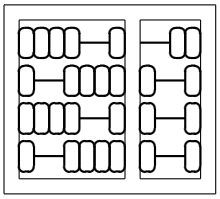


Estruturas de Dados
e
Técnicas de Programação

Versão 1.12

Agosto de 2004

Cláudio L. Lucchesi
Tomasz Kowaltowski

INSTITUTO DE COMPUTAÇÃO
UNIVERSIDADE ESTADUAL DE CAMPINAS

Instituto de Computação
UNICAMP
Caixa Postal 6176
13083-970 Campinas, SP
`http://www.ic.unicamp.br/~{lucchesi,tomasz}`

Prefácio

Este texto é o resultado da tentativa de reunir o material preparado pelos autores durante os vários anos em que têm ministrado as disciplinas sobre estruturas de dados, na UNICAMP e em várias outras instituições. Os capítulos e seções que compõem o texto estão em estágios distintos de elaboração. Alguns têm apenas seus títulos enunciados, outros estão bastante completos. Os autores esperam que, com o tempo, o texto possa evoluir para um livro completo.

A preocupação principal do texto é com as estruturas de dados fundamentais e os algoritmos para a sua manipulação, incluindo noções básicas de eficiência. Os exemplos procuram utilizar o conceito de tipos abstratos de dados, sempre que aplicável. Entretanto, a metodologia de programação não é um assunto aprofundado neste texto, já que trata-se de programação em *pequena escala* (*programming-in-the-small*). Apenas a título de completude está previsto um capítulo final sobre programação orientada a objetos.

Os autores decidiram utilizar nos exemplos a linguagem PASCAL, apesar da disseminação atual da linguagem C e dos seus descendentes. A principal razão é didática. Infelizmente, vários conceitos de programação que são razoavelmente simples e claros em PASCAL, são mais obscuros ou inexistentes em C. Entre eles, podemos citar o conceito geral de tipos e os mecanismos de passagem de parâmetros. Não temos dúvida que qualquer aluno de Computação que domine estes conceitos numa linguagem, possa aprender facilmente qualquer outra linguagem de programação.

Durante os vários anos em que foi acumulado este material, muitos colegas e alunos contribuíram com observações e sugestões, e não seria possível agradecer explicitamente a todos eles. Os autores gostariam de destacar, entretanto, as contribuições do saudoso *Claudinho* – Claudio Sergio Da Rós de Carvalho – cuja morte prematura interrompeu o projeto de coautoria, numa primeira tentativa de organizar um texto como este.

Cláudio L. Lucchesi e Tomasz Kowaltowski

Campinas, SP, Março de 1997

Sumário

Prefácio **iii**

1 Introdução **1**
1.1 Análise de algoritmos 1
1.2 Tipos primitivos de dados 3
1.3 Execução de programas 4
1.4 Tipos abstratos de dados 6
1.5 Exercícios 6

2 Estruturas Seqüenciais **9**
2.1 Representação linearizada 9
2.2 Formas especiais 10
2.3 Exercícios 11

3 Estruturas ligadas **13**
3.1 Listas ligadas simples 13
3.2 Listas circulares 15
3.3 Listas duplamente ligadas 16
3.4 Exemplo 1: manipulação de polinômios 17
3.5 Exemplo 2: matrizes esparsas 19
3.6 Exercícios 22

4 Estruturas Lineares **25**
4.1 Estruturas lineares 25
4.2 Pilhas 26
4.3 Exemplo de aplicação: manipulação de expressões 31
4.4 Filas 34
4.5 Exercícios 35

5 Recursão **39**
5.1 Recursão e repetição 39
5.2 Eliminação da recursão 42
5.3 Exemplo: análise sintática 45
5.4 Exercícios 48

6 Árvores binárias **51**
6.1 Definições, terminologia e propriedades . . . 54
6.2 Implementação de árvores binárias . . . 56
6.3 Percurso de árvores binárias . . . 59
6.4 Exemplo de aplicação: árvores de busca . . . 68
6.5 Exercicios . . . 71

7 Árvores gerais **75**
7.1 Definições e terminologia . . . 75
7.2 Representação de árvores . . . 76
7.3 Percursos de florestas . . . 78
7.4 Exercícios . . . 79

8 Árvores de busca **81**
8.1 Árvores de altura balanceada . . . 81
8.2 Árvores de tipo *B* . . . 88
8.3 Árvores digitais . . . 98
8.4 Filas de prioridade . . . 100
8.5 Exercícios . . . 105

9 Listas Gerais **107**
9.1 Conceitos e implementação . . . 107
9.2 Exemplo: polinômios em múltiplas variáveis . . . 112
9.3 Exercicios . . . 113

10 Espalhamento **115**
10.1 Conceitos básicos . . . 115
10.2 Funções de espalhamento . . . 116
10.3 Endereçamento aberto . . . 117
10.4 Encadeamento . . . 119
10.5 Exercícios . . . 120

11 Gerenciamento de Memória **121**
11.1 Gerenciamento elementar . . . 121
11.2 Contagem de referências . . . 122
11.3 Coleta de lixo . . . 124
11.4 Alocação de blocos de memória variáveis . . . 127
11.5 Alocação controlada . . . 128
11.6 Exercícios . . . 130

12 Processamento de Cadeias e Textos **133**
12.1 Representação de cadeias . . . 133
12.2 Noções de compactação de cadeias . . . 133
12.3 Busca de padrões . . . 133
12.4 Noções de criptografia . . . 133
12.5 Exercícios . . . 133

13 Algoritmos de Ordenação **135**
13.1 Ordenação por transposição 135
13.2 Ordenação por inserção 136
13.3 Ordenação por seleção 136
13.4 Ordenação por intercalação 137
13.5 Ordenação por distribuição 137
13.6 Exercícios 138

14 Programação Orientada a Objetos **143**
14.1 Conceitos básicos 143
14.2 Implementação 143
14.3 Exercícios 143

Lista de Figuras

1.1 Registros de ativação para o Prog. 1.2 4
1.2 Registros de ativação e *heap* para o Prog. 1.3 5
1.3 Registros de ativação para o Prog. 1.4 6

3.1 Representação de uma matriz esparsa 21

5.1 O problema de torres de Hanoi 41

6.1 Árvore genealógica 52
6.2 Árvore de descendentes 53
6.3 Exemplo de árvore binária 54
6.4 Árvores binárias distintas 55
6.5 Representação de árvores binárias com campos *esq* e *dir* 56
6.6 Árvore binária com o campo *pai*. 57
6.7 Árvore binária completa de altura 4 58
6.8 Árvore binária quase completa de altura 4 58
6.9 Árvore binária de busca contendo números 68
6.10 Árvore binária de busca contendo nomes dos meses 69

7.1 Exemplo de árvore 76
7.2 Uma floresta com três árvores 77
7.3 Representação binária de uma floresta com três árvores 77

8.1 Exemplos de árvores AVL 81
8.2 Árvores de Fibonacci 82
8.3 Funcionamento esquemático de discos magnéticos 89
8.4 Exemplo de árvore B de ordem 3 90
8.5 Exemplo de árvore digital 99
8.6 Representação modificada de árvore digital 99
8.7 Exemplo de autômato 100
8.8 Exemplo de fila de prioridade 100
8.9 Operação de subida de um elemento no *heap* 101
8.10 Operação de descida de um elemento no *heap* 102

9.1 Implementação compartilhada de listas 108
9.2 Implementação de listas com cópia 109

9.3 Representação dos polinômios em múltiplas variáveis 113

10.1 Resultados das técnicas de endereçamento aberto 118

11.1 Blocos em uso e disponível (marcas de fronteira) 128
11.2 Árvore para o sistema de blocos conjugados com $m = 4$ 129
11.3 Blocos em uso e disponível (sistema conjugado) 130
11.4 Árvore para blocos conjugados de Fibonacci com $F_7 = 13$ 131

Lista de Programas

1.1 Procedimento *Ordena* . . . 3
1.2 Exemplo de programa . . . 4
1.3 Exemplo de programa com alocação de memória dinâmica . . . 5
1.4 Exemplo de programa recursivo . . . 6
3.1 Rotinas para listas ligadas simples . . . 14
3.2 Rotinas para listas ligadas circulares . . . 15
3.3 Busca com sentinela numa lista circular . . . 16
3.4 Rotinas para listas circulares duplamente ligadas . . . 18
3.5 Função de soma de polinômios . . . 20
3.6 Recuperação de valor e atribuição em matriz esparsa . . . 24
4.1 Implementação ligada de pilhas . . . 28
4.2 Implementação seqüencial de pilhas . . . 29
4.3 Solução do problema de balanceamento de parênteses . . . 30
4.4 Transformação da notação infixa para pós-fixa . . . 33
4.5 Implementação ligada de filas . . . 37
4.6 Implementação seqüencial de filas . . . 38
5.1 Cálculo da função fatorial . . . 40
5.2 Cálculo da série de Fibonacci . . . 40
5.3 Solução do problema de torres de Hanoi . . . 41
5.4 Exemplo de recursão mútua . . . 42
5.5 Esboço de um procedimento recursivo . . . 43
5.6 Esboço da transformação do procedimento recursivo do Prog. 5.5 . . . 44
5.7 Transformação infixa para pós-fixa (versão incompleta) . . . 46
5.8 Procedimeto *Fator* . . . 47
6.1 Procedimento de inserção à esquerda . . . 57
6.2 Procedimento de percurso recursivo em pré-ordem . . . 60
6.3 Procedimento de percurso em largura utilizando uma fila . . . 60
6.4 Pré-ordem com pilha explícita obtida por transformação . . . 62
6.5 Pré-ordem com pilha explícita análoga ao percurso em largura . . . 63
6.6 Pré-ordem mais eficiente com pilha explícita . . . 64
6.7 Pós-ordem com pilha explícita . . . 65
6.8 Algoritmo de Deutsch, Schorr e Waite . . . 67
6.9 Busca e inserção numa árvore binária de busca . . . 70
6.10 Pré-ordem com pilha explícita . . . 72
6.11 Algoritmo de Lindstrom e Dwyer . . . 73

8.1 Esboço da rotina de busca e inserção em árvore AVL 86
8.2 Rotina de busca e inserção em árvoreB 91
8.3 Esboço da rotina auxiliar de busca e inserção em árvoreB 92
8.4 Operações básicas para *heap* 103
8.5 Rotinas de manipulação de *heaps* 104
9.1 Contagem de átomos de uma lista 110
9.2 Contagem de átomos de uma lista – versão mais geral 111
11.1 Gerenciamento elementar de memória 122
11.2 Rotina de desalocação para contagem de referências 123
11.3 Algoritmo de marcação de uma estrutura 125
11.4 Coleta de nós marcados 125
11.5 Rotinas de compactação 126
13.1 Algoritmo *bubblesort* 136
13.2 Algoritmo *quicksort* 137
13.3 Ordenação por inserção simples 138
13.4 Ordenação por seleção simples 139
13.5 Algoritmo *heapsort* 140
13.6 Ordenação iterativa por intercalação 141
13.7 Ordenação recursiva por intercalação 142
14.1 Exemplo de utilização de objetos *(continua)* 144
14.2 Exemplo de utilização de objetos *(continuação)* 145

Capítulo 1

Introdução

O objetivo deste Capítulo é introduzir ou rever alguns conceitos que serão fundamentais para o resto do texto. Estão, entre eles, noções elementares de eficiência de algoritmos, implementação de tipos primitivos na memória do computador, dinâmica de execução de programas e tipos abstratos de dados.

1.1 Análise de algoritmos

Um dos aspectos mais importantes do desenvolvimento de algoritmos é a escolha correta da estrutura de dados a ser utilizada para representar os objetos a serem manipulados. Tomemos o seguinte exemplo de problema: achar o k-ésimo elemento de uma seqüência. Conforme a escolha da estrutura, há duas maneiras óbvias de resolver o problema:

```
...
x := a[k]
...
```

```
...
p := a;  i := 1;
while i≠k do
    begin
    p := p↑.prox;
    i := i+1
    end;
x := p↑.info
...
```

O primeiro trecho o faz em tempo constante; no segundo, o número de operações é proporcional ao valor de k.

Este tipo de *análise de eficiência* de programas pode ser realizado de várias maneiras. Consideremos os exemplos:

```
...
x := a+b
...
```

(a)

```
...
for i:=1 to n do
        x := a+b
...
```

(b)

```
...
for i:=1 to n do
      for j:=1 to n
            x := a+b
...
```

(c)

Realizaremos dois tipos de análise para os três exemplos. Na primeira, contaremos apenas o número de atribuições explícitas executadas; na segunda, mais detalhada, contaremos as atribuições, as operações aritméticas e as comparações. Adotaremos uma hipótese simplificadora, supondo que cada uma destas operações consome uma unidade de tempo. Supondo as implementações padrão de comandos em PASCAL e $n \geq 0$, obteríamos então os seguintes resultados:

	a	*b*	*c*
análise simples	1	n	n^2
análise detalhada	2	$5n+2$	$5n^2+5n+2$

(Lembre que o comando **for** inclui o comando de incremento de *i* equivalente ao comando $i := i+1$ incluído nesta contagem.)

Para valores crescentes de *n*, predominam os termos de ordem mais alta, assim que podemos adotar os resultados $5n$ e $5n^2$ para as duas últimas entradas da tabela. Uma vez que não foram fixados os valores da unidade de tempo, podemos concluir que os resultados da tabela são, para valores suficientemente grandes de *n*, aproximadamente proporcionais aos tempos de execução. Neste caso, os resultados 1 e 2, n e $5n$, bem como n^2 e $5n^2+5n+2$ são na realidade iguais, pois diferem apenas na constante de proporcionalidade. Este fato é indicado através de uma notação conveniente de acordo com a seguinte definição:

> *Dizemos que* $g(n) = \mathrm{O}(f(n))$ *se existirem constantes* c_0 *e* n_0 *tais que* $g(n) < c_0 f(n)$ *para todo* $n > n_0$.

Esta definição traduz a idéia de que a função *g*, a partir de uma constante n_0, é menor do que *f* multiplicada por uma constante de proporcionalidade c_0. Por exemplo:

$$\begin{aligned} c &= \mathrm{O}(1) && \text{para qualquer constante } c \\ 2 &= \mathrm{O}(1) \\ 5n+2 &= \mathrm{O}(n) \\ 5n^2+5n+2 &= \mathrm{O}(n^2) \\ n^k &= \mathrm{O}(n^{k+1}), && k \geq 0 \end{aligned}$$

(A última igualdade ilustra o fato de que, pela definição, a função *g* pode ter crescimento muito mais lento do que *f*.)

Devido a estas considerações, as nossas análises tenderão, em geral, a ser apenas aproximadas, indicando o comportamento em termos da função O. Deve-se tomar cuidado, entretanto, quanto à escolha correta das operações a serem contadas.

A título de exemplo faremos uma análise de um procedimento mais completo indicado no Prog. 1.1. O procedimento *Ordena* é um exemplo de algoritmo de ordenação por seleção; este assunto será coberto de maneira mais profunda no Cap. 13.

Numa análise mais superficial, contaremos apenas os comandos de atribuição explícitos. Não é difícil perceber que a repetição controlada pela variável *k* é executada $n-i$ vezes, para cada valor de *i*. Dentro desta

Programa 1.1 Procedimento *Ordena*

```
type vetor = array[1..nMax];

procedure Ordena(var v: vetor; n: integer);
    var i,k,t: integer;
begin
for i:=1 to n-1 do
    begin
    j := i;
    for k:=j+1 to n do
        if v[k]<v[j] then j := k;
    t := v[i]; v[i] := v[j]; v[j] := t
    end
end;
```

repetição, no pior caso, é executado sempre um comando de atribuição. Assim, o número de atribuições numa execução do corpo da repetição mais externa controlada pela variável *i* pode ser avaliado em $n-i+4$, para os valores $i = 1, 2, \ldots, n-1$. O total será dado então pela somatória:

$$\sum_{i=1}^{n-1}(n-i+4) = \frac{n^2}{2} + \frac{7n}{2} - 4$$

ou seja, o número de atribuições é da ordem de $\mathrm{O}(n^2)$(veja também o Exercício 4).

A tabela seguinte indica a importância deste tipo de análise. Deve-se notar que algoritmos que têm comportamento exponencial, isto é, $\mathrm{O}(2^n)$, são inúteis sob o ponto de vista prático.

n	$\log_2 n$	$n \log_2 n$	n^2	n^3	2^n
1	0	0	1	1	2
2	1	2	4	8	4
4	2	8	16	64	16
8	3	24	64	512	256
16	4	64	256	4096	65536
32	5	160	1024	32768	4.294.967.296

1.2 Tipos primitivos de dados

Explicar: *bits*, *bytes*, notação hexadecimal, caracteres, inteiros, reais, booleanos, etc.

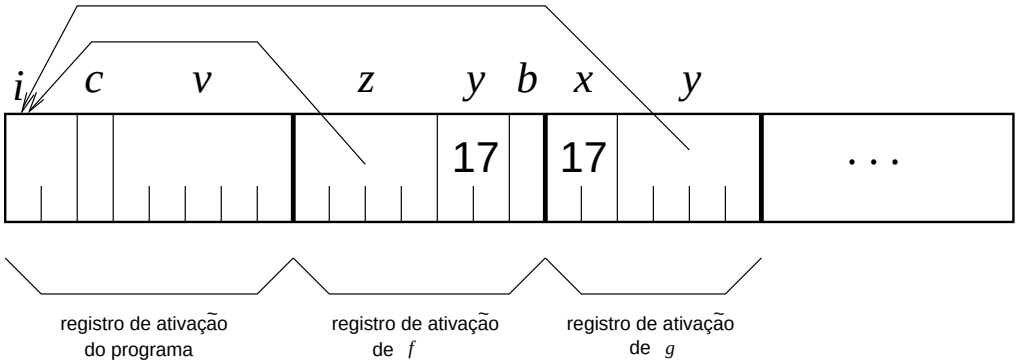
1.3 Execução de programas

Nesta seção apresentaremos alguns exemplos de *layout* da pilha de execução de programas. Nestes exemplos supusemos que um inteiro ocupa dois *bytes*, um caractere um *byte* e um endereço de variável ou um apontador, quatro *bytes*. Na realidade, em todos eles foram omitidos alguns pormenores que não são relevantes para esta discussão.

Programa 1.2 Exemplo de programa

```
program Exemplo1;
    var
        i: integer;  c: char;  v: array[1..5] of char;
    procedure g(x: integer; var y: integer);
    begin
    y := x
    end;

    procedure f(var z: integer);
        var  y: integer; b: char;
    begin
    y := 17; g(y,z)
    end;
begin
f(i)
end.
```

Figura 1.1 Registros de ativação para o Prog. 1.2



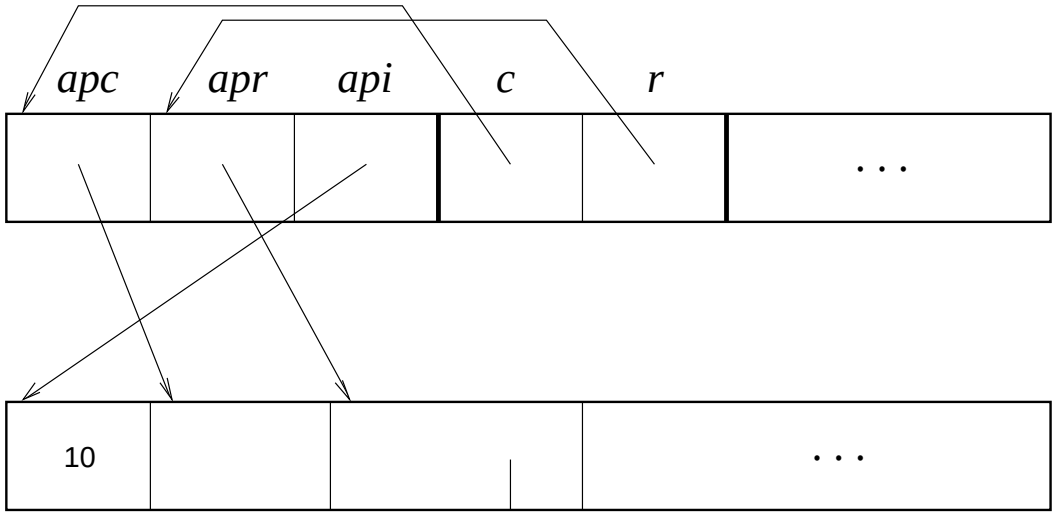
Neste primeiro exemplo, representamos os registros de ativação na da pilha durante a execução do

procedimento *g*.

Programa 1.3 Exemplo de programa com alocação de memória dinâmica

```
program Exemplo2;
    type
        Cadeia = array[1..5] of char;  ApCadeia = ↑Cadeia;
        Reg = record nome: Cadeia; idade: integer end;
        ApReg = ↑Reg;
    var apc: ApCadeia;  apr: ApReg;  api: ↑integer;

    procedure Aloca(var c: ApCadeia; var r: ApReg);
    begin
    new(api); api↑ := 10; new(c); new(r);
    end;

begin
Aloca(apc,apr); dispose(apc);
dispose(apr); dispose(api)
end.
```

Figura 1.2 Registros de ativação e *heap* para o Prog. 1.3

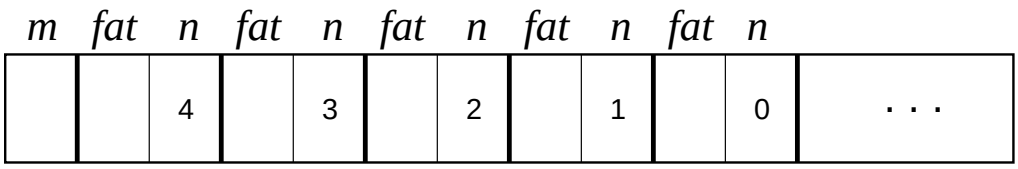


Neste exemplo, a pilha indica o estado das variáveis no fim da execução do procedimeto *Aloca*.

Programa 1.4 Exemplo de programa recursivo

```
program Exemplo3;

    var m: integer;

    function fat(n: integer): integer;
    begin
    if n=0
        then fat := 1
        else fat := n*fat(n-1)
    end;

begin
    m := fat(4)
end;
```

Figura 1.3 Registros de ativação para o Prog. 1.4



Neste caso, está indicado o instante em que há o número máximo de registros de ativação na pilha ($n = 0$); *fat* indica uma pseudo-variável para guardar o resultado da função.

1.4 Tipos abstratos de dados

Usar um exemplo para explicar: número complexos?

1.5 Exercícios

1. Análise de algoritmos envolve, com freqüência, o cálculo de somatórias de séries. Prove por indução os seguintes resultados:

   - $\sum_{i=1}^{n} i = n(n+1)/2, \quad n \geq 1$

- $\sum_{i=1}^{n} i^2 = n(n+1)(2n+1)/6, \quad n \geq 1$
- $\sum_{i=0}^{n} x^i = (x^{n+1} - 1)/(x-1), \quad x \neq 1, \quad n \geq 1$

2. Justifique os resultados obtidos para a análise detalhada dos três exemplos de trechos de programa na Seção 1.1 (pág. 2).

3. Determine quantas vezes é executado o comando de atribuição $x := x + 1$ no trecho de programa abaixo, em função de n:

```
for i:=1 to n do
    for j:=1 to i do
        for k:=1 to j do
            x := x+1
```

4. Faça uma análise mais detalhada da eficiência do procedimento *Ordena* apresentado no Prog. 1.1.

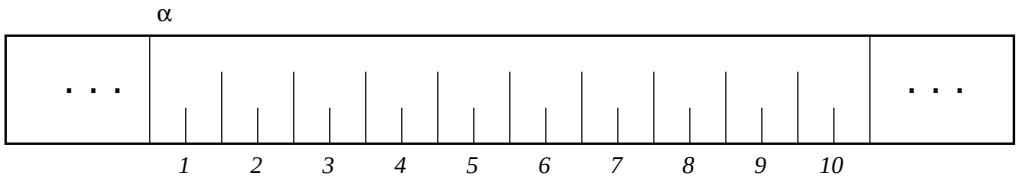
Capítulo 2

Estruturas Seqüenciais

Este Capítulo será dedicado à implementação seqüencial de estruturas, ou seja, àquela em que componentes ocupam posições consecutivas de memória, possibilitando o cálculo direto dos seus endereços.

2.1 Representação linearizada

Consideremos a declaração '*a*: **array**$[1..10]$ **of** *integer*'. Supondo que cada inteiro ocupa dois *bytes*, obtém-se a seguinte representação na memória:



Se denotarmos por α o endereço do primeiro *byte* do vetor, é claro que o endereço do *i*-ésimo elemento pode ser obtido por:

$$\text{ender}(a[i]) = \alpha + 2(i-1)$$

No caso geral de declaração '*b*: **array**$[p..u]$ **of** T' vale a fórmula

$$\text{ender}(b[i]) = \alpha + |T|(i-p)$$

onde $|T|$ denota o tamanho do objeto do tipo *T* em *bytes*. Esta fórmula pode ser reescrita como

$$\text{ender}(b[i]) = \beta + |T|i$$

onde $\beta = \alpha - |T|p$ tem um valor constante, uma vez tenha sido alocado o espaço para o vetor *b*. Desta maneira, percebe-se que o cálculo do endereço de um elemento do vetor envolve uma multiplicação e uma soma.

O caso de vetores multidimensionais (ou matrizes) é semelhante. Em particular em PASCAL, bem como algumas outras linguagens, a declaração '*c*: **array**$[p1..u1, p2..u2]$ **of** T' é equivalente a '*c*: **array**$[p1..u1]$ **of array**$[p2..u2]$ **of** T'. Desta maneira, utilizando o mesmo raciocínio acima, chega-se à fórmula

$$\text{ender}(c[i1, i2]) = \alpha + |T|(u2 - p2 + 1)(i1 - p1) + |T|(i2 - p2)$$

que também pode ser reescrita como

$$\text{ender}(c[i1, i2]) = \beta + t_1 i1 + t_2 i2$$

com

$$\begin{aligned} \beta &= \alpha - |T|((u2 - p2 + 1)p1 + p2) \\ t_1 &= |T|(u2 - p2 + 1) \\ t_2 &= |T| \end{aligned}$$

Conclui-se, desta maneira, que o cálculo do endereço envolve, neste caso, duas multiplicações e duas somas.

Fórmulas semelhantes podem ser desenvolvidas para os casos de dimensões maiores – veja o Exercício 2.

2.2 Formas especiais

Existem aplicações em que aparecem matrizes que têm formas especiais. Um exemplo são as matrizes triangulares, inferiores ou superiores. Dizemos que uma matriz é triangular inferior se ela é quadrada e todos os seus elementos acima da diagonal principal são nulos, como no exemplo seguinte:

$$\begin{vmatrix} -10 & 0 & 0 & 0 & 0 & 0 \\ 8 & 20 & 0 & 0 & 0 & 0 \\ 0 & 5 & 7 & 0 & 0 & 0 \\ 25 & -3 & 1 & 0 & 0 & 0 \\ 5 & 10 & 8 & -4 & 10 & 0 \\ 10 & 8 & -3 & 1 & 9 & 25 \end{vmatrix}$$

Uma maneira natural seria representar estas matrizes como exibido acima, com todos os elementos, utilizando, neste caso, a declaração '*a*: **array**$[1..6, 1..6]$ **of** *integer*'. Esta representação acarreta, entretanto, um desperdício de $(n-1)n/2$ elementos, onde *n* é a ordem da matriz. Além disto, algoritmos que manipulam este tipo de matriz terão que processar também os elementos nulos.

Uma alternativa é linearizar a matriz *a* e representá-la por meio de um vetor *ra*, de $n(n+1)/2$ elementos não necessariamente nulos. No caso do exemplo são 21 elementos: '*ra*: **array**$[1..21]$ **of** *integer*'. Será estabelecida então uma correspondência entre os elementos da matriz *a* e da sua representação linearizada *ra*, que pode ser tanto por linha como por coluna. No caso de representação por linha, teremos:

$$\begin{array}{ccccccccc} a[1,1] & & a[2,1] & & a[2,2] & \ldots & a[6,1] & \ldots & a[6,6] \\ \updownarrow & & \updownarrow & & \updownarrow & & \updownarrow & & \updownarrow \\ ra[1] & & ra[2] & & ra[3] & \ldots & ra[16] & \ldots & ra[21] \end{array}$$

O cálculo dos índices correspondentes é bastante simples. Os trechos de programa abaixo comparam o uso das duas implementações:

```
    var
      a: array [1..6,1..6] of integer;
      s,i,j: integer;

begin
    ...
    a[4,3] := 75;
    ...
    s := a[i,j]
    ...
end
```

```
    var
       ra: array [1..21] of integer;
       s,i,j: integer;

    function IndiceTriangular(i,j: integer): integer;
    begin
    if i<j
       then IndiceTriangular := 0
       else  IndiceTriangular := (i-1)*i div 2 + j
    end;

begin
    ...
    ra[IndiceTriang(4,3)] := 75;
    ...
    if i≥j
       then  s := ra[IndiceTriangular(i,j)]
       else   s := 0
    ...
end
```

O teste de índices antes de chamar a função pode ser evitado, conforme mostra o Exercício 3 no fim deste capítulo.

Existem outros casos especiais de matrizes que podem ser tratados de maneira análoga, como matrizes simétricas, diagonais e de banda (veja os Exercícios 6 a 8). Veremos também, na Seção 3.5, uma maneira bem diferente de implementar matrizes quando a grande maioria dos seus elementos é nula, sem que haja uma regra que indique quais são esses elementos (matrizes esparsas).

2.3 Exercícios

1. Considere a seguinte declaração em PASCAL:

   c: **array**[*p1..u1, p2..u2*] **of** *T*;

   Explique porque, nesta linguagem, os elementos da matriz são obrigatoriamente alocados *por linha*, os seja na seguinte ordem:

   $c[p1, p2]$ $c[p1, p2+1]$ $c[p1, p2+2]$ ... $c[p1, u2]$

$c[p1+1, p2]$	$c[p1+1, p2+1]$	$c[p1+1, p2+2]$	$\ldots$	$c[p1, u2]$
$\ldots$	$\ldots$	$\ldots$	$\ldots$	$\ldots$
$c[u1, p2]$	$c[u1, p2+1]$	$c[u1, p2+2]$	$\ldots$	$c[u1, u2]$

2. Considere a seguinte forma de declaração em PASCAL de matriz n-dimensional:

   $$d\text{: } \textbf{array}[p1..u1, p2..u2, ..., pn..un] \textbf{ of } T$$

   Estabeleça uma fórmula geral para calcular o endereço do elemento $d[i1, i2, \ldots, in]$. Quantas operações de soma e multiplicação são necessárias neste caso?

3. A implementação de matrizes triangulares sugerida na pág. 11 exige um teste *antes* de cada chamada da função *IndiceTriangular*. Mostre como modificar a representação para que o teste possa ser escondido dentro da própria função. Sugestão: acrescente mais uma posição especial ao vetor de representação *ra*, e preencha-o com um valor constante.

4. Escreva um procedimento que aceita duas matrizes triangulares inferiores e produz a sua soma, também triangular. O que acontece no caso da operação produto?

5. Sugira uma maneira eficiente de implementar simultaneamente *duas* matrizes triangulares de mesma ordem. Sugestão: Coloque-as numa mesma matriz retangular de tamanho conveniente.

6. Uma matriz quadrada *a* é dita *simétrica* se para todos os valores dos índices tem-se $a[i, j] = a[j, i]$. Sugira uma maneira mais eficiente de implementá-la do que a comum.

7. Uma matriz quadrada *a* é dita *diagonal* se para todos os valores distintos de i e j, tem-se $a[i, j] = 0$. Sugira uma maneira mais eficiente de implementá-la do que a comum.

8. Dizemos que uma matriz quadrada *a* é *de banda* (p, q) se $a[i, j] = 0$ para todos os valores de i e j tais que $i - j > p$ ou $j - i > q$. Note que uma matriz diagonal tem a banda $(0, 0)$; uma matriz comum tem a banda $(n-1, n-1)$.

   (a) esboce uma matriz quadrada 8×8, de banda $(4, 3)$;

   (b) sugira uma maneira mais eficiente de implementar matrizes de banda do que as comuns, usando a linearização.

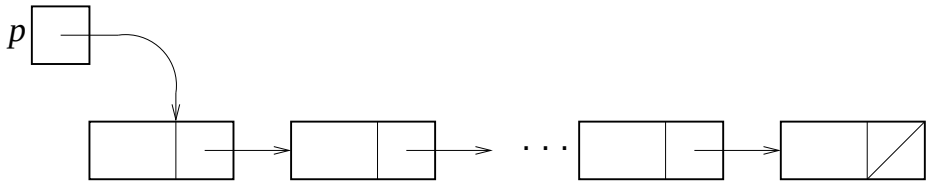
Capítulo 3

Estruturas ligadas

Introduziremos neste Capítulo as principais técnicas de construção de estruturas ligadas. Apresentaremos também dois exemplos típicos de utilização.

3.1 Listas ligadas simples

Uma lista ligada simples apontada por uma variável de programa *p* pode ser representada esquematicamente por:

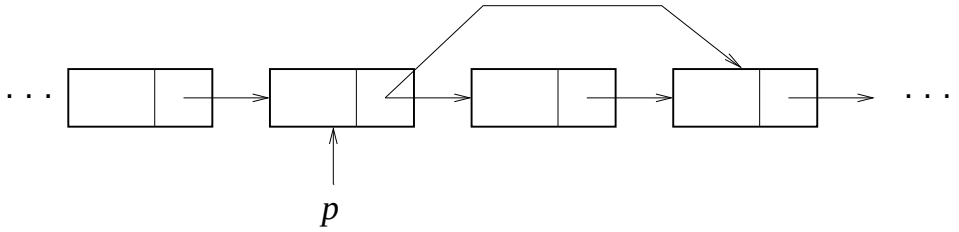


As declarações típicas para estabelecer os tipos necessários são:

```
type
    ApRegLista = ↑RegLista;
    RegLista = record
        info: T;
        prox: ApRegLista
    end;
    Lista = ApRegLista;
```

onde *T* é o tipo da informação que se deseja guardar nos elementos da lista. Note que utilizamos dois nomes – *Lista* e *ApRegLista* – para dois tipos aparentemente iguais. A intenção é frisar a diferença entre o primeiro, que é um tipo abstrato que está sendo implementado, e o segundo que é um tipo auxiliar utilizado na implementação. Neste caso particular (e muitos outros), estes tipos são equivalentes.

As operações básicas de inserção e remoção de nós genéricos na lista podem ser esquematizadas como indicado a seguir:



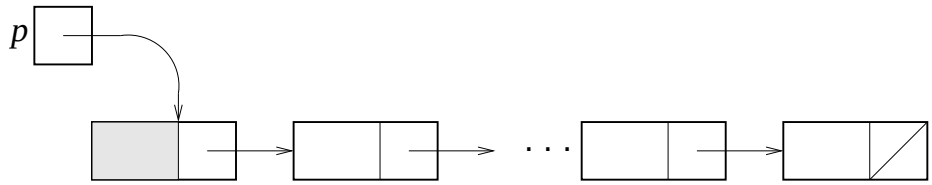
Os procedimentos correspondentes podem ser vistos no Prog. 3.1.

Programa 3.1 Rotinas para listas ligadas simples

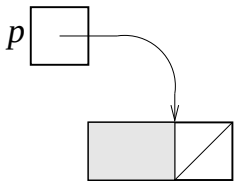
```
procedure InsereLista(p: Lista; x: T);
    var q: ApRegLista;
begin
new(q);
with q↑ do
    begin info := x; prox := p↑.prox end;
p↑.prox := q
end;
```

```
procedure RemoveLista(p: Lista);
    var q: ApRegLista;
begin
q := p↑.prox;
p↑.prox := q↑.prox;
dispose(q)
end;
```

A convenção dos procedimentos é que o parâmetro *p* deve apontar para o nó que *precede* o nó a ser inserido ou removido. Note-se, entretanto, que os procedimentos não funcionam para o caso em que deseja-se fazer a operação no início da lista. Este problema sugere a introdução de um primeiro nó fictício na lista, denominado *nó-cabeça*. O campo de informação deste nó não será utilizado para guardar os elementos da lista; em algumas aplicações poderá ser usado com outra finalidade como veremos mais adiante. Uma lista com nó-cabeça apontada por uma variável *p* pode ser representada esquematicamente como:



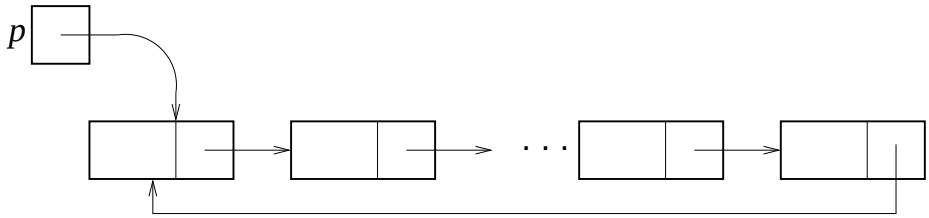
No caso de lista vazia, haverá apenas o nó cabeça:



Desta maneira, os procedimentos indicados funcionam mesmo quando se trata do primeiro nó da lista – neste caso o parâmetro *p* deve apontar para o nó-cabeça.

3.2 Listas circulares

Em algumas aplicações é conveniente utilizar *listas circulares* como esboçado a seguir:



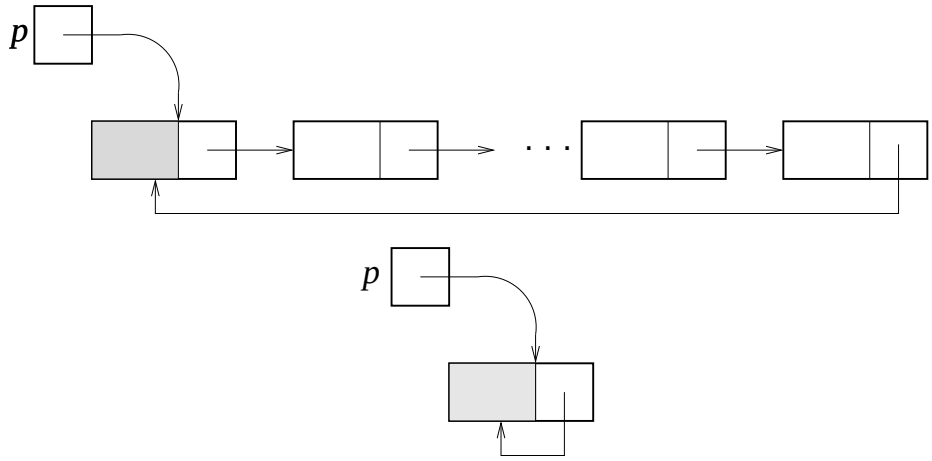
As operações de inserção e de remoção podem ser programadas como indicado no Prog. 3.2

Programa 3.2 Rotinas para listas ligadas circulares

```
procedure InsereCircular(var p: Lista; x: T);
    var  q: ApRegLista;
begin
new(q);
with q↑ do
    begin
    info := x;
    if p=nil
        then begin prox := q;  p := q  end
        else  begin
                prox := p↑.prox;
                p↑.prox := q
                end
    end
end;
```

```
procedure RemoveCircular(var p: Lista);
    var  q: ApRegLista;
begin
q := p↑.prox;
if q=p
    then p := nil
    else  p↑.prox := q↑.prox;
dispose(q)
end;
```

Estes procedimentos funcionam em qualquer caso, adotando-se o valor **nil** para a lista vazia. Esta convenção não é muito coerente pois **nil** não satisfaz a definição de lista circular. Este fato está refletido numa certa complicação aparente nos procedimentos. Uma maneira de simplificar a convenção é introduzir, também neste caso, um nó-cabeça (está indicada também uma lista circular vazia):



Uma vez que esta lista nunca fica de fato vazia, podem ser usados os mesmos procedimentos de inserção e remoção indicados na seção anterior no Prog. 3.1.

Uma vantagem adicional de haver nó-cabeça é a simplificação de alguns algoritmos. Por exemplo, um procedimento de busca numa lista circular poderia utilizar o campo de informação do nó-cabeça como *sentinela* conforme demonstrado no Prog. 3.3. Supusemos neste caso que o parâmetro *p* aponta para o nó-cabeça e que o procedimento devolve o apontador para o primeiro nó que contém o dado *x*; caso ele não exista, devolve **nil**. Note que o uso de sentinela evita testes de fim de lista dentro da repetição.

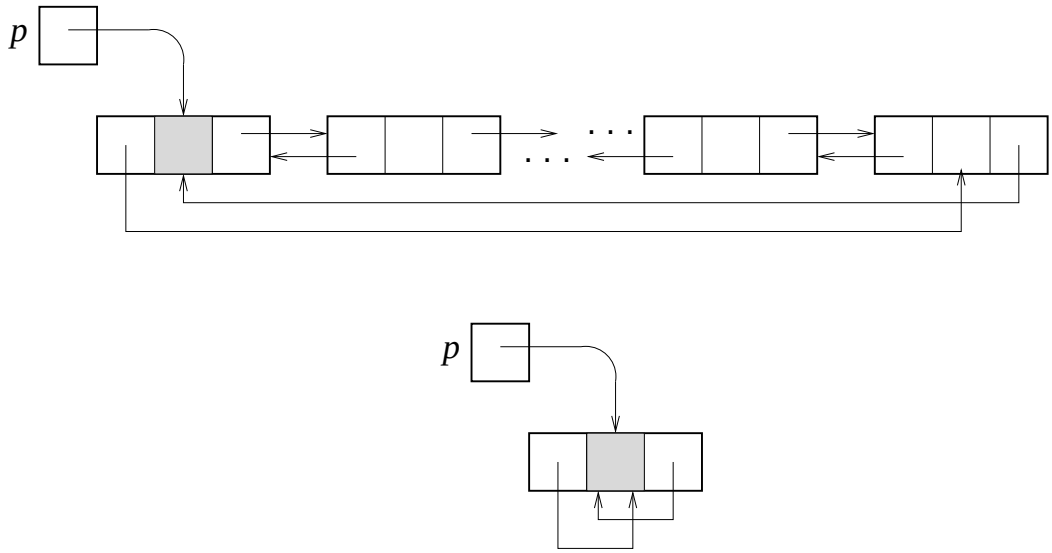
Programa 3.3 Busca com sentinela numa lista circular

```
function BuscaCircular(p: Lista; x: T): ApRegLista;
    var q: ApRegLista;
begin
      p↑.info := x;  q := p;
repeat
    q := q↑.prox
until q↑.info=x;
if q=p
    then BuscaCircular := nil
    else BuscaCircular := q
end;
```

3.3 Listas duplamente ligadas

As listas ligadas, tanto simples como circulares, apresentam alguns problemas. Por exemplo, dado o apontador para um nó, é difícil removê-lo da lista; no caso da lista simples, não se conhece o seu predecessor, no caso circular ele pode ser encontrado mas ao custo proporcional ao número de nós na lista (veja, entretanto, o Exercício 1). Uma alternativa é a utilização de listas *duplamente ligadas*. Também neste caso podemos

tornar as listas circulares e incluir um nó-cabeça, como ilustrado na figura seguinte:



As declarações típicas para implementar estas listas seriam:

```
type
    ApRegListaDupla = ↑RegListaDupla;
    RegListaDupla = record
        info: T;
        esq,dir: ApRegListaDupla
    end;
    ListaDupla = ApRegListaDupla;
```

As operações de inserção e de remoção nestas listas são ligeiramente mais complicadas, mas permitem, por exemplo, a remoção do próprio nó passado como argumento, como indicado no Prog 3.4.

3.4 Exemplo 1: manipulação de polinômios

O problema de manipulação de polinômios é um bom exemplo de problema em que é interessante utilizar as técnicas ligadas apresentadas nas seções anteriores. Um polinômio de grau $n \geq 0$ pode ser denotado por:

$$P(x) = a_n x^n + a_{n-1} x^{n-1} + \ldots + a_1 x^1 + a_0 x^0$$

onde $a_n \neq 0$, exceto possivelmente no caso $n = 0$.

Uma primeira alternativa natural para representar polinômios de grau máximo *grauMax* seria através de um vetor do tipo *Polinomio* conforme as declarações:

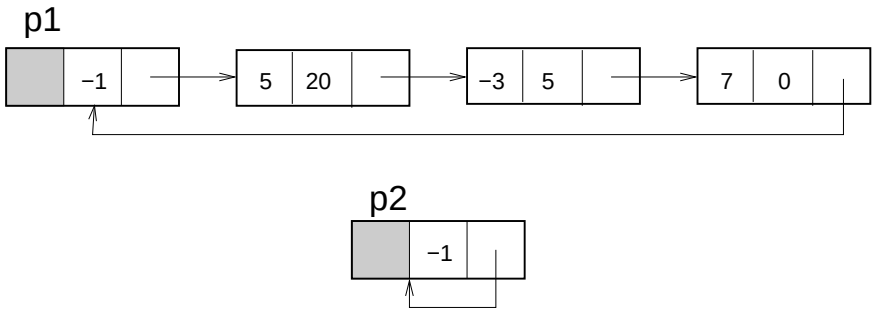
Programa 3.4 Rotinas para listas circulares duplamente ligadas

```
procedure InsereDuplaDireita(p: ListaDupla; x: T);
    var q: ApRegListaDupla;
begin
new(q);
with q↑ do
    begin
    dir := p↑.dir; esq := p;
    p↑.dir↑.esq := q; p↑.dir := q;
    info := x;
    end;
end;
```

```
procedure RemoveDupla(p: ListaDupla);
begin
p↑.esq↑.dir := p↑.dir;
p↑.dir↑.esq := p↑.esq;
dispose(p)
end;
```

```
type
    Termo = record
        coef: real;
        expo: integer
    end;
    Polinomio = array [0..grauMax] of Termo;
```

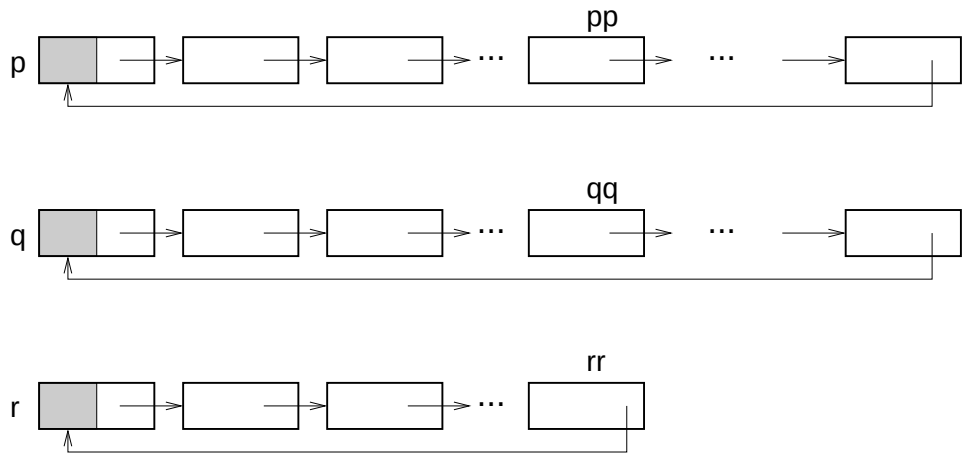
Com esta representação, a implementação de operações como soma, produto e outros sobre polinômios é bastante simples. Um primeiro problema desta representação é que ela exige um limite *a priori* para o grau máximo do polinômio. Isto se deve ao fato de que algumas linguagens de programação, como PASCAL, não permitem que vetores tenham seus limites calculados dinamicamente durante a execução. Um outro problema é a possível ineficiência caso o grau do polinômio seja alto, mas poucos dos seus termos sejam não nulos. Neste caso é mais conveniente utilizar a representação ligada. Adotaremos listas circulares com nó-cabeça, a fim de utilizar a técnica de sentinelas. A figura a seguir representa os polinômios: $P_1(x) = 5x^{20} - 3x^5 + 7$ e $P_2(x) = 0$.



Supusemos, nesta implementação, que cada nó representa um termo com coeficiente não nulo. Os termos estão em ordem decrescente dos expoentes; o nó-cabeça tem expoente -1 que será conveniente para implementar as operações que manipulam os polinômios. As declarações para esta estrutura seriam:

```
type
    ApTermo = ↑Termo;
    Termo   = record
        coef : real;
        expo: integer;
        prox: ApTermo;
    end;
    Polinomio = ApTermo;
```

A título de exemplo, mostraremos a construção da rotina que produz a soma de dois polinômios. Ela percorrerá simultaneamente as duas listas que representam os polinômios dados, somando os termos com expoentes iguais e copiando aqueles que não têm correspondente no outro. A figura a seguir indica a situação genérica encontrada durante o ciclo repetitivo principal da função:



Nesta figura, as variáveis *pp* e *qq* contêm apontadores para os próximos termos dos polinômios dados *p* e *q* cujos expoentes serão comparados; *rr* aponta para último termo inserido no resultado *r*. A rotina está apresentada no Prog. 3.5. Ela precisa ser completada com procedimentos convenientes: *CriaPolinomioNulo*, *InsereTermo* e *AvançaTermo* – veja o Exercício 2. Note-se a utilização de sentinela como o último termo de cada polinômio, evitando um tratamento especial para o fim de cada uma das listas.

3.5 Exemplo 2: matrizes esparsas

Um outro exemplo clássico de utilização de técnicas ligadas vistas nas seções anteriores deste capítulo é a representação de *matrizes esparsas*. Dizemos que uma matriz é esparsa se apenas uma pequena fração dos seus elementos é diferente de zero, sem que haja uma regra simples para determinar os elementos possivelmente não nulos. Este tipo de matrizes é comum em aplicações numéricas onde a fração de elementos não nulos pode ser da ordem de 1 a 5%. Nestes casos, uma representação que evite a inclusão de elementos nulos pode economizar espaço e tempo de cálculo, pois os elementos nulos não terão que ser processados.

Uma maneira de implementar matrizes esparsas foi sugerida por Knuth [**?**], e esta apresentação está baseada nela. Consideremos o seguinte exemplo de matriz:

Programa 3.5 Função de soma de polinômios

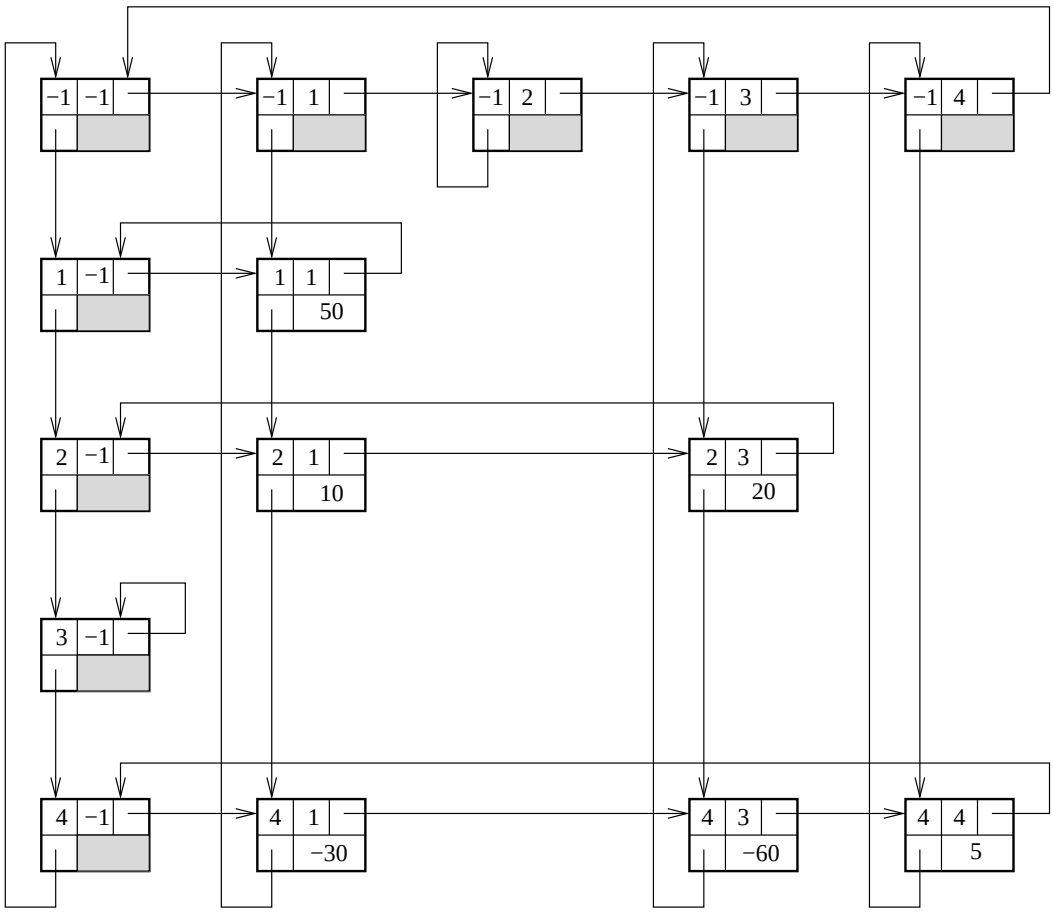
```
procedure SomaPolinomios(p,q: Polinomio; var r: Polinomio);
    var pp, qq, rr: ApTermo; ep, eq: integer; cr: real;
begin
CriaPolinomioNulo(r);
pp := p↑.prox; qq := q↑.prox; rr := r;
repeat
    ep := pp↑.expo; eq := qq↑.expo;
    if ep>eq
        then begin
                InsereTermo(pp↑.coef,ep,rr);
                AvancaTermo(rr); AvancaTermo(pp);
                end
    else if ep<eq
        then begin
                InsereTermo(qq↑.coef,eq,rr);
                AvancaTermo(rr); AvancaTermo(qq)
                end
    else if ep≠−1
        then begin
                cr := pp↑.coef+qq↑.coef;
                if cr≠0.0
                    then begin InsereTermo(cr,ep,rr); AvancaTermo(rr) end;
                AvancaTermo(pp); AvancaTermo(qq)
                end
until (ep=−1) and (eq=−1);
rr↑.prox := r
end;
```

$$\begin{vmatrix} 50 & 0 & 0 & 0 \\ 10 & 0 & 20 & 0 \\ 0 & 0 & 0 & 0 \\ -30 & 0 & -60 & 5 \end{vmatrix}$$

A sua representação poderia ser aquela indicada na Fig. 3.1.

Note-se que nesta representação cada nó indica um termo não nulo da matriz e pertence a duas listas: uma correspondente à sua linha (apontadores horizontais) e outra à sua coluna (apontadores verticais). Cada lista é organizada como uma lista ligada circular com nó cabeça, com os termos em ordem de colunas (nas listas das linhas) ou em ordem de linhas (nas listas das colunas). Os valores de sentinelas para os campos das linhas ou colunas são -1 (supõe-se que os índices da matriz são de 1 a n). Os próprios nós-cabeça das

Figura 3.1 Representação de uma matriz esparsa



linhas e das colunas constituem listas ligadas circulares, com um nó-cabeça comum; um apontador para este nó representará a matriz inteira.

Note-se que esta é uma representação geral, simétrica em relação às linhas e às colunas. Ela poderia ser simplificada para uma aplicação dada, utilizando por exemplo apenas as listas das linhas. Note-se também que a matriz não precisa ser quadrada.

As seguintes declarações podem ser utilizadas para implementar esta representação:

```
type
    ApRegEsparsa = ↑RegEsparsa;
    RegEsparsa = record
        linha, coluna: integer;
        valor: real;
        direita, abaixo: ApRegEsparsa
    end;
    Matriz = ApRegEsparsa;
```

Supondo que tenhamos uma matriz quadrada de ordem n, com r elementos não nulos, a representação utilizará $2n + r + 1$ nós ao invés de n^2 posições de número reais. É claro que cada nó ocupará muito mais *bytes* do que um real, mas poderá haver uma economia apreciável de memória se o valor de r for realmente muito menor que n^2. Analogamente, para certas operações com matrizes poderá haver economia de tempo, pois os elementos nulos não serão processados explicitamente.

A título de exemplo, o Prog. 3.6 mostra as rotinas básicas de atribuição e de recuperação de um valor de uma matriz esparsa. Note que para completar estas rotinas falta elaborar os procedimentos *InsereMatriz* e *RemoveMatriz*. Estes procedimentos são semelhantes àqueles indicados no Prog. 3.1 para listas ligadas, mas agora a operação é realizada em duas listas, sendo fornecidos os predecessores do nó em cada lista. Veja também Exercícios 4 a 7.

3.6 Exercícios

1. Explique o funcionamento do procedimento *RemoveLista*:

```
procedure RemoveLista(p: Lista);
    var q: ApRegLista;
begin
q := p↑.prox;
p↑ := q↑;
dispose(q)
end;
```

   Quais são as condições em que este procedimento pode ser utilizado?

2. Complete a declaração do procedimento *SomaPolinomios* do Prog. 3.5 com os procedimentos auxiliares.

3. Escreva os procedimentos que implementam as operações de subtração, multiplicação e divisão de polinômios, análogos ao procedimento de soma indicado no Prog. 3.5.

4. Escreva as rotinas que faltam no Prog. 3.6.

5. Note que na representação de matrizes esparsas indicada na Fig. 3.1 estão incluídos também os nós-cabeça das linhas e das colunas vazias, como por exemplo a linha 3. Estes nós poderiam ser eliminados às custas de alguma complicação adicional nos procedimentos que manipulam as matrizes. Refaça as rotinas indicadas no Prog. 3.6 para verificar este fato.

6. Supondo a representação de matrizes esparsas ilustrada na Fig. 3.1, escreva as seguintes rotinas:

   - **procedure** *InicializaMatriz(***var** *a: Matriz; m,n: integer);*
     Este procedimento cria uma nova matriz a, com m linhas e n colunas, com todos os elementos iguais a zero.
   - **procedure** *LiberaMatriz(***var** *a: Matriz);*
     Libera toda a memória dinâmica utilizada pela matriz a.

- **procedure** *SomaMatrizes(***var** *a,b,c: Matriz);*
  Soma as matrizes a e b, deixando o resultado em c. Supõe que as matrizes são compatíveis quanto ao número de linhas e de colunas.
- **procedure** *MultiplicaMatrizes(***var** *a,b,c: Matriz);*
  Multiplica as matrizes a e b, deixando o resultado em c. Supõe que as matrizes são compatíveis para esta operação.

7. Escreva um programa que testa as rotinas desenvolvidas no exercício anterior. Sugestão: Estabeleça uma convenção para a entrada de dados e escreva as rotinas de entrada e saída convenientes.

Programa 3.6 Recuperação de valor e atribuição em matriz esparsa

```
function ValorMatriz(var a: Matriz; i,j: integer): real;
    var p: ApRegEsparsa; k: integer;
begin
p := a;
for k:=1 to i do p := p↑.abaixo;
repeat
    p := p↑.direita
until (p↑.coluna≥j) or (p↑.coluna=−1);
if j≠p↑.coluna
    then ValorMatriz := 0.0
    else ValorMatriz := p↑.valor
end;

procedure AtribuiMatriz(var a: Matriz; i,j: integer; x: real);
    var p,q,pp,qq: ApRegEsparsa; k: integer;
begin
p := a; q := a;
for k:=1 to i do p := p↑.abaixo;
for k:=1 to j do q := q↑.direita;
repeat
    pp := p; p := p↑.direita
until (p↑.coluna≥j) or (p↑.coluna=−1);
repeat
    qq := q; q := q↑.abaixo
until (q↑.linha≥i) or (q↑.linha=−1);
if p↑.coluna=j
    then if x≠0.0
            then p↑.valor := x
            else RemoveMatriz(a,pp,qq)
    else if x≠0.0 then InsereMatriz(a,i,j,x)
end; { AtribuiMatriz }
```

Capítulo 4

Estruturas Lineares

Exploraremos neste Capítulo o conceito abstrato de listas, ou seja seqüências ordenadas de valores e suas operações. Veremos também algumas estruturas mais restritas, como pilhas e filas, de fundamental importância em Computação.

4.1 Estruturas lineares

Muitas aplicações envolvem representação e manipulação de seqüências ordenadas de objetos

$$x_1, x_2, \ldots, x_n.$$

Conforme visto nos capítulos anteriores, podemos adotar duas representações naturais para estas coleções: seqüencial (vetores) ou ligada. A escolha dependerá das características dos objetos e das operações às quais deverão ser submetidas as seqüências. Entre as várias operações possíveis sobre seqüências, podemos citar:

- selecionar e modificar o k-ésimo elemento;
- inserir um novo elemento entre as posições k e $k + 1$;
- remover o k-ésimo elemento;
- concatenar duas seqüências;
- desdobrar uma seqüência;
- copiar uma seqüência;
- determinar o tamanho de uma seqüência;
- buscar um elemento que satisfaz uma propriedade;
- ordenar uma seqüência;
- aplicar um procedimento a todos os elementos de uma seqüência;
- ...

É fácil perceber, por exemplo, que a operação de inserção ou de remoção numa seqüência é mais complicada no caso de representação seqüencial; por outro lado, com esta representação, é mais simples e eficiente a seleção do k-ésimo elemento da seqüência. Concluímos que, em qualquer aplicação, devemos estudar, em primeiro lugar, as operações que serão realizadas sobre os objetos. Conforme veremos nas seções seguintes, muitas vezes poderemos adotar representações alternativas, sendo que quase sempre haverá um compromisso entre a eficiência, memória utilizada, generalidade, facilidade de implementação, etc.

As seções seguintes serão dedicadas a listas especiais em que é possível executar apenas algumas das operações indicadas.

4.2 Pilhas

Consideremos o seguinte problema: dada uma seqüência constituída de caracteres '`(`', '`)`', '`[`' e '`]`', decidir se estes parênteses estão corretamente balanceados. Não definiremos de maneira exata o que é uma seqüência balanceada de parênteses, mas daremos alguns exemplos (veja, entretanto, o Exercicio 1):

Correto	Incorreto
ϵ	`(`
`()`	`)`
`[()]`	`[)`
`[]()[()[]]`	`()()[`
`((((([[]]))))`	`)(`

(O símbolo ϵ denota a seqüência vazia.)

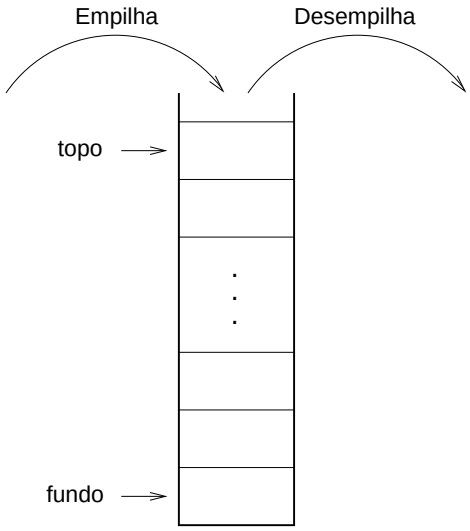
Uma primeira idéia para resolver o problema seria supor que a seqüência é representada como um vetor de n caracteres, e que o procedimento percorre o vetor "apagando" dois caracteres consecutivos que se balanceiam. Caso a seqüência seja balanceada, o resultado final deve ser a cadeia vazia. Não é difícil verificar que o número de operações para esta solução seria, no pior caso, da ordem de $\mathrm{O}(n^2)$. Uma outra idéia é utilizar uma estrutura de dados especial, denominada *pilha*.[1] Será um caso particular de lista linear, em que as operações de inserção e de remoção podem ser feitas somente numa única extremidade da lista. No caso do nosso problema, o procedimento deverá percorrer a seqüência de caracteres dada da esquerda para a direita. Os caracteres '`(`'e '`[`' serão inseridos na pilha. No caso dos caracteres '`)`'e '`]`', será verificado se o caractere que está na extremidade da pilha é do mesmo tipo que o que está sendo processado. No caso positivo, ele será removido e o procedimento continua. No caso negativo, a seqüência não era balanceada. No fim, a seqüência será considerada correta se não há mais símbolos a processar e a pilha está vazia.

Ilustramos a seguir os passos do procedimento para a seqüência '`([([][()])])`':

[1] Esta pilha é semelhante à pilha de execução vista na Seção 1.3.

Pilha	*Resto da seqüência*
Vazia	([([][()])])
(	[([][()])])
([	([][()])])
([(	[][()])])
([[[	][()])])
([(	[()])])
([[[	()])])
([[[(	)])])
([[[	])])
([[	)])
([	])
(	)
Vazia	ϵ

De uma maneira geral, diremos que a pilha é uma estrutura linear cujas extremidades serão denominadas *fundo* e *topo*. As operações de inserção (empilhamento) e remoção (desempilhamento) podem ser realizadas somente pelo topo da pilha. A figura seguinte dá uma idéia esquemática da estrutura:



Como em muitos outros contextos, podemos implementar uma pilha utilizando tanto a representação seqüencial (um vetor) quanto a ligada (uma lista). A escolha dependerá muito da aplicação. Indicamos nos Progs. 4.1 e 4.2 as declarações que implementam as duas opções. Adotaremos o ponto de vista de um tipo abstrato de dados, ou seja, um programa poderá utilizar qualquer uma das implementações contanto que as operações sobre a pilha sejam realizadas sempre através dos procedimentos e funções indicados. Supõe-se que existe uma rotina *TrataErro* a ser executada quando acontece um imprevisto. Note-se também que foi utilizada sistematicamente a passagem por variável para parâmetros do tipo *Pilha*, para evitar, no caso da implementação seqüencial, a cópia do vetor inteiro.

Programa 4.1 Implementação ligada de pilhas

```
type
    ApRegPilha = ↑RegPilha;
    RegPilha = record
        info: T;
        prox: ApRegPilha
    end;
    Pilha = ApRegPilha;

function PilhaVazia(var p: Pilha): boolean;
begin
PilhaVazia := p=nil
end;

procedure Empilha(var p: Pilha; x: T);
    var q: ApRegPilha;
begin
new(q);
with q↑ do
    begin
    info := x; prox := p; p := q
    end
end;

procedure InicializaPilha(var p: Pilha);
begin
p := nil
end;

procedure LiberaPilha(var p: Pilha);
    var q: ApRegPilha;
begin
while p≠nil do
    begin
    q := p; p := p↑.prox; dispose(q)
    end
end;

procedure Desempilha(var p: Pilha; var x: T);
    var q: ApRegPilha;
begin
if p=nil then TrataErro;
q := p;
with q↑ do
    begin x := info; p := prox end
dispose(q)
end;
```

Programa 4.2 Implementação seqüencial de pilhas

```
type
    Pilha = record
        topo: 0..TamMax;
        elemen: array [1..TamMax] of T
    end;

function PilhaVazia(var p: Pilha): boolean;
begin
PilhaVazia := p.topo=0
end;

procedure Empilha(var p: Pilha; x: T);
begin
with p do
    if topo=TamMax
        then TrataErro
        else begin
                topo := topo+1;
                elemen[topo] := x
                end
end;

procedure InicializaPilha(var p: Pilha);
begin
p.topo := 0
end;

procedure LiberaPilha(var p: Pilha);
begin
{ nada }
end;

procedure Desempilha(var p: Pilha; var x: T);
begin
with p do
    if topo= 0
        then TrataErro
        else begin
                x := elemen[topo];
                topo := topo-1
                end
end;
```

A título de exemplo, mostramos no Prog. 4.3 a implementação da função que determina se uma cadeia de caracteres representa uma seqüência balanceada de parênteses, conforme exposto no início desta seção. Suporemos que a cadeia é representada por um vetor de caracteres e que a informação contida nos elementos da pilha é do tipo *char*. Adotaremos também a convenção de que a cadeia termina com o caractere especial '#'. Note-se que a solução não depende da opção utilizada para implementar a pilha, que foi programada seguindo a idéia de tipo abstrato de dados. (Veja também o Exercício 5.6.)

Programa 4.3 Solução do problema de balanceamento de parênteses

```
type
    Cadeia = array [1..ComprMax] of char;
    ...
function Balanceada(var c: Cadeia): boolean;
    var
        p: Pilha; t: char; i: 0..ComprMax;
        resultado, continua: boolean;
begin
InicializaPilha(p); i := 1;
resultado := false; continua := true;
while continua do
    begin
    case c[i] of
      '#': begin
             resultado := PilhaVazia(p);
             continua := false
             end;
      '(','[': Empilha(p,c[i]);
      ')': if PilhaVazia
             then continua := false
             else begin
                    Desempilha(p,t);
                    if t≠'(' then continua := false
                    end;
      ']': if PilhaVazia
             then continua := false
             else begin
                    Desempilha(p,t);
                    if t≠'[' then continua := false
                    end;
    end;
    i := i+1
    end
    LiberaPilha(p);
    Balanceada := resultado
end;
```

4.3 Exemplo de aplicação: manipulação de expressões

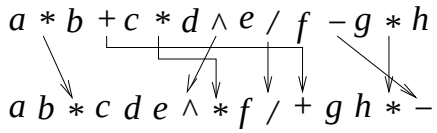
Expressões aparecem em linguagens de programação como PASCAL, ou então em sistemas que realizam manipulação simbólica. É comum nestes casos adotar-se representrações alternativas para expressões, diferentes daquelas usadas normalmente. Duas alternativas bastante comuns são a *notação pós-fixa* e a *notação pré-fixa*.[2]

Simplificaremos a nossa discussão supondo que todos os operadores usados nas expressões são diádicos, ou seja, requerem dois operandos. Em notação comum, chamada também de *notação infixa*, o operador fica entre os dois operandos. Como os próprios nomes indicam, em *notação pós-fixa* o operador segue os dois operandos enquanto que em *notação pré-fixa* o operador precede os dois operandos. Consideremos alguns exemplos das três notações:

infixa	*pós-fixa*	*pré-fixa*
a	a	a
$a + b$	$ab+$	$+ab$
$a + b * c$	$abc*+$	$+a*bc$
$(a + b) * c$	$ab+c*$	$*+abc$

Deve-se notar que as notações pré-fixa e pós-fixa não necessitam de parênteses. A própria forma da expressão indica a ordem das operações como ilustrado pelos exemplos. Em notação infixa, utilizamos, por convenção, as prioridades dos operadores e a sua associatividade à esquerda ou à direita. Quando desejamos um resultado que é contrário a estas convenções, utilizamos os parênteses.

Um problema de interesse óbvio é o de transformação entre estas notações. Consideremos o caso de transformação da notação infixa para a pós-fixa, e tomemos um exemplo um pouco mais complexo:



onde $a \wedge b$ denota a^b. Nota-se que os nomes das variáveis são copiados da entrada infixa para a saída pós-fixa na mesma ordem. Entretanto, os operadores parecem "esperar" a chegada de um outro operador, para que seja tomada a decisão qual o operador seguinte que vai para a saída. Não é difícil perceber que esta decisão está baseada nas prioridades dos operadores e que os operadores que são "lembrados" seguem uma disciplina de pilha, esperando até que apareça na entrada um de prioridade menor ou igual. Podemos imaginar, portanto, um algoritmo utilizando uma pilha auxiliar que faz a transformação. Os passos deste algoritmo para o exemplo mostrado estão indicados a seguir:

[2] Estas notações são chamadas também de *notação polonesa* e *notação polonesa reversa*, em homenagem ao matemático polonês Jan Łukasiewicz que as introduziu no início deste século.

Saída	*Pilha*	*Entrada*
		$a*b+c*d \wedge e/f-g*h$
a		$*b+c*d \wedge e/f-g*h$
a	$*$	$b+c*d \wedge e/f-g*h$
ab	$*$	$+c*d \wedge e/f-g*h$
$ab*$		$+c*d \wedge e/f-g*h$
$ab*$	$+$	$c*d \wedge e/f-g*h$
$ab*c$	$+$	$*d \wedge e/f-g*h$
$ab*c$	$+*$	$d \wedge e/f-g*h$
$ab*cd$	$+*$	$\wedge e/f-g*h$
$ab*cd$	$+*\wedge$	$e/f-g*h$
$ab*cde$	$+*\wedge$	$/f-g*h$
$ab*cde\wedge$	$+*$	$/f-g*h$
$ab*cde\wedge*$	$+$	$/f-g*h$
$ab*cde\wedge*$	$+/$	$f-g*h$
$ab*cde\wedge*f$	$+/$	$-g*h$
$ab*cde\wedge*f/$	$+$	$-g*h$
$ab*cde\wedge*f/+$		$-g*h$
$ab*cde\wedge*f/+$	$-$	$g*h$
$ab*cde\wedge*f/+g$	$-$	$*h$
$ab*cde\wedge*f/+g$	$-*$	h
$ab*cde\wedge*f/+gh$	$-*$	
$ab*cde\wedge*f/+gh*$	$-$	
$ab*cde\wedge*f/+gh*-$		

Pode-se perceber que, numa implementação completa, o algoritmo teria que prever os casos iniciais (pilha vazia) e finais (entrada vazia).

A generalização do algoritmo para expressões que têm parênteses é relativamente simples e, na realidade, semelhante ao tratamento das expressões balanceadas da Seção 4.2. Um abre-parêntese deverá ser empilhado e tratado como o fundo de uma pilha menor, até que seja encontrado o fecha-parêntese correspondente; todos os operadores acima do abre-parêntese, deverão ser transferidos então para a saída. Um esboço do procedimento que transforma expressões infixas em pós-fixas está indicado no Prog. 4.4. O funcionamento correto do procedimento depende da definição conveniente de uma função *Prioridade* que indica a relação entre o símbolo de entrada e o símbolo que está no topo da pilha. Note-se que esta relação deve incluir o símbolo '(' quando na pilha (veja também o Exercício 2). O procedimento deve ser completado com algumas rotinas óbvias.

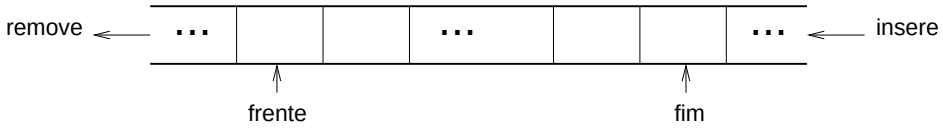
As outras transformações são um pouco mais complicadas e também serão deixadas para exercícios.

Programa 4.4 Transformação da notação infixa para pós-fixa

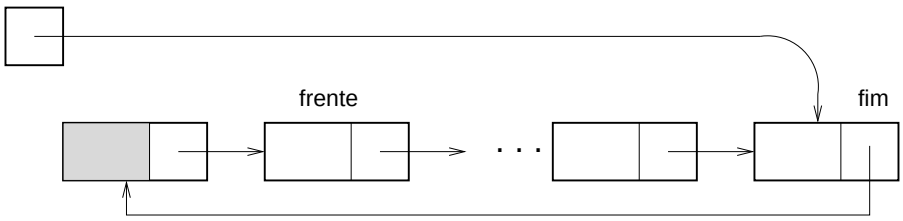
```
type
    Cadeia = array [1..ComprMax] of char;
    ...
procedure InPos(var expr: Cadeia);
    var
        p: Pilha;
        pe: 0..ComprMax;
        corr,aux: char;
        fim: boolean;
begin
InicializaPilha(p);
Empilha(p,'(');
pe := 1;
repeat
    corr := expr[pe]; inc(pe);
    case corr of
        'a'..'z':
                Sai(corr);
        ')','#':
                repeat
                    Desempilha(p,aux);
                    if aux≠'('
                        then Sai(aux)
                until aux='(';
        '+','-','*','/','^':
                begin
                fim := false;
                repeat
                    Desempilha(p,aux);
                    if Prioridade(corr,aux)
                        then begin
                                Empilha(p,aux);
                                Empilha(p,corr);
                                fim := true
                                end
                        else Sai(aux)
                until fim
                end
    end {case}
until corr='#';
LiberaPilha(p)
end; {InPos}
```

4.4 Filas

Uma *fila* é uma estrutura linear restrita em que todas as inserções são realizadas numa extremidade (*fim*) e todas as remoções na outra extremidade (*frente*). Esquematicamente podemos podemos visualizar uma fila como:

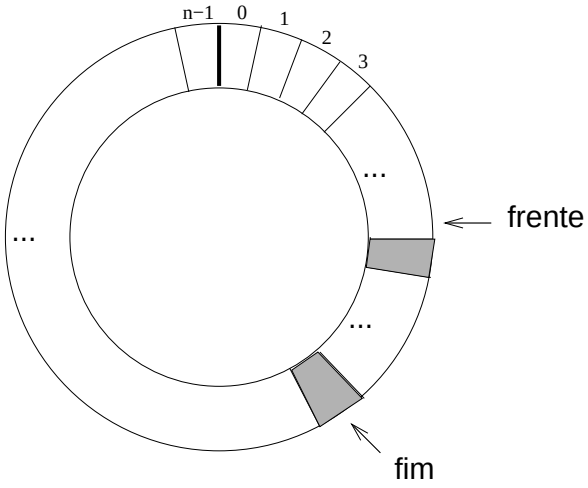


Como no caso de pilhas, temos as duas alternativas naturais de implementação: seqüencial e ligada, sendo que a decisão dependerá do contexto de utilização. No caso da implementação ligada, é conveniente utilizar uma lista circular com nó-cabeça, como esboçado a seguir:



Neste caso, é interessante que a variável que indicará a fila aponte para o seu último nó. Desta maneira, com um apontador, tanto a frente como o fim da fila tornam-se acessíveis de maneira eficiente. O Prog. 4.5 mostra a implementação desta opção, sob a forma de um tipo abstrato de dados.

No caso da implementação seqüencial, há o problema inerente de previsão *a priori* do número máximo de elementos que poderá haver na fila. Fora esta restrição, uma solução natural é a adoção de um vetor "circular", como indicado no esboço:



Por convenção, o índice *frente* aponta para a posição que precede o primeiro elemento da fila, enquanto que o índice *fim* aponta para o último elemento. Com isto, algumas expressões a serem usadas nos procedimentos ficarão mais simples. Conseqüentemente, poderemos adotar para inicializar uma fila os valores:

$$\mathit{frente} = \mathit{fim} = 0$$

Entretanto, a condição para detetar que não há mais espaço na fila é:

$$\textit{frente} = \textit{fim}$$

ou seja, não podemos distinguir entre uma fila cheia e uma fila vazia! A solução mais simples para este problema é sacrificar uma posição do vetor e usar como condição de fila cheia (veja também o Exercício 4):

$$\textit{frente} = (\textit{fim} + 1) \bmod n$$

Com estas convenções, podemos reprogramar o tipo abstrato de dados *Fila* como indicado no Prog. 4.6.

A estrutura de fila aparece em muitas aplicações. Em particular, veremos um exemplo na Seção 6.3 aplicado ao percurso em largura de árvores binárias.

4.5 Exercícios

1. Defina de maneira precisa o conceito de seqüência balanceada de perênteses. Sugestão: Use definições indutivas (isto é, recursivas).

2. Implemente o algoritmo de transformação da notação infixa para pós-fixa descrito na Seção 4.3, incluindo expressões com operadores diádicos e parênteses, bem como o tratamento de erros. Uma sugestão para a implementação da função *Prioridade* é a construção de uma tabela que associa a cada símbolo considerado um número inteiro, conforme o símbolo esteja na entrada ou no topo da pilha. A comparação entre estes números indicará a prioridade em cada caso. Um exemplo desta tabela para os símbolos considerados seria:

Símbolo	Pilha	Entrada
$\wedge$	3	4
$*$, $/$	2	2
$+$, $-$	1	1
$($	0	4

   Foi adotada nesta tabela a convenção de dar prioridade ao símbolo da pilha quando os números são iguais (associação à esquerda). Note também a maneira de tratar o parêntese '$($' para que ele seja enquadrado no caso geral de operadores; o parêntese '$)$' é tratado de maneira especial como indicado no Prog. 4.4.

3. A notação pós-fixa pode ser estendida para operadores monádicos (de um operando), entretanto eles introduzem ambigüidade. Por exemplo, as expressões infixas distintas '$a - (-b)$' e '$-(a - b)$' seriam traduzidas para a mesma expressão pós-fixa '$ab - -$'. A fim de evitar este problema, os operadores devem utilizar símbolos distintos na tradução. Adotando-se o símbolo $\sim$ para o operador unário $-$, teríamos:

$$\begin{array}{ll} a - (-b) & ab \sim - \\ -(a - b) & ab- \sim \end{array}$$

Complete a implementação do exercício anterior para incluir os operadores monádicos '$-$' e '$+$', usando esta representação.

4. A implementação seqüencial de filas proposta neste capítulo sacrifica uma posição do vetor para distinguir entre uma fila cheia e uma fila vazia. Uma outra solução seria incluir no registro que representa a fila uma variável booleana que indicasse se a fila está cheia ou não. Reescreva os procedimentos do Prog. 4.6 de acordo com esta sugestão.

5. Em algumas aplicações é interessante utilizar o conceito de *fila dupla*. Nesta estrutura, as inserções e remoções podem ser feitas em qualquer uma das duas extremidades da seqüência. Proponha implementações ligada e seqüencial para esta estrutura e escreva as rotinas básicas para manipulá-la, análogas às dos Progs. 4.5 e 4.6.

Programa 4.5 Implementação ligada de filas

```
type
    ApRegFila = ↑RegFila;
    RegFila = record
        info: T;
        prox: ApRegFila
    end;
    Fila = ApRegFila;

function FilaVazia(var p: Fila): boolean;
begin
FilaVazia := p=p↑.prox
end;

procedure InsereFila(var p: Fila; x: T);
    var  q: ApRegFila;
begin
new(q);
with q↑ do
    begin
    info := x; prox := p↑.prox;
    p↑.prox := q; p := q
    end
end;

procedure InicializaFila(var p: Fila);
begin
new(p); p↑.prox := p
end;

procedure LiberaFila(var p: Fila);
    var  q,r: ApRegFila;
begin
q := p↑.prox;
while q≠p do
    begin
    r := q↑.prox; dispose(q); q :=r
    end;
dispose(p)
end;

procedure RemoveFila(var p: Fila; var x: T);
    var  q,r: ApRegFila;
begin
q := p↑.prox;
if p=q  then  TrataErro;
r := q↑.prox;  x := r↑.info;
q↑.prox := r↑.prox;
if r=p then p:= q;
dispose(r)
end;
```

Programa 4.6 Implementação seqüencial de filas

```
type
    Fila = record
        frente, fim: 0..TamMax;
        elemen: array [0..TamMax−1] of T
    end;

function FilaVazia(var p: Fila): boolean;
begin
FilaVazia := p.frente=p.fim
end;

procedure InsereFila(var p: Fila; x: T);
begin
with p do
    begin
    if frente=((fim+1) mod TamMax)
        then TrataErro;
    fim := (fim+1) mod TamMax;
    elemen[fim] := x
end;

procedure InicializaFila(var p: Fila);
begin
with p do
  begin frente := 0; fim := 0 end
end;

procedure LiberaFila(var p: Fila);
    var q,r: ApRegFila;
begin
{ nada }
end;

procedure RemoveFila(var p: Fila; var x: T);
begin
if FilaVazia(p) then TrataErro;
with p do
    begin
    frente := (frente+1) mod TamMax;
    x := elemen[frente]
    end
end;
```

Capítulo 5

Recursão

O objetivo deste Capítulo é discutir a *recursão*, um mecanismo muito poderoso de definição e programação, aplicável em muitas situações distintas como tratamento de árvores (veja Cap. 6), aplicações simbólicas, implementação de linguagens de programação, inteligência artificial e outros.

Na primeira seção deste capítulo veremos alguns exemplos de recursão e discutiremos quando ela deve ser utilizada. Na seção seguinte veremos como a recursão explícita pode ser substituída por uma pilha do tipo visto na Sec. 4.2. Finalmente, na última seção introduziremos um exemplo mais elaborado de aplicação da recursão.

5.1 Recursão e repetição

Dizemos que uma rotina é *recursiva* se a sua definição envolve uma chamada a ela mesma. Note-se que, neste sentido, o termo *recursão* é equivalente ao termo *indução* utilizado por matemáticos. Um exemplo clássico e muito simples de definição indutiva é a função fatorial que pode ser caracterizada por:

$$\text{fat}(n) = \begin{cases} 1 & \text{se } n = 0 \\ n \times \text{fat}(n-1) & \text{se } n > 0 \end{cases}$$

Obviamente, esta definição indutiva pode ser transformada numa declaração recursiva em PASCAL como no Prog. 5.1. Por outro lado, é muito fácil escrever uma versão não recursiva da mesma função, como indicado no mesmo programa. Uma análise das duas versões mostra que o número de operações executadas é, em ambos os casos, da ordem de $\mathrm{O}(n)$. Entretanto, na versão iterativa são evitadas as chamadas recursivas e a conseqüente criação de registros de ativação, conforme visto na Seção 1.3. Desta maneira, a constante de proporcionalidade certamente será menor para esta versão.

Um outro exemplo clássico é o cálculo da série de Fibonacci. Esta série pode ser definida por:

$$\begin{aligned} \text{fibo}(0) &= 0 \\ \text{fibo}(1) &= 1 \\ \text{fibo}(n) &= \text{fibo}(n-1) + \text{fibo}(n-2) \quad \text{se } n > 1 \end{aligned}$$

Novamente, a transformação desta definição num programa recursivo é trivial conforme indicado no Prog. 5.2. Por outro lado, é bastante simples escrever uma versão não recursiva da função, indicada no mesmo programa. Analisemos a eficiência das duas formulações, calculando o número de operações de soma $S(n)$

Programa 5.1 Cálculo da função fatorial

```
function fat(n: integer): integer;
begin
if n=0
   then fat := 1
   else fat := n*fat(n-1)
end;
```

```
function fat(n: integer): integer;
   var i,s: integer;
begin
s := 1;
for i:=1 to n do s := s*i;
fat := s
end;
```

realizadas. Para a versão recursiva, podemos concluir:

$$\mathrm{S}(n) = \begin{cases} 0 & \text{se } n \leq 1 \\ \mathrm{S}(n-1) + \mathrm{S}(n-2) + 1 & \text{se } n > 1 \end{cases}$$

A partir desta formulação, pode-se provar por indução finita (veja o Exercício 1) que

$$\mathrm{S}(n) = \mathrm{fibo}(n+1) - 1$$

Por outro lado, pode-se mostrar que $\mathrm{fibo}(i) \approx ((1+\sqrt{5})/2)^i/\sqrt{5}$. Assim, para valores grandes de n, temos $\mathrm{S}(n) \approx 1,6^n/1,4$. Por exemplo, para calcular $\mathrm{fibo}(100)$ seriam necessárias mais de 10^{20} somas! É simples verificar, entretranto, que para a versão iterativa o número de somas é apenas n. Obviamente, neste caso não faz sentido utilizar a versão recursiva.

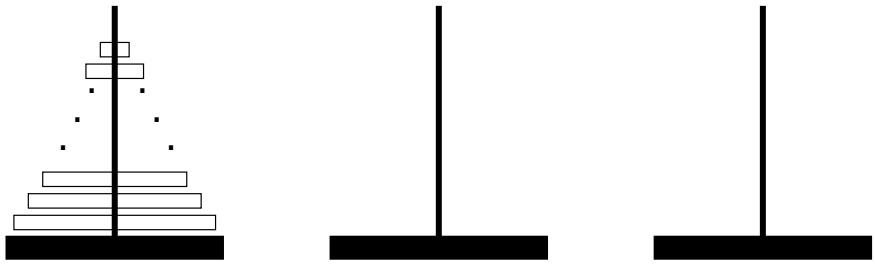
Programa 5.2 Cálculo da série de Fibonacci

```
function fibo(n: integer): integer;
begin
if n≤1
   then fibo := n
   else fibo := fibo(n-1)+fibo(n-2)
end;
```

```
function fibo(n: integer): integer;
var f1,f2,f3,k : integer;
begin
f1 := 0; f2 := 1;
for k:=1 to n do
   begin
   f3 := f1+f2; f1 := f2; f2 := f3
   end;
fibo := f1
end;
```

Um outro exemplo interessante e clássico de problema que tem uma solução simples utilizando formulação recursiva é o de torres de Hanoi. Suponha que temos três hastes, denominadas *A*, *B* e *C*. Na haste *A* há 64 discos de tamanhos distintos, com furos no centro, em ordem de tamanhos crescentes; as outras hastes estão vazias, conforme indicado na Fig. 5.1. O problema consiste em transferir todas os discos da haste *A* para a *B*, utilizando a haste *C* como auxiliar, seguindo as seguintes regras:

Figura 5.1 O problema de torres de Hanoi



- somente um disco é movido de cada vez;
- um disco maior nunca pode ser colocado em cima de um disco menor.

Pensando de maneira recursiva, não é difícil perceber que a solução pode ser obtida conforme indicado no Prog. 5.3, no qual são impressos todos os movimentos. A chamada inicial seria, neste caso, *TorresDeHanoi('A','B','C',64)*.

Programa 5.3 Solução do problema de torres de Hanoi

```
procedure TorresDeHanoi(origem, destino, aux: char; n: integer);
begin
if n>0
   then begin
        TorresDeHanoi(origem,aux,destino,n−1);
        writeln('Mova de ',origem,' para ',destino);
        TorresDeHanoi(aux,destino,origem,n−1)
        end
end;
```

Os exemplos vistos neste capítulo utilizam a *recursão direta*, ou seja, a rotina que está sendo definida depende diretamente da sua própria definição. Em algumas aplicações é interessante utilizar a *recursão mútua* em que duas ou mais rotinas dependem mutuamente uma da outra. As funções f e g são um exemplo muito simples deste conceito:

$$f(n) = \begin{cases} 0 & \text{se } n = 0 \\ g(n-1) & \text{se } n > 0 \end{cases}$$

$$g(n) = \begin{cases} 1 & \text{se } n = 0 \\ f(n-1) & \text{se } n > 0 \end{cases}$$

Note-se que, na realidade, $f(n) = n \bmod 2$ e $g(n) = (n+1) \bmod 2$. Na última seção deste capítulo veremos um exemplo mais interessante de recursão mútua.

A transformação de definições mutuamente recursivas em programas é muito simples. No caso de PASCAL, obtém-se para este exemplo o Prog. 5.4. Note-se que a declaração **forward** é exigida pelo compilador.

Programa 5.4 Exemplo de recursão mútua

```
function g(n: integer): forward;

function f(n: integer): integer;
begin
if n=0
    then f := 0
    else  f := g(n−1)
end;

function g(n: integer): integer;
begin
if n=0
    then g := 1
    else  g := g(n−1)
end;
```

5.2 Eliminação da recursão

Nem sempre é possível, como no caso de funções de cálculo de fatoriais e de números de Fibonacci na seção anterior, eliminar a formulação recursiva da solução de um problema. Entretanto, existem situações nas quais é conveniente eliminar o uso da recursão e substituí-la pela utilização de uma pilha explícita. As versões de rotinas obtidas são, em geral, mais complexas mas costumam ser mais eficientes, no que diz respeito ao coeficiente de proporcionalidade, por evitarem várias operações como chamadas e retornos de rotinas, criação de registros de ativação, etc.

Consideremos o esboço de procedimento recursivo típico apresentado no Prog. 5.5. A sua tradução para um procedimento sem recursão mas com uma pilha explícita está indicada no Prog. 5.6.

Basicamente, a versão transformada do procedimento simula a execução do procedimento original. Antes de simular uma chamada recursiva, os valores das variáveis locais (incluindo os parâmetros) são salvos na pilha, juntamente com uma indicação de qual das chamadas será simulada. A seguir são calculados os valores dos parâmetros para a nova chamada e o ciclo de simulação é repetido. No retorno de uma chamada simulada, são restaurados os valores das variáveis locais e executadas as ações pendentes daquela chamada. O cálculo de novos valores dos parâmetros deve ser feito com cuidado pois eles podem depender dos valores anteriores; para isto são usadas as variáveis temporárias *t1, t2, ...*.

Deve-se notar que este esquema de tradução é bastante geral, mas alguns cuidados precisam ser tomados

na presença de parâmetros passados por variável. O esquema não leva em consideração nenhuma propriedade do algoritmo que está sendo transformado nem da estrutura dos seus dados. Traduções mais simples e mais eficientes podem ser conseguidas em casos específicos como exemplificado por algoritmos de percurso de árvores binárias na Seção 6.3.

Programa 5.5 Esboço de um procedimento recursivo

```
procedure Exemplo(x1,x2, ...: ...);
    var  y1, y2, ...: ...;
begin
Ci;
if E(...)
    then  C
    else  begin
              C1;
              Exemplo(e11,e12,...);
              C2;
              Exemplo(e21,e22,...);
              C3;
              ...
              Cm;
              Exemplo(em1,em2,...);
              Cf
              end
end;
```

Programa 5.6 Esboço da transformação do procedimento recursivo do Prog. 5.5

```
type  Chamadas = (chamada1, chamada2,chamada3,...);

procedure Exemplo(x1,x2, ...: ...);
    type Acoes = (entrada,retorno,saida);
    var  y1, y2, ...: ...;
         f: Pilha;  ch: Chamadas;  acao: Acoes;  t1, t2: ...;
begin
    InicializaPilha(f); acao := entrada;
    repeat
        case acao of
          entrada: begin
                     Ci;
                     if E(...)
                       then begin C; acao := retorno end
                       else  begin
                               C1;
                               Empilha(f,x1,...,y1,...,chamada1);
                               t1 := e11;  t2 := e12;  ...
                               x1 := t1;  x2 := t2; ...
                               end
                   end;
          retorno: if PilhaVazia(f)
                       then acao := saida
                       else  begin
                               Desempilha(f,x1,...,y1,...,ch);
                               case ch of
                                   chamada1: begin
                                               C2;
                                               Empilha(f,x1,...,y1,...,chamada2);
                                               t1 := e21;  t22 := e22;  ...
                                               x1 := t1;  x2 := t2; ...
                                               acao := entrada
                                               end;
                                   chamada2: begin
                                               C3;
                                               Empilha(f,x1,...,y1,...,chamada3);
                                               t1 := e31;  t2 := e32;  ...
                                               x1 := t1;  x2 := t2; ...
                                               acao := entrada
                                               end;
                                   ...
                                   chamadam: Cf
                               end { case }
                               end
        end { case }
    until acao=saida
end;
```

5.3 Exemplo: análise sintática

Vimos na Seção 4.3 um exemplo interessante de manipulação de expressões ao desenvolver um programa que transforma expressões em notação infixa para a notação pós-fixa, utilizando uma pilha de operadores. Conforme estudamos neste capítulo, existe uma relação muito próxima entre a utilização de pilha e a recursão; na realidade, a pilha é uma maneira de implementar a recursão. É natural, portanto, que o programa de transformação visto possa ser expresso em termos recursivos, tornando-o mais fácil de compreender, apesar de eventualmente um pouco menos eficiente.

Consideremos inicialmente as expressões infixas que podem ser construídas com varíaveis representadas por letras minúsculas, os operadores diádicos '$+$', '$-$', '$*$', '$/$' e os parênteses '$($' e '$)$' (omitimos, por enquanto, o operador de exponenciação '$\wedge$'). Não é difícil notar que uma expressão e sempre tem a forma

$$e = t_1 \oplus t_2 \oplus \cdots \oplus t_n, \qquad n \geq 1$$

onde os t_i representam os *termos* da expressão e, e o símbolo '$\oplus$' um dos operadores '$+$' ou '$-$'. Análogamente, um termo t pode ser descrito como

$$t = f_1 \otimes f_2 \otimes \cdots \otimes f_n, \qquad n \geq 1$$

onde os f_i representam os *fatores* do termo t, e '$\otimes$' representa um dos operadores '$*$' ou '$/$'. Finalmente, podemos descrever um fator f como sendo ou uma letra ou uma expressão entre parênteses:

$$f = x \qquad \text{ou} \qquad f = (\, e \,)$$

onde x denota qualquer letra. (Fica subentendido, apesar de não estar implícito nesta notação, que nos dois casos os operadores estão associados à esquerda.)

Deve-se notar que utilizamos definições mutuamente recursivas para definir o conjunto de expressões. Esta forma de descrevê-las leva naturalmente a um conjunto de rotinas recursivas para processá-las com alguma finalidade. No contexto de teoria de linguagens formais e de teoria de compilação, este tipo de descrição recebe nome de *gramática*, e a análise de expressões descritas por uma gramática é denominada *análise sintática*.

Uma aplicação possível desta idéia é a implementação da transformação de expressões infixas para pós-fixas através de três rotinas mutuamente recursivas, como indicado no Prog. 5.7. Um ponto importante a destacar é que cada um dos três procedimentos *Expressao*, *Termo* e *Fator* é responsável por processar a parte da cadeia de símbolos que corresponde à respectiva construção; além disto, cada chamada de um destes procedimentos avança a variável *pe* até o caractere que segue a construção processada. A deteção de erros não é completa e será deixada para o Exercício 5. Deve-se observar também a maneira natural como estes procedimentos seguem a descrição recursiva das expressões indicadas acima. Um outro ponto a notar é o instante em que os operadores são transferidos para a saída, produzindo o efeito de associação à esquerda dos quatro operadores.

Completemos agora o nosso programa com o operador de exponenciação '$\wedge$'. Podemos modificar a definição de fator através de

$$f = p_1 \wedge p_2 \wedge \cdots \wedge p_n, \qquad n \geq 1$$

$$p = x \qquad \text{ou} \qquad p = (\, e \,)$$

onde p denota os elementos *primários* de um fator.

Programa 5.7 Transformação infixa para pós-fixa (versão incompleta)

```
type
    Cadeia = array [1..ComprMax] of char;
    ...

procedure InPos(var e: Cadeia);
    var corr: char;  pe: 0..ComprMax;

    procedure Expressao; forward;

    procedure Termo; forward;

    procedure Fator;
    begin
    corr := e[pe];
    case corr of
        'a'..'z': begin
                  Sai(corr);
                  inc(pe)
                  end;
        '(': begin
                  inc(pe);
                  Expressao;
                  if e[pe]=')'
                      then inc(pe)
                  end
          else Erro
        end
    end; {Fator}

    procedure Termo;
        var op: char;  fim: boolean;
    begin
    Fator;
    fim := false;
    repeat
        corr := e[pe];
        if (corr='*') or (corr='/')
            then begin
                  op := corr;
                  inc(pe);
                  Fator;
                  Sai(op)
                  end
            else fim := true
    until fim
    end; {Termo}

    procedure Expressao;
        var op: char;  fim: boolean;
    begin
    Termo;
    fim := false;
    repeat
        corr := e[pe];
        if (corr='+') or (corr='-')
            then begin
                  op := corr;
                  inc(pe);
                  Termo;
                  Sai(op)
                  end
            else fim := true
    until fim
    end; {Expressao}

begin
pe := 1;
Expressao;
if e[pe]<>'#'
    then Erro
end; {InPos}
```

Aparentemente, a extensão do Prog. 5.7 para incluir a exponenciação seria muito simples, bastando para isto mudar o nome do procedimento *Fator* para *Primario*, e escrever um novo procedimento *Fator*, análogo a *Expressao* e *Termo*. Entretanto, o procedimento *Fator* escrito desta maneira trataria o operador '$\wedge$' como se fosse associado à esquerda o que não seria correto. Uma análise mais cuidadosa indica que seria necessário utilizar uma pilha local ao próprio procedimento para guardar todas as ocorrências do operador '$\wedge$' dentro do fator corrente, e desempilhá-los e copiar para a saída no fim da sua execução. Neste caso particular, a pilha não seria realmente necessária, pois bastaria anotar o número de ocorrências de '$\wedge$', já que temos um único operador neste nível de precedência.

Esta solução, apesar de satisfatória neste caso particular, não seria conveniente em geral, se houvesse uma variedade maior de operadores. Afinal, estamos utilizando a recursão justamente para evitar o uso de uma pilha explícita! Uma outra maneira de resolver o problema é modificar a definição de fator:

$$f = p \qquad \text{ou} \qquad f = p \wedge f$$

Note-se que agora a definição de f é *diretamente* recursiva, e indica explicitamente que o operador '$\wedge$' deve ser associado à direita. Segundo esta definição, podemos escrever uma nova versão do procedimento *Fator* indicada no Prog. 5.8. Note-se que o procedimento *Fator* não pode decidir imediatamente qual das duas alternativas, p ou $p \wedge f$ deve ser escolhida. Entretanto, as duas alternativas têm o mesmo início p o que permite que a decisão seja postergada até que seja processada a parte comum.

Programa 5.8 Procedimeto *Fator*

```
procedure Fator;
begin
Primario;
if e[pe]='∧'
    then begin
            inc(pe);
            Fator;
            Sai('∧')
            end
end; {Fator}
```

Uma idéia que pode surgir naturalmente neste momento é a transformação análoga das descrições das expressões e dos termos para:

$$e = t \qquad \text{ou} \qquad e = e \oplus t$$

$$t = f \qquad \text{ou} \qquad t = t \otimes f$$

Observe-se como foi feita a definição para indicar que neste caso os operadores são associados à esquerda. Entretanto, a transformação direta destas definições em procedimentos, análoga ao caso de *Fator*, não parece ser possível. Por exemplo, as duas alternativas para expressões, t e $e \oplus t$, têm partes iniciais diferentes e não está claro como tomar a decisão. Se o procedimento *Expressao* chamasse inicialmente o procedimento *Termo*, não estaria claro como proceder com outros termos se existissem; se ele chamasse inicialmente a si mesmo, entraria numa repetição infinita!

Esta dificuldade é devida ao fato das definições de expressões e de termos conterem *recursão esquerda*, isto é, o conceito que está sendo definido aparece imediatamente no início da definição. No caso do fator, temos a *recursão direita*, isto é, o conceito aparece no fim da definição. O método de análise sintática apresentado nesta seção pertence à categoria de métodos *descendentes* que não são aplicáveis em casos de recursão esquerda, sem que haja uma manipulação adicional das definições levando a formas como a indicada inicialmente nesta seção. Deve-se mencionar que existem outras categorias de análise, mais gerais, que não apresentam essas limitações.

5.4 Exercícios

1. Prove, por indução finita, que o número de somas $\mathrm{S}(n)$ realizado pela versão recursiva da função *fibo* do Prog. 5.2 é dado por:
$$\mathrm{S}(n) = \mathrm{fibo}(n+1) - 1$$

2. Mostre que

   (a) o número de movimentos impressos pelo procedimento *TorresDeHanoi* indicado no Prog. 5.3 é $2^n - 1$;

   (b) este é o número mínimo de movimentos para resolver o problema.

3. Aplique as idéias da Seção 5.2 para eliminar a recursão do procedimento *TorresDeHanoi* exibido no Prog. 5.3.

4. Generalize a idéia de eliminação da recursão aplicando-a ao caso de recursão mútua como indicado no exemplo das funções f e g da pág. 41. Sugestão: Utilize uma única pilha, mas inclua no código de retorno a informação sobre a origem da chamada.

5. Complete o Prog. 5.7 de maneira a incluir todos os operadores diádicos e monádicos e a detetar todas as condições de erro.

6. Considere a seguinte definição de todas as seqüências de parênteses balanceados (veja também a Seção 4.2):

$$\begin{aligned} b &= \epsilon \\ b &= (\,b\,)\,b \\ b &= [\,b\,]\,b \end{aligned}$$

   onde ϵ denota a seqüência vazia, e cada linha indica uma alternativa para a seqüência. Escreva um programa que usa recursão para reconhecer as cadeias que correspondem a esta definição.

7. Considere a seguinte definição de todas as seqüências de símbolos que denotam expressões em notação pré-fixa:

$$
\begin{aligned}
p &= x \\
p &= \& p \\
p &= \sim p \\
p &= + p\, p \\
p &= - p\, p \\
p &= * p\, p \\
p &= / p\, p \\
p &= \wedge p\, p
\end{aligned}
$$

onde x denota uma letra minúscula. Escreva um programa que usa recursão para transformar as expressões pré-fixas para pós-fixas.

8. Refaça o exercício anterior para a transformação de expressões pré-fixas para in-fixas. Sugestão: É mais fácil produzir expressões infixas com parênteses para cada operador, mesmo que sejam supérfluos. Por exemplo, '`+a*bc`' produziria $(a + (b * c))$.

9. Refaça o exercício anterior eliminando os parênteses supérfluos. Sugestão: Cada rotina recursiva deve receber como argumento uma informação sobre o operador que está sendo processado.

Capítulo 6

Árvores binárias

Neste capítulo, bem como no próximo, estudaremos árvores, uma estrutura de dados que aparece com freqüência em muitas aplicações. Intuitivamente, uma árvore representa uma organização da informação em que os dados estão numa relação hierárquica. Na realidade, o uso de árvores para representar relações precede de muito o aparecimento de computadores. Exemplos mais conhecidos são as *árvores genealógicas*[1] que indicam os antepassados de um indivíduo. Assim a Fig. 6.1 indica a árvore genealógica de um cão de raça de propriedade de um dos autores. Esta árvore poderia ser estendida (quase!) indefinidamente com a inclusão de mais gerações de antepassados, mas em geral não o faremos por falta de informação suficiente, ou por falta de interesse, ou por falta de espaço!

Note-se que, pela própria natureza da informação representada na Fig. 6.1, para cada indivíduo há dois antepassados imediatos. Note-se, também, que alguns indivíduos estão repetidos, refletindo um fato comum em árvores genealógicas. Como veremos mais adiante, se insistíssemos em manter uma única cópia do nome de cada indivíduo, com várias linhas apontando para esta cópia, a estrutura de dados deixaria de ser propriamente uma árvore. No caso desta figura, são as *ocorrências* dos nomes dos indivíduos, e não os indivíduos, que constituem a árvore.

A Fig. 6.2 apresenta um outro exemplo de árvore, desta vez relacionada com assuntos lingüísticos: a árvore de alguns idiomas que descendem de uma hipotética língua indo-européia. Neste caso, cada língua pode ter um número variável e, em princípio ilimitado, de descendentes diretos. Devido a limitações de espaço, bem como a diferenças de opinião entre os lingüistas, a informação contida na árvore está incompleta e inexata.

Existem muitos outros exemplos de uso de árvores, como a organização hierárquica de uma empresa, a classificação biológica das espécies, e assim por diante. Nas seções que seguem, examinaremos árvores binárias. O Capítulo 7 será dedicado a árvores gerais, como a da Fig. 6.2.

[1] Em inglês utiliza-se o termo *pedigree*.

Figura 6.1 Árvore genealógica

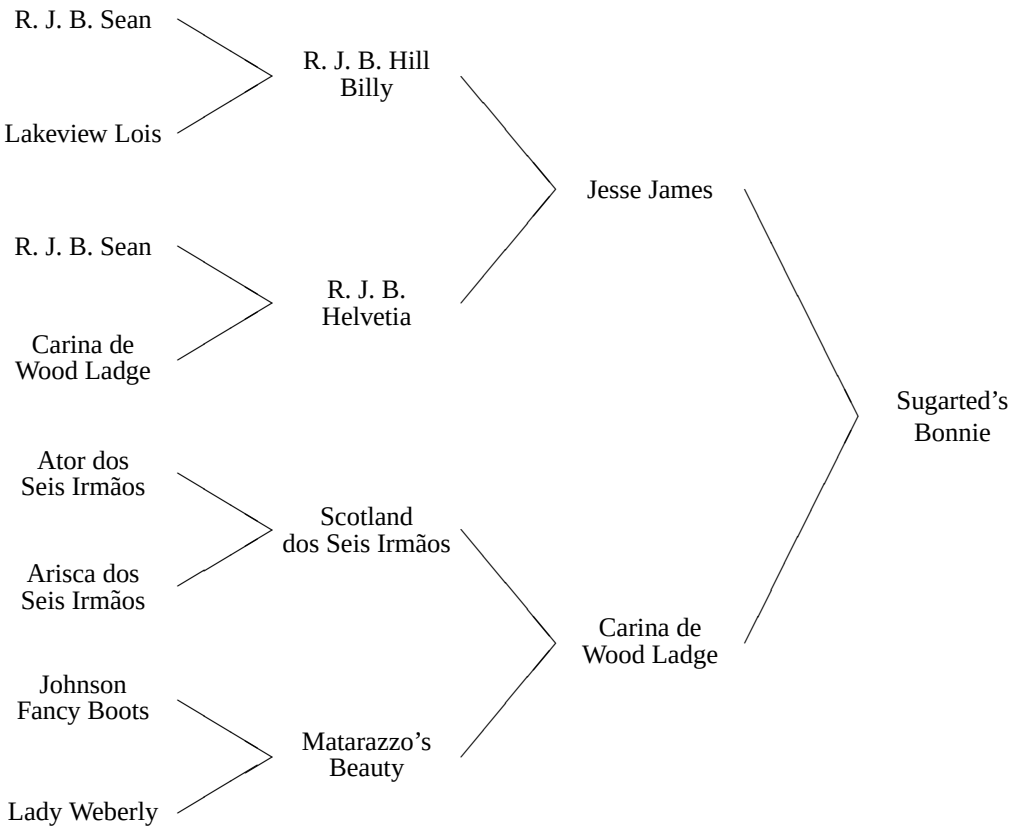
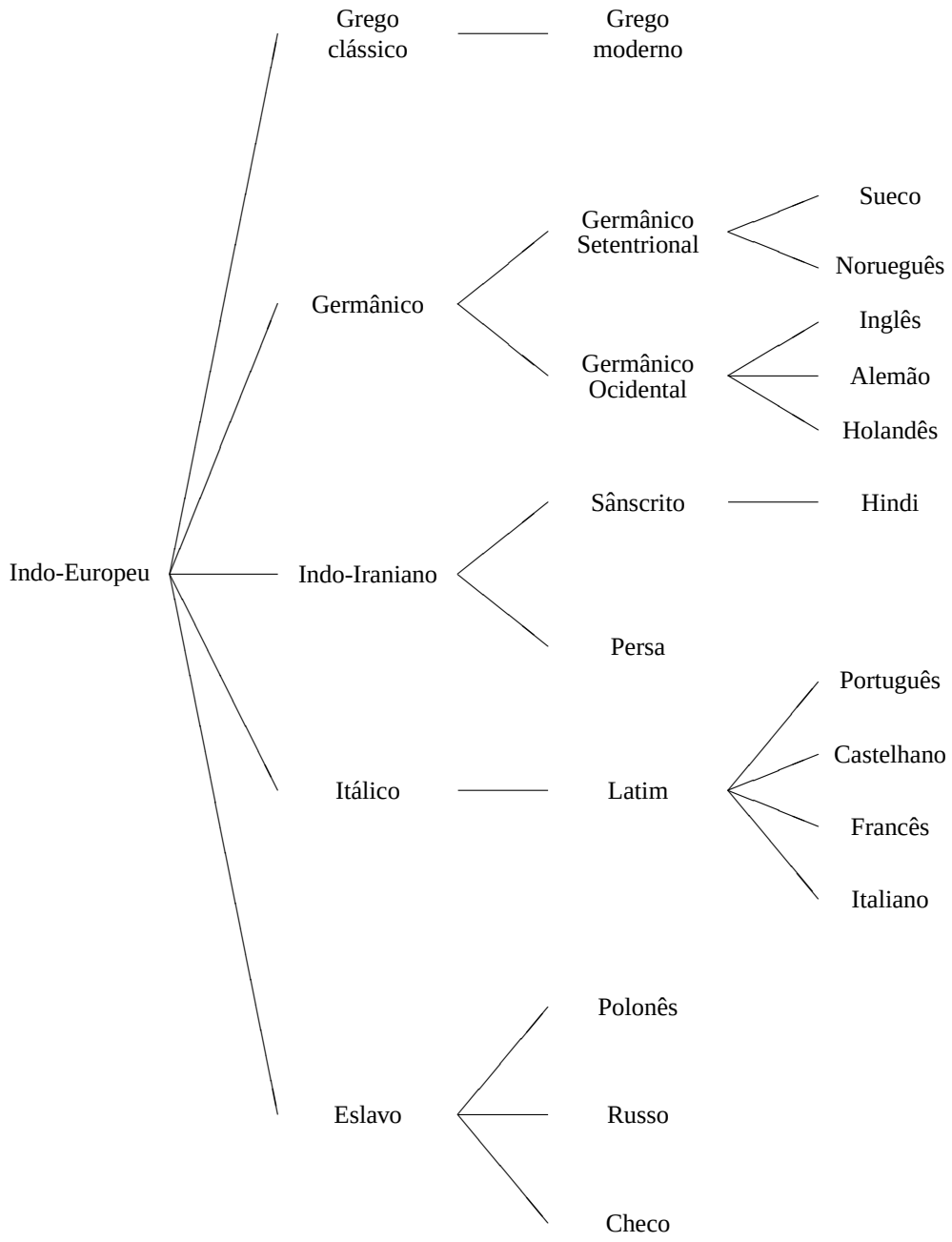


Figura 6.2 Árvore de descendentes



6.1 Definições, terminologia e propriedades

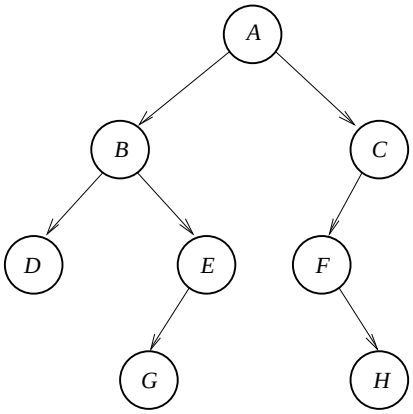
Uma das variantes mais importantes de árvores são as *árvores binárias*, das quais a Fig. 6.1 é um exemplo. De maneira mais precisa, adotaremos a seguinte definição:

> *Uma árvore binária T é um conjunto finito de objetos denominados* nós *(ou* vértices*), tais que*
>
> ou (a) *T é vazio,*
>
> ou (b) *T consiste de um nó distinto, chamado* raiz *de T, e de duas árvores binárias disjuntas T_e e T_d, chamadas* subárvores esquerda *e* direita *de T (ou da raiz de T), respectivamente.*

Note-se que a definição de árvores binárias é recursiva.[2]

A Fig. 6.3 mostra um outro exemplo de árvore binária, em que os nós foram rotulados com letras. Nas aplicações, cada nó será rotulado com alguma informação relevante ao problema em questão.

Figura 6.3 Exemplo de árvore binária



Existem algumas convenções e terminologia que têm sido adotadas quase universalmente com relação às árvores, se bem que nem sempre de maneira muito coerente.[3] Neste texto, indicaremos geralmente as árvores com as suas raízes no topo da figura, e com as subárvores "crescendo" para baixo. Como é natural, as subárvores esquerda e direita aparecem representadas do lado esquerdo e direito da raiz. Deve-se notar que as duas árvores binárias da Fig. 6.4 são distintas (veja, entretanto, as definições do Capítulo 7).

O *grau* de um nó de uma árvore binária é definido como sendo o número de subárvores não vazias deste nó (portanto zero, um ou dois). Assim, os nós A, F e H da Fig. 6.3 têm graus 2, 1 e 0, respectivamente. Os nós de grau igual a zero são chamados de *nós terminais* ou *folhas* da árvore. Os outros nós são os *nós internos* ou *nós não-terminais*. O conjunto de folhas da árvore binária da Fig. 6.3 é $\{D, G, H\}$.

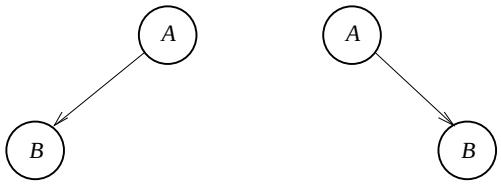
Ao descrever árvores binárias utilizam-se com freqüência termos inspirados por árvores que representam relações de parentesco. Assim, o nó que é a raiz de uma árvore é chamado de *pai*[4] dos nós que são as raízes das suas subárvores; na Fig. 6.3, A é o pai de B e C; E é o pai de G. Inversamente, falamos em nós que são

[2] Alguns autores preferem uma definição ligeiramente diferente que não admite árvores binárias vazias.

[3] O conceito de árvore adotado em Computação é distinto mas relacionado com o conceito adotado em Teoria dos Grafos.

[4] Como é praxe em português, as formas masculinas como *pai* ou *irmão* são utilizadas como traduções dos termos neutros ingleses *parent* ou *sibling*, sem nenhuma preferência por um dos sexos.

Figura 6.4 Árvores binárias distintas



filhos de um outro nó; ainda na Fig. 6.3, B e C são os filhos de A; G é o único filho de E. Os filhos de um mesmo nó são chamados de *irmãos*; assim, B e C são irmãos, como o são D e E.

As relações de *pai* e *filho* são estendidas de maneira natural para *antepassado* e *descendente*. Todos os nós da árvore da Fig. 6.3 (inclusive A) são considerados descendentes de A, e A é o antepassado de todos eles (inclusive de si mesmo). B é um dos antepassados de B e de G; H é um descendente de C. Obviamente, poderíamos definir, se fosse conveniente, noções como *primo*, *tio*, e assim por diante.

O *nível* de um nó numa árvore binária é um número natural obtido aplicando-se a seguinte definição recursiva:

(a) *a raiz da árvore binária tem o nível 1;*

(b) *se um nó tem o nível n, então os seus filhos têm o nível $n + 1$.*

A *altura* (ou a *profundidade*) de uma árvore binária é o nível máximo atribuído a um nó da árvore. Na Fig. 6.3, os nós A, C, E e H têm os níveis 1, 2, 3 e 4, respectivamente; a altura da árvore é 4. A altura de uma árvore binária vazia é igual a zero.[5]

Existem várias propriedades quantitativas de árvores binárias que são importantes para as suas aplicações. Citaremos apenas algumas, deixando as suas demonstrações para os exercícios. Quase todas as demonstrações são bastante simples, utilizando em geral a indução matemática.

1. Uma árvore binária com n nós utiliza $n - 1$ apontadores não nulos (utilizando a representação como a da Fig. 6.3).

2. O número máximo de nós de uma árvore binária de altura h é $2^h - 1$.

3. As alturas máxima e mínima de uma árvore binária de n nós são dadas por n e $\lceil \log_2(n+1) \rceil$, respectivamente.[6]

4. Sejam n_0 e n_2 os números de nós com zero e dois filhos, respectivamente, numa árvore binária. Então $n_0 = n_2 + 1$.

5. O número b_n de árvores binárias distintas com n nós é dado por $b_n = \frac{1}{n+1} \begin{pmatrix} 2n \\ n \end{pmatrix}$. Pode-se mostrar que, para valores crescentes de n, o valor aproximado é dado por $b_n = 4^n / n\sqrt{\pi n}$.[7]

[5] Alguns autores associam à raiz o nível zero, e consideram a altura de uma árvore binária vazia como sendo indefinida.

[6] A função $\lceil r \rceil$ indica o *teto* de r, isto é, o mínimo inteiro maior ou igual a r.

[7] Uma demonstração desta propriedade pode ser encontrada, por exemplo, em Knuth [**?**].

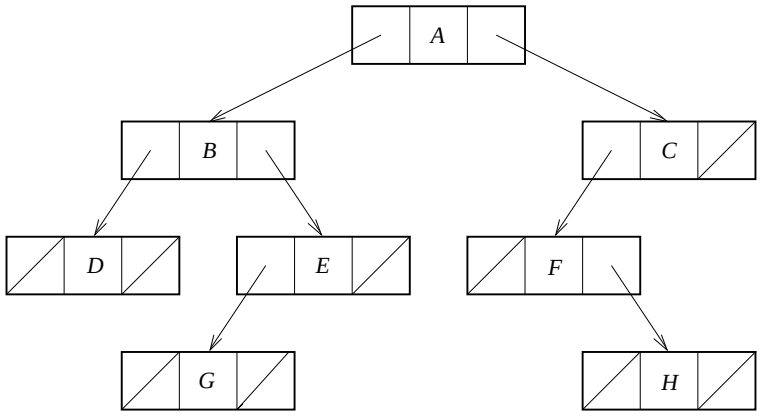
6.2 Implementação de árvores binárias

A própria maneira de representar as árvores binárias da Fig. 6.3 sugere uma implementação através de uma estrutura ligada utilizando apontadores. Dependendo da aplicação, podemos incluir ou não certas informações nesta representação. A Fig. 6.5 mostra a mesma árvore da Fig. 6.3, representada utilizando-se apontadores e tornando fácil a recuperação das informações *filho esquerdo* e *filho direito* para cada nó. O tipo de nó utilizado poderia ser indicado num programa através das declarações:

```
type
    ApReg = ↑Reg;
    Reg = record
        info: M;
        esq, dir: ApReg
    end;
    ArvBin = ApReg;
```

onde M é algum tipo que corresponde à informação a ser contida em cada nó. Suporemos também, na maior parte das aplicações, que é conhecido o apontador à raiz da árvore binária através do qual pode-se ter acesso à árvore inteira. Este apontador poderá estar contido, por exemplo, numa variável do programa, ou no campo apontador de um outro nó.

Figura 6.5 Representação de árvores binárias com campos *esq* e *dir*



Esta representação permite realizar com facilidade as operações de inserção ou remoção de nós, contanto que elas sejam bem definidas. No caso da inserção poderíamos utilizar, por exemplo, um procedimento como indicado no Prog. 6.1 onde foi convencionada uma inserção do novo nó como filho esquerdo do nó apontado por p; a subárvore esquerda de p tornou-se a subárvore esquerda do novo nó. Outros procedimentos poderiam ser definidos de maneira semelhante.

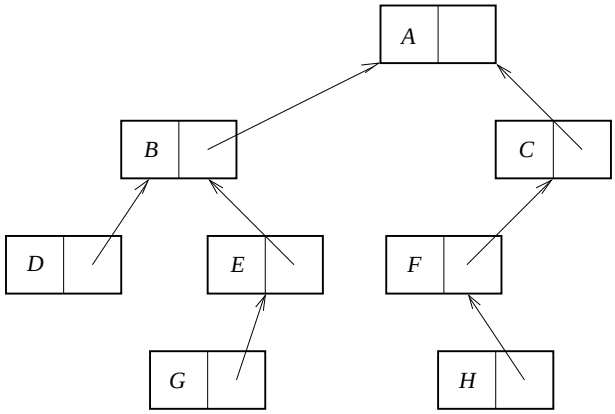
Conforme a aplicação, poderão ser utilizadas representações ligadas diferentes. A Fig. 6.6 mostra ainda a mesma árvore da Fig. 6.3, mas contendo apenas o apontador para a representação do nó *pai*. Note-se que um algoritmo que utilizasse esta representação teria necessidade de obter a informação sobre a localização de vários nós da árvore (ao menos de todas as folhas); um apontador para a raiz não seria suficiente neste

Programa 6.1 Procedimento de inserção à esquerda

```
procedure InsereEsq(p: ArvBin; x: M);
    var q: ApReg;
begin
new(q);
with q↑ do
    begin info := x; esq := p↑.esq; dir := nil end;
p↑.esq := q
end;
```

caso. Além disto, dado um nó qualquer, não seria possível decidir se ele é o filho esquerdo ou direito do seu pai. Caso esta informação fosse exigida pela aplicação, seria necessário colocar, por exemplo, uma marca adicional em cada nó.

Figura 6.6 Árvore binária com o campo *pai*.



Em certos casos, podem ser utilizadas representações mais eficientes para árvores binárias. Dizemos que uma árvore binária de altura h é *completa*[8] se ela contiver $2^h - 1$ nós (ou seja, o número máximo de nós – veja a Propriedade 2 da Seção 6.1). A Fig. 6.7 mostra uma árvore completa de altura 4. É fácil verificar que podemos associar a cada nó x de uma árvore binária completa um índice $f(x)$ tal que:

(a) se x é raiz, então $f(x) = 1$;

(b) se x_e e x_d são os filhos esquerdo e direito de x, então $f(x_e) = 2f(x)$ e $f(x_d) = 2f(x)+1$;

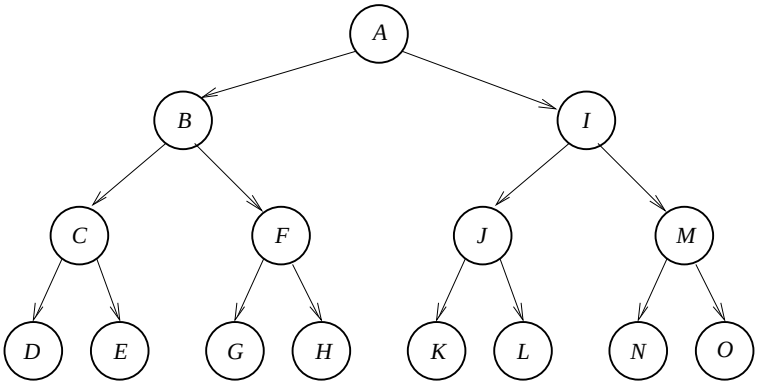
(c) se x não é raiz, e x_p é o pai de x, então $f(x_p) = \lfloor f(x)/2 \rfloor$.[9]

Nestas condições, podemos representar a árvore da Fig. 6.7 por um vetor de nós, como indicado abaixo:

[8] Em inglês *full binary tree*.

[9] A função $\lfloor r \rfloor$ indica o *piso* de r, isto é, o máximo inteiro menor ou igual a r.

Figura 6.7 Árvore binária completa de altura 4

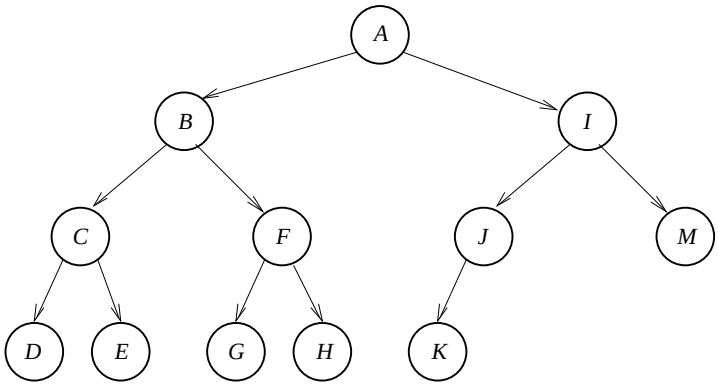


1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
A	B	I	C	F	J	M	D	E	G	H	K	L	N	O

É óbvio que, nesta representação, dado o índice de um nó qualquer, é simples e eficiente achar os seus filhos e o seu pai, se existirem. Além disto, com a eliminação dos apontadores pode haver economia considerável do espaço utilizado para armazenar a árvore. Esta representação pode ser estendida de maneira óbvia para árvores *quase completas* que são aquelas semelhantes às completas, em que está faltando um certo número de nós no final da ordem indicada pelo índice $f(x)$ definido acima. A Fig. 6.8 mostra um exemplo.

Existem várias aplicações em que árvores completas e quase completas com esta representação seqüencial podem ser utilizadas. Outras maneiras de representar árvores binárias serão vistas nas seções seguintes e nos exercícios.

Figura 6.8 Árvore binária quase completa de altura 4



6.3 Percurso de árvores binárias

Várias aplicações de árvores binárias exigem o exame sistemático das informações contidas em todos os nós da árvore. Dizemos que tais aplicações realizam um *percurso*[10] (ou uma *travessia*) da árvore, e que cada nó recebe uma ou mais *visitas*. A ordem em que os nós devem ser percorridos depende da aplicação. Existem, entretanto, duas classes de percursos que aparecem com mais freqüência, e que serão estudadas de maneira mais detalhada.

Uma classe de percursos de árvores binárias é constituída por percursos *em profundidade*, que seguem a própria definição recursiva destas árvores. Três desses percursos poderiam ser chamados de canônicos, e estão definidos a seguir, para o caso de árvores binárias não vazias:

- *Pré-ordem:*

  Visitar a raiz
  Percorrer a subárvore esquerda em pré-ordem
  Percorrer a subárvore direita em pré-ordem

- *Pós-ordem:*

  Percorrer a subárvore esquerda em pós-ordem
  Percorrer a subárvore direita em pós-ordem
  Visitar a raiz

- *Inordem (ou ordem simétrica):*

  Percorrer a subárvore esquerda em inordem
  Visitar a raiz
  Percorrer a subárvore direita em inordem

No caso da árvore da Fig. 6.3 obteríamos as seguintes ordens de percurso:

Pré-ordem: A,B,D,E,G,C,F,H

Pós-ordem: D,G,E,B,H,F,C,A

Inordem: D,B,G,E,A,F,H,C

Estas definições podem ser traduzidas diretamente em procedimentos recursivos que implementam os percursos. Supondo a implementação de árvores da Seção 6.2 com os campos *esq* e *dir*, podemos utilizar o procedimento apresentado no Prog. 6.2 para realizar o percurso em pré-ordem, utilizando recursão.

Os outros percursos em profundidade podem utilizar procedimentos análogos, trocando-se simplesmente a posição do comando de visita ao nó p. É fácil verificar que os procedimentos realizam um número de operações proporcional ao tamanho da árvore apontada por p (isto é, ao número de nós). A memória utilizada é proporcional ao número máximo de chamadas recursivas atingido em algum instante da execução. Este número é igual à altura da árvore, sendo no máximo igual ao seu número de nós. Mais adiante estudaremos maneiras mais eficientes de implementar estes percursos, diminuindo possivelmente os fatores de proporcionalidade mas sem afetar o comportamnento descrito.

Dizemos que o percurso de uma árvore binária é realizado *em largura* (ou *em nível*) se os seus nós são visitados em ordem dos seus níveis, isto é, em primeiro lugar o nível 1 (a raiz), depois o nível 2, depois o

[10] Em inglês *traversal*.

Programa 6.2 Procedimento de percurso recursivo em pré-ordem

```
procedure PreOrdem(p: ArvBin);
begin
if p≠ nil then
    with p↑ do
        begin
        Visita(p);
        PreOrdem(esq);
        PreOrdem(dir)
        end
end;
```

nível 3, e assim por diante. Dentro de cada nível, a ordem costuma ser da esquerda para a direita, quando representada como na Fig. 6.3. Assim, para aquela árvore, o percurso em largura produziria a seguinte ordem das visitas aos nós: A,B,C,D,E,F,G,H.

Uma maneira simples de implementar o algoritmo de percurso em largura é utilizar uma *fila* como definida na Seção 4.4, para "lembrar", em ordem certa, os nós que ainda não foram processados (inicialmente, a raiz). No procedimento do Prog. 6.3 supõe-se que os elementos da fila são do tipo *ApReg*, e que todos os procedimentos auxiliares foram definidos convenientemente.

Programa 6.3 Procedimento de percurso em largura utilizando uma fila

```
procedure PercursoEmLargura(p: ArvBin);
    var f: Fila; q: ApReg;
begin
InicializaFila(f);
InsereFila(f,p);
repeat
    Remove(f,q);
    if q≠nil then with q↑ do
        begin
        Visita(q);
        InsereFila(f,esq);
        InsereFila(f,dir)
        end
until FilaVazia(f)
end;
```

É fácil verificar que o tempo de execução do procedimento *PercursoEmLargura* é basicamente proporcional ao número de nós existentes na árvore. É interessante notar também que, no caso de árvores binárias

completas ou quase completas, com sua representação seqüencial da seção anterior, o algoritmo de percurso em largura reduz-se a uma simples enumeração seqüencial dos elementos do vetor que representa a árvore!

Os percursos em profundidade definidos nesta seção sugerem algumas outras representações lineares de árvores que podem ser muito úteis em algumas aplicações, como por exemplo a representação de árvores em arquivos. Note-se que, dado o resultado de um dos percursos da árvore, não é possível reconstruí-la sem alguma informação adicional. Pode-se mostrar, entretanto (veja os Exercícios 5 e 6), que esta reconstrução é possível se forem conhecidos os resultados de se percorrer a árvore em pré-ordem e inordem (ou em pós-ordem e inordem). Podemos descrever, assim, uma árvore binária fornecendo duas ordens de percurso. Uma outra maneira de descrever uma árvore consiste em fornecer apenas uma das ordens de percurso com alguma informação adicional. Por exemplo, podemos indicar, para cada nó, quais são as suas subárvores não vazias. Adotando a notação X_{ij}, com valores de i (ou j) iguais a *zero* se a subárvore esquerda (ou direita) for vazia, e iguais a *um* caso contrário, obtém-se para a árvore da Fig. 6.3, no caso da pré-ordem, a representação:

$$A_{11}B_{11}D_{00}E_{10}G_{00}C_{10}F_{01}H_{00}$$

Ao invés de utilizar os índices, podemos usar uma descrição parentética, na qual (no caso da inordem, por exemplo) cada nó é descrito por um abre-parêntese, a descrição da sua subárvore esquerda, o rótulo do nó, a descrição da subárvore direita, e por um fecha-parêntese; uma árvore vazia fica descrita por $()$. Novamente, no caso do mesmo exemplo de árvore binária, obteríamos a descrição:

$$((((()D())B(((()G())E()))A(((()F(()H()))C()))$$

Este tipo de descrição parentética pode ser obtido facilmente com percurso em profundidade da árvore.

Os procedimentos de percurso em profundidade descritos acima são muito óbvios e bastante eficientes, pelo menos em princípio, pois os seus tempos de execução e a memória utilizada no pior caso são essencialmente proporcionais ao tamanho da árvore. Mesmo assim, podem-se aproveitar as particularidades dos procedimentos e a estrutura de dados utilizada para introduzir várias melhorias, às vezes de maneira muito engenhosa.

Em primeiro lugar, poderíamos aplicar aos procedimentos de percurso em profundidade as técnicas vistas na Seção 5.2 para eliminar o uso da recursão, substituindo-a por uma pilha explícita. Mesmo se a transformação for aplicada sem levar em consideração as características de cada procedimento, poderá haver uma economia apreciável no tempo de execução e na memória máxima utilizada. O tempo seria economizado diminuindo-se o número de operações necessárias para processar cada nó, que na formulação anterior incluía chamadas recursivas do procedimento, que podem ser operações relativamente caras em algumas implementações. As chamadas das rotinas que manipulam a pilha explícita poderiam ser implementadas de maneira mais eficiente, ou simplesmente substituídas por comandos correspondentes.[11] Além disso, conforme foi visto na Seção 1.3, para cada chamada recursiva, o sistema cria na sua pilha uma nova área de dados locais para um procedimento, que é o seu *registro de ativação*. Esta área é normalmente muito maior do que o espaço que será utilizado para cada nó, com uma pilha explícita. Como resultado, teríamos ainda procedimentos com tempo e espaço lineares, mas com coeficientes de proporcionalidade significativamente menores. Como exemplo deste tipo de transformação, mostramos no Prog. 6.4 o percurso em pré-ordem. Supusemos neste caso que os elementos da pilha são especificados pelas declarações abaixo. A adaptação deste procedimento para a inordem e a pós-ordem é muito simples (veja o Exercício 16).

[11] Esta substituição é denominada às vezes *expansão em linha*.

```
    type
        Chamadas   = (chamada1, chamada2);
        ElemPilha =  record
            info: ApReg;
            chamada: Chamadas
        end;
```

Programa 6.4 Pré-ordem com pilha explícita obtida por transformação

```
procedure PreOrdem(p: ArvBin);
    type
        Acoes = (entrada,retorno,saida);
    var
        f: Pilha; q: ApReg; ch: Chamadas;
        acao: Acoes;
begin
    InicializaPilha(f); acao := entrada;
    repeat
        case acao of
            entrada: if p≠nil
                        then begin
                                Visita(p); Empilha(f,p,chamada1);
                                p := p↑.esq
                                end
                        else  acao := retorno;
            retorno: if PilhaVazia(f)
                        then acao := saida
                        else  begin
                                Desempilha(f,p,ch);
                                case ch of
                                    chamada1: begin
                                                Empilha(f,p,chamada2);
                                                p := p↑.dir;
                                                acao := entrada
                                                end;
                                    chamada2: { nada }
                                end
                                end
    until acao=saida
end;
```

A versão do procedimento mostrada no Prog. 6.4 foi obtida de maneira quase “mecânica”, sem levar em consideração a natureza do procedimento recursivo em questão nem a estrutura que ele está manipulando. É

interessante notar que no caso da pré-ordem podemos obter uma outra versão muito simples do procedimento através de uma pequena transformação do procedimento *PercursoEmLargura*: basta substituir a fila por uma pilha, e inverter a ordem de inserção dos elementos, como indicado no Prog. 6.5.

Programa 6.5 Pré-ordem com pilha explícita análoga ao percurso em largura

```
procedure PreOrdem(p: ArvBin);
    var f: Pilha;
begin
InicializaPilha(f);
Empilha(f,p);
repeat
    Desempilha(f,p);
    if p≠nil then with p↑ do
        begin
        Visita(p);
        Empilha(f,dir)
        Empilha(f,esq);
        end
until PilhaVazia(f)
end;
```

Uma análise destas versões mostra que há muitos empilhamentos seguidos imediatamente de desempilhamentos, além do empilhamento de apontadores nulos. A versão do Prog. 6.6, um pouco mais elaborada, elimina esta ineficiência.

No caso dos outros percursos em profundidade, também podemos aproveitar as suas peculiaridades para diminuir mais ainda os coeficientes de proporcionalidade. Por exemplo, no caso da pós-ordem é necessário distinguir quando o algoritmo está voltando depois de percorrer a subárvore esquerda ou direita do nó corrente. Esta distinção pode ser feita comparando-se o apontador a uma destas subárvores com o apontador ao último nó visitado, como mostrado no Prog. 6.7.

Programa 6.6 Pré-ordem mais eficiente com pilha explícita

```
procedure PreOrdem(p: ArvBin);
    var f: Pilha; fim: boolean;
begin
InicializaPilha(f); fim := false;
repeat
    if p≠nil
        then with p↑ do
            begin
            Visita(p);
            if dir≠nil then Empilha(f,dir);
            p := esq
            end
        else if PilhaVazia(f)
                then fim := true
                else Desempilha(f,p)
until fim
end;
```

Programa 6.7 Pós-ordem com pilha explícita

```
procedure PosOrdem(p: ArvBin);
    var f: Pilha; ultimo: ApReg; sobe: boolean;
begin
InicializaPilha(f);
repeat
    while p≠nil do
        begin
        Empilha(f,p); p := p↑.esq
        end;
    sobe := true; ultimo := nil;
    while sobe and (not PilhaVazia(f)) do
        begin
        Desempilha(f,p);
        if p↑.dir≠ultimo
            then begin
                    Empilha(f,p); p := p↑.dir; sobe := false
                    end
            else begin
                    Visita(p); ultimo := p
                    end
        end
until PilhaVazia(f)
end;
```

Uma outra idéia interessante pode ser aplicada para reduzir o espaço utilizado por algoritmos de percurso em profundidade. Ao invés de manter a pilha como uma estrutura de dados separada, pode-se embuti-la dentro da própria árvore, a um custo que em certos casos pode ser desprezível. O algoritmo do Prog. 6.8, conhecido com o nome de Deutsch, Schorr e Waite (veja por exemplo Knuth [**?**]), implementa esta idéia ao inverter os apontadores que estão no caminho desde a raiz da árvore até o nó corrente. O método exige, entretanto, a existência nos nós de um campo adicional para distinguir entre as possibilidades de inversão do apontador esquerdo ou direito. Esta marca é implementada possivelmente aproveitando-se algum espaço não utilizado do próprio nó (um *bit* é suficiente). O Exercício 18 sugere uma outra solução em que as marcas formam uma pilha externa à árvore, mas ocupam um espaço bem menor do que a pilha de apontadores. Na versão do Prog. 6.8 indicamos de maneira óbvia onde devem ser colocados os comandos de visita para a obtenção dos três percursos em profundidade. Os tipos foram redefinidos para incluir o campo *marca*.

Deve-se notar como, durante o percurso, a variável t aponta para uma lista ligada de nós que constituem a pilha (as ligações entre os elementos da lista usam os campos *esq* ou *dir* conforme a *marca*); a variável p aponta sempre para o nó corrente.

Uma observação importante sobre o algoritmo de Deutsch, Schorr e Waite é que ele modifica temporariamente a estrutura de dados percorrida, mas deixa-a finalmente intacta, exceto possivelmente pelos campos de marca. Este fato pode tornar impraticável o uso do algoritmo em certas situações, como por exemplo

numa base de dados compartilhada por vários processos concorrentes. Por outro lado, caso seja possível incluir as marcas nos nós, o algoritmo realiza o percurso utilizando memória auxiliar constante (apenas as variáveis do programa).

Programa 6.8 Algoritmo de Deutsch, Schorr e Waite

```
type
    Marcas = (MarcaEsq, MarcaDir);
    ApReg = ↑Reg;
    Reg =
        record
            info: M;
            marca: Marcas;
            esq, dir: ApReg
        end;
    ArvBin = ApReg;

procedure DeutschSchorrWaite(p: ArvBin);
  var t,q: ApReg;  sobe: boolean;
begin
t := nil;
repeat
    while p≠nil do with p↑ do { desce 'a esquerda }
        begin
        PreVisita(p);  marca := MarcaEsq;
        q := esq;  esq := t;
        t :=p;  p := q
        end;
    sobe := true;
    while sobe and (t≠nil) do with t↑ do
        begin
        case marca of
        MarcaEsq: begin  { desce 'a direita }
                        InVisita(t);  sobe := false;
                        marca := MarcaDir;  q := p; p := dir;
                        dir := esq; esq := q
                        end;
        MarcaDir: begin { sobe }
                        PosVisita(t);
                        q := dir;  dir := p;
                        p := t;  t := q
                        end
        end
until t=nil
end;
```

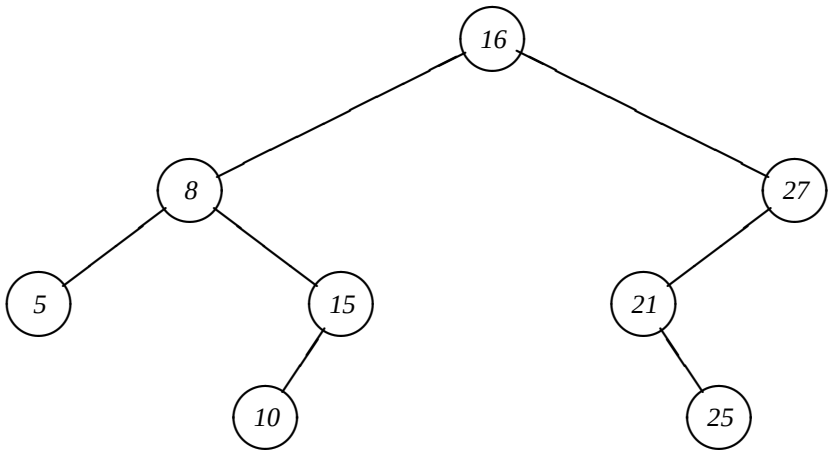
6.4 Exemplo de aplicação: árvores de busca

Árvores binárias têm muitas aplicações, e algumas delas serão vistas mais adiante. Nesta seção, mostraremos uma aplicação simples que será explorada de maneira mais completa no Capítulo 8.

Suponhamos que, numa árvore binária, os campos de informação contêm objetos que podem ser classificados de acordo com alguma ordem total. Como exemplos poderíamos tomar números inteiros (ordenados de maneira óbvia) ou cadeias de caracteres (em ordem alfabética ou lexicográfica[12]). Dizemos, então, que a árvore é uma *árvore binária de busca* se em cada nó da árvore vale a propriedade de que, usando a ordem especificada, o valor contido no seu campo de informação *segue* todos os valores contidos nos nós da subárvore esquerda e *precede* todos os valores contidos nos nós da subárvore direita.

A Fig. 6.9 mostra um exemplo de árvore binária de busca que contém os números inteiros $5, 8, 10, 15, 16, 21, 25, 27$; na Fig. 6.10 temos um exemplo de árvore binária que contém os nomes abreviados dos meses, em ordem alfabética.

Figura 6.9 Árvore binária de busca contendo números

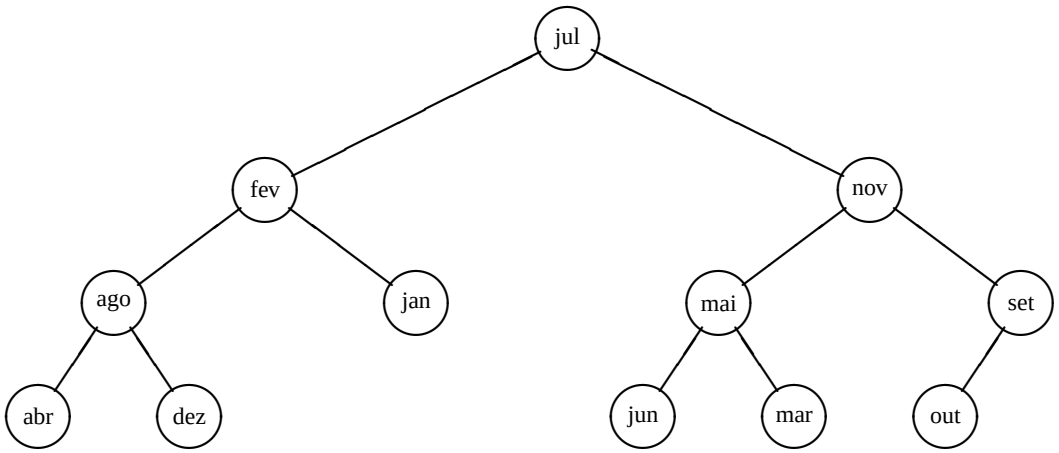


Árvores binárias de busca podem ser utilizadas para manter de maneira eficiente tabelas de elementos. É muito simples, por exemplo, a verificação da presença ou não de um elemento na tabela. O Prog. 6.9 mostra o exemplo de um procedimento que verifica se um elemento x está contido na árvore de busca apontada por p, e caso contrário o insere; o valor final da variável *pertence* indica se o elemento já pertencia à árvore. Supusemos neste exemplo a implementação da Seção 6.2. Utilizamos neste caso a formulação recursiva que torna a elaboração mais simples. O uso da recursão, neste caso, não é estritamente necessário, pois a busca pode ser realizada com uma simples malha repetitiva (veja o Exercício 23).

Deve-se notar que o número de operações executado pelo procedimento *BuscaInsere* é proporcional, no pior caso, à profundidade da árvore. De acordo com as propriedades da Seção 6.1, a profundidade de uma árvore binária com n nós está entre $\lceil \log_2(n+1) \rceil$ e n. Dizemos que uma árvore binária está balanceada

[12] *Ordem lexicográfica* é uma generalização da ordem alfabética que inclui também outros caracteres que não letras.

Figura 6.10 Árvore binária de busca contendo nomes dos meses



se ela tem profundidade mínima, ou seja $\lceil \log_2(n+1) \rceil$. Na prática, procura-se manter as árvores de busca *balanceadas* de tal maneira que suas profundidades sejam próximas da mínima (veja Seção 8.1). Pode-se mostrar também que árvores de busca obtidas por inserção em ordem aleatória têm profundidades médias bastante próximas desta profundidade mínima.

A remoção de um elemento de uma árvore binária de busca é ligeiramente mais complicada. Caso o elemento esteja num nó que é uma folha da árvore, basta substituí-lo por uma subárvore vazia; no caso de um nó com apenas uma subárvore vazia, a remoção ainda é simples, substituindo-se o apontador a este nó por um apontador à sua única subárvore. Finalmente, quando o elemento a ser removido está num nó de grau dois e, portanto, com duas subárvores não vazias, podemos substituir o valor a ser removido pelo maior valor que o precede (ou o menor valor que o segue) e remover o outro nó. É fácil verificar que este outro nó tem o grau zero ou um, recaindo-se no caso anterior (veja também o Exercício 24).

Programa 6.9 Busca e inserção numa árvore binária de busca

```
procedure BuscaInsere(var p: ArvBin; x: T; var pertence: boolean);
begin
if p=nil
   then begin
          pertence := false; new(p);
          with p↑ do
             begin
             esq := nil;  dir := nil;  info := x
             end
          end
   else with p↑ do
          if x=info
             then pertence := true
          else if x<info
             then BuscaInsere(esq,x,pertence)
             else  BuscaInsere(dir,x,pertence)
end;
```

6.5 Exercicios

1. Demonstre as Propriedades 1 a 4 da Seção 6.1 (pág. 55).

2. Dizemos que uma árvore é *estritamente binária* se todos os seus nós têm graus 0 ou 2. Qual é a relação entre o número total n de nós de uma árvore binária estrita e os números n_0 e n_2 (veja a Propriedade 4 da Seção 6.1).

3. Caracterize as árvores binárias para as quais a pré-ordem e a inordem produzem os mesmos resultados.

4. Caracterize as árvores binárias para as quais a pré-ordem e o percurso em largura produzem os mesmos resultados.

5. Mostre que nenhuma das três ordens de percurso em profundidade determina sozinha a árvore binária da qual ela resulta. Mostre que a pré-ordem junto com a pós-ordem também não é suficiente. Sugestão: Exiba contra-exemplos para cada caso.

6. Mostre que a pré-ordem e a inordem (ou a pós-ordem e a inordem) são suficientes para determinar uma árvore binária. Sugestão: Descreva os algoritmos de reconstrução, mostrando que em cada passo há somente uma escolha.

7. Dada uma árvore binária T, define-se o *número de Strahler* $S(T)$ por:

   se T é vazia então $S(T) = 0$;
   se T não é vazia, e T_e e T_d são as duas subárvores de T, então:

$$S(T) = \begin{cases} \max\{S(T_e), S(T_d)\} & \text{se } S(T_e) \neq S(T_d) \\ S(T_e) + 1 & \text{caso contrário} \end{cases}$$

   (a) Escreva um procedimento que calcula os números de Strahler para todas as subárvores de uma árvore binária, armazenando-os nas suas respectivas raízes.

   (b) Enumere as árvores binárias com o número mínimo de nós, tal que o seu número de Strahler é 3?

   (c) Escreva um procedimento de percurso em profundidade de árvores binárias que, utilizando os números precalculados de Strahler, minimiza a altura máxima da pilha escolhendo convenientemente em cada passo a subárvore esquerda ou direita para ser percorrida em primeiro lugar.

8. Suponha que é dada uma coleção de nós para representar de maneira padrão uma árvore binária. Os campos *esq* destes nós formam uma lista ligada que representa a pré-ordem de uma árvore binária; os campos *dir* formam a lista (dos mesmos nós) que representa a inordem da mesma árvore. As duas listas são dadas por apontadores aos seus primeiros nós. Escreva um procedimento que reconstrói a árvore binária correspondente, sem gerar nós adicionais.

9. Escreva procedimentos para a obtenção de descrições lineares (veja pág. 61) de árvores binárias com representação padrão, de acordo com os vários percursos em profundidade.

Programa 6.10 Pré-ordem com pilha explícita

```
type
   ProxTarefa = (FazVisita, FazPercurso);
   RegPilha = record info: ApReg; tarefa: ProxTarefa end;

procedure PreOrdem(p: ArvBin);
   var f: Pilha; q: ApReg; t: ProxTarefa;
begin
InicializaPilha(f); Empilha(f,p,FazPercurso);
repeat
   Desempilha(f,q,t);
   case t of
     FazVisita:   Visita(q);
     FazPercurso: if q≠ nil
                        begin
                        Empilha(f,dir,FazPercurso);
                        Empilha(f,esq,FazPercurso);
                        Empilha(f,q,FazVisita)
                        end
   end
until PilhaVazia(f)
end;
```

10. Escreva procedimentos para reconstruir árvores binárias a partir das suas descrições lineares (veja página 61).

11. Escreva procedimentos para realizar os três percursos em profundidade de árvores binárias completas e quase completas, representadas seqüencialmente. Não use recursão nem estruturas de dados auxiliares.

12. Note que o procedimento *PreOrdem* do Prog. 6.2 executa chamadas recursivas mesmo para as subárvores vazias. Reescreva o procedimento para evitar estas chamadas. Discuta a economia de tempo e espaço obtida.

13. Uma outra maneira elegante de realizar o percurso em pré-ordem está indicada no Prog. 6.10, onde está incluída a declaração do tipo *RegPilha* da informação que deve estar guardada na pilha. Escreva procedimentos análogos para realizar os outros dois percursos em profundidade.

14. O que se pode afirmar sobre o tamanho máximo alcançado pela fila do procedimento *PercursoEmLargura* do Prog. 6.3?

15. Reescreva o procedimento *PercursoEmLargura* do Prog. 6.3 de maneira a eliminar a inserção de apontadores nulos na fila.

16. Escreva procedimentos para inordem e pós-ordem, análogos ao procedimento *PreOrdem* do Prog. 6.4.

Programa 6.11 Algoritmo de Lindstrom e Dwyer

```
procedure LindstromDwyer(p: ArvBin);
    var vazia,q,t: ApReg;
begin
new(vazia); t := vazia;
repeat
    if p=nil
        then begin q := p; p := t; t := q end
        else begin
                Visita(p);
                q := t; t := p; p := p↑.esq;
                t↑.esq := t↑.dir; t↑.dir := q
                end
until p=vazia;
dispose(vazia)
end;
```

17. Escreva um procedimento para inordem análogo ao procedimento *PreOrdem* do Prog. 6.6.

18. Implemente o algoritmo de Deutsch, Schorr e Waite do Prog. 6.8, utilizando em lugar das marcas em cada nó, uma pilha externa de marcas que indica os campos invertidos no caminho da raiz ao nó corrente.

19. Considere o procedimento do Prog. 6.11, proposto por Lindstrom [**?**] e Dwyer [**?**], para percorrer árvores binárias utilizando a representação ligada comum. Explique o resultado do percurso realizado pelo procedimento. É possível fazer com que o procedimento realize um dos três percursos canônicos em profundidade?

20. Considere uma representação de árvores binárias proposta por Siklóssy em [**?**] em que o campo *esq* de cada nó n contém o resultado da expressão

$$pai(n) \oplus filho\_esq(n)$$

e analogamente para o campo *dir*. Nestas expressões, $pai(n)$, $filho\_esq(n)$ e $filho\_dir(n)$ denotam os *endereços* dos respectivos nós, e $\oplus$ denota a operação de *ou-exclusivo* entre os dígitos na representação binária do endereço. Quando algum dos nós não existe, utiliza-se o valor nulo. Note-se que, para quaisquer valores x e y, valem $x \oplus y \oplus x = y$ e $x \oplus y \oplus y = x$. Suponha também que cada nó contém um campo de marca que indica se este nó é o filho esquerdo ou direito do seu pai. (Suponha que o nó raiz é o filho esquerdo do seu pai, que não existe). Escreva os procedimentos de percursos canônicos em profundidade de árvores representadas desta maneira. Os procedimentos não devem modificar a árvore durante o percurso, nem utilizar estruturas de dados auxiliares cujo tamanho possa depender da própria árvore.

21. Ache exemplos de ordens em que poderiam ter sido inseridos os elementos nas árvores binárias de busca das Figs. 6.9 e 6.10. Existe alguma maneira sistemática de achar estas ordens?

22. O exercício anterior apresenta várias soluções; ache uma caracterização de todas elas.

23. Reescreva o procedimento do Prog. 6.9 sem usar recursão nem pilha.

24. Escreva um procedimento que remove um elemento dado de uma árvore binária de busca; após a remoção a árvore deve manter a propriedade de ser de busca.

25. O conceito de árvores binárias pode ser generalizado[13] para múltiplas subárvores, adotando-se a definição: dado um inteiro positivo n, uma árvore n-ária T é um conjunto finito de objetos (*nós* ou *vértices*) tais que:

    ou (a) T é vazio,

    ou (b) T consiste de um nó distinto, chamado *raiz* de T, e de n árvores n-árias disjuntas $T_1, T_2, \ldots, T_n$, chamadas *primeira, segunda, . . ., n-ésima subárvores* de T (ou da raiz de T), respectivamente.

    Note que uma representação padrão de árvores n-árias poderia utilizar nós declarados por:

```
type
    ApReg = ↑Reg;
    Reg = record
        info: M;
        filhos: array[1..n] of ApReg
    end;
    ArvNaria = ApReg;
```

    Estenda, sempre que possível as noções e os algoritmos estudados neste capítulo para o caso de árvores n-árias.

[13] Um outro tipo de generalização será visto no capítulo seguinte.

Capítulo 7

Árvores gerais

O capítulo anterior foi dedicado a um estudo bastante completo de árvores binárias, uma estrutura que aparece com freqüência nas aplicações. Entretanto existem situações nas quais uma árvore binária não constitui uma descrição natural das relações entre os objetos a serem manipulados; um exemplo desta situação aparece na Fig. 6.2 referente à família de línguas indo-européias. Cada nó desta figura poderia ter um número variável e, em princípio, ilimitado de descendentes diretos. Outros exemplos são as árvores dos descendentes de um indivíduo, da organização hierárquica de uma empresa e da classificação biológica de espécies.

Nas seções deste capítulo estudaremos as definições, propriedades, representações eficientes e os algoritmos aplicáveis a essas árvores gerais,[1] fazendo referências freqüentes aos conceitos correspondentes relativos às árvores binárias.

7.1 Definições e terminologia

Uma *árvore geral* (ou simplesmente uma *árvore*) T é um conjunto finito e não vazio de objetos (*nós* ou *vértices*), tais que T consiste de um nó distinto, chamado de raiz de T, e de $m \geq 0$ árvores disjuntas $T_1, T_2, \ldots, T_m$, chamadas *subárvores* de T (ou da raiz de T).

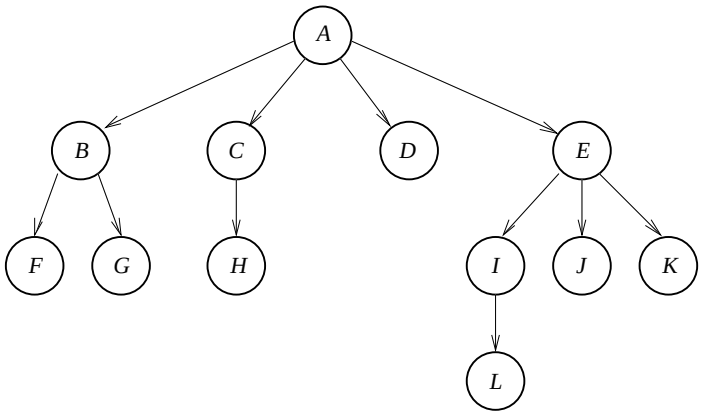
Note-se que esta definição não admite o conceito de árvore vazia; por outro lado, um nó pode não ter subárvores ($m = 0$). Está implícita nesta definição uma *ordem* das subárvores $T_1, T_2, \ldots, T_m$; alguns autores preferem usar neste caso o termo *árvores ordenadas*.

É interessante, neste contexto, introduzir mais um conceito: uma *floresta* (ou uma *arborescência*) é uma seqüência finita de árvores. Dada uma floresta F, podemos escrever, portanto, $F = (T_1, T_2, \ldots, T_m)$ onde $m \geq 0$ e cada T_i representa uma árvore da floresta F. Deve-se notar que a definição de florestas também é recursiva, pois uma árvore pode ser vista como a sua raiz juntamente com a floresta das suas subárvores. A Fig. 7.1 mostra um exemplo de árvore; retirando-se a raiz A, obtém-se uma floresta de quatro árvores, com as raízes B, C, D e E.

Os conceitos de *grau*, *folha*, *nó interno*, *pai*, *filho*, *irmão*, *antepassado*, *descendente*, *nível* e *altura* definidos na Seção 6.1 podem ser adaptados de maneira natural às árvores gerais e serão deixados para exercícios.

[1] Esta generalização deve ser contrastada com as árvores n-árias introduzidas no Exercício 25 do Cap. 6.

Figura 7.1 Exemplo de árvore



7.2 Representação de árvores

Por analogia com árvores binárias, poderíamos representar árvores gerais por meio de nós que incluíssem vetores de apontadores, prevendo algum número máximo *grauMax* de filhos:

```
type
    ApReg = ↑Reg;
    Reg = record
        info: M;
        grau: 0..grauMax;
        filhos: array[1..grauMax] of ApReg
    end;
    Arvore = ApReg;
```

onde o campo *grau* indica o número de filhos realmente existentes. Esta representação teria dois inconvenientes. Em primeiro lugar, é necessário prever um limite máximo para o grau de cada nó, o que pode ser difícil em algumas aplicações. Em segundo lugar, pode haver um grande desperdício de memória se o grau de cada nó for, em geral, muito menor do que esse máximo. Em alguns sistemas (inclusive algumas implementações de PASCAL), é possível obter-se o efeito de uma declaração como

```
type
    ApReg = ↑Reg;
    Reg = record
        info: M;
        grau: 0..grauMax;
        filhos: array[] of ApReg
    end;
    Arvore = ApReg;
```

onde o número de componentes do vetor *filhos* é indicado quando é alocada uma variável dinâmica do tipo *Reg*. Com esta facilidade ficariam resolvidos os dois problemas citados. Entretanto, tornam-se mais

Figura 7.2 Uma floresta com três árvores

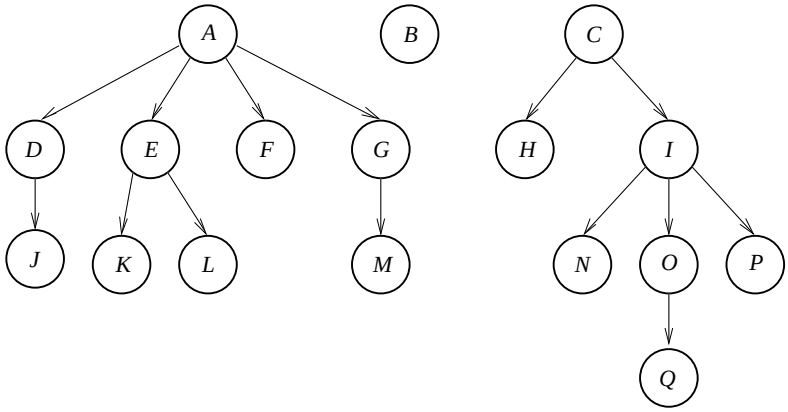
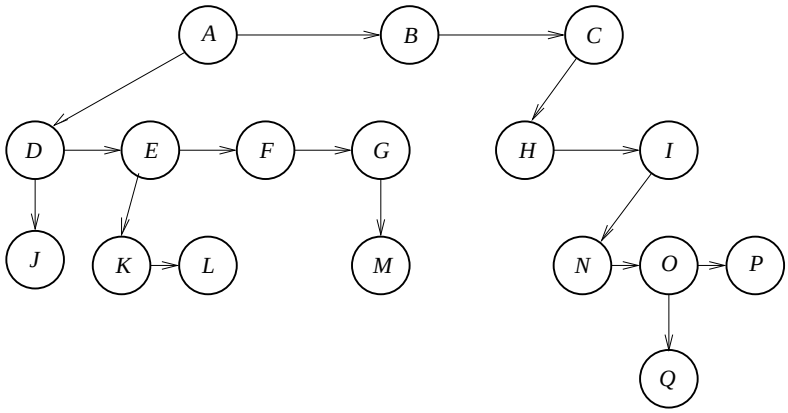


Figura 7.3 Representação binária de uma floresta com três árvores



complicadas as operações sobre árvores que possivelmente modificam o grau de um nó, como a inserção e a remoção.

Uma outra alternativa para representar árvores gerais é a utilização de árvores binárias para representar florestas de árvores. Considere a floresta de três árvores indicada na Fig. 7.2. Suponha agora que cada nó da foresta é ligado por uma aresta horizontal ao seu irmão seguinte, quando existir, e por uma aresta descendente ao seu primeiro filho, também quando este existir. Ao mesmo tempo são eliminadas as arestas originais. O resultado desta operação aparece na Fig. 7.3. (Note que as raízes A, B e C das três árvores são consideradas irmãs.)

Se considerarmos as arestas descendentes como sendo as que apontam para filhos esquerdos e as horizontais como as que apontam para filhos direitos, a Fig 7.3 pode ser interpretada como uma árvore binária, a partir da qual pode-se reconstruir facilmente a floresta original (veja o Exercício 2). A fim de enfatizar esta utilização de árvores binárias, poderíamos utilizar nós declarados por:

```
type
    ApReg = ↑Reg;
    Reg = record
        info: M;
        irmao, filhos: ApReg
    end;
    Arvore = ApReg;
```

A transformação indicada neste exemplo pode ser definida de maneira mais precisa, seguindo a própria estrutura recursiva dos objetos envolvidos. Seja $F = (T_1, T_2, \ldots, T_m)$ uma floresta com $m \geq 0$. A árvore binária $\mathbf{B}(F)$ que representa F é definida por:

(a) árvore binária vazia se F é uma floresta vazia ($m = 0$);

(b) árvore binária cuja raiz contém a mesma informação da raiz de T_1; cuja subárvore esquerda é dada por $\mathbf{B}((T_{11}, T_{12}, \ldots, T_{1m_1}))$ onde $(T_{11}, T_{12}, \ldots, T_{1m_1})$ é a floresta das subárvores de T_1; e cuja subárvore direita é dada por $\mathbf{B}((T_2, \ldots, T_m))$.

Pode-se definir de maneira análoga uma transformação inversa que associa a cada árvore binária T uma floresta $\mathbf{F}(T)$ por ela representada (veja o Exercício 2).

A representação binária assim definida resolve os problemas das outras representações sugeridas no início desta seção. Como veremos na seção seguinte, existe também uma correspondência natural entre percursos de florestas e os percursos em profundidade de árvores binárias. Note-se também que uma árvore geral pode ser tratada como uma floresta unitária, ou seja constituída de uma única árvore.

Deve-se notar, também, que a representação de florestas por meio de árvores binárias pode ser interpretada como uma lista de listas, semelhante às listas generalizadas a serem vistas no Cap. 9.

7.3 Percursos de florestas

Analogamente às árvores binárias, várias aplicações de árvores gerais e de florestas exigem um exame sistemático do conteúdo de todos os nós. Costuma-se definir, portanto, algumas ordens de percurso naturais. Seja $F = (T_1, T_2, \ldots, T_m)$ uma floresta não vazia, onde $F_1 = (T_{11}, T_{12}, \ldots, T_{1m_1})$ é a floresta das subárvores de T_1; teremos então:

- *Pré-ordem de florestas:*

  Visitar a raiz de T_1
  Percorrer a floresta F_1 em pré-ordem de florestas
  Percorrer a floresta $(T_2, \ldots, T_m)$ em pré-ordem de florestas

- *Pós-ordem de florestas:*

  Percorrer a floresta F_1 em pós-ordem de florestas
  Percorrer a floresta $(T_2, \ldots, T_m)$ em pós-ordem de florestas
  Visitar a raiz de T_1

- *Inordem de florestas:*

  Percorrer a floresta F_1 em inordem de florestas
  Visitar a raiz de T_1
  Percorrer a floresta $(T_2, \ldots, T_m)$ em inordem de florestas

No caso da Fig. 7.2 obteríamos os seguintes resultados:

Pré-ordem: A,D,J,E,K,L,F,G,M,B,C,H,I,N,O,Q,P

Pós-ordem: J,L,K,M,G,F,E,D,Q,P,O,N,I,H,C,B,A

Inordem: J,D,K,L,E,F,M,G,A,B,H,N,Q,O,P,I,C

Deve-se notar que os mesmos resultados seriam obtidos aplicando-se os percursos correspondentes para árvores binárias à árvore da Fig. 7.3 que representa a floresta da Fig. 7.2. Esta é uma propriedade geral que resulta das definições das transformações **B** e **F** (veja o Exercício 7).

Uma outra observação importante refere-se à interpretação intuitiva dos percursos de florestas. Em pré-ordem, a raiz de uma árvore geral é visitada imediatamente *antes* da visita a todos os seus descendentes; em inordem de florestas, uma raiz é visitada imediatamente *após* a visita a todos os seus descendentes; finalmente em pós-ordem de florestas, não há uma interpretação tão natural. Deve-se notar, portanto, que a *pré-ordem de florestas* tem uma interpretação intuitiva semelhante à *pré-ordem de árvores binárias*, enquanto que a *inordem de florestas* tem uma interpretação semelhante à *pós-ordem de árvores binárias*! Este fato leva alguns autores a chamarem a *inordem* de florestas de *pós-ordem*, e a não definirem a terceira ordem de percurso.

A implementação dos percursos depende da representação utilizada para florestas. No caso da representação binária, podemos utilizar todos os variantes estudados na Seção 6.3. No caso de outras representações, a implementação é bastante óbvia e será deixada para exercício.

O *percurso em largura* de florestas e de árvores gerais pode ser definido de maneira semelhante às árvores binárias. A sua implementação será deixada para exercícios.

7.4 Exercícios

1. Redefina os conceitos definidos na Seção 6.1 relativos a árvores binárias para as árvores gerais.

2. Defina a transformação $\mathbf{F}(T)$ mencionada na Seção 7.2.

3. Mostre que $\mathbf{F}(\mathbf{B}(F)) = F$ e $\mathbf{B}(\mathbf{F}(T)) = T$ para qualquer floresta F e qualquer árvore binária T.

4. Transforme para representação binária a árvore geral da Fig 6.2 referente às línguas indo-européias.

5. Por que, no caso de árvores gerais, foi conveniente introduzir-se o conceito de floresta?

6. Suponha que T é uma árvore binária completa de altura h. O que se pode afirmar sobre a forma geral da floresta $\mathbf{F}(T)$?

7. Demonstre que os três percursos em profundidade definidos para florestas são equivalentes aos percursos correspondentes de árvores binárias quando aplicados às suas representações binárias.

8. Escreva os procedimentos de percurso em profundidade para árvores gerais representadas por nós declarados na página 76.

9. Escreva os procedimentos de percurso em largura de árvores gerais representadas por nós declarados na página 76.

10. Escreva os procedimentos de percurso em largura de florestas com representação binária.

11. Proponha uma representação de florestas utilizando o conceito de listas generalizadas a ser visto no Cap. 9.

Capítulo 8

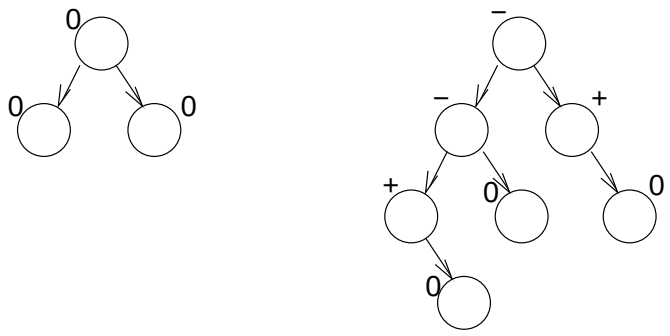
Árvores de busca

O objetivo deste Capítulo é o estudo de aplicações de árvores no armazenamento e recuperação eficiente de informação. Na primeira seção veremos como podemos manipular árvores de busca binárias, de modo a garantir um comportamento logarítmico. Na seção seguinte, veremos como estruturar informação no caso de utilização de dispositivos de memória de massa, tipicamente discos rígidos. Finalmente, veremos o uso de árvores binárias quase completas para resolver um problema de prioridades.

8.1 Árvores de altura balanceada

Já vimos na Seção 6.4 que árvores binárias podem ser usadas para guardar e recuperar informações, com número de operações proporcional à altura da árvore, ou seja variando aproximadamente, entre $\log_2 n$ e n. Veremos nesta seção, como esta manipulação pode ser realizada de maneira a garantir a altura da ordem de $\mathrm{O}(\log_2 n)$.[1]

Figura 8.1 Exemplos de árvores AVL

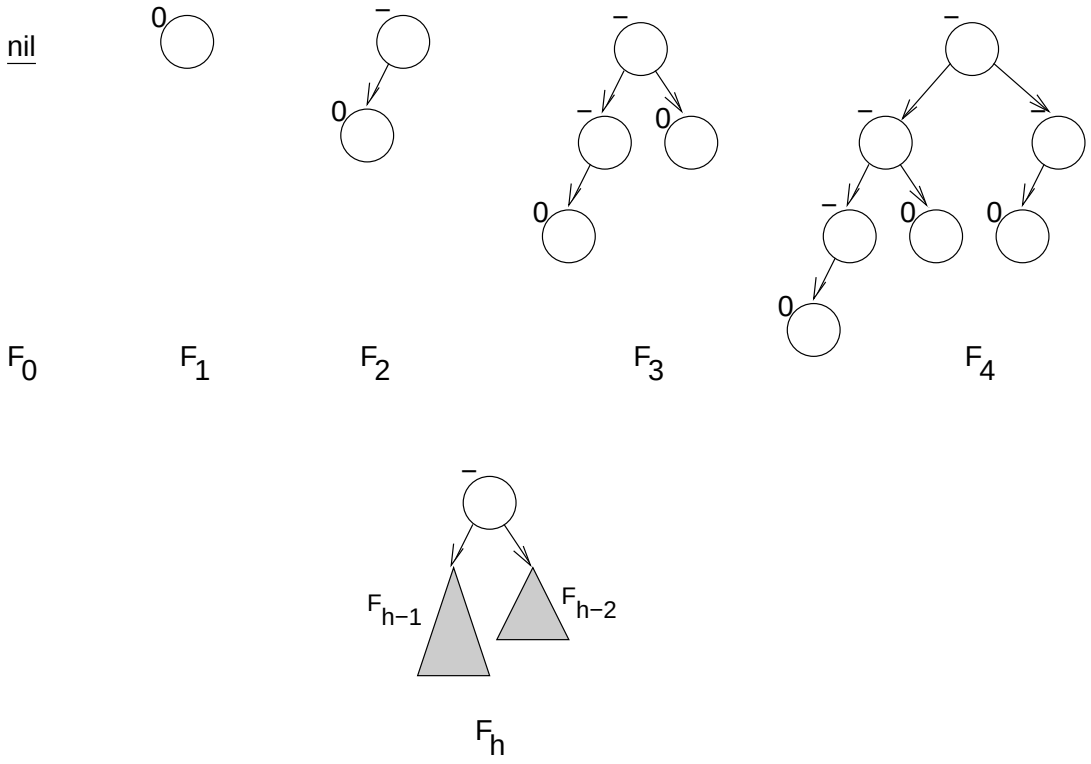


Dizemos que uma árvore binária de busca é *de altura balanceada* ou do tipo *AVL*[2] se para todos os seus nós vale a propriedade de que a diferença de alturas entre as subárvores esquerda e direita é no máximo 1,

[1] Escreveremos com frqüência $\mathrm{O}(\log n)$, pois $\log_2 n = \log_2 b \log_b n = \mathrm{O}(\log_b n)$ para qualquer base $b > 1$.

[2] Dos nomes dos seus inventores: G. M. Adel'son-Vel'skiĭ e E. M. Landis.

Figura 8.2 Árvores de Fibonacci



em valor absoluto. A diferença entre as alturas direita e esquerda é chamada *fator de balanceamento*. Nos exemplos da Fig. 8.1, os símbolos '$-$','0' e '$+$' denotam os fatores -1, 0 e $+1$.

A fim de calcular a altura máxima de uma árvore AVL, consideremos a família de árvores F_h com número mínimo de nós $N(h)$, fixada a altura h. Sem perda de generalidade, suporemos que a subárvore esquerda é sempre a mais alta. Esta família de árvores, chamadas de *árvores de Fibonacci*, está exibida na Fig. 8.2.

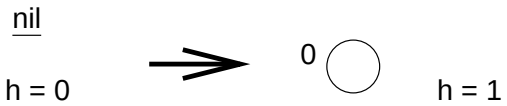
É fácil concluir que o número de nós de uma árvore de Fibonacci de altura h é dado por:

$$N(h) = N(h-1) + N(h-2) + 1, \quad h \geq 2$$

de onde resulta $N(h) = f_{h+2} - 1$, onde f_i é o i-ésimo número de Fibonacci. Lembrando que $f_i \approx ((1+\sqrt{(}5))/2)^i/\sqrt{(}5)$, demonstra-se que uma árvore de Fibonacci com n nós tem a altura aproximada de $1,44\log_2(n+2)$, que é o pior caso para árvores AVL.

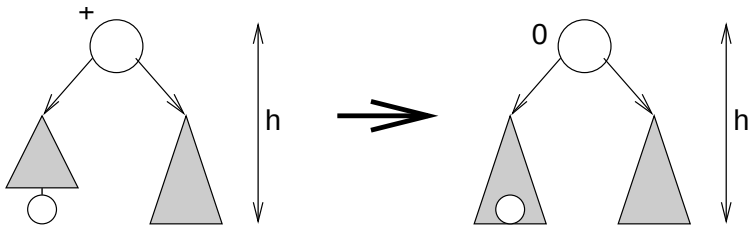
Explicaremos a seguir o algoritmo de busca e inserção numa árvore AVL, de maneira a preservar esta propriedade. Suporemos um algoritmo recursivo que pára tão logo a altura da árvore não muda, após a inserção:

Caso 1: Inserção em árvore vazia



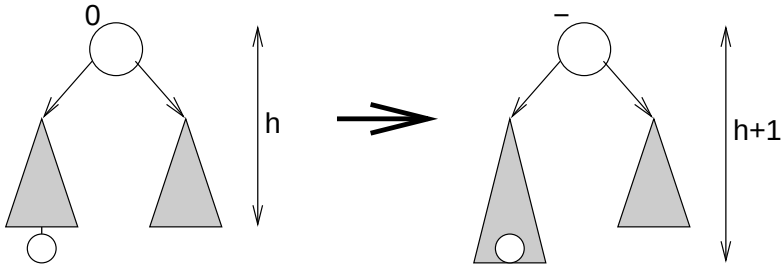
A altura final aumenta.

Caso 2: Inserção do lado mais baixo



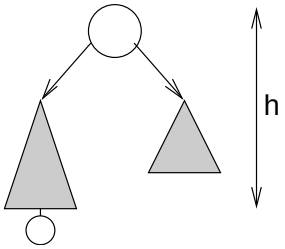
A altura final permanece inalterada (o processo de inserção pára).

Caso 3: Inserção quando as duas alturas são iguais



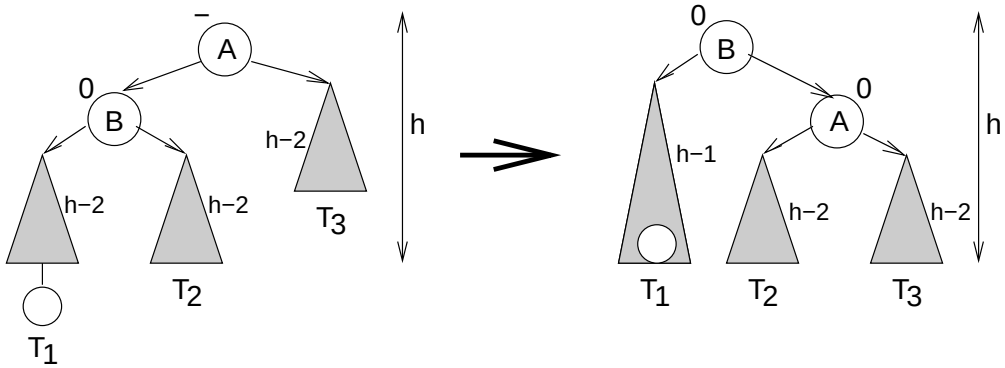
A altura final aumenta.

Caso 4: Inserção do lado mais alto



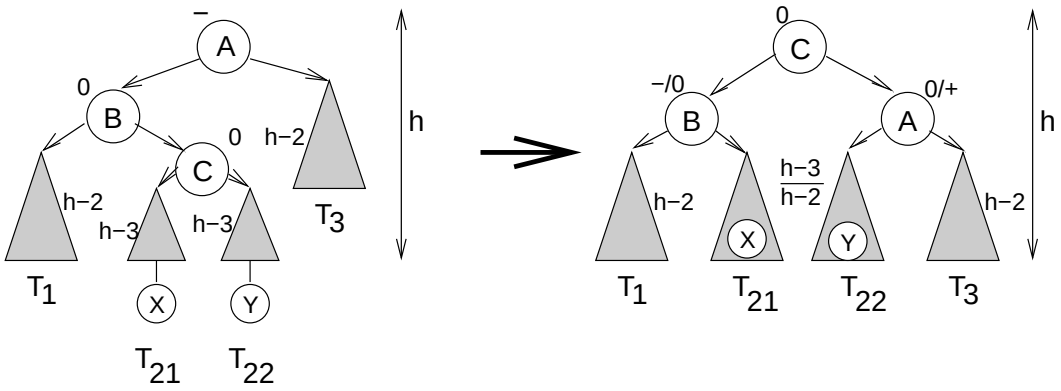
Deixa de ser AVL. Há dois subcasos:

(4a): Caminho LL (esquerdo, esquerdo) – rotação simples



A altura final permanece inalterada (o processo de inserção pára).

(4b): Caminho LR (esquerdo, direito) – rotação dupla



O nó inserido pode ser X ou Y. Quando h=2, as árvores T_1 e T_3 são vazias, e o nó inserido é o próprio nó C; neste caso as árvores T_{21} e T_{22} são vazias.

A altura final permanece inalterada (o processo de inserção pára)

Existem também os casos simétricos 2', 3' e 4' quando a inserção é realizada do lado direito da raiz. O Prog. 8.1 mostra o esboço da rotina recursiva.

É simples verificar que o número de operações necessárias para inserir um valor numa árvore AVL é proporcional à sua altura. De acordo com o que vimos, o valor da altura não passa de aproximadamente $1,44 \log_2 n$, o que garante a eficiência do algoritmo.

O algoritmo de remoção de um valor, preservando-se as propriedades de árvores AVL, é um pouco mais complexo e compõe-se de duas partes. Em primeiro lugar, a remoção de um nó arbitrário é substituída pela remoção de uma folha. Para tanto, existem três casos possíveis:

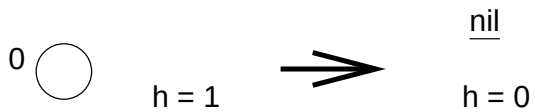
1. o nó tem grau zero e, portanto, já é uma folha;

2. o nó tem grau um – pela propriedade AVL, a sua única subárvore é necessariamente constituída por uma folha, cujo valor é copiado para o nó pai; o nó a ser eliminado é a folha da subárvore;

3. o nó tem grau dois – o seu valor é substitído pelo maior valor contido na sua subárvore esquerda (ou o menor valor contido na sua subárvore direita); o nó que continha o menor (ou maior) valor copiado tem necessariamente grau zero ou um, recaindo num dos casos anteriores.

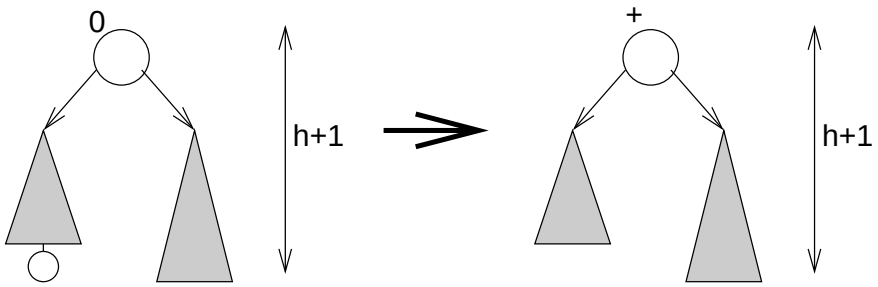
A segunda parte do algoritmo consiste, portanto, na remoção de uma folha. O processo é semelhante à inserção e ilustrado a seguir de maneira análoga. Supõe-se, em cada caso, que o algoritmo retorna da remoção de um nó numa subárvore esquerda, com diminuição de altura, e é analisado o efeito para a raiz corrente. Se houver novamente diminuição de altura, o algoritmo continua retornando pelo caminho de descida; caso contrário, o algoritmo pára.

Caso 1: Remoção de uma folha



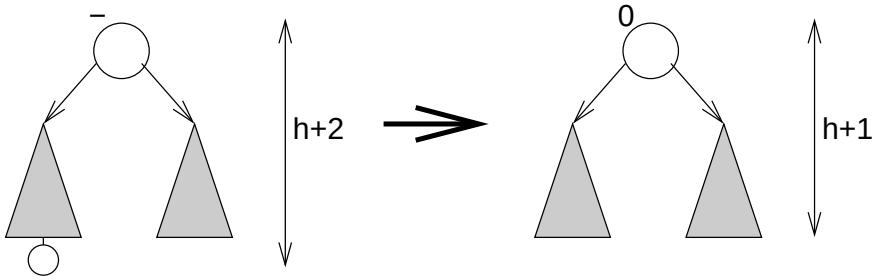
A altura diminui (o processo continua).

Caso 2: Alturas iguais



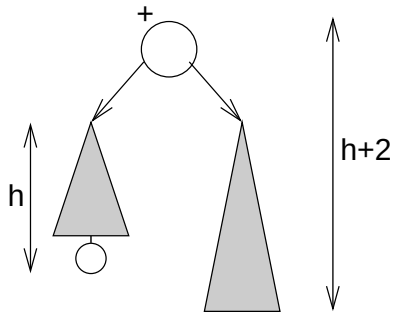
A altura permanece inalterada (o processo pára).

Caso 3: Remoção do lado mais alto



A altura diminui (o processo continua).

Caso 4: Remoção do lado mais baixo

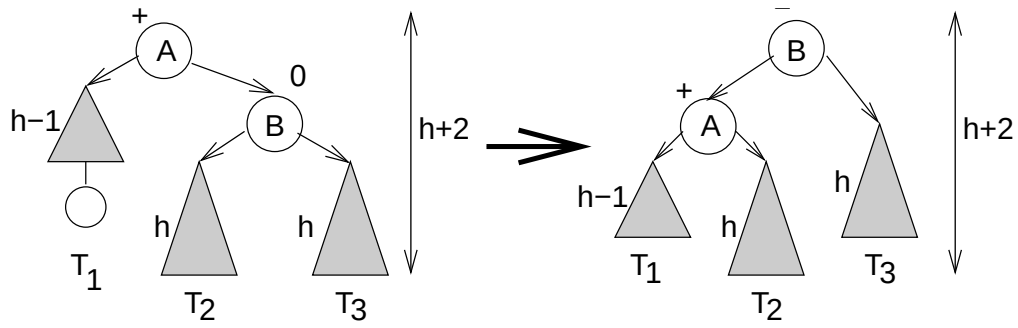


Há três subcasos, dependendo do fator de balanceamento do filho direito da raiz:

Programa 8.1 Esboço da rotina de busca e inserção em árvore AVL

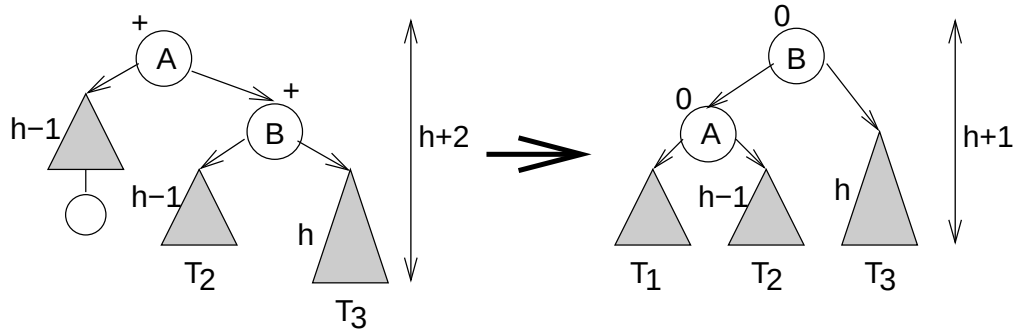
```
procedure BuscaInsere(var p: ArvAVL; k: TipoChave;
        inf: TipoInfo; var h: boolean; var q: Arvore);
  { O parâmetro h indica se a altura p da árvore cresceu;
    q apontará para o nó que contém a chave k.}
  var p1,p2: ArvAVL;
begin
if p=nil then {Não achou}
    begin
    new(p); h := true;
    with p↑ do
        begin
        bal := zero; chave := k; info := inf;
        esq := nil; dir := nil
        end
    end
else
    if k<p↑.chave then {Desce à esquerda}
        begin BuscaInsere(p↑.esq,k,inf,h,q);
        if h then {Subárvore esquerda ficou mais alta}
            case p↑.bal do
            mais: begin p↑.bal := zero; h := false end;
            zero: p↑.bal := menos;
            menos: {Rebalancear}
                begin
                p1 := p↑.esq;
                if p1↑.bal=menos then {Rotação simples – LL}
                    begin ... end
                else     {Rotação dupla – LR}
                    begin ... end;
                p↑.bal := zero; h := false
                end
            end {case}
        end
    else if k>p↑.chave then {Desce à direita}
        begin {Análogo ao caso de descer à esquerda} end
    else h := false {Achou!}
end;
```

(4a): Fator “0” (rotação simples RR)



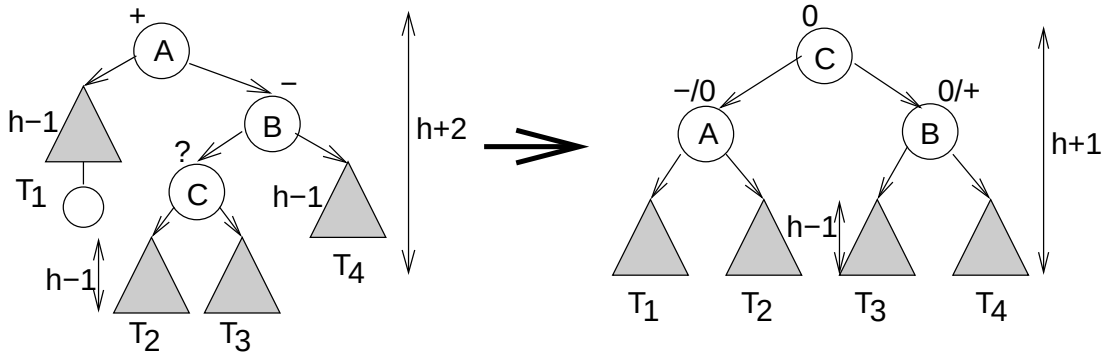
A altura permanece inalterada (o processo pára).

(4b): Fator “+” (rotação simples RR)



A altura diminui (o processo continua).

(4c): Fator “-” (rotação dupla RL)



A altura diminui (o processo continua).

8.2 Árvores de tipo *B*

Vimos na seção anterior que é possível utilizar árvores binárias para armazenar e recuperar informação, garantindo que o tempo de execução das rotinas será da ordem $\mathrm{O}(log_2 n)$. Entretanto, quando se trata de dados arquivados em dispositivos de memória externa, como discos rígidos, mesmo este desempenho não é satisfatório. A Fig. 8.3 indica, esquematicamente, o funcionamento de um disco magnético e ajuda a entender a disparidade entre tempos de acesso.

O tempo total de um acesso ao disco é constituído pela soma de três parcelas:

- tempo de busca[3] ΔS
- tempo de latência ΔL
- tempo de transferência de dados ΔT

Os tempos ΔS e ΔL variam aleatoriamente, conforme a localização dos dados no disco e a posição do mecanismo de acesso no início da operação; ΔT depende basicamente da quantidade de dados a serem transferidos. A soma destas três parcelas costuma ser ordens de grandeza maior do que os tempos de acesso à memória principal.

Por outro lado, note-se, por exemplo, que altura mínima de uma árvore binária que contém 1.024 nós é 10; para uma árvore com 1 milhão de nós, esta altura será de 20. O número de acessos à estrutura nos algoritmos de busca e inserção é proporcional a esta altura. Este número de acessos é aceitável para estruturas residentes na memória principal, mas torna-se proibitivo para memórias externas. Nestes casos, o tempo de execução de algoritmos é dominado pelo tempo de acesso, assim a preocupação principal é diminuir o número deles. Uma solução óbvia para este problema é diminuir a altura da árvore, aumentando significativamente o grau dos seus nós. Além disto, deve-se garantir que a altura não crescerá além de um certo limite. Uma das estruturas mais utilizadas neste caso são as *árvores do tipo B* (ou *árvores B*), semelhantes às árvore n-árias definidas no Cap. 6.

T é uma árvore B de ordem $b \geq 2$ *se*

1. *todas as folhas de T têm o mesmo nível;*
2. *cada nó interno tem um número variável r de registros e $r+1$ de filhos, onde*

$$\begin{cases} \lfloor b/2 \rfloor \leq r \leq b & \textit{se nó} \neq \textit{raiz} \\ 1 \leq r \leq b & \textit{se nó = raiz} \end{cases}$$

3. *cada folha tem um número variável r de registros obedecendo à mesma restrição do item anterior;*
4. *os campos de informação contidos nos registros obedecem à propriedade de árvores de busca.*

[3] Em inglês *seek*.

Figura 8.3 Funcionamento esquemático de discos magnéticos

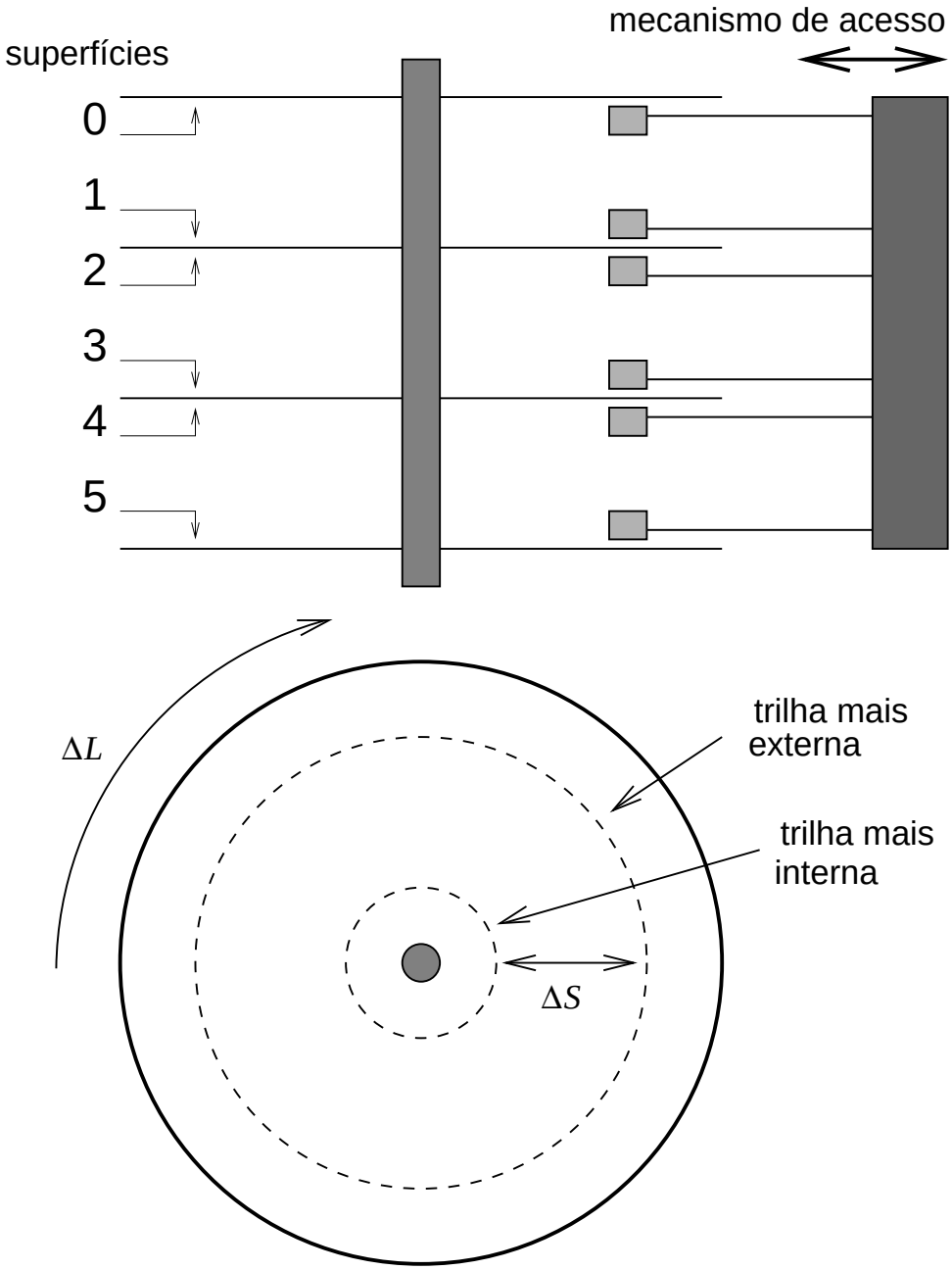
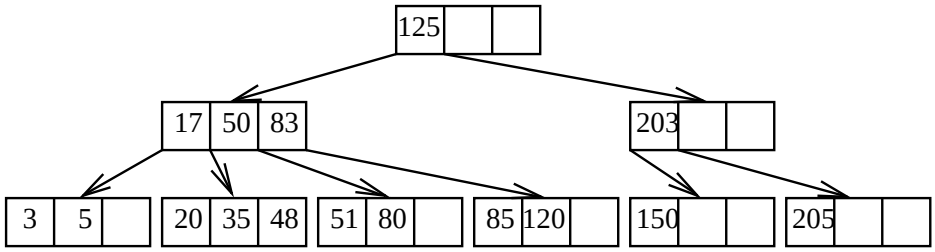


Figura 8.4 Exemplo de árvore B de ordem 3



A Fig. 8.4 apresenta um exemplo de árvore B de ordem 3.[4] Note que, neste caso, cada *nó* da árvore contém de 1 a 3 *registros* de dados. Na prática, as ordens são muito maiores, sendo tipicamente 255 ou 511. A tabela seguinte indica os números mínimos e máximos de registros de dados que podem estar contidos em níveis 1 a 4 de uma árvore B de ordem 255.

	mínimo		máximo	
nível	nós	registros	nós	registros
1	1	1	1	1×255
2	2	2×127	256	256×255
3	2×128	$2 \times 128 \times 127$	256^2	$256^2 \times 255$
4	2×128^2	$2 \times 128^2 \times 127$	256^3	$256^3 \times 255$
Total	33.027	4.194.303	16.843.009	4.294.967.295

Os números desta tabela mostram que, numa árvore de ordem 255, de altura 4, podemos guardar entre 4×10^6 e 4×10^9 registros de dados. Isto certamente cobre a maior parte de aplicações. Note-se que para uma árvore de altura 5, o mínimo seria da ordem de 5×10^7. Conclui-se que o número de acessos para encontrar um dado numa árvore B típica não passa, na prática, de 4. Se supusermos que a raiz reside sempre na memória principal, este número reduz-se a 3. Note-se que uma árvore binária de altura mínima que contivesse 4×10^6 registros teria a altura aproximada de 22.

A implementação de árvores B pode utilizar as técnicas já vistas nos Caps. 6 e 7. Por exemplo, um nó de uma árvore B de ordem 255 poderia seguir as declarações:

```
const ordem = 255;

type
    ApReg = ↑Reg;
    Reg = record
        numregs: 0..ordem;
        filhos: array[0..ordem] of ApReg
        info: array[1..ordem] of M;
    end;
    ArvB = ApReg;
```

Supõe-se que os elementos do nó seguem a representação natural indicada na Fig. 8.4. Assim, a informação contida em *info*[1] é maior do que as informações contidas na subárvore *filhos*[0] e menor do que as contidas na subárvore *filhos*[1]. Os campos de informação dentro de um nó estão em ordem crescente. Deve-se

[4] Alguns autores utilizam um conceito diferente de ordem.

observar que simplificamos a discussão supondo que a árvore está sendo manipulada na memória, pois utiliza ainda o conceito de apontadores. Numa implementação real, com a árvore em disco, utilizaríamos os endereços no disco, análogos aos apontadores.

O algoritmo de busca em árvore B é bastante elementar, análogo às árvores de busca do Cap. 6 e não será apresentado aqui. Note-se apenas que, feito acesso a um nó, a busca do valor dado no vetor *info* pode ser feita utilizando-se a técnica de *busca binária* uma vez que os dados estão ordenados.

O algoritmo de inserção é um pouco mais complexo e será apresentado através de uma rotina auxiliar recursiva *InsereRecArvB(p,s,x,h)* esboçada no Prog. 8.3 e alguns diagramas que esclarecem os vários casos que ocorrem neste processo. Os parâmetros p e x denotam a árvore e o valor a ser inserido. Ao retornar, a variável h indicará se houve necessidade de quebra do nó em dois nós; neste caso, a variável x conterá o valor que deve ficar entre os valores contidos nos novos nós, e a variável s apontará para o nó que deve ficar à direita de x; por construção, o nó anterior ficará à esquerda. O valor devolvido pela função será falso se o valor x fornecido já ocorre na árvore. O Prog. 8.2 indica a chamada inicial da rotina de inserção.

Programa 8.2 Rotina de busca e inserção em árvoreB

```
function InsereArvB(var p: ArvB; var x: M): Boolean;
{ Devolve 'false' se 'x' já ocorre na árvore 'p'. }
   var h: Boolean;  q,s: ArvB;
begin
InsereArvB := InsereRecArvB(p,s,x,h);
if h
   then begin
           new(q);
           with q↑ do
              begin
              numregs := 1;
              filhos[0] := p;  filhos[1] := s;
              info[1] := x
              end;
           p := q
           end
end;
```

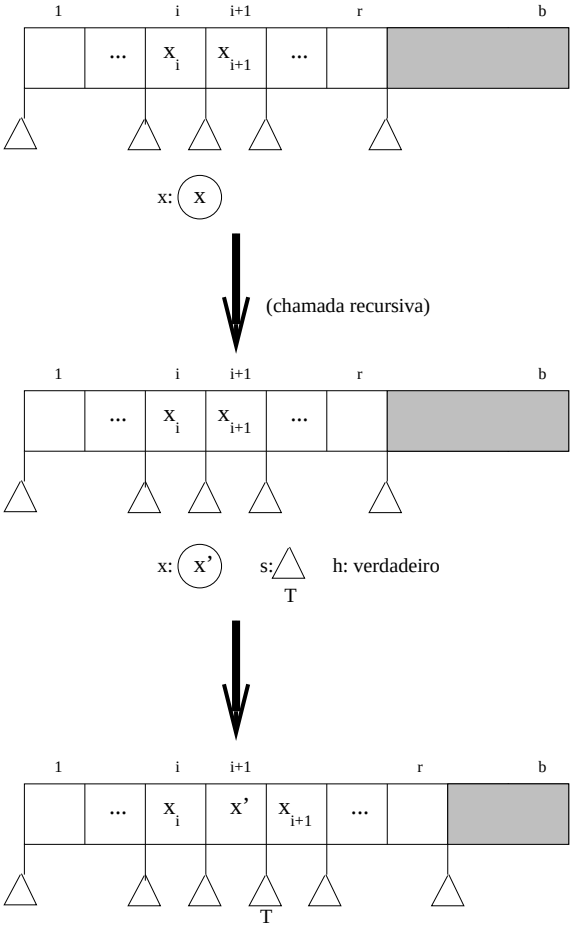
Programa 8.3 Esboço da rotina auxiliar de busca e inserção em árvoreB

```
function InsereRecArvB(var p,s: ArvB; var x: M; var h: Boolean): Boolean;
   var i: 0..ordem;
begin
if p=nil
   then begin
          h := true;
          s := nil;
          InsereRecArvB := true
          end
   else with p↑ do
        begin
        i := {menor índice 'j' tal que 'x≤info[j]'; senão, 'j=(numregs+1)'};
        if (i≤ordem) and (x=info[i])
           then begin
                  h := false;
                  InsereRecArvB := false
                  end
           else begin
                InsereRecArvB := InsereRecArvB(filhos[i−1],s,x,h);
                if h
                   then begin
                        {Insere 'x' e 's' na posição 'i' dos vetores 'info'
                          e 'filhos'; se necessário, supõe que existe uma
                          posição adicional em cada vetor};
                        inc(numregs);
                        if numregs≤ordem
                           then h := false
                           else begin
                                {Quebra o nó 'p'; a primeira metade
                                  permanece em 'p'; o valor do meio é
                                  atribuído a 'x'; a segunda metade é
                                  copiada para um novo nó alocado em 's'};
                                h := true
                                end
                        end
                end
        end
end;
```

Caso 1: Árvore vazia

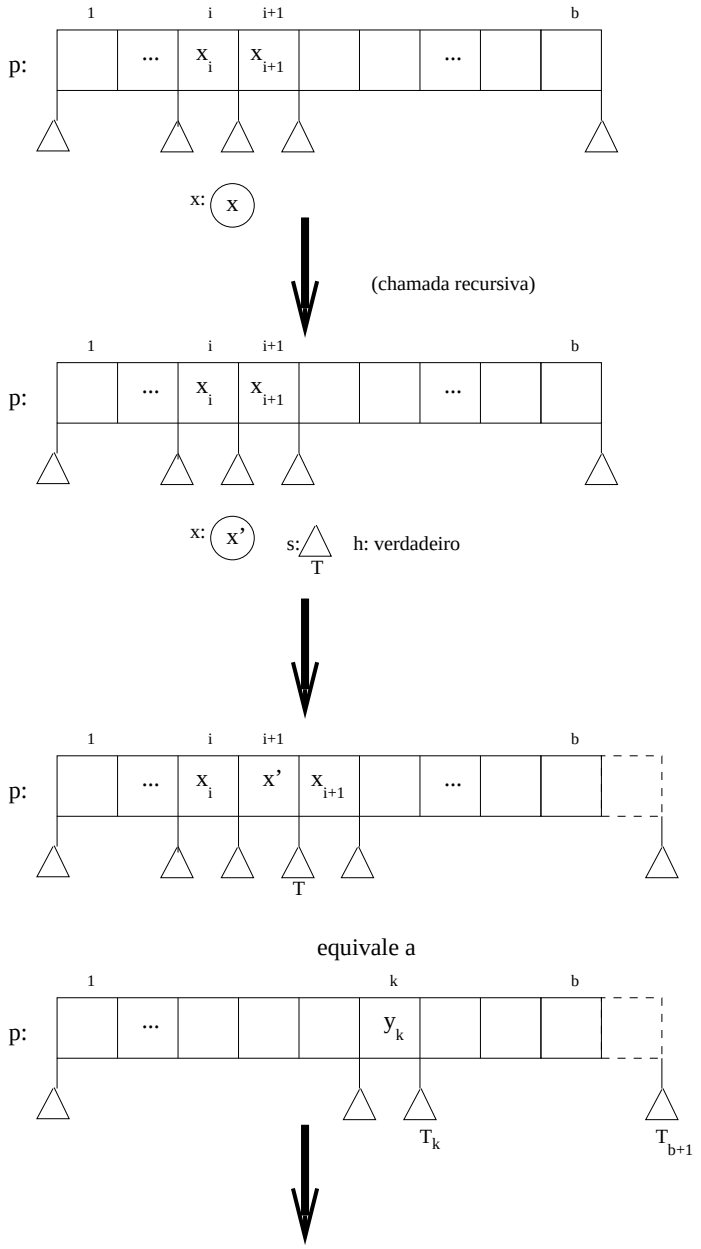
h: verdadeiro s : **nil**

Caso 2: $r < b$



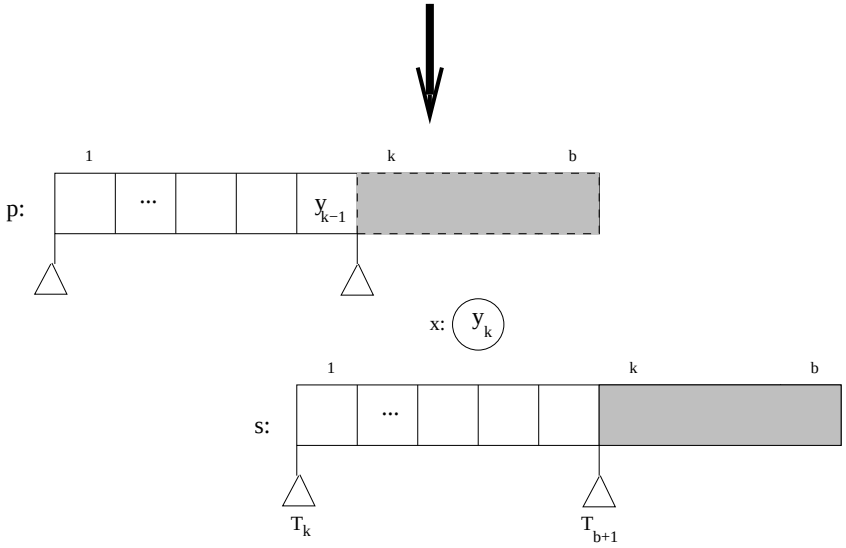
h: falso

Caso 3: $r = b$



onde $k = \lceil b/2 \rceil + 1$.

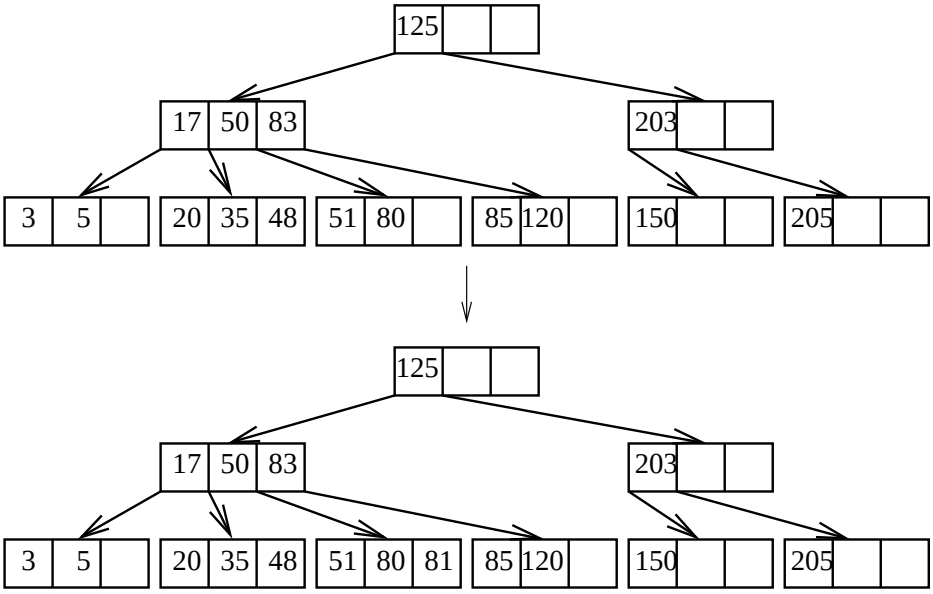
(continua)



h: verdadeiro

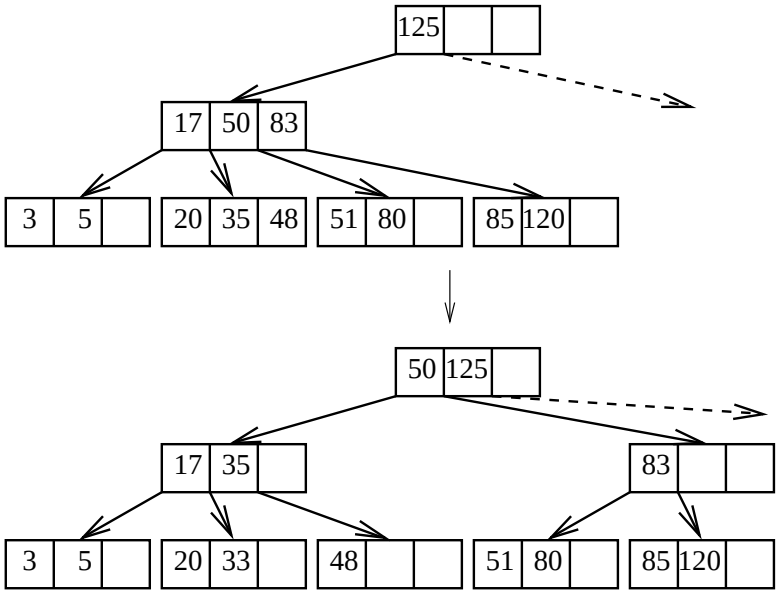
A seguir, o algoritmo de inserção é apresentado através de exemplos.

Caso 1: Inserção do elemento 81 – a folha tem espaço para mais um elemento:



É fácil ver que neste caso haverá, no máximo, $h + 1$ acessos ao disco (h leituras e 1 gravação) onde h é a altura da árvore.

Caso 2: Inserção do elemento 33 – Estouro da folha provoca subida da mediana dos $b + 1$ registros e divisão dos restantes em dois nós, com $\lceil b/2 \rceil$ e $\lfloor b/2 \rfloor$ registros, respectivamente:

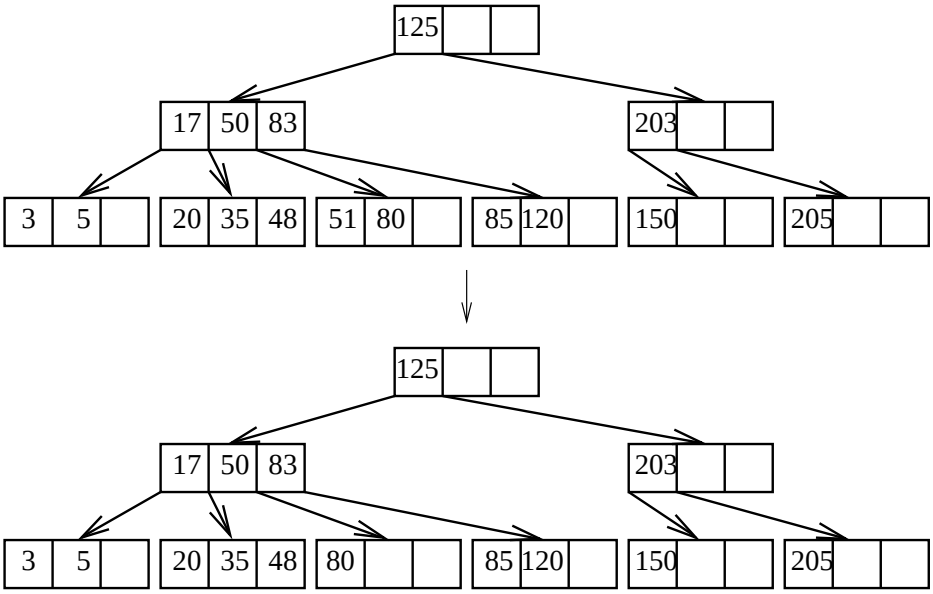


Neste caso, haverá, no máximo h leituras e $2h + 1$ gravações, supondo que há estouro em todos os níveis.

Note-se que estas operações podem envolver uma movimentação razoável de dados dentro dos nós, mas elas serão realizadas na memória principal, assim que em termos de eficiência o fator dominante será ainda o número de acessos ao disco que claramente é da ordem $\mathrm{O}(h)$, ou seja $\mathrm{O}(log_b n)$.

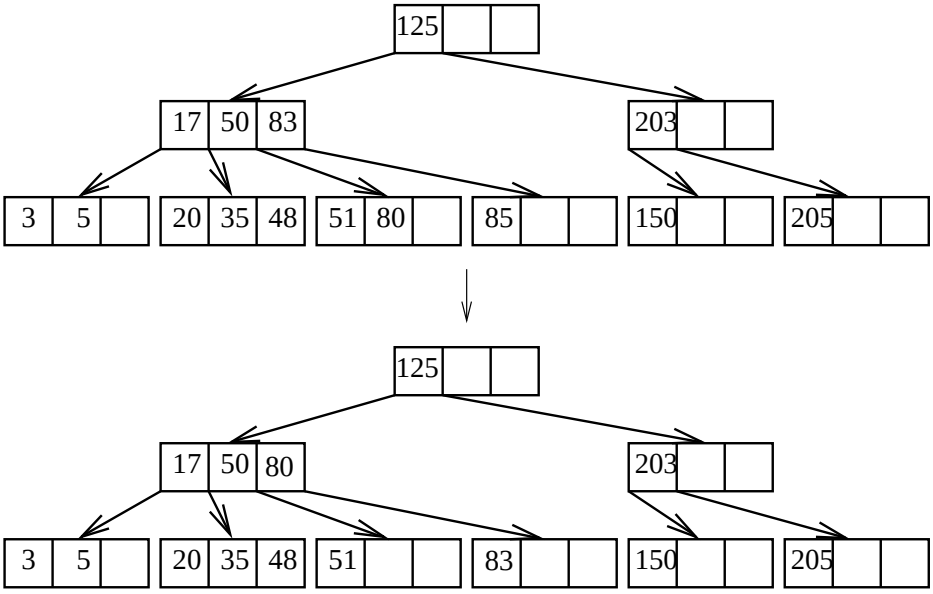
O algoritmo de remoção segue a mesma filosofia de manter as propriedades de árvore B. Uma remoção sempre pode ser reduzida à remoção de um elemento de uma folha; caso ele não esteja na folha, ele pode ser trocado com o primeiro elemento que o segue (ou o último que o precede) na árvore – este elemento estará numa folha. Por exemplo, se quisermos remover o elemento 83 da Fig. 8.4, poderemos colocar em seu lugar o elemento 85 (ou 80), caindo assim no caso de remoção de uma folha. O algoritmo de remoção será apresentado somente através de exemplos.

Caso 1: Remoção do elemento 51 – a folha ainda tem o número mínimo $\lfloor b/2 \rfloor$ de registros:



Neste caso, haverá, h leituras e 1 gravação.

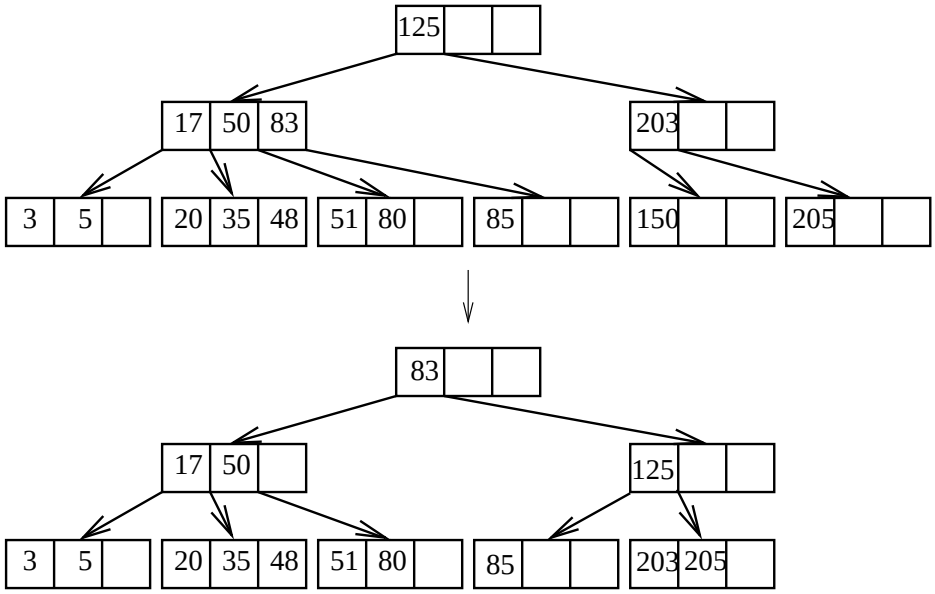
Caso 2: Remoção do elemento 85 – o número de elementos cai abaixo do mínimo $\lfloor b/2 \rfloor$, mas um dos nós irmãos tem mais do que o mínimo de elementos, permitindo um "empréstimo" através do nó pai; no caso serão movidos 80 e 83:



Neste caso, haverá, no pior caso, $h + 2$ leituras e 3 gravações.

Caso 3: Remoção do elemento 150 – o número de elementos cai abaixo do mínimo $\lfloor b/2 \rfloor$ e não há possibilidade de "empréstimo"; neste caso existe um nó irmão que tem $\lfloor b/2 \rfloor$ registros, e juntando este nó com a folha em questão e incluindo o elemento do nó pai que separa os dois, obtém-se um novo nó

cheio; este processo pode ser propagado até a raiz:



Neste caso, haverá, no pior caso, $3h - 2$ leituras e $2h - 1$ gravações.

Conclui-se que a remoção também pode ser realizada com número de acessos da ordem $\mathrm{O}(h)$.

Deve-se notar, finalmente, que existem variantes de árvores B, como B^+ e B^* descritos na literatura – veja, por exemplo, [**?**].

8.3 Árvores digitais

Conjuntos de cadeias de caracteres podem ser representados por *árvores digitais* em que as arestas são rotuladas com caracteres consecutivos das cadeias. Por exemplo, o conjunto das palavras inglesas:

```
a    an   and  are   as   at    be
but  by   for  from  had  have  he
her  his  i    in    is   it    no
not  of   on   or
```

pode ser representado pela árvore da Fig 8.5. Nesta representação, os nós da forma ● indicam o fim de uma cadeia.

Uma representação possível para árvores digitais é utilizar árvores n-árias de grau do tamanho do alfabeto utilizado (por exemplo, 26) com uma marca booleana para indicar um nó ● :

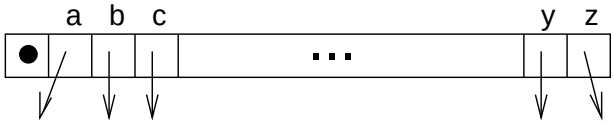
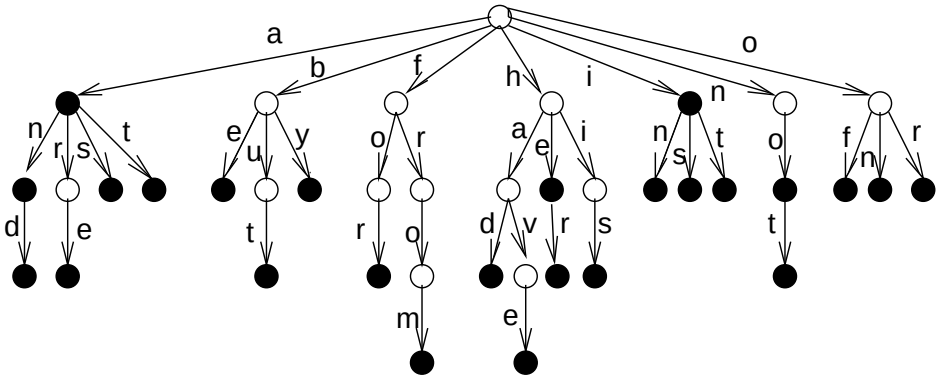
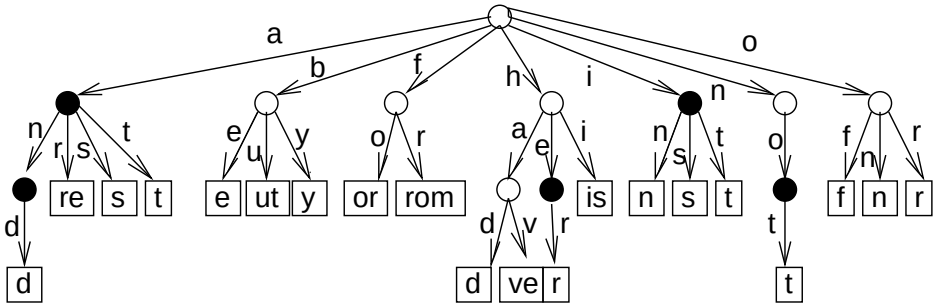


Figura 8.5 Exemplo de árvore digital



Com esta representação, o apontador pode ser selecionado utilizando-se o próprio caractere como índice; desta maneira, o tempo de busca torna-se proporcional ao comprimento da cadeia. Há, entretanto, um grande desperdício de memória: dos $39 \times 26 = 1014$ campos apontadores do exemplo, apenas 38 são não nulos. Este desperdício de memória pode ser reduzido em parte utilizando um formato especial para as folhas, sem o vetor; no caso do exemplo, haveria redução para $19 \times 26 = 494$ campos apontadores, dos quais 38 não nulos. Pode-se também modificar a representação, com folhas especiais, juntando cadeias quando há uma única possibilidade de continuar na árvore; no caso do exemplo, restariam 12 nós comuns com 312 campos apontadores, dos quais 31 não nulos, como mostra a Fig. 8.6.

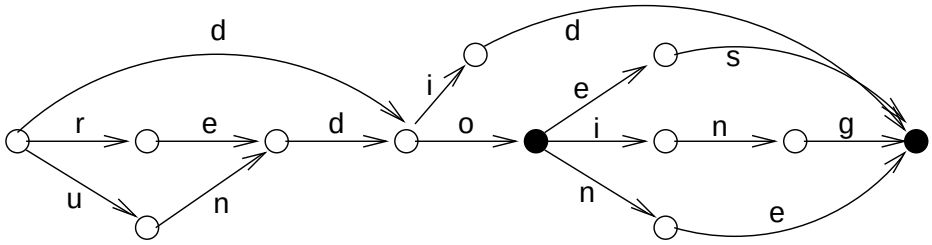
Figura 8.6 Representação modificada de árvore digital



Árvores digitais são eficientes na fatoração de prefixos comuns das cadeias representadas. Em casos em que há também muito compartilhamento de sufixos comuns, é interessante utilizar-se autômatos finitos acíclicos minimais. Por exemplo, as 15 formas dos verbos ingleses do, redo e undo seriam representadas pelo autômato da Fig. 8.7.

Supondo a utilização de folhas especiais (sem vetores), este autômato teria $11 \times 26 = 286$ campos apontadores, dos quais 16 não nulos; a árvore digital correspondente teria $26 \times 26 = 676$ campos apontadores, dos quais 37 não nulos. Deve-se notar, entretanto, que a construção de um autômato como este é muito mais complicada e demorada do que de uma árvore digital.

Figura 8.7 Exemplo de autômato



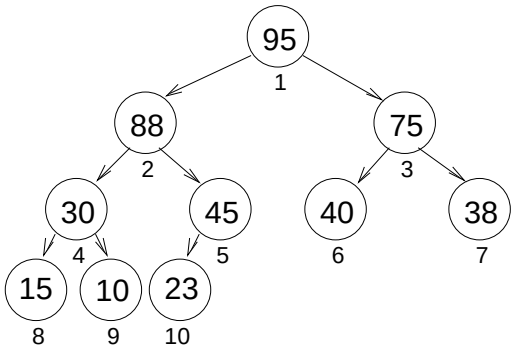
8.4 Filas de prioridade

Uma outra aplicação de árvores binárias é exemplificada por *filas de prioridade* usadas para selecionar ou remover eficientemente o maior (ou o menor) elemento de uma coleção.

Uma *fila de prioridade* é uma árvore binária que tem as seguintes propriedades:

1. a árvore é *completa* ou *quase completa*;
2. em cada nó da árvore, o valor da chave é maior do que os valores das chaves dos filhos.

Figura 8.8 Exemplo de fila de prioridade



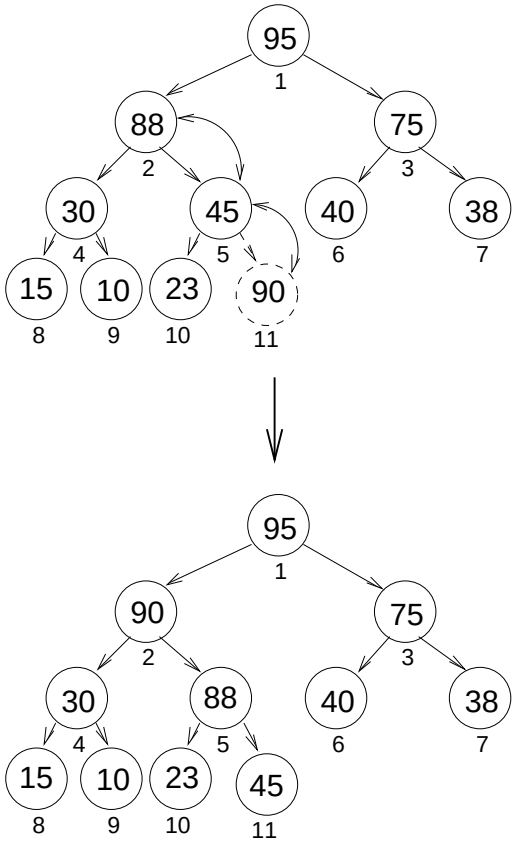
A Fig. 8.8 mostra um exemplo desta estrutura. Conforme foi visto na Seção 6.2, árvores quase completas podem ser implementadas eficientemenete utilizando vetores; neste caso a fila de prioridade costuma ser chamada de *heap*. Ela pode ser percorrida facilmente utilizando-se as fórmulas vistas na pág. 57. No caso do nosso exemplo, teríamos:

95	88	75	30	45	40	38	15	10	23
1	2	3	4	5	6	7	8	9	10

A determinação do maior elemento da fila é obviamente trivial e muito eficiente pois é o elemento que está na raiz da árvore, isto é, na primeira posição do *heap*. Verificaremos que as operações de inserção e remoção de elementos do *heap* também podem ser realizadas com eficiência, mantendo-se a propriedade da estrutura. Primeiramente, veremos através de exemplos duas operações básicas de *subida* e de *descida* de elementos na árvore.

Suponhamos que na primeira posição disponível (isto é, 11) do nosso exemplo foi colocado um novo elemento cujo valor é 90. Este elemento pode ser deslocado, por meio de trocas sucessivas, no sentido da raiz da árvore, até encontrar a posição em que será satisfeita a definição de *heap*. A Fig. 8.9 ilustra os estados inicial e final desta operação.

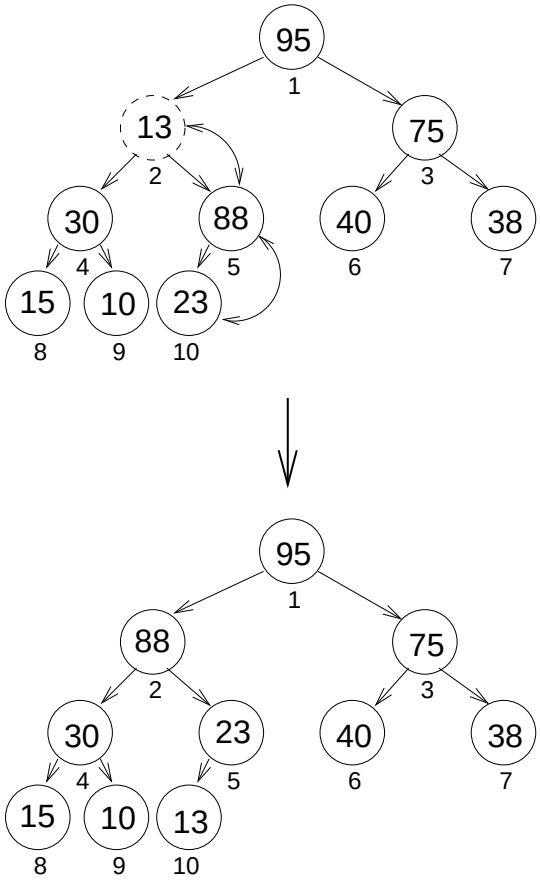
Figura 8.9 Operação de subida de um elemento no *heap*



Analogamente, suponhamos que temos um *heap* no qual apenas um elemento não satisfaz a definição por não ser maior do que os elementos contidos nos seus filhos. Podemos deslocar este elemento na direção das folhas até que ele encontre uma posição válida. Fig. 8.10 ilustra esta operação.

A programação destas operações é simples, adotando-se, por exemplo, as declarações:

Figura 8.10 Operação de descida de um elemento no *heap*




```
const nMax = ...;

type
    Indice = 0 .. nMax;
    Heap = record
        vetor: array [1..nMax] of T;
        n: Indice  { Comprimento corrente }
    end;
```

As rotinas básicas estão indicadas no Prog. 8.4. É fácil verificar que cada uma das rotinas realiza da ordem de $\mathrm{O}(\log n)$ operações, proporcional à altura da árvore.

Utilizando estas rotinas podemos programar as operações de construção inicial, inserção e remoção de um *heap*, conforme mostra o Prog. 8.5. Note-se que a remoção é sempre feita na raiz, devolvendo o elemento máximo. No caso de construção, indicamos duas versões diferentes. Pode-se mostrar que o procedimento *ConstroiHeap* realiza $\mathrm{O}(n \log n)$ operações enquanto que o procedimento *OutroConstroiHeap* realiza $\mathrm{O}(n)$ operações. Veremos na Seção 13.3 como estas rotinas podem ser usadas para obter um algoritmo eficiente de ordenação.

Programa 8.4 Operações básicas para *heap*

```
procedure Sobe(var a: Heap; m: Indice);
    var j: Indice; x: T;
begin
with a do
    begin
    x := vetor[m];
    j := m div 2;
    while (j≥1) and (vetor[j]<x) do
        begin
        vetor[m] := vetor[j];
        m := j; j := j div 2
        end;
    vetor[m] := x
    end
end;
```

```
procedure Desce(var a: Heap; m: Indice);
    var k: Indice; continua: boolean; x: T;
begin
with a do
    begin
    x := vetor[m]; k := 2*m;
    continua := true;
    while continua and (k≤n) do
        begin
        if k<n then
            if vetor[k]<vetor[k+1]
                then k := k+1;
        if x<vetor[k]
            then
                begin
                vetor[m] := vetor[k];
                m := k;
                k := 2*k
                end
            else continua := false
        end;
    vetor[m] := x
    end
end;
```

Programa 8.5 Rotinas de manipulação de *heaps*

```
procedure ConstroiHeap(var a: Heap);
    var m: Indice;
begin
with a do
    for m:=2 to n do Sobe(a,m)
end;

procedure InsereHeap(var a: Heap; x: T);
begin
with a do
    begin
    n := n+1;
    vetor[n] := x;
    Sobe(a,n)
    end
end;

procedure OutroConstroiHeap(var a: Heap);
    var m: Indice;
begin
with a do
    for m:=n div 2 downto 1 do Desce(a,m)
end;

function RemoveHeap(var a: Heap): T;
begin
with a do
    begin
    RemoveHeap := vetor[1];
    vetor[1] := vetor[n];
    n := n-1; Desce(a,1)
    end
end;
```

8.5 Exercícios

1. Demonstre que o número de nós de uma árvore de Fibonacci de altura h descrita na Seção 8.1 é dado por $N(h) = f_{h+2} - 1$.

2. Complete a rotina exibida no Prog. 8.1.

3. Escreva uma função que verifica se uma árvore dada tem alturas balanceadas. Suponha que a árvore não contém os fatores de balanceamento. Sugestão: A função deve devolver também a altura da árvore.

4. Implemente o algoritmo de remoção de árvores AVL seguindo a explicação da Seção 8.1.

5. Podemos generalizar o conceito de árvores de altura balanceada, impondo que o fator de balanceamento seja no máximo k, em valor absoluto, sendo este um valor prefixado.

   - Calcule a altura máxima de uma árvore deste tipo, em função de k e do número de nós n.
   - Desenvolva os algoritmos de inserção e de remoção análogos aos discutidos para as árvores AVL.

6. Descreva numa notação mais precisa os algoritmos de inserção e de remoção em árvores B, sem entrar em muitos pormenores de programação.

7. Justifique as conclusões obtidas sobre os números de acesso ao disco para os algoritmos de inserção e de remoção em árvores B.

Capítulo 9

Listas Gerais

Este capítulo cobrirá uma generalização dos conceitos de estruturas lineares vistas no Cap. 4.

9.1 Conceitos e implementação

Dizemos que uma *lista generalizada* é uma seqüência linear em que cada elemento ou é um *átomo*, ou é uma lista generalizada. O conceito de átomo depende da aplicação, mas poderá ser um inteiro, um nome, um registro de informações ou outra estrutura qualquer que não seja tratada como lista neste contexto. Note-se que o conceito de lista generalizada foi definido de maneira recursiva.

Utilizaremos, às vezes, uma notação parentética para descrever listas generalizadas. Nesta notação, cada lista terá a forma

$$(\alpha_1, \alpha_2, \ldots, \alpha_n)$$

onde α_i denota um átomo ou uma lista generalizada, conforme a definição acima. Indicaremos alguns exemplos destas listas, dando um nome a cada uma:

A: ((4,7),(4,7,(8)))
B: ((1,4),(7,8))
C: (3,B,B)
D: (5,8,D)
E: ()

As listas *A*, *B*, *C*, *D* e *E* têm, respectivamente, 2, 2, 3, 3 e 0 elementos.

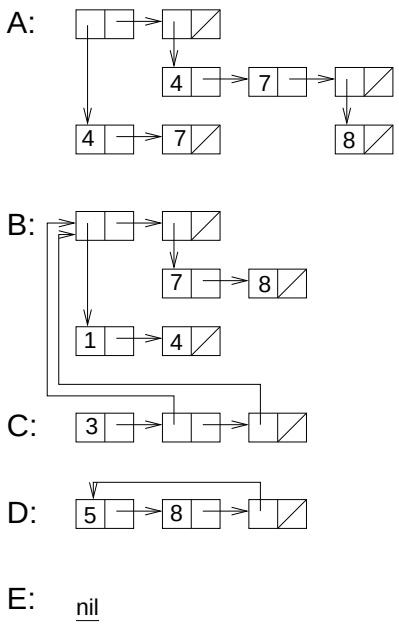
As listas *C* e *D* podem ser expandidas com as definições correspondentes:

C: (3,((1,4),(7,8)),((1,4),(7,8)))
D: (5,8,(5,8,(5,8,(...))))

Note-se que a lista D, apesar de ter três elementos, inclui um número infinito de inteiros. Este fato deve-se obviamente à definição recursiva desta lista.

Para representar na memória as listas generalizadas, podemos usar as várias técnicas vistas no Cap. 3. Por exemplo, usando as listas ligadas simples, poderíamos representar os exemplos acima como indicado na Fig. 9.1. Neste caso, utilizamos a representação *compartilhada* em que as várias listas repetidas são representadas uma única vez como é o caso da lista B. Uma outra opção, necessária em algumas aplicações,

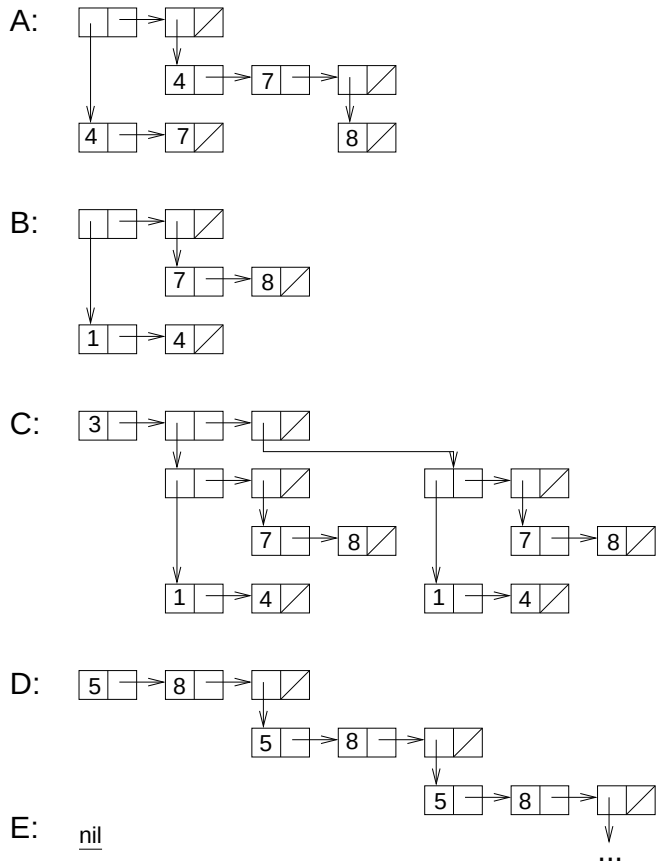
Figura 9.1 Implementação compartilhada de listas



seria expandir as definições das listas; neste caso teríamos a representação da Fig. 9.2. Como é natural, neste caso, não poderemos representar adquadamente a lista D. Note-se que nos dois casos haverá necessidade de incluir em cada nó da estrutura uma marca que indique se o elemento é um átomo ou uma lista.

A implementação destas idéias e a formulação de algoritmos que manipulam estas listas resulta naturalmente da discussão acima. Indicamos no Prog. 9.1 a implementação de uma função que conta os átomos de uma lista. Esta versão não funciona para listas que têm alguma circularidade como é o caso da lista D. Além disto, a contagem dos átomos é realizada como se a representação fosse feita com cópia. Por exemplo, o resultado seria 9 para a lista C qualquer que fosse a representação. O Prog. 9.2 traz uma outra solução que funciona para qualquer lista e cujo resultado indica exatamente o número de átomos usado na representação. Esta versão supõe que todos os campos *visitado* da estrutura têm inicialmente o valor *false*; estes campos têm o seu valor modificado para *true* durante a execução.

Figura 9.2 Implementação de listas com cópia



Programa 9.1 Contagem de átomos de uma lista

```
type
    ApReg = ↑Reg;
    ListaGen = ApReg;
    Reg = record
        prox: ApReg;
        case ElementoAtomo: boolean of
            true: (atomo: integer);
            false: (lista: ListaGen)
    end;

function ContaAtomos(p: ListaGen): integer;
    var  s: integer;
begin
    s := 0;
    while p≠nil do with p↑ do
        begin
            case ElementoAtomo of
                true:  s := s+1;
                false: s := s+ContaAtomos(lista)
            end;
        p := prox
        end;
    ContaAtomos := s
end;
```

Programa 9.2 Contagem de átomos de uma lista – versão mais geral

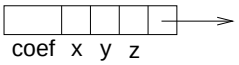
```
type
    ApReg = ↑Reg;
    ListaGen = ApReg;
    Reg = record
        visitado: boolean;
        prox: ApReg;
        case ElementoAtomo: boolean of
            true: (atomo: integer);
            false: (lista: ListaGen)
    end;

function ContaAtomos(p: ListaGen): integer;
    var  s: integer;
begin
    s := 0;
    while p≠nil do with p↑ do
        begin
        if not visitado then
            begin
                visitado := true;
                case ElementoAtomo of
                    true:  s := s+1;
                    false: s := s+ContaAtomos(lista)
                end
            end;
            p := prox
        end;
    ContaAtomos := s
end;
```

9.2 Exemplo: polinômios em múltiplas variáveis

Uma aplicação interessante de listas generalizadas é a manipulação de polinômios em múltiplas variáveis que estende o exemplo da Seção 3.4. Consideremos um exemplo de polinômio em variáveis x, y e z:

$$P(x,y,z) = x^{10}y^3z^2 + 2x^8y^3z^2 + 3x^8y^2z^2 + x^4y^4z - 6x^3y^4z + 2yz$$

Uma primeira idéia de representação poderia adotar para cada termo um nó da forma:



Entretanto, esta representação é muito inflexível, fixando o número de variáveis. Uma segunda representação utiliza as idéias da seção anterior. O polinômio em $k \geq 1$ variáveis será transformado em um polinômio em uma variável, com coeficientes que são polinômios em $k-1$ variáveis, e assim por diante. O nosso exemplo pode ser reescrito como:

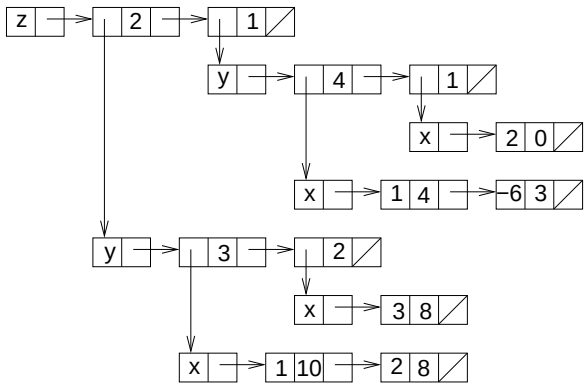
$$P(x,y,z) = ((x^{10} + 2x^8)y^3 + 3x^8y^2)z^2 + ((x^4 - 6x^3)y^4 + 2x^0y)z$$

Esta transformação sugere as representações indicadas na Fig. 9.3. A alternativa (a) utiliza mais memória mas é mais uniforme e, portanto, facilita a programação das rotinas de manipulação. A implementação destas representações poderia utilizar nós do tipo:

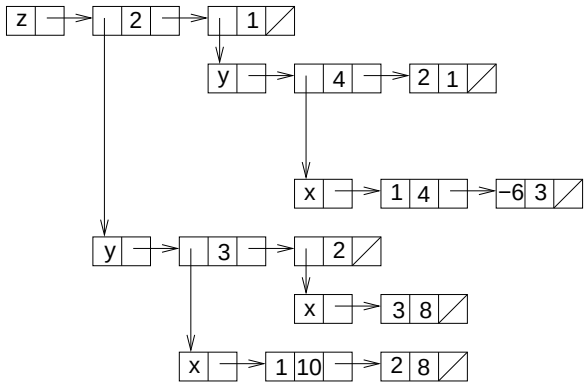
```
type
    ApTermo = ↑Termo;
    Polinomio = ApTermo;
    Termo = record
        prox: Polinomio;
        case MarcaCabeca: boolean of
            true: (variavel: char);
            false:
                (expoente: integer;
                  case CoefInteiro: boolean of
                      true: (CoefInt: integer);
                      false: (CoefPolin: Polinomio)
                )
        end;
```

Utilizando esta representação, pode-se elaborar as rotinas usuais de manipulação de polinômios: soma, multiplicação, divisão etc. Várias delas serão semelhantes ao caso de polinômios de uma variável, mas utilizarão recursão para operar sobre os coeficientes que são, por sua vez, polinômios.

Figura 9.3 Representação dos polinômios em múltiplas variáveis



(a)



(b)

9.3 Exercicios

1. Transforme os programas 9.1 e 9.2 eliminando os comandos **while** e introduzindo recursão dupla.

2. Escreva uma função que verifica se uma lista generalizada tem compartilhamento ou circularidades na sua representação.

3. Escreva uma função que calcula o número de nós utilizado na representação de uma lista generalizada e que funciona em qualquer caso.

4. Escreva procedimentos que copiam listas generalizadas representadas como na Seção 9.1. Considere duas versões: uma que faz uma cópia exata e outra que ao copiar expande todas as sublistas.

5. Escreva as rotinas de soma e multiplicação para polinômios em múltiplas variáveis, conforme indicado na Seção 9.2.

Capítulo 10

Espalhamento

A técnica de *espalhamento*[1] permite construção eficiente de tabelas em que deseja-se armazenar informação constituída de chaves, chamadas neste contexto de *nomes*, e de *atributos* correspondentes, sem utilizar estruturas em árvore, mas baseadas na representação das próprias chaves.

Como em outros casos de implementação de tabelas (veja Cap. 8), as operações fundamentais são a busca, inserção e remoção de informações.

10.1 Conceitos básicos

Uma *tabela de espalhamento* é uma matriz com $b \geq 1$ linhas e $s \geq 1$ colunas, como esquematizado a seguir. Dado um nome x, a linha desta matriz é selecionada calculando-se um índice $f(x)$, onde f é a *função de espalhamento*. No caso do exemplo ($b = 7$ e $s = 3$) temos $f(\texttt{'joão'}) = 3$. As colunas 2 e 3 são usadas quando há mais de um nome mapeado pela função f num mesmo valor. Em muitos casos utilizam-se tabelas com $s = 1$, ou seja apenas uma coluna.

		1	2	3
	0			
	1			
	2			
$f(\texttt{'joão'}) \rightarrow$	3	`joão`		
	4			
	5			
	6			

Suponhamos, a título de exemplo, que a função f calcula o índice baseado na posição da primeira letra do alfabeto no nome, isto é, 0 para a letra ‘`a`’, 1 para ‘`b`’, etc. A tabela seguinte ($b = 26$ e $s = 2$) indica o resultado de inserção de vários nomes (foram ignorados os acentos):

[1] Em inglês *hashing* ou *scattering*.

	1	2
0	antônio	átila
1		
2	carlos	célio
3	douglas	
4	ernesto	estêvão
5		
	...	
24		
25	zoroastro	

Dependendo da localização de espaços livres na tabela, podemos continuar inserindo mais nomes, como por exemplo, '`diogo`' que entraria na posição [3,2] e '`francisco`' que entraria na posição [5,1]. Entretanto, não seria possível inserir '`eduardo`' pois as duas posições da linha 4 já estão ocupadas. Neste caso, temos uma *colisão*.

Por outro lado, é claro que a função de espalhamento escolhida para o exemplo é muito ingênua. Mesmo que o valor de s fosse muito maior, podemos imaginar que linhas da tabela correspondentes às letras '`a`' ou '`m`' seriam esgotadas muito antes daquelas correspondentes a '`q`' ou '`x`'.

Podemos concluir que a utilização de tabelas de espalhamento apresenta dois problemas fundamentais:

- escolha da função de espalhamento
- tratamento das colisões

Estes dois aspectos serão discutidos nas seções seguintes.

10.2 Funções de espalhamento

A escolha de uma função de espalhamento deve satisfazer a duas propriedades:

- eficiência
- bom espalhamento

Esta última significa que os valores produzidos pela função devem ter um comportamento aparentemente aleatório, independentemente do valor do nome, diminuindo a probabilidade de haver conflitos enquanto houver espaço razoável na tabela.

Citaremos, sem muita discussão, apenas alguns exemplos de funções de espalhamento. Na realidade, os métodos indicados podem ser combinados, resolvendo alguns problemas de cada um. Outros métodos podem ser encontrados na literatura especializada.

1. *Divisão:* o nome, tratado como um número na base 26, é dividido por um número p relativamente primo $f(x) = x \bmod p$. Por exemplo, para $p = 51$ teríamos:

$$\begin{aligned} f(\texttt{carlos}) &= ((((2 \times 26 + 0) \times 26 + 17) \times 26 + 11) \times 26 + 14) \times 26 + 18) \bmod 51 \\ &= 24.069.362 \bmod 51 = 14 \end{aligned}$$

ou seja, o nome ‘carlos’ deveria ser inserido na linha 14 da tabela, se houver espaço. Note-se que a utilização deste tipo de função implica em adotar também o valor p como o número de linhas da tabela. Uma outra observação é que o cálculo da função pode ser otimizado, utilizando-se aritmética modular o que evitaria operações com muitos dígitos.

2. *Seleção de algarismos:* o nome é tratado como uma seqüência de algarismos ou de *bytes* ou de *bits*, e uma subseqüência é selecionada para representar o índice. Por exemplo, suponhamos que todos os nomes são representados como a seqüência de dígitos $x = d_0 d_1 \cdots d_{11}$ em alguma base conveniente; uma escolha seria $f(x) = d_3 d_5 d_9$.

   Retomemos o exemplo do nome ‘carlos’. A sua representação poderia ser 020017111418. Supusemos que cada letra é indicada por dois dígitos que indicam a posição no alfabeto, ou seja 00 para ‘a’, 01 para ‘b’, etc. Teríamos então $f(\texttt{carlos}) = 074$.

   Este método deveria ser combinado com outros para resolver o problema de comprimentos variáveis dos nomes.

3. *Dobramento:* novamente, o nome é tratado como uma seqüência de algarismos ou de *bytes* ou de *bits*, e algumas subseqüências são combinadas por operações convenientes para produzir um índice. Por exemplo, suponhamos que todos os nomes são representados como a seqüência de dígitos $x = d_0 d_1 \cdots d_{11}$ em alguma base conveniente; uma escolha seria $f(x) = d_3 d_5 d_9 \oplus d_8 d_7 d_{10}$, onde $\oplus$ denota a operação de *ou exclusivo bit a bit*.

10.3 Endereçamento aberto

Um dos métodos de tratar o problema de colisões é o *endereçamento aberto*. Neste caso, o algoritmo procura uma outra posição disponível na tabela para guardar o dado, de maneira que ele possa ser recuperado sem muito problema quando necessário.

Veremos três variantes desta técnica. Nos três casos, utilizaremos o mesmo exemplo que consistirá da inserção dos nomes antônio, carlos, douglas, célio, armando, zoroastro, átila, alfredo, nesta ordem, ainda utilizando a função de espalhamento baseada na primeira letra. A tabela terá os parâmetros $s = 1$ e $b = 26$.

- *Reespalhamento linear:* se houver colisão, procura-se a próxima posição consecutiva livre da tabela, ou seja, são utilizados os índices $(f(x) + i) \bmod b, (i = 0, 1, \cdots)$. A aritmética modular é utilizada para tornar a tabela “circular”, recomeçando a busca desde o início, se necessário.

  Os resultados do exemplo estão indicados na coluna (a) da Fig. 10.1.

  Um dos problema criados por este tipo de tratamento é o aparecimento de aglomerações de entradas em partes da tabela.

- *Reespalhamento quadrático:* a busca de uma posição livre segue a fórmula $(f(x) + i^2) \bmod b, (i = 0, 1, \cdots)$.

  Os resultados do exemplo estão indicados na coluna (b) da Fig. 10.1.

- *Reespalhamento duplo:* a busca de uma posição livre segue a fórmula $(f(x) + i \times g(x)) \bmod b, (i = 0, 1, \cdots)$, onde $g(x)$ é uma segunda função de espalhamento conveniente.

  Usando, por exemplo, uma função $g(x) = (c \bmod 3) + 1$, onde c é o código numérico da segunda letra do nome x, obteríamos para o mesmo exemplo a coluna (c) da Fig. 10.1.

Figura 10.1 Resultados das técnicas de endereçamento aberto

0	antônio
1	armando
2	carlos
3	douglas
4	célio
5	átila
6	alfredo
7	
8	
9	
...	
25	zoroastro

(a)

0	antônio
1	armando
2	carlos
3	douglas
4	átila
5	
6	célio
7	
8	
9	alfredo
...	
25	zoroastro

(b)

0	antônio
1	
2	carlos
3	douglas
4	célio
5	
6	armando
7	
8	átila
9	alfredo
...	
25	zoroastro

(c)

Note-se que os algoritmos de busca utilizados no caso de tabelas construídas desta maneira têm que seguir o mesmo cálculo para verificar se um nome pertence à tabela. Por exemplo, no caso do espalhamento quadrático, se o nome procurado for '`carolina`', o algoritmo calculará os índices: 2, $2+1^2 = 3$, $2+2^2 = 6$ e $2+3^2 = 11$, verificando que o nome não aparece em nenhuma posição, e parando a busca quando aparece uma posição livre.

Um problema importante no caso de utilização de endereçamento aberto é o de remoção de entradas da tabela. Por exemplo, se removermos da tabela (b) da Fig. 10.1 o nome '`armando`', a busca descrita acima não poderá encontrar mais o nome '`átila`'. Uma maneira de resolver o problema seria substituir um nome eliminado por uma marca especial, denominada às vezes "lápide", que indica que a posição está livre mas que não pára a busca. Assim, a remoção de '`armando`' que foi mencionada produziria:

0	antônio
1	+++++++
2	carlos
3	douglas
4	átila
5	
6	célio
7	
8	
9	alfredo
...	
25	zoroastro

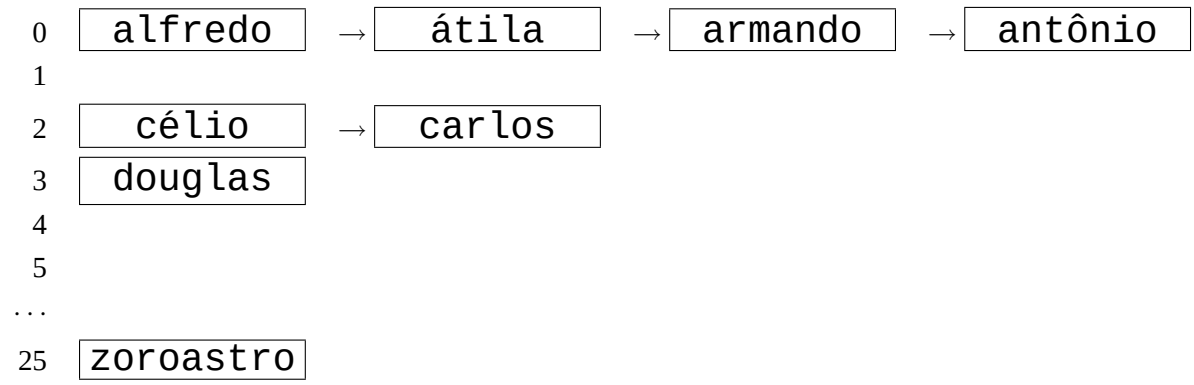
Finalmente, deve-se mencionar que esta técnica pode ser bastante eficiente, em média, mas depende de uma boa escolha da função de espalhamento e do tamanho da tabela em relação ao número de entradas ocupadas. A análise precisa desta eficiência é bastante complexa, mas pode-se mostrar que satisfeitas algumas condições, o número médio de comparações de nomes numa busca de um elemento que está na tabela é $(2 - \alpha)/(2 - 2\alpha)$, onde α denota o *fator de carga* que é a fração de posições da tabela ocupadas. Supondo uma tabela com 1000 posições, teríamos os seguintes resultados para alguns valores:

100	1,06
200	1,13
300	1,21
400	1,33
500	1,50
600	1,75
700	2,17
800	3,00
900	5,50
950	10,5

Como era de se esperar o número médio de comparações começa a crescer rapidamente quando o fator de carga começa a ser apreciável. Por outro lado, a eficiência é muito boa para fatores de carga razoáveis, até 60% ou 70%. Note-se que numa árvore binária de busca com 500 nós e de altura mínima, o número de comparações poderia chegar a 9.

10.4 Encadeamento

Uma outra técnica para tratar de colisões é o *encadeamento* que consiste em utilizar listas ligadas para manter numa única posição da tabela todas as entradas que produzem o mesmo valor da função de espalhamento. Utilizando ainda o mesmo exemplo dos nomes `antônio`, `carlos`, `douglas`, `célio`, `armando`, `zoroastro`, `átila`, `alfredo`, inseridos nesta ordem, ainda utilizando a função de espalhamento baseada na primeira letra, obteríamos a tabela:



Neste caso, as entradas da tabela são na realidade apontadores para os primeiros nós de cada lista. A implementação das listas pode utilizar as várias técnicas descritas no Cap. 3.

A eficiência desta implementação de encadeamento também depende do tamanho b da tabela e do número de entradas existentes n. Tomando $\alpha = n/b$, pode-se mostrar que o número médio de comparações de nomes numa busca de um elemento que está na tabela é $1 + \alpha/2$. Usando o mesmo exemplo da seção anterior, podemos apresentar a seguinte tabela (note que podemos ter $\alpha > 1$):

100	1,05
200	1,10
400	1,20
500	1,25
1000	1,50
2000	2,00

Podemos ver que a degradação de eficiência com o crescimento do fator de carga é mais lenta do que no caso de endereçamento aberto. Além disto, não existe o problema de a tabela ficar totalmente preenchida. A remoção de entradas pode usar as técnicas já conhecidas em relação a listas ligadas.

10.5 Exercícios

Capítulo 11

Gerenciamento de Memória

Linguagens e sistemas apresentam duas maneiras distintas de tratar os problemas de gerenciamento da memória dinâmica: *explícita* e *implícita*. Um exemplo de gerenciamento explícito é a linguagem PASCAL em que os pseudo-procedimentos *New* e *Dispose* são utilizados pelo próprio programador para requisitar e devolver ao sistema blocos de memória dinâmica de tamanho conveniente. O sistema não verifica se as variáveis apontadoras, quando usadas, referem-se a blocos corretamente alocados através de *New*, nem se os blocos liberados através de *Dispose* são realmente desnecessários; toda a responsabilidade cabe ao programador. Mesmo nestes sistemas, deve existir um mecanismo que permite manter as informações sobre a memória já alocada e aquela disponível para alocação.

Em sistemas com gerenciamento automático ou implícito, o programador requisita blocos de memória através de construções apropriadas. O sistema é responsável pela manutenção de todas as informações e pela liberação de eventuais blocos de memória que não podem ser mais utilizados pelo programa por serem inacessíveis. Neste último caso, é realizada a tarefa de identificação e coleta de blocos de memória que já foram utilizados mas tornaram-se inacessíveis ao programa. Linguagens como LISP e PROLOG utilizam este tipo de implementação. Algumas técnicas serão vistas nas seções seguintes.

11.1 Gerenciamento elementar

Uma exemplo elementar de gerenciamento de memória explícito feito pelo próprio programador está indicado no Prog. 11.1. Supõe-se, para fins deste exemplo, que um programa em PASCAL utiliza nós de somente um tipo que contêm pelo menos um campo apontador, no caso *prox*. O programador utiliza os procedimentos *Aloca* e *Desaloca*. O primeiro, chama o pseudo-procedimento *New* somente quando necessário. Ao invés de desalocar a memória através de *Dispose*,[1] o programador chama o procedimento *Desaloca* que "coleta" os nós a serem descartados numa lista ligada global de nós disponíveis, apontada pela variável *disp*. O valor desta variável deve ser inicializado no começo da execução do programa através do procedimento *InicializaAloca*.

A abordagem deste exemplo pode ser estendida a nós de vários tipos diferentes, mas a sua implementação em PASCAL apresenta alguns problemas, como a necessidade de manutenção de várias listas disponíveis e conjuntos de procedimentos, correspondentes a cada tipo. Além disto, o aproveitamento de memória pode tornar-se ineficiente quando há disponibilidade de nós de um tipo mas não de outro.

[1] Em alguns sistemas mais antigos, o procedimento *Dispose* não fazia nada!

Veremos, nas seções seguintes, algumas maneiras de resolver este tipo de problemas. A sua programação é feita, em geral, fora do sistema de execução da linguagem PASCAL padrão que não não permite acesso ao seu sistema de gerenciamento de memória. Mesmo assim, a título de clareza, os algoritmos serão apresentados em notação mais próxima possível desta linguagem.

Programa 11.1 Gerenciamento elementar de memória

```
type
   ApReg = ↑Reg;
   Reg = record
            ...
            prox: ApReg;
            ...
            end;
var
   disp: ApReg;

procedure InicializaAloca;
begin
disp := nil
end;

procedure Aloca(var p: ApReg);
begin
if disp≠nil
   then begin  p := disp;  disp := disp↑.prox end
   else  New(p)
end;

procedure Desaloca(p: ApReg);
begin
p↑.prox := disp;  disp := p
end;
```

11.2 Contagem de referências

Uma maneira aparentemente simples de implementar o gerenciamento automático de memória dinâmica é a utilização de *contadores de referências*.[2] Esta técnica supõe que cada bloco dinâmico possui um campo adicional no qual é mantido um contador de número de apontadores (referências) a este bloco. Quando o bloco é alocado, este campo recebe o valor um. O bloco pode ser desalocado quando o valor do contador atinge o valor zero.

Neste caso, a implementação de operações que envolvem variáveis apontadoras deve ser revista. Consideremos o comando de atribuição da forma '$p := q$', onde p e q são duas variáveis apontadoras do mesmo tipo; suponhamos que as duas foram inicializadas e apontam para dois blocos de memória válidos. Neste caso, o compilador de PASCAL deveria traduzir este comando para a seguinte seqüência:

[2] Em inglês *reference counts*.

```
if q≠nil
    then q↑.refs := q↑.refs+1;
if p≠nil then
    begin
    p↑.refs := p↑.refs−1;
    if p↑.refs=0
        then DesalocaRefs(p)
    end;
p :=q
```

onde *DesalocaRefs* é uma rotina que trata de desalocar um bloco de memória. Se o bloco a ser desalocado contém apontadores, os contadores correspondentes também têm que ser atualizados, e podem aparecer outros blocos para desalocação. O Prog. 11.2 mostra como poderia ser implementada esta rotina. Note-se que ela precisa conhecer, de alguma maneira, quais são os campos apontadores existentes dentro de cada bloco. Para simplificar a exposição, suporemos que cada registro apontado tem um vetor de apontadores, descrito de maneira óbvia. Suporemos também que a memória liberada é coletada numa lista disponível como indicado na seção anterior, através do primeiro apontador.

Programa 11.2 Rotina de desalocação para contagem de referências

```
type
    ApReg = ↑Reg;
    Reg =
      record
      refs: Integer;
      ...
      numAps: Integer;
      aponts: array [1..MaxAps] of ApReg;
      ...
      end;

procedure DesalocaRefs(p: ApReg);
    var q: ApReg; i: Integer
begin
with p↑ do
    for i:=1 to numAps do
        begin
        q := aponts[i];
        if q≠nil then
            begin
            q↑.refs := q↑.refs−1;
            if q↑.refs=0 then DesalocaRefs(q)
            end
        end;
    p↑.aponts[1] := disp;
    disp := p
end;
```

Os repetidos testes de apontadores nulos poderiam ser eliminados através de um truque de programação, utilizando-se ao invés de **nil** um nó especial com valor do contador muito alto, que nunca atingisse o valor zero. Mesmo assim, o custo desta técnica é relativamente alto, pois transformou uma simples operação de atribuição numa seqüência de operações bastante complexas, de duração imprevisível, pois o procedimento *DesalocaRefs* é recursivo.

Entretanto, o problema mais grave desta técnica reside no fato de não ser aplicável no caso de estruturas de dados que têm circularidades. Por exemplo, numa lista ligada circular, os contadores de referências dos seus nós nunca atingirão o valor zero, mesmo que não haja variáveis apontando para algum nó da lista.

Este fato torna a técnica utilizável somente em situações especiais. Em situações em que o problema de chamadas recursivas não é muito freqüente, esta técnica tem a vantagem de distribuir o tempo adicional de gerenciamento de memória de maneira mais uniforme ao longo da execução dos programas.

11.3 Coleta de lixo

Uma outra técnica de implementação de gerenciamento automático de memória é comumente chamada de *coleta de lixo*.[3] Ela supõe que a memória dinâmica é alocada de acordo com as necessidades do programa até que ela seja esgotada. No momento em que esta situação ocorre, o sistema executa uma série de operações que têm por finalidade identificar todos os blocos de memória ainda acessíveis ao programa e liberar a memória dos outros blocos.

A coleta de lixo padrão envolve duas fases principais: marcação dos blocos acessíveis e compactação ou coleta dos blocos que não estão em uso. Nesta última fase, a escolha entre a compactação e a coleta depende de alguns fatores que serão comentados mais adiante. Para fins de exposição, suporemos que a memória dinâmica pode ser tratada de duas maneiras diferentes. Na primeira fase, o acesso aos blocos de memória será aquele normalmente usado em PASCAL. Na segunda fase, suporemos que a memória dinâmica é constituída por um vetor *MemDin* de registros consecutivos. A fim de simplificar a exposição inicial, suporemos também que cada registro tem um certo número de campos apontadores, conhecidos pelo programa e representado por um vetor, analogamente à forma utilizada na seção anterior. Além disto, cada registro terá um campo booleano *marca*, com valor inicial falso, para indicar se o correspondente bloco de memória já foi marcado.

A fase de marcação corresponde, basicamente, a um simples percurso em pré-ordem das estruturas de dados acessíveis a partir das variáveis do programa. O problema de visitas repetidas a um nó devidas a compartilhamento e circularidades será resolvido utilizando-se o próprio campo *marca*, analogamente à técnica usada no Prog. 9.2. O procedimento *Marcar* está apresentado no Prog. 11.3. Ele deve ser executado para cada valor apontador armazenado em alguma variável do programa. É bastante fácil verificar que o número total de chamadas do procedimento *Marcar* será igual ao número de campos apontadores acessíveis a serem percorridos. A eficiência da fase de marcação pode ser melhorada através da eliminação da recursão como ilustrado para percursos em pré-ordem na Seção 6.3. Caso seja necessário minimizar o espaço para a pilha, pode-se adaptar o algoritmo de Deutsch, Schorr e Waite de Prog. 6.8. Na realidade, este algoritmo foi desenvolvido originalmente neste contexto.

Terminada a fase de marcação, trataremos a memória dinâmica como um vetor conveniente, descrito anteriormente, a fim de recuperar a memória que não foi marcada. A sua declaração poderia ter a forma

var *MemDin*: *array*[1..*MemDinMax*] **of** *Reg*;

Uma técnica muito simples seria fazer a coleta numa lista disponível, como já foi feito nas seções anteriores, através do primeiro campo apontador. Esta alternativa está apresentada no Prog. 11.4. Utilizamos neste programa o operador &, que não existe em PASCAL, mas é análogo ao da linguagem C, e devolve o endereço da variável correspondente, isto é, um apontador. Note-se que o procedimento *Coleta* atribui novamento valores falsos a todas as marcas, para que, numa próxima coleta de lixo, todas elas tenham o valor esperado. É simples verificar que esta fase de gerenciamento tem um tempo de execução proporcional

[3] Em inglês *garbage collection*.

Programa 11.3 Algoritmo de marcação de uma estrutura

```
type
  ApReg = ↑Reg;
  Reg =
    record
    marca: Boolean;
    ...
    numAps: Integer;
    aponts: array[1..MaxAps] of ApReg;
    ...
    end;

procedure Marcar(p: ApReg);
    var  i: Integer;
begin
if p≠nil then
  with p↑ do
    if not marca then
        begin
        marca := true;
        for i:=1 to numAps do Marcar(aponts[i])
        end
end;
```

ao tamanho total da memória dinâmica, que pode ser muito maior do que o tamanho da memória realmente em uso.

Programa 11.4 Coleta de nós marcados

```
procedure Coleta;
    var i: Integer;
begin
disp := nil
for i:=1 to MemDinMax do
    with MemDin[i] do
        if marca
            then marca := false
            else  begin
                    aponts[1] := disp;
                    disp := &(MemDin[i])
                    end
end;
```

Esta técnica de coleta funciona bem em caso de registros de tipo uniforme, ou pelo menos, de mesmo comprimento. Ela pode ser adaptada para registros de tamanho variável, contanto que o sistema saiba identificar os campos apontadores de cada registro. Além disto, cada registro deverá conter uma indicação relativa ao seu comprimento para viabilizar o percurso seqüencial do procedimento *Coleta*. Por outro lado, a coleta de blocos de memória de tamanho variável traz vários problemas de gerenciamento a serem discutidos na Seção 11.4.

Uma alternativa conveniente para esta técnica de coleta é a *compactação* de memória. Ela consiste, basicamente, em deslocar todos os blocos marcados para o início (ou o fim) da memória dinâmica, deixando o espaço disponível sob a forma de um único bloco. Novamente, a fim de simplificar a exposição, suporemos

inicialmente que trata-se de registros de tipo uniforme, como no caso da coleta simples. A extensão para o caso de registros de tamanho variável será deixado para um exercício (vide Exercício 1).

Suporemos que, para fins de compactação, cada registro possui, além dos outros campos, um campo apontador auxiliar que denominaremos *destino* pois conterá o endreço para o qual o registro deverá ser movido. A compactação será implementada em três passos. No primeiro passo, serão preenchidos os campos *destino* de cada registro marcado. No segundo passo, serão modificados os campos apontadores de cada registro para refletir a posição final dos registros correspondentes. Finalmente, no terceiro passo, os registros serão movidos para a sua posição final. O Prog. 11.5 mostra os procedimentos correspondentes aos três passos. Deve-se notar que falta indicar ainda a atualização das variáveis apontadoras do programa para o valor indicado pelos campos *destino* correspondentes. Desta vez, a variável *disp* apontará para o primeiro bloco disponível; outros blocos poderão ser alocados seqüencialmente. Não é difícil, novamente, verificar que a compactação tem um tempo de execução proporcional ao tamanho total da memória dinâmica, que pode ser muito maior do que o tamanho da memória realmente em uso.

Programa 11.5 Rotinas de compactação

```
type
  ApReg = ↑Reg;
  Reg =
    record
    marca: Boolean;
    destino: ApReg;
    ...
    numAps: Integer;
    aponts: array[1..MaxAps] of ApReg;
    ...
    end;

var disp: ApReg;

procedure Atualiza;
  var i,k: Integer;
begin
for i:=1 to MemDinMax do
  with MemDin[i] do
    if marca then
      for k:=1 to numAps do
        aponts[k] := aponts[k]↑.destino;
end;

procedure CalculaDestino;
    var i,j: Integer;
begin
j := 1;
for i:=1 to MemDinMax do
    with MemDin[i] do
        if marca then
            begin
            destino := &(MemDin[j];)
            j := j+1
            end;
disp := &(MemDin[j])
end;

procedure Move;
    var i: Integer;
begin
for i:=1 to MemDinMax do
  with MemDin[i] do
        if marca then
            begin
            marca := false;
            destino↑ := MemDin[i];
            end
end;
```

11.4 Alocação de blocos de memória variáveis

Conforme foi visto nas seções anteriores, o resultado da coleta de memória disponível pode ser uma lista ligada de blocos que não estão em uso, ou então um bloco compactado único de memória cujo tamanho é o máximo possível, pois inclui toda a memória disponível naquele momento.

A obtenção de uma lista ligada é mais simples e eficiente, e é uma solução muito boa no caso de blocos de tamanho fixo.[4] No caso de blocos de tamanho variável, pode-se chegar a uma situação em que o total de memória disponível seria suficiente para uma dada requisição, mas nenhum bloco existente é suficientemenete grande.

Mesmo que não ocorra este caso extremo, pode ser freqüente a situação em que o bloco encontrado na lista disponível é maior do que necessário. Se ele for alocado na íntegra, haverá um desperdício de memória. Se a política de alocação for a de quebrar o bloco em dois, alocando o tamanho exato e recolocando a sobra do bloco na lista disponível, haverá a tendência de *fragmentação* de memória, isto é, formação de blocos muito pequenos com pouca probabilidade de reutilização. Esta tendência poderá ser especialmente acentuada se o critério de alocação for o de *melhor candidato*,[5] isto é, de procurar o bloco de tamanho mínimo, mas maior ou igual ao necessário. Além disto, a busca deste candidato pode levar um tempo excessivo.

Na prática, observa-se que o critério de *primeiro candidato*,[6] em que procura-se o primeiro bloco da lista de tamanho maior ou igual ao necessário, funciona melhor. Para prevenir a proliferação de blocos pequenos ele pode ser complementado com a idéia de não quebrar mais blocos que deixariam sobras menores do que um valor pré-determinado. Uma outra modificação neste esquema consiste em tornar a lista disponível circular e iniciar a busca de um novo bloco sempre na posição seguinte na lista àquela utilizada numa alocação anterior, evitando a acumulação de blocos pequenos no início da lista. Este tipo de estratégia recebe o nome de seleção do *próximo candidato*.[7] Mesmo assim, ele pode acabar com uma lista disponível contendo um número grande de blocos pequenos.

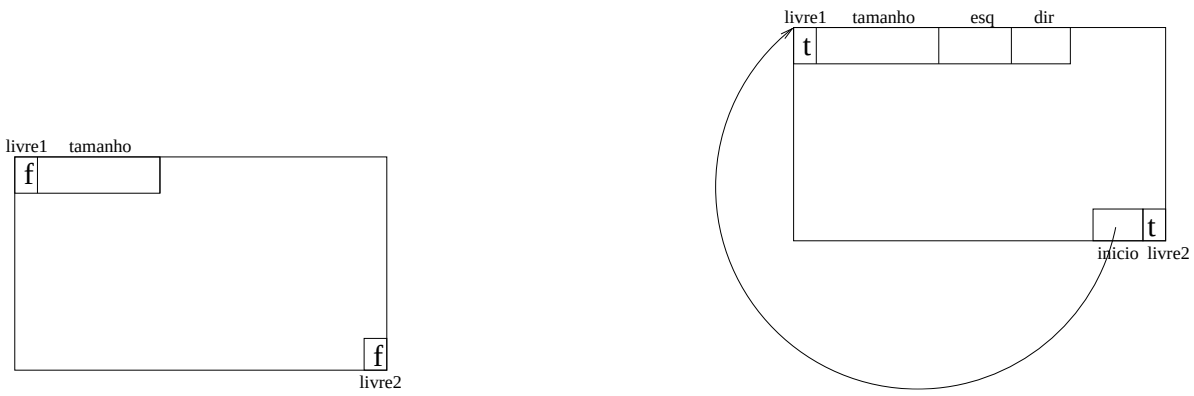
Este esquema de lista disponível pode ser melhorado significativamente juntando-se, sempre que possível, blocos contíguos que não são reconhecidos como um único bloco. Entretanto, a identificação de blocos contíguos pode ser bastante ineficiente. Uma das técnicas propostas para resolver o problema é a de *marcas de fronteira*.[8] Neste caso, blocos de memória em uso e disponíveis são representados como indicado na Fig. 11.1.

Nesta representação, os campos *tamanho* e *livre1* ocorrem no início, e o campo *livre2* no fim de qualquer bloco, disponível ou não. Os campos *esq* e *dir* ocorrem após o campo *livre1* somente em blocos que não estão em uso e são usados para formar uma lista duplamente ligada de blocos disponíveis. O campo *inicio* ocorre no fim de bloco disponível, precedendo imediatamente o campo *livre2*, e aponta para o início do próprio bloco. As letras `f` e `t` denotam, respectivamente, os valores booleanos *false* e *true*. Note que os valores dos campos *livre1* e *livre2* são sempre iguais. Esta disposição dos campos permite que, dado o endereço (apontador) a um bloco, seja fácil verificar o estado e o início dos dois blocos contíguos.

As operações de alocação e desalocação são bastante eficientes. Para alocar um bloco novo, basta percorrer a lista disponível à procura de um bloco de tamanho suficiente. Se a diferença de tamanhos for abaixo de um limite pré-determinado, o bloco é alocado na íntegra. Caso contrário, a memória em

[4] Mesmo neste caso pode ser preferível realizar compactação para diminuir a movimentação de páginas da memória virtual.
[5] Em inglês *best fit*.
[6] Em inglês *first fit*.
[7] Em inglês *next fit*.
[8] Em inglês *boundary tags*.

Figura 11.1 Blocos em uso e disponível (marcas de fronteira)



excesso constitui um novo bloco que é reinserido na lista disponível e marcado de maneira apropriada. Na operação de desalocação de um bloco, a existência das marcas de fronteira permite verificar rapidamente se um ou ambos dos seus dois blocos contíguos também estão disponíveis. Conforme o caso, um ou ambos são juntados ao bloco que está sendo liberado. A existência da lista duplamente ligada permite operações eficientes de inserção e remoção na lista. Os detalhes de implementação serão deixados para um exercício (vide Exercício 2).

11.5 Alocação controlada

As idéias sugeridas na seção anterior são, em geral, bastante eficientes, mas não garantem a eliminação da fragmentação, e podem gastar um tempo excessivo na busca de próximo candidato à alocação. Uma outra idéia de alocação controlada de memória é o *sistema de blocos conjugados binários*.[9] Para simplificar a exposição, suporemos que a memória é constituída de um certo número de blocos de tamanho mínimo, numerados de 0 a $2^m - 1$, ou seja, há 2^m blocos. Suporemos também que 2, 4, 8, ..., blocos contíguos podem ser combinados em blocos maiores, seguindo a árvore binária indicada na Fig. 11.2 (caso de $m = 4$). Nesta árvore conceitual, as folhas indicam os blocos de tamanho mínimo e os nós internos correspondem a blocos obtidos pela junção de 2, 4, 8 e 16 blocos mínimos. Os valores de k são os níveis da árvore; em cada nível, o bloco tem o tamanho 2^k. Dizemos que dois blocos de mesmo tamanho são *conjugados* se os nós correspondentes da árvore são irmãos.

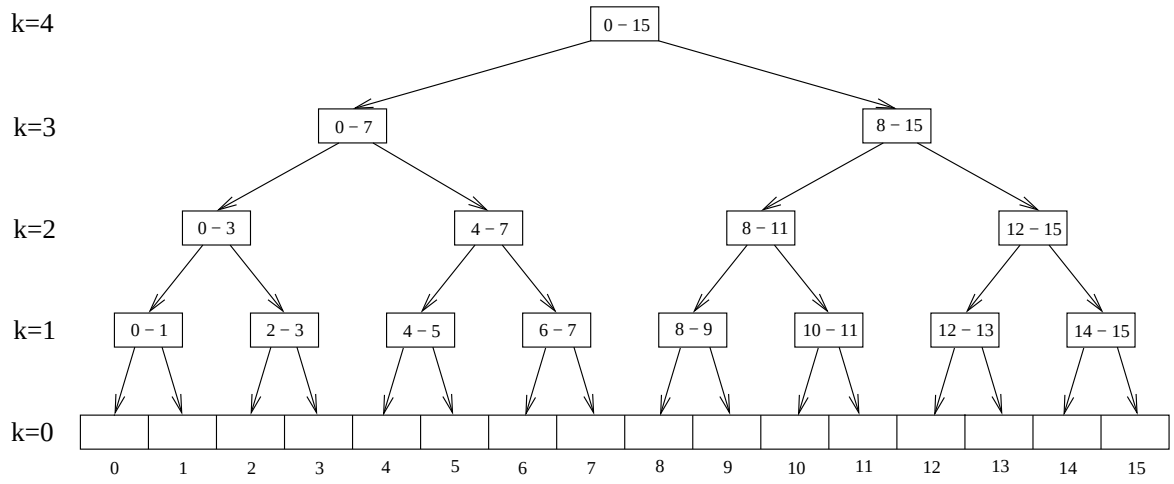
Verifica-se que é fácil reconhecer se dois índices i e j denotam duas áreas conjugadas de tamanho 2^k. É suficiente para isto tomar a representação binária dos números i e j: elas devem diferir apenas no k-ésimo *bit*, contado da direita para a esquerda, começando com zero. No exemplo indicado na figura, tomemos $i = 8$, $j = 10$ e $k = 1$; neste caso, $i_{[2]} = 1000$ e $j_{[2]} = 1010$.[10] As duas representações diferem apenas no *bit* número 1. Conclui-se que os blocos apontados por 8 e 10, de tamanho 2, são conjugados, podendo ser juntados num único bloco de tamanho 4, fato este confirmado pela figura. Tomemos um outro exemplo,

[9] Em inglês *binary buddy system*.

[10] A notação $n_{[2]}$ indica aqui a representação do número n em base 2.

com $i = 4$, $j = 8$ e $k = 2$. Neste caso, $i_{[2]} = 0100$ e $j_{[2]} = 1000$, e as duas representações diferem em mais de um *bit*, não correspondendo a blocos conjugados. A figura confirma que os blocos 4 e 8 de tamanho 4 não correspondem a irmãos na árvore.

Figura 11.2 Árvore para o sistema de blocos conjugados com $m = 4$



A alocação e desalocação de blocos segue um esquema relativamente simples. Somente podem ser alocados os blocos que fazem parte da árvore. Os blocos disponíveis estão arranjados em $m + 1$ listas duplamente ligadas; a k-ésima lista contém os blocos disponíveis de tamanho 2^k. Os apontadores para os nós cabeça das listas formam um vetor numerado de 0 a m. Numa operação de alocação que necessita de tamanho n, aloca-se um bloco de tamanho $k = \lceil \log_2 n \rceil$. Se não existir um bloco deste tamanho, procura-se um bloco de tamanho 2^{k+1} que é quebrado em dois blocos conjugados de tamanho 2^k. Caso este também não exista, aplica-se uma operação semelhante a um bloco de tamanho 2^{k+2}, etc. Para inicializar o sistema, é formado um único bloco de tamanho 2^m.

Numa operação de desalocação de um bloco de tamanho 2^k, verifica-se se o seu bloco conjugado está disponível; se estiver, os dois blocos são juntados, formando um bloco de tamanho 2^{k+1}. Caso o bloco conjugado deste novo bloco também esteja disponível, é formado um novo bloco de tamanho 2^{k+2}, e o processo é repetido. O bloco final é inserido na lista correspondente ao seu tamanho. Para que estes algoritmos possam ser executados, cada bloco, em uso ou disponível, deverá conter certas informações conforme indicado na Fig. 11.3. Os detalhes de implementação serão deixados para um exercício (vide Exercício 3). Note-se que a árvore da Fig. 11.2 é apenas conceitual e não existe fisicamente na memória durante a execução dos algoritmos.

Figura 11.3 Blocos em uso e disponível (sistema conjugado)

Em uso Disponível

11.6 Exercícios

1. Esboce a implementação de compactação de memória para blocos de tamanho variável, conforme sugerido na Seção 11.3.

2. Esboce a implementação do esquema de alocação e desalocação de blocos utilizando marcas de fronteira, conforme explicado na Seção 11.4.

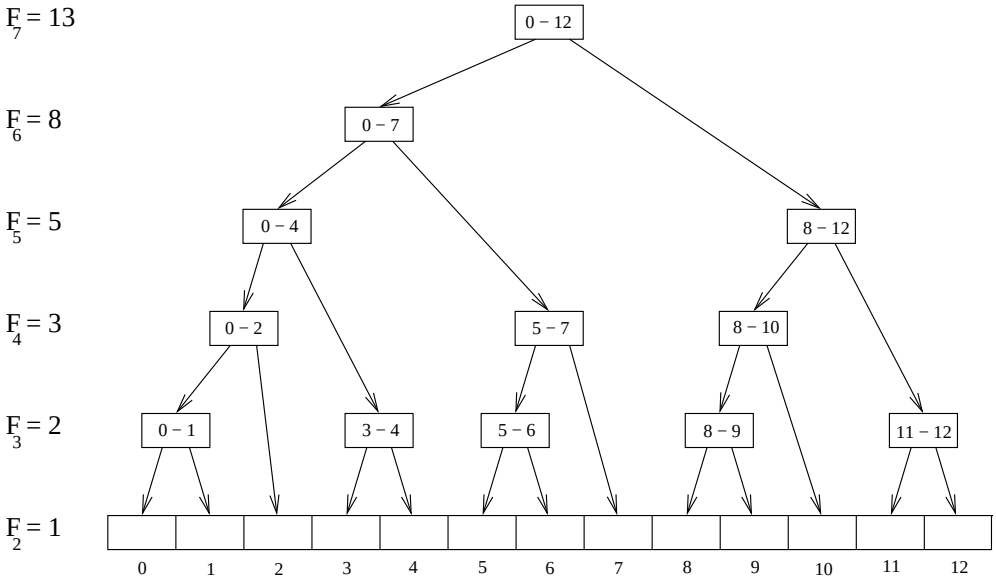
3. Escreva as rotinas para implementar a alocação e a desalocação de memória utilizando o sistema de blocos conjugados explicado na Seção 11.5.

4. O sistema de blocos conjugados binários como explicado na Seção 11.5 supõe que a memória dinâmica tem tamanho 2^m para algum m. Sugira como implementar o sistema quando isto não acontece. Sugestão: Adote uma maneira diferente de inicializar as listas dos blocos disponíveis; lembre que todo número positivo pode ser expresso como uma soma de potências distintas de 2.

5. O esquema de blocos conjugados utiliza o fato de que dois blocos contíguos de tamanho 2^k podem ser juntados num bloco de tamanho 2^{k+1}. Uma outra idéia seria utilizar os números de Fibonacci que têm a propriedade $F_n = F_{n-1} + F_{n-2}$.[11] Nesta versão do esquema, um bloco de nível n seria obtido pela junção de dois blocos, um de nível $n-1$ e outro de nível $n-2$. Complete os detalhes deste esquema e esboce as rotinas correspondentes. A Fig. 11.4 ilustra esta técnica para memória disponível de 13 blocos ($n = 7$).

[11] Lembre que $F_0 = 0$ e $F_1 = 1$.

Figura 11.4 Árvore para blocos conjugados de Fibonacci com $F_7 = 13$



Capítulo 12

Processamento de Cadeias e Textos

12.1 Representação de cadeias

12.2 Noções de compactação de cadeias

12.3 Busca de padrões

12.4 Noções de criptografia

12.5 Exercícios

Capítulo 13

Algoritmos de Ordenação

Neste capítulo serão cobertos vários algoritmos de ordenação. A ordenação é um dos problemas fundamentais em Computação e existem vários algoritmos para resolvê-lo. Pode-se demonstrar que um algoritmo baseado apenas em comparações de chaves exige da ordem de $\mathrm{O}(n \log n)$ operações de comparação, no pior caso.

A fim de uniformizar e simplificar a apresentação, suporemos em todo este capítulo que deseja-se ordenar um vetor de dados especificado pelo tipo *Vetor* e utilizaremos o procedimento auxiliar *Troca*:

```
const nMax = ...;

type
    Indice = 0 .. nMax;
    Vetor = record
        dados: array [1..nMax] of T;
        n: Indice  { Comprimento corrente }
    end;

procedure Troca(var x, y: T);
    var  t: T;
begin  t := x;  x := y;  y := t  end;
```

Deve-se observar que cobriremos métodos de ordenação aplicáveis a seqüências representadas na memória principal (ordenação interna). A ordenação de arquivos mantidos em dispositivos externos (ordenação externa) requer métodos especiais. Notaremos quando o método pode ser adaptado para esta aplicação.

13.1 Ordenação por transposição

Algoritmos de ordenação por transposição baseiam-se em trocas repetidas de elementos que estão fora de ordem. O algoritmo muito simples conhecido como *bubblesort* está apresentado no Prog. 13.1 – veja também a Seção 1.1. Já sabemos que este algoritmo realiza da ordem de $\mathrm{O}(n^2)$ operações, ou seja, não é muito adequado, exceto para sequüências de poucas dezenas de elementos.

Um outro algoritmo de ordenação por transposição é conhecido como *quicksort*. A análise de sua eficiência é bastante complexa, mas pode-se demonstrar que, o número médio de operações realizadas é da ordem de $\mathrm{O}(n \log n)$ enquanto que no pior caso este número é $\mathrm{O}(n^2)$. Apesar disto, é um dos métodos

Programa 13.1 Algoritmo *bubblesort*

```
procedure BubbleSort(var v: Vetor);
        var  i,j: Indice;
begin
with v do
    begin
    for i:=n downto 2 do
        begin
        { Troca vizinhos fora de ordem e deixa, em definitivo, em dados[i] }
        { o máximo de dados[1..i] }
        for j:=1 to i-1 do
            if dados[j]>dados[j+1] then Troca(dados[j],dados[j+1]);
        end
    end
end;
```

muito usados na prática, por ter o coeficiente de proporcionalidade relativamente pequeno. O algoritmo está apresentado no Prog. 13.2.

13.2 Ordenação por inserção

Os algoritmos por inserção supõem que uma parte da seqüência de dados já está ordenada, e inserem mais um elemento no lugar apropriado. O Prog 13.3 apresenta um exemplo deste tipo de algoritmo, denominado *inserção simples*. Pode-se demonstrar que este algoritmo executa da ordem de $\mathrm{O}(n^2)$ de operações, sendo portanto pouco recomendável sob o ponto de vista prático.

13.3 Ordenação por seleção

Os algoritmos por seleção supõem que os menores elementos da seqüência de dados já estão ordenados, buscando a seguir o menor elemento da seqüência restante para incluí-lo. O Prog 13.4 apresenta um exemplo deste tipo de algoritmo, denominado *seleção simples*. Pode-se demonstrar que este algoritmo executa da ordem de $\mathrm{O}(n^2)$ de operações, sendo também pouco recomendável sob o ponto de vista prático.

Um outro algoritmo por seleção é conhecido como *heapsort* – veja Prog. 13.5 – e utiliza as idéias da Seção 8.4. Demonstra-se que o algoritmo realiza da ordem $\mathrm{O}(n \log n)$ operações o que o torna um dos algoritmos ótimos para esta aplicação.

Programa 13.2 Algoritmo *quicksort*

```
procedure QuickSortAux(var v: Vetor; esq, dir: Indice);
        var i,j: Indice; pivot: T;
begin { Supõe esq ≤ dir }
with v do
    begin
    i := esq; j := dir; pivot := dados[(esq+dir) div 2];
    repeat
        while dados[i]<pivot do i := i+1;
        while dados[j]>pivot do j := j−1;
        if i≤j then begin
            Troca(dados[i],dados[j]);
            i := i+1; j := j−1
        end
    until i>j;
    if esq<j then QuickSortAux(v,esq,j);
    if dir>i then QuickSortAux(v,i,dir)
    end
end;

procedure QuickSort(var v: Vetor);
    var i: Indice;
begin
QuickSortAux(v,1,v.n)
end;
```

13.4 Ordenação por intercalação

Os algoritmos desta classe fazem a intercalação de seqüências já ordenadas de comprimentos crescentes. Demonstra-se que os algoritmos realizam da ordem $\mathrm{O}(n \log n)$ operações o que os torna algoritmos ótimos para esta aplicação. Apresentaremos duas versões de ordenação por intercalação: a iterativa e a recursiva. Para aplicações na memória, a primeira é certamente mais adequada. Entretanto, a segunda pode ser reformulada facilmente para ordenação de arquivos externos. Os algoritmos estão apresentados nos Progs. 13.6 e 13.7

13.5 Ordenação por distribuição

Programa 13.3 Ordenação por inserção simples

```
procedure Insercao(var v: Vetor);
        var  i,j: integer; p: T;
begin
with v do
    begin
    for i:=1 to n-1 do
        begin
        { insere dados[i + 1] em dados[1..i] }
        p := dados[i+1]; j := i;
        while (j>=1) and (p<dados[j]) do
            begin dados[j+1] := dados[j]; j := j-1 end;
        dados[j+1] := p
        end
    end
end;
```

13.6 Exercícios

Programa 13.4 Ordenação por seleção simples

```
procedure Selecao(var v: Vetor);
    var  i,j, p: integer;
begin
with v do
    begin
    for i:=1 to n−1 do
        begin
        { coloca em dados[i] o mínimo de dados[i..n − 1] }
        p := i;
        for j:=i+1 to n do
            if dados[j]<dados[p] then p :=j;
        Troca(dados[i],dados[p])
        end
    end
end;
```

Programa 13.5 Algoritmo *heapsort*

```
procedure DesceRaiz(var v: Vetor; raiz,ultimo: Indice);
{ Supõe que subárvores da raiz constituem 'heaps' }
   var  x: T; i,j: Indice; continua: boolean;
begin
with v do
   begin
   x := dados[raiz];  continua := true;
   j := raiz;  i := 2*j;
   while continua and (i<=ultimo) do
      begin
      if (i<ultimo) and (dados[i]<dados[i+1])
         then  i := i+1;
      if x<dados[i]
         then begin dados[j] := dados[i]; j := i; i := 2*i end
         else continua := false
      end;
   dados[j] := x
   end
end;

procedure HeapSort(var v: Vetor);
   var  i: Indice;
begin
with v do
   begin
         for i:= n div 2 downto 1 do DesceRaiz(v,i,n);
         for i:= n downto 2 do
            begin  Troca(dados[1],dados[i]);  DesceRaiz(v,1,i-1) end
   end
end;
```

Programa 13.6 Ordenação iterativa por intercalação

```
procedure IntercalaIteAux(var v,w: Vetor; esq, dir, ld: Indice);
    var i,j,k: Indice;
begin
{ Intercala os vetores v.dados[esq..dir − 1] e v.dados[dir..ld − 1] }
{ em w.dados[esq..ld − 1] }
i := esq; j := dir; k := esq;
while (i<dir) and (j<ld) do begin
    if v.dados[i]≤v.dados[j]
        then begin w.dados[k] := v.dados[i]; i := i+1 end
        else begin w.dados[k] := v.dados[j]; j := j+1 end;
    k := k+1
end;
while (i<dir) do
    begin w.dados[k] := v.dados[i]; i := i+1; k := k+1 end;
while (j<ld) do
    begin w.dados[k] := v.dados[j]; j := j+1; k := k+1 end;
end;

procedure IntercalaIterativo(var v: Vetor);
    var d, esq, dir, ld: integer; par: boolean; w: Vetor;
begin
{ Ordena de 2 em 2, de 4 em 4, ..., usando intercalação }
with v do begin
    d :=1; par := false; w.n := v.n;
    while d<n do begin
        esq := 1; par := not par;
        repeat
            dir := esq+d; ld := dir+d;
            if dir>n
                then begin dir := n+1; ld := n end { direito vazio }
                else if ld>(n+1) then ld := n+1;
            if par
                then IntercalaIteAux(v,w,esq,dir,ld)
                else IntercalaIteAux(w,v,esq,dir,ld);
            esq := dir+d;
        until esq>n;
        d := 2∗d
        end;
    if par then dados := w.dados;
    end
end;
```

Programa 13.7 Ordenação recursiva por intercalação

```
procedure IntercalaRecAux(var u,v,w: Vetor);
    var  i,j,k: Indice;
begin
{ Intercala os vetores u e v, deixando resultados em w }
with w do begin
    i := 1;  j := 1; n := u.n+v.n;
    for k:=1 to n do begin
        if (i≤u.n) and (j≤v.n)
            then if u.dados[i]≤v.dados[j]
                    then begin dados[k] := u.dados[i]; i := i+1 end
                    else begin dados[k] := v.dados[j]; j := j+1 end
            else  if i≤u.n
                    then begin dados[k] := u.dados[i]; i := i+1 end
                    else begin dados[k] := v.dados[j]; j := j+1 end
        end
    end
end;

procedure IntercalaRecursivo(var v: Vetor);
        var i: Indice; v1,v2: Vetor;
begin
with v do begin
    if n>1 then begin
        v1.n := n div 2; v2.n := n−v1.n;
        for i:=1 to v1.n do v1.dados[i] := dados[i];
        for i:=1 to v2.n do v2.dados[i] := dados[i+v1.n];
        IntercalaRecursivo(v1); IntercalaRecursivo(v2);
        IntercalaRecAux(v1,v2,v)
        end;
    end
end;
```

Capítulo 14

Programação Orientada a Objetos

14.1 Conceitos básicos

O Prog. 14.1 (com continuação em 14.2) traz um exemplo de utilização do paradigma de orientação a objetos, no contexto da linguagem *TurboPascal*. O exemplo ilustra, de maneira bastante elementar, conceitos de *herança* e de *métodos virtuais*.

14.2 Implementação

14.3 Exercícios

Programa 14.1 Exemplo de utilização de objetos *(continua)*

```
program Figuras;

type

  TFigura = class
    posx,posy: Real;
    function Area: Real; virtual; abstract;
    procedure Desenha(escala: Real); virtual; abstract;
  end;

  TRetangulo = class(TFigura)
    larg,alt: Real;
    function Area: Real; override;
    procedure Desenha(escala: Real); override;
  end;

  TCirculo = class(TFigura)
    raio: Real;
    function Area: Real; override;
    procedure Desenha(escala: Real); override;
  end;

  procedure TRetangulo.Desenha(escala: Real);
  begin
  {Este procedimento está incompleto}
  Writeln('Desenho de um retângulo de largura ',larg:5:2,
          ' e altura ',alt:5:2);
  Writeln('Posição: ',posx:5:2,',',posy:5:2)
  end;

  function TRetangulo.Area: Real;
  begin
  Area := larg*alt
  end;

  function TCirculo.Area: Real;
  begin
  Area := Pi*sqr(raio)
  end;
```

Programa 14.2 Exemplo de utilização de objetos *(continuação)*

```
    procedure TCirculo.Desenha(escala: Real);
    begin
    {Este procedimento está incompleto}
    Writeln('Desenho de um círculo de raio ',raio:5:2);
    Writeln('Posição: ',posx:5:2,',',posy:5:2)
    end;

    procedure DesenhaImprimeArea(f: TFigura);
    begin
    Writeln;
    f.Desenha(2.0);
    Writeln('Area: ',f.Area:6:2)
    end;

procedure Translacao(f: TFigura; dx,dy: Real);
    begin
    with f do
        begin
        posx := posx+dx;
        posy := posy+dy
        end
    end;

var
    fig1,fig2: TFigura;
    ret: TRetangulo;
    circ: TCirculo;
begin
    ret := TRetangulo.Create;
    ret.posx := 0.0;  ret.posy := 0.0;
    ret.larg := 5.0;  ret.alt := 10.0;
    circ := TCirculo.Create;
    circ.posx := 3.0;  circ.posy := 10.0;
    circ.raio := 5.0;
    fig1 := ret;
    fig2 :=circ;
    Translacao(ret,2.0,3.0);
    Translacao(fig2,3.0,1.0);
    DesenhaImprimeArea(fig1);
    DesenhaImprimeArea(fig2)
end.
```