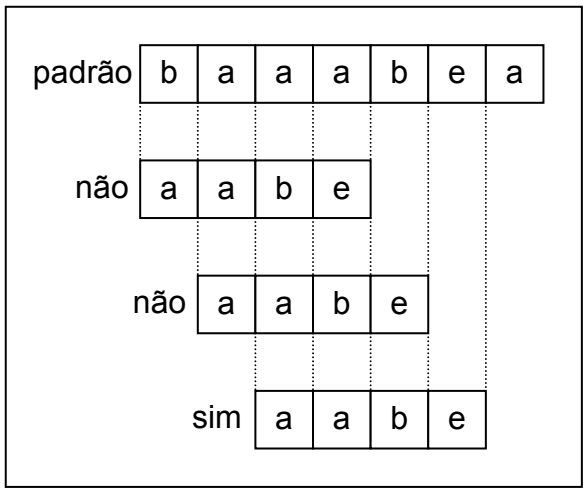


Figura 5.9 – O algoritmo da força bruta

O algoritmo a seguir descreve o processo. A variável ***L*** indica o número de caracteres, na cadeia ***X***, que antecedem cada subcadeia considerada. A variável lógica ***TESTE*** é utilizada para informar a ocorrência ou não de casamento. Para tornar o método mais eficiente, utilizou-se o princípio de abandonar a verificação de uma subcadeia no meio do processo, se for detectado que um casamento não é possível. Por exemplo, se o ***i-ésimo*** caractere da subcadeia considerada não coincide com o ***i-ésimo*** caractere de ***Y***, não há necessidade de comparar os caracteres de ordem ***i + 1, i + 2, ..., m***.

```
função CASAMENTO (X, Y : caractere): inteiro;
{retorna a posição onde ocorre o casamento e 0 caso não ocorra
casamento de cadeias de caracteres}
var
   L, N, M, I : inteiro;
   TESTE : lógico;

   N ← COMPRIMENTO(X); {COMPRIMENTO é função que retorna o}
   M ← COMPRIMENTO(Y); (comprimento da variável parâmetro}

   para L ← 0 até (N – M) faça
      I ← 1;
      TESTE ← Verdadeiro;
      enquanto ((I <= M) e (TESTE)) faça
         se (X[L + I] = Y[I]) então
            I ← I + 1;
         senão
            TESTE ← Falso;
         fim se;
      fim enquanto;

      se (TESTE) então
         CASAMENTO ← L + 1;
         exit;
      fim se;
   fim para;

   CASAMENTO ← 0;

fim função CASAMENTO;
```

5.2.4 O ALGORITMO DE KNUTH, MORRIS E PRATT

Por volta de 1970, D. E. Knuth, J. H. Morris e V. R. Pratt inventaram um algoritmo que exige, essencialmente, cerca de ***N*** comparações de caracteres, mesmo nos piores casos. O novo algoritmo é baseado no fato de que, no início de cada comparação de um novo padrão, informações valiosas podem estar sendo descartadas e, talvez, algumas destas informações poderão ser utilizadas para se agilizar a busca no texto da cadeia.

O exemplo a seguir auxiliará na compreensão do algoritmo. Dada uma cadeia ***X*** = BABABAEABAEABAEAABAEABAEAAEBA, desejamos saber se a

subcadeia ***Y*** = ABAEABAEAAE está contida na cadeia ***X***.

A primeira etapa do algoritmo de Knuth, Morris e Pratt consiste em descobrir padrões de comportamento na subcadeia ***Y***. Estes padrões de comportamento poderão ser armazenados num vetor e depois utilizados para agilizar a busca. A Figura 5.10 mostra o processo de obtenção deste vetor.

O algoritmo que analisa a subcadeia ***Y*** armazenando os padrões de comportamento num vetor é apresentado a seguir.

```
const TAM = 11; {constante do tamanho da subcadeia Y}
tipo VET_PADROES = vetor [1..TAM] de inteiro;

procedimento FIND_PADROES (Y : caractere;
                           var PADROES: VET_PADROES);
var  I, J, K, M    : inteiro;

   para I ← 1 até TAM faça
      PADROES[I] ← 0;
   fim para;
   J ← 0;
   K ← 1;
   M ← COMPRIMENTO(Y); {ou M ← TAM}

   enquanto (K < M) faça
      se (Y[K + 1] = Y[J + 1]) então
         K ← K + 1;
         J ← J + 1;
         PADROES[K] ← J;
      senão
         se (J = 0) então
            K ← K + 1;
            PADROES[K] ← 0;
         senão
            J ← PADROES[J];
         fim se;
      fim se;
   fim enquanto;
fim procedimento FIND_PADROES;
```

Y	A	B	A	E	A	B	A	E	A	A	E	
	-											⇒ PADROES[1] = 0
	-	-										⇒ PADROES[2] = 0
	-	-	A									⇒ PADROES[3] = 1
	-	-	-	-								⇒ PADROES[4] = 0
	-	-	-	-	A							⇒ PADROES[5] = 1
	-	-	-	-	A	B						⇒ PADROES[6] = 2
	-	-	-	-	A	B	A					⇒ PADROES[7] = 3
	-	-	-	-	A	B	A	E				⇒ PADROES[8] = 4
	-	-	-	-	A	B	A	E	A			⇒ PADROES[9] = 5
	-	-	-	-	-	-	-	-	-	A		⇒ PADROES[10] = 1
	-	-	-	-	-	-	-	-	-	-	-	⇒ PADROES[11] = 0

Figura 5.10 – Os valores de PADROES[K]

A segunda etapa do algoritmo consiste em utilizar-se do vetor PADROES para efetuar a busca da subcadeia propriamente dita. Neste processo nem todas as subcadeias de ***X*** com comprimento igual à subcadeia ***Y*** necessitam ser comparadas com ***Y***. A Figura 5.11 mostra as comparações efetuadas pelo algoritmo de Knuth, Morris e Pratt na busca da subcadeia ***Y*** em ***X***.

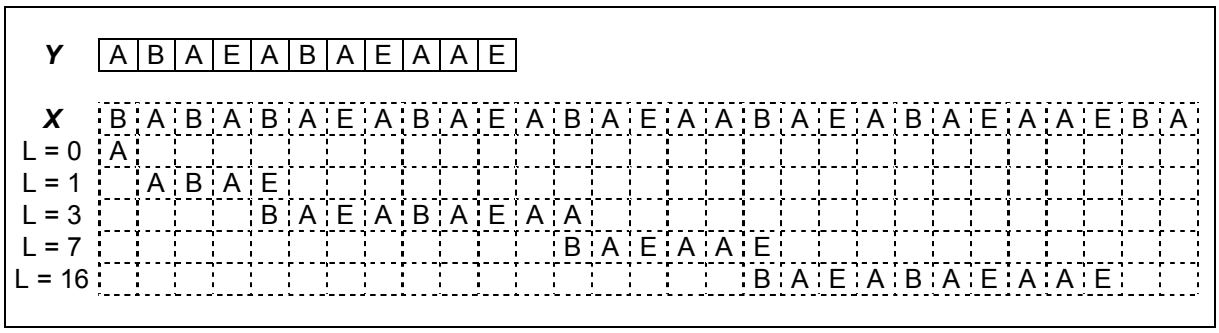


Figura 5.11 – Comparações efetuadas pelo algoritmo de Knuth, Morris e Pratt

O algoritmo utilizado para verificar o casamento entre as cadeias ***X*** e ***Y*** é apresentado a seguir.

```
função CASAMENTO_KMP (X, Y : caractere): inteiro;
{retorna a posição onde ocorre o casamento e 0 caso não ocorra
casamento de cadeias de caracteres}

var
   PADROES       : VET_PADROES;
   N, M, I, J    : inteiro;
   TESTE         : lógico;

   N ← COMPRIMENTO(X);
   M ← COMPRIMENTO(Y);
   I ← 0;
   J ← 0;
   FIND_PADROES(Y, PADROES);

   enquanto ((I - J) <= (N - M)) faça
      TESTE ← Verdadeiro;

      enquanto ((J < M) e (TESTE)) faça
         se (X[I + 1] = Y[J + 1]) então
            I ← I + 1;
            J ← J + 1;
         senão
            TESTE ← Falso;
         fim se;
      fim enquanto;
      se (TESTE) então
         CASAMENTO_KMP ← I - M + 1;
         exit;
      fim se;
```

```
        se (J = 0) então
           I ← I + 1;
        senão
           J ← PADROES[J];
        fim se;
    fim enquanto;
    CASAMENTO_KMP ← 0;
fim função CASAMENTO_KMP;
```

5.3 ESPALHAMENTOS

Esta seção apresenta uma eficiente técnica empregada para agilizar o processo de pesquisa de informações, denominada ***espalhamento*** ou ***"hashing"***. Veremos que uma das principais características desta técnica de armazenamento e recuperação de informações é que ela não requer ordenação, o que torna a inserção tão rápida quanto o acesso!

5.3.1 FUNDAMENTOS

Seja ***P*** o conjunto que contém todos os elementos de um determinado universo, possivelmente infinito. Chamamos ***espalhamento*** ou ***"hashing"*** ao particionamento de ***P*** em um número finito de classes $P_1, P_2, P_3, ..., P_n$, com $n > 1$, tal que:

1. $\bigcup_{i=1}^{n} P_i = P$

Isto é, a união de todas as ***n*** classes deve resultar no próprio conjunto ***P***; significando que para todo elemento $k \in P$ deve existir uma classe P_i, $1 \leq i \leq n$, tal que $k \in P_i$.

2. $\bigcap_{i=1}^{n} P_i = \varnothing$

Isto é, a intersecção de todas as ***n*** classes deve resultar em um conjunto vazio, significando que não existem $k \in P$ e $1 \leq i < j \leq n$, tal que $k \in P_i$ e $k \in P_j$. Em outras palavras, um mesmo elemento não pode pertencer a mais de uma classe ao mesmo tempo.

A correspondência unívoca entre os elementos do conjunto ***P*** e as ***n*** classes sugere a existência de uma função h, através da qual é feito o particionamento (Figura 5.12)

A função $h: P \rightarrow [1 .. n]$, que leva cada elemento de ***P*** à sua respectiva classe, é chamada função de espalhamento ou ***função hashing***. Sendo ***k*** um elemento qualquer do conjunto ***P***, o seu ***valor de hashing***, $h(k)$, determina a que classe ele

pertence. Pode-se garantir que:

$$k \in P_{\hbar(k)}$$

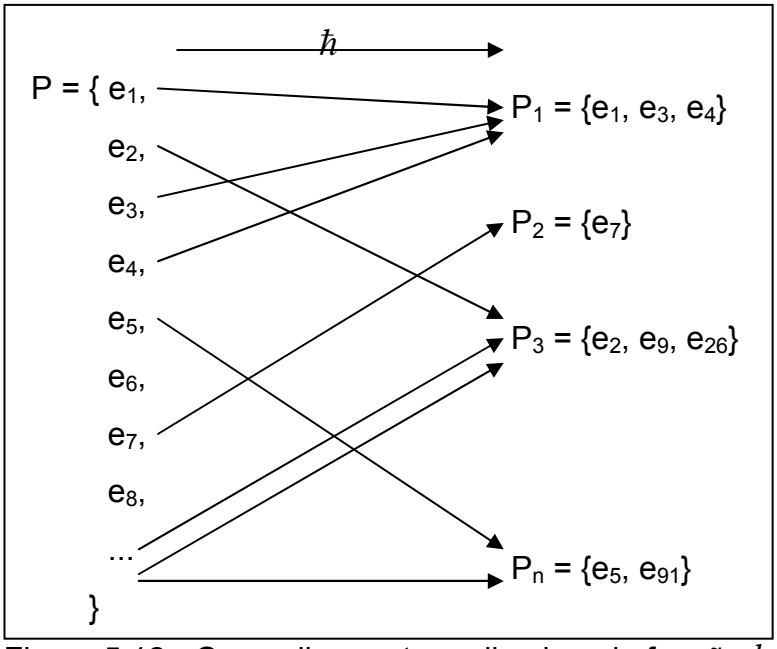


Figura 5.12 - O espalhamento realizado pela função $\hbar$

A cardinalidade de um conjunto ***C***, representada por **|*C*|**, equivale ao número de elementos existentes em ***C***. Se, num certo espalhamento, a diferença absoluta entre as cardinalidades de quaisquer duas classes for no máximo 1, dizemos que a função de espalhamento utilizada é ótima.

Função Ótima $\Leftrightarrow$ ABS(|P_i| - |P_j|) $\leq$ 1; para 1 $\leq$ i < j $\leq$ n

Por exemplo, sejam o conjunto ***P*** = {a, b, c, d, e, f, g, h} e a função ***$\hbar$: P $\rightarrow$ [1 .. 3]***, que " espalha" os elementos de ***P*** entre três classes distintas e que fornece os seguintes valores de hashing:

$\hbar$ (a) = 2; $\hbar$ (b) = 1; $\hbar$ (c) = 3; $\hbar$ (d) = 3; $\hbar$ (e) = 1; $\hbar$ (f) = 2; $\hbar$ (g) = 3; $\hbar$ (h) = 1;

De acordo com os valores de hashing obtidos para os elementos, temos as classes ***P_1*** = {b, e, h}, ***P_2*** = {a, f} e ***P_3*** = {c, d, g}, cujas cardinalidades são, respectivamente, |***P_1***| = 3, |***P_2***| = 2 e |***P_3***| = 3. Pode-se dizer que a função ***$\hbar$*** empregada neste espalhamento, é ótima pois todas as classes têm aproximadamente a mesma quantidade de elementos, ou seja, o espalhamento foi ***uniforme***.

Quando a função utilizada no espalhamento de um conjunto ***P*** é ótima, a menor classe criada tem no mínimo (|***P***| div ***n***) elementos e a maior, no máximo (|***P***| div ***n***) + 1. Considerando o nosso exemplo, onde |***P***| = 8 e ***n*** = 3, podemos constatar que a menor cardinalidade obtida foi |***P_2***| = (8 div 3) = 2, enquanto a maior foi |***P_1***| = |***P_3***| = (8 div 3) + 1 = 3.

Suponha agora um espalhamento no qual uma função ***P*** ótima é utilizada para particionar um conjunto ***P*** em |***P***| classes distintas. Naturalmente, existindo |***P***|

classes, teremos tantas classes quantos forem os elementos de ***P***. Aliando isto ao fato de que o espalhamento foi realizado por uma função ótima (o espalhamento é uniforme), é evidente que cada classe terá um único elemento de ***P***. Dizemos que o espalhamento assim obtido é ***perfeito***.

Se um espalhamento não é perfeito, existe pelo menos uma classe com mais de um elemento, isto é, existem $x \in P$ e $y \in P$, tal que $x \neq y$ mas $h(x) = h(y)$; neste caso, dizemos que ***x*** e ***y*** são ***sinônimos***.

5.3.2 APLICABILIDADE DO ESPALHAMENTO

Em termos práticos, a grande vantagem do espalhamento está no fato de que, dado um elemento ***k*** de um conjunto ***P***, o valor de hashing $h(k)$ pode ser calculado em tempo constante, fornecendo imediatamente a classe da partição de ***P*** em que o elemento se encontra. Se considerarmos ***P*** uma coleção de elementos a ser pesquisada, é fácil perceber que o processo será muito mais eficiente se a pesquisa for restrita a uma pequena parte do conjunto ***P*** (uma única classe). Suponha o caso de procurar o nome "Maria" em um conjunto de nomes próprios:

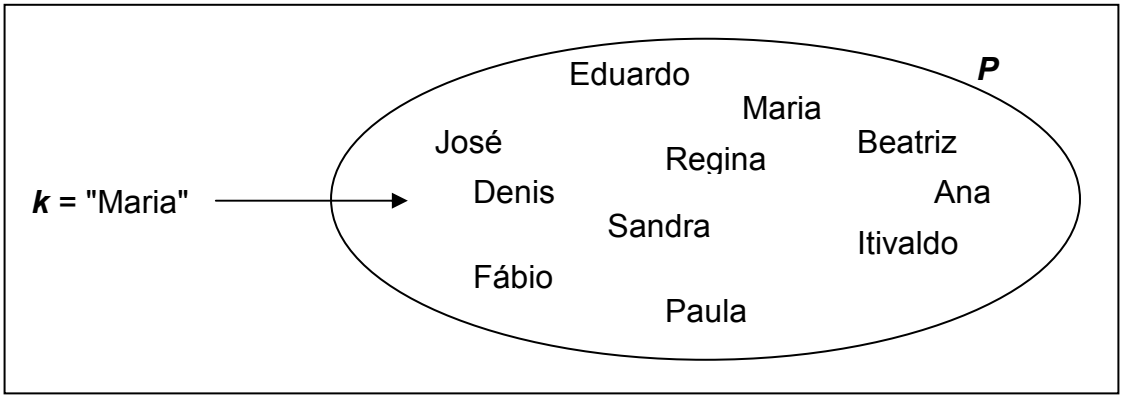


Figura 5.13 - Pesquisa sem espalhamento

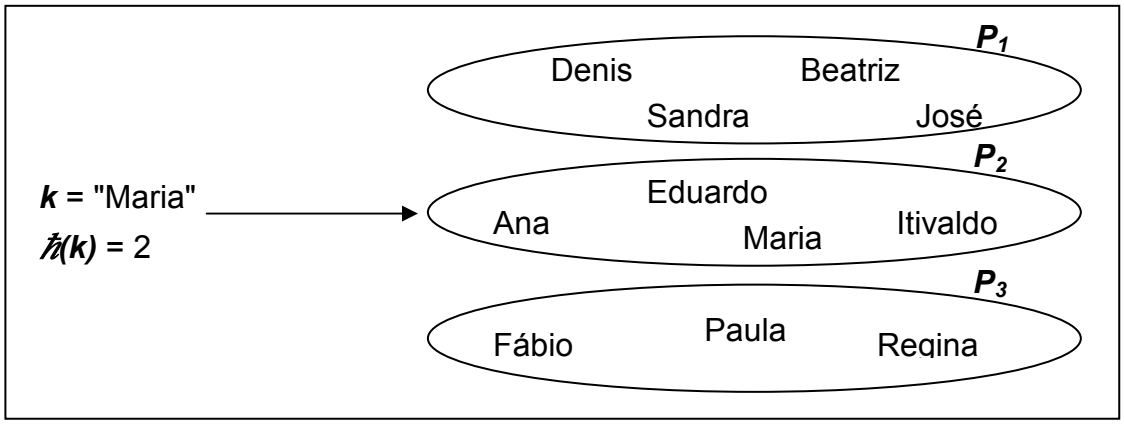


Figura 5.14 - Pesquisa com espalhamento

O espalhamento pode ser usado como uma técnica de redução do espaço de busca; o processo de pesquisa será tão mais eficiente quanto menores forem as partições. Se o espalhamento for perfeito, então o valor de hashing calculado dará

imediatamente a localização do elemento desejado; neste caso, temos um ***acesso direto*** ou randômico, como também é chamado, sendo este o tipo de busca mais eficiente de que dispomos.

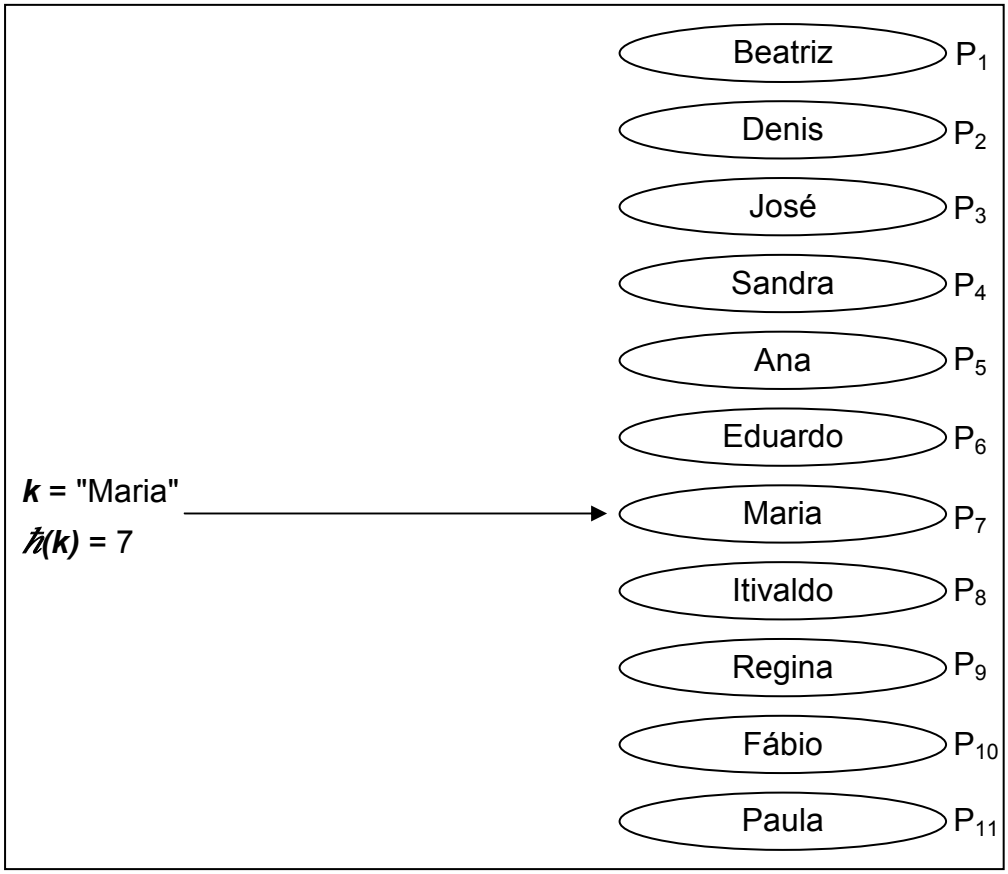


Figura 5.15 - Acesso direto ou randômico através de hashing

5.3.3 TABELAS DE ESPALHAMENTO

Tabela de espalhamento é a estrutura de dados que implementa o espalhamento para aplicações em computador. Ela pode ser representada por um vetor onde cada posição, denominada ***encaixe***, mantém uma classe de partição. O número de encaixes na tabela deve coincidir com o número de classes criadas pela função de espalhamento. Considerando que um conjunto ***P*** seja espalhado em $P_1, P_2, P_3, ..., P_n$ classes distintas, um vetor ***T[1 .. n]*** pode representar o espalhamento de maneira satisfatória, bastando associar a cada elemento ***T[i]*** uma classe P_i, $1 \leq i \leq n$ correspondente.

Admitindo a existência de elementos sinônimos em ***P***, é de se esperar que durante o espalhamento ocorra pelo menos uma ***colisão***, ou seja, é possível que tenhamos que armazenar um elemento numa posição da tabela que já se encontre ocupada por outro valor. Existem diversas formas de se resolver este problema, entretanto, a mais comumente utilizada é chamada tratamento de colisão por ***espalhamento externo***.

O espalhamento externo parte do princípio que existirão muitas colisões durante o carregamento das chaves numa tabela e que, na verdade, cada encaixe ***T[i]*** armazenará não um único elemento, mas uma coleção de elementos sinônimos. Como a quantidade de elementos sinônimos em cada classe pode variar bastante, a lista

encadeada será bastante útil. No espalhamento aberto, a tabela de hashing será um vetor cujos elementos são ponteiros para listas encadeadas que representam as classes de espalhamento.

```
const N = 5;
tipo ELEM = caractere;
tipo CLAS = ^NODO;
     NODO = registro
               CHAVE : ELEM;
               PROX : CLAS;
     fim registro;

tipo TABHSH = vetor [1..N] de CLAS;

var
   T : TABHSH;
```

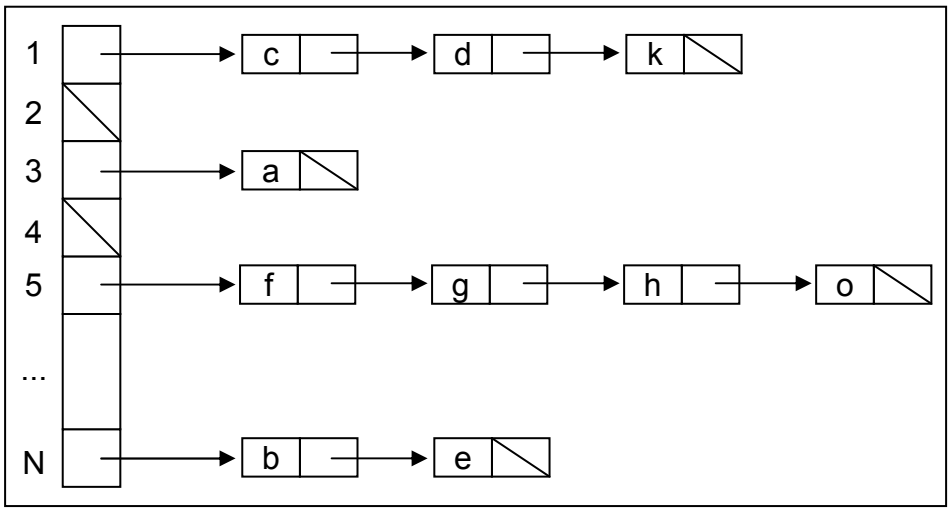


Figura 5.16 - Uma tabela de espalhamento ou hashing

O espalhamento é denominado externo porque as chaves são armazenadas "fora" da tabela de hashing, numa área de memória alocada dinamicamente.

5.3.4 FUNÇÕES DE ESPALHAMENTO

Espalhamento pode ser visto como um método de busca que permite acessar dados diretamente, através de uma função que transforma uma chave ***k*** em um endereço físico, relativo ou absoluto, ***h(k)***.

Uma função de espalhamento ideal seria aquela capaz de mapear ***n*** chaves em exatamente ***n*** endereços, sem a ocorrência de colisões. Existem ***n!*** formas de se obter este mapeamento ideal, o que poderia nos levar a crer que tais funções ideais são facilmente encontradas. Considerando-se, entretanto, que existem n^n formas possíveis de atribuir ***n*** chaves a ***n*** endereços, a probabilidade de se obter um espalhamento perfeito é mínima ($n! / n^n$). Na prática, devemos nos contentar com funções capazes de fazer um mapeamento razoável, distribuindo-se as ***n*** chaves de forma mais ou menos uniforme entre ***m*** endereços, tal que ***m < n*** (sempre teremos

colisões).

Muito esforço tem sido dedicado em busca de funções eficientes, que transformem rapidamente uma chave num endereço e que possibilitem uma distribuição uniforme das chaves dentro de um determinado espaço de armazenamento (encaixes da tabela de hashing), sendo que nenhum dos métodos propostos garantem a eficiência máxima desejada. Felizmente, porém, um dos mais simples métodos existentes, o método da divisão, tem mostrado na prática um desempenho bastante satisfatório, sendo recomendado como o melhor método para uso geral.

5.3.5 O MÉTODO DA DIVISÃO

O método da divisão consiste basicamente em realizar uma divisão inteira e tomar o seu resto. Para entendermos melhor como ele funciona, vamos utilizá-lo para espalhar as chaves 54, 21, 15, 46, 7, 33, 78, 9, 14, 62, 95 e 87 numa tabela contendo ***N = 5*** encaixes. A função definida a seguir, que transforma as chaves em endereços relativos, está baseada no método da divisão:

```
função DH (CHV: inteiro) : inteiro;

   DH ← (CHV mod N) + 1;

fim função DH;
```

É evidente que numa divisão inteira por ***N*** não se pode obter resto maior nem igual a ***N***; logo, o operador ***mod*** resultará sempre em valores no intervalo de ***0*** a ***N - 1***. Como a tabela oferece ***N*** encaixes numerados de ***1*** a ***N***, devemos adicionar uma unidade ao resto calculado, de modo a obtermos valores na faixa de ***1*** a ***N***.

Veja a seguir, os valores de hashing gerados pela função:

- DH(54) = (54 mod 5) + 1 = 5;
- DH(15) = (15 mod 5) + 1 = 1;
- DH(7) = (7 mod 5) + 1 = 3;
- DH(78) = (78 mod 5) + 1 = 4;
- DH(14) = (14 mod 5) + 1 = 5;
- DH(95) = (95 mod 5) + 1 = 1;
- DH(21) = (21 mod 5) + 1 = 2;
- DH(46) = (46 mod 5) + 1 = 2;
- DH(33) = (33 mod 5) + 1 = 4;
- DH(9) = (9 mod 5) + 1 = 5;
- DH(62) = (62 mod 5) + 1 = 3;
- DH(87) = (87 mod 5) + 1 = 3;

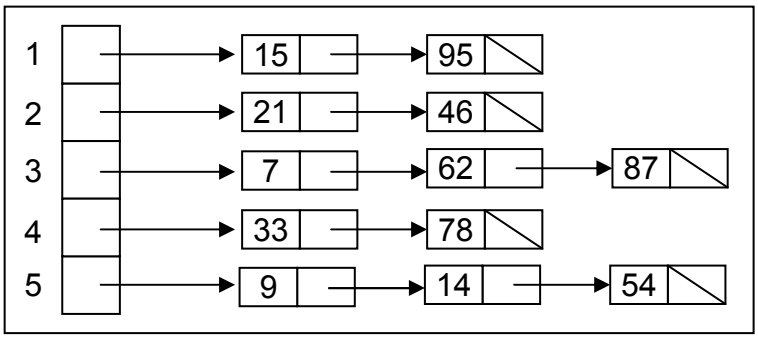


Figura 5.17 - Espalhamento de chaves numéricas

5.3.6 TRANSFORMAÇÃO DE CHAVES ALFANUMÉRICAS

O método da divisão não pode ser usado diretamente quando as chaves são alfanuméricas; neste caso, temos antes que transformá-las em valores numéricos que poderão ser divididos por ***N***.

Uma forma simples de se transformar uma chave alfanumérica num valor numérico consiste em considerar cada caractere da chave como um valor inteiro (correspondente ao seu código ASCII) e realizar uma soma com todos eles:

```
função ADH (CHV: caractere): inteiro;
var
   I, SOMA : inteiro;

   SOMA ← 0;

   para I ← 1 até COMPRIMENTO(CHV) faça
      SOMA ← SOMA + ORD(CHV[I]); {ORD retorna o código ASCII}
   fim para;                      {do parâmetro}

   ADH ← (SOMA mod N) + 1;

fim função ADH;
```

Vamos supor agora que pretendemos espalhar em uma tabela com ***N = 7*** encaixes as chaves *Thais, Edu, Bia, Neusa, Lucy, Rose, Yara, Decio* e *Sueli*. Calculando a função ***ADH( )*** para cada uma das chaves, obtemos os valores relacionados a seguir e a tabela ilustrada na Figura 5.18. Observe que se não fossem as chaves *Decio* e *Sueli*, o espalhamento seria perfeito.

- ADH('Thais') = 2;
- ADH('Bia') = 3;
- ADH('Lucy') = 1;
- ADH('Yara') = 6;
- ADH('Sueli') = 4.
- ADH('Edu') = 7;
- ADH('Neusa') = 5;
- ADH('Rose') = 4;
- ADH('Decio') = 2;

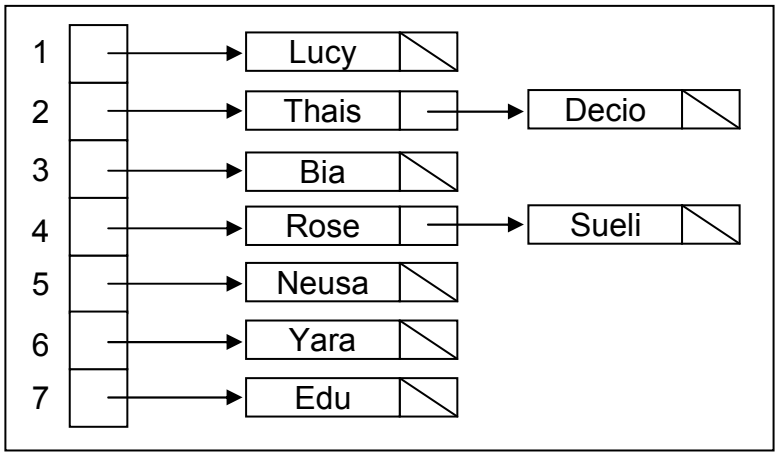


Figura 5.18 – Espalhamento de chaves alfanuméricas

Considere agora um conjunto de chaves alfanuméricas onde cada uma delas é apenas uma permutação dos mesmos caracteres básicos; por exemplo: *ABC,*

ACB, BAC, BCA, CAB e *CBA*. Neste caso, se estas chaves forem espalhadas pela função ***ADH( )***, numa tabela com ***N = 7*** encaixes, teremos:

- ADH('ABC') = 3;
- ADH('ACB') = 3;
- ADH('BAC') = 3;
- ADH('BCA') = 3;
- ADH('CAB') = 3;
- ADH('CBA') = 3.

Claramente, se as chaves são todas compostas pelos mesmos caracteres, e a transformação da chave é baseada na soma dos seus "caracteres", então todas as chaves serão mapeadas para a mesma classe (não há espalhamento). Não poderíamos ter uma situação mais indesejável! Isto acontece porque a transformação por simples somatório não leva em consideração a posição em que os caracteres aparecem na chave.

Para resolver este problema, vamos usar um algoritmo bastante eficiente para a transformação de chaves alfanuméricas denominado ***somatório com deslocamentos***. Este algoritmo associa a cada caractere da chave uma quantidade de bits que deverá ser deslocada à esquerda no seu código ASCII, antes de ele ser adicionado à soma total. As quantidades a serem deslocadas à esquerda variam de 0 a 7, de forma cíclica, conforme esquematizado na Figura 5.19

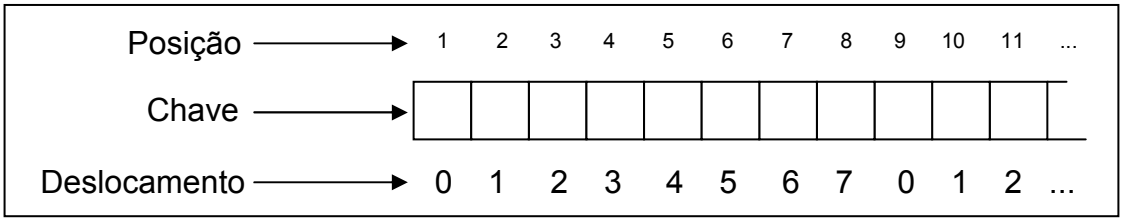


Figura 5.19 – Quantidade de bits a ser deslocada para cada posição da chave

No somatório simples, qualquer que seja a posição ocupada por um caractere ***X*** dentro da chave, ele é sempre transformado no número ***ORD(X)***. No somatório com deslocamento, entretanto, o valor numérico associado a ***X*** dependerá não somente do seu código, mas também da posição que ele ocupa dentro da chave. Seu valor será:

ORD(X) deslocado à esquerda de ***[(posição – 1) mod 8]*** bits

Para entender melhor como este algoritmo funciona, vamos tomar como exemplo, o código ASCII da letra ***'A'*** (65) na sua representação binária:

$$ORD('A') = (0\ 1\ 0\ 0\ 0\ 0\ 0\ 1)_B$$

Se a letra ***'A'*** aparecer na primeira posição da chave, então o seu valor numérico será o seu próprio código ASCII, pois, para a primeira posição, o deslocamento é 0 e nada será alterado. Porém, se ela aparecer na segunda posição, seu código deverá ser deslocado de 1 bit à esquerda; se aparecer na terceira, 2 bits; na quarta, 3 e assim sucessivamente conforme especificado na Figura 5.19.

- 1ª posição: $(0\ 1\ 0\ 0\ 0\ 0\ 0\ 1)_B$ shl 0 = $(0\ 1\ 0\ 0\ 0\ 0\ 0\ 1)_B$ = 65;
- 2ª posição: $(0\ 1\ 0\ 0\ 0\ 0\ 0\ 1)_B$ shl 1 = $(1\ 0\ 0\ 0\ 0\ 0\ 1\ 0)_B$ = -126;
- 3ª posição: $(0\ 1\ 0\ 0\ 0\ 0\ 0\ 1)_B$ shl 2 = $(0\ 0\ 0\ 0\ 0\ 1\ 0\ 0)_B$ = 4;

- 4ª posição: $(0\ 1\ 0\ 0\ 0\ 0\ 0\ 1)_B$ shl 3 = $(0\ 0\ 0\ 0\ 1\ 0\ 0\ 0)_B$ = 8;
- 5ª posição: $(0\ 1\ 0\ 0\ 0\ 0\ 0\ 1)_B$ shl 4 = $(0\ 0\ 0\ 1\ 0\ 0\ 0\ 0)_B$ = 16;
- 6ª posição: $(0\ 1\ 0\ 0\ 0\ 0\ 0\ 1)_B$ shl 5 = $(0\ 0\ 1\ 0\ 0\ 0\ 0\ 0)_B$ = 32;
- 7ª posição: $(0\ 1\ 0\ 0\ 0\ 0\ 0\ 1)_B$ shl 6 = $(0\ 1\ 0\ 0\ 0\ 0\ 0\ 0)_B$ = 64;
- 8ª posição: $(0\ 1\ 0\ 0\ 0\ 0\ 0\ 1)_B$ shl 7 = $(1\ 0\ 0\ 0\ 0\ 0\ 0\ 0)_B$ = -128;
- 9ª posição: $(0\ 1\ 0\ 0\ 0\ 0\ 0\ 1)_B$ shl 0 = $(0\ 1\ 0\ 0\ 0\ 0\ 0\ 1)_B$ = 65;

Em Pascal, utilizam-se os operadores ***shl*** (shift to left) e ***shr*** (shift to right) para se deslocar bits de um valor inteiro, respectivamente, à esquerda e à direita. Observe que o 8º bit (da direita para a esquerda) é usado como bit de sinal, sendo que se este bit está ligado, o byte representa um valor negativo em complemento de 2.

A função de hashing para chaves alfanuméricas com permutação usa o somatório com deslocamentos e o método da divisão, em conjunto:

```
função SDH (CHV: caractere): inteiro;
var
   P, SOMA : inteiro;

   SOMA ← 0;

   para P ← 1 até COMPRIMENTO(CHV) faça
      SOMA ← SOMA + ORD(CHV[P]) shl ((P - 1) mod 8);
   fim para;

   SDH ← (ABS(SOMA) mod N) + 1;

fim função SDH;
```

Agora sim as chaves com permutação podem ser espalhadas numa tabela com 7 encaixes:

- SDH('ABC') = 6;
- SDH('ACB') = 2;
- SDH('BAC') = 7;
- SDH('BCA') = 6;
- SDH('CAB') = 7;
- SDH('CBA') = 5.

Embora o somatório com deslocamentos apareça como uma forma de resolver o problema de chaves com permutação, podemos empregá-lo sempre que tivermos que manipular chaves alfanuméricas; ainda que elas não sejam permutações do mesmo conjunto de caracteres. Compare os resultados obtidos no espalhamento (***N = 7***) a seguir:

Chave	ADH(chave)	SDH(chave)
Thais	2	1
Denise	6	1
Carlos	4	3
Fabio	6	4
Paula	3	2
Silvio	1	7
Roberto	6	5
Luis	1	6
Denis	3	3
Yara	6	4

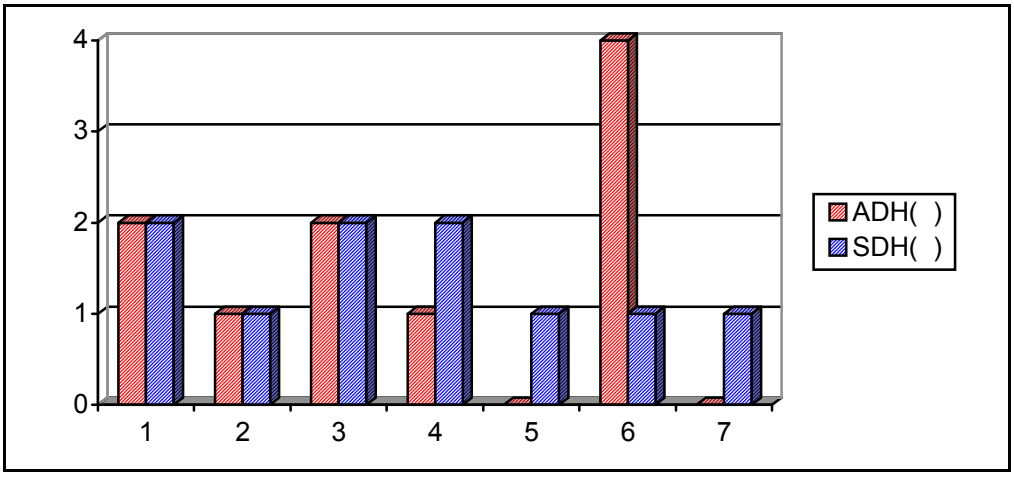


Figura 5.20 – Comparação entre os algoritmos ***ADH( )*** e ***SDH( )***

5.3.7 OPERAÇÕES BÁSICAS SOBRE TABELA DE ESPALHAMENTO

Dada uma tabela de espalhamento ***T***, a operação ***HINIT( )*** inicia a tabela no estado "vazio". Como o espalhamento é externo, isto é, a tabela armazena apenas ponteiros para listas encadeadas que contêm chaves sinônimas, a operação de inicialização simplesmente anula todos os ***N*** ponteiros (encaixes) existentes na tabela:

```
procedimento HINIT (var T: TABHSH);
var
   I : inteiro;

   para I ← 1 até N faça
      T[I] ← nil;
   fim para;

fim procedimento HINIT;
```

A operação ***HINS(T, K)*** insere a chave ***K*** no $\hbar$(***K***)-ésimo encaixe da tabela ***T***. Admitindo-se que a operação ***INS( )***, que realiza a inserção em lista ordenada, se encontra disponível, a operação de inserção numa tabela de espalhamento pode ser codificada como a seguir:

```
procedimento HINS (var T: TABHSH; K: ELEM);

   INS(T[H(K)], K);

fim procedimento HINS;
```

Note que, como os elementos do vetor T são ponteiros, a expressão ***T[H(K)]*** resulta num ponteiro para a lista ordenada onde a chave ***K*** deve ser inserida. A inserção propriamente dita é realizada pela rotina ***INS( )***.

Analogamente, as operações ***HREM(T, K)*** e ***HFND(T, K)***, respectivamente empregadas para remoção e pesquisa, podem ser codificadas com base nas operações de remoção e busca em listas ordenadas. Lembre-se de que, sendo ***L*** uma

lista ordenada, ***REM(L, K)*** exclui a chave ***K*** da lista ***L*** e ***FIND(L, K)*** retorna um ponteiro para o nodo que contém a chave ***K***, ou ***nil*** se a chave não for encontrada em ***L***.

```
procedimento HREM (var T: TABHSH; K: ELEM);

   REM(T[h(K)], K);

fim procedimento HREM;


função HFND (var T: TABHSH; K: ELEM): lógico;

   HFND ← (FND(T[h(K)], K) <> nil);

fim procedimento HFND;
```

Vale lembrar que qualquer estrutura de dados capaz de armazenar coleções de dados pode ser usada para resolver as colisões que ocorrem durante o processo de carga da tabela de hashing, assim, árvores binárias poderiam ter sido escolhidas ao invés das listas ordenadas.