À

Maria, esposa e companheira,

Patricia e Paula, nossas filhas.

Dados Internacionais de Catalogação na Publicação (CIP)
(Cimara Brasileira do Livro, SP, Brasil)

Ziviani, Nivio
Projeto de algoritmos com implementações Pascal • C / Nivio Ziviani. -- **4. ad. -- São Paulo :** Pioneira, **1999. --** (Pioneira Informática)

Bibliografia.
ISBN 85-221-

1. Algoritmos 2. C (Linguagem de programação para computadores) 3. Dados - Estruturas (Ciência da computação). 4. PASCAL (Linguagem de programação para computadores) I. Título.II. Série

98-5286 **CDD-005.1**

Índices para catálogo sistemático:

1. Algoritmos : **Computadores : Programação : Processamento de dados 005.1**

Projeto de Algoritmos

Com Implementações em Pascal e C

PIONEIRA INFORMÁTICA

Projeto de Algoritmos

Com Implementações em Pascal e C

Nivio Ziviani, Ph.D.

Professor Titular
Universidade Federal de Minas Gerais
Departãmento de Ciência dã Computação

4[4] Edição

EDITORA PIONEIRA
São Paulo

Este livro foi composto, revisado.
• paginado pelo autor.
A Pioneira, a partir dos fotolitos, imprimiu-o.

Capa do
Riccardo Fanucchi

02515-050 — Praça Dirceu de Lima, 313
Telefone: (011) **858-3199 —** Fax: (011) **858-0443 — São Paulo — SP**
e-mail: pioneira@editorapioneira.com.br

Impresso no
Brasil *Printed in*
Brazil

Prefácio

Este livro apresenta uma introdução ao estudo de algoritmos computacionais. As principais técnicas de projeto de algoritmos são ensinadas através da explicação detalhada de algoritmos e estruturas de dados para o uso eficiente do computador. Estas explicações são mantidas o mais simples possível, mas sem perder a profundidade e o rigor matemático.

O conteúdo é dirigido principalmente para ser utilizado como livro-texto em cursos sobre algoritmos e estruturas de dados. Pelo fato de apresentar muitas implementações de algoritmos práticos o texto é igualmente útil para profissionais engajados no desenvolvimento de sistemas de computação e de programas de aplicação. Os algoritmos são apresentados através de refinamentos sucessivos até o nível de uma implementação na linguagem Pascal, o que permite que qualquer pessoa com um mínimo de experiência em programação possa ler o código.

Conteúdo

O livro apresenta as principais técnicas utilizadas para a implementação de estruturas de dados básicas e de algoritmos para ordenação e pesquisa em memória primária e memória secundária. Os tópicos estão agrupados em cinco capítulos, cada um com o seguinte conteúdo: (i) conceito de algoritmo, estrutura de dados e tipo abstrato de dados, técnicas de análise de desempenho de algoritmos, linguagem Pascal; (ii) estruturas de dados básicas: listas lineares, pilhas e filas; (iii) métodos de ordenação em memória primária: por inserção, por seleção, shellsort, quicksort e heapsort, e em memória secundária: intercalação balanceada; (iv) métodos de pesquisa em memória primária: pesquisa seqüencial, pesquisa binária, árvores de pesquisa e *hashing; (v)* métodos de pesquisa em memória secundária: seqüencial indexado e árvores B.

O estudo do comportamento dos algoritmos tem um papel decisivo no projeto de algoritmos eficientes. Por isso, são apresentadas informações sobre as características de desempenho de cada algoritmo apresentado. Entretanto, a parte matemática utilizada para apresentar os resultados analíticos

é autocontida e exige muito pouco conhecimento matemático prévio para ser entendida.

A linguagem de programação utilizada para apresentação do refinamento final dos algoritmos apresentados é a linguagem Pascal. A vantagem de se usar a linguagem Pascal é que os programas se tornam fáceis de ser lidos e de ser traduzidos para outras linguagens. Além disso, todos os algoritmos implementados na linguagem Pascal são também implementados na linguagem C. Todo programa Pascal de um capítulo tem um programa C correspondente no apêndice.

Ao Leitor

O material apresentado é adequado para ser utilizado como livro texto em cursos de graduação em Ciência da Computação e em cursos de extensão para formação de Programadores na área de Algoritmos e Estruturas de Dados. E recomendável que o estudante já tenha tido um curso de programação (ou experiência equivalente) em uma linguagem de alto nível, tal como Pascal ou C, assim como conhecimentos de utilização de sistemas de computação.

Versões anteriores deste livro foram utilizadas na Universidade Federal de Minas Gerais. A disciplina Algoritmos e Estruturas de Dados II do Curso de Bacharelado em Ciência da Computação, com carga horária de 60 horas e um semestre de duração, possui a seguinte ementa: tipos abstratos de dados; introdução a análise de algoritmos; listas lineares, pilhas e filas; ordenação: seleção direta, inserção direta, shellsort, quicksort, heapsort, mergesort e radixsort; pesquisa em tabelas: seqüencial, binária e transformação de chave *(hashing);* árvores de pesquisa: sem balanceamento, com balanceadamento, tries e patricia. Os tópicos ordenação externa, pesquisa em memória secundária e um estudo mais elaborado de análise de algoritmos fazem parte da disciplina Algoritmos e Estruturas de Dados III, do mesmo Curso.

Ao final de cada capítulo são incluídos exercícios. Alguns exercícios são do tipo questões curtas, para testar os conhecimentos básicos sobre o material apresentado. Outros exercícios são do tipo questões mais elaboradas, podendo exigir do leitor um trabalho de vários dias, devendo ser realizado em casa ou em laboratório. Assim, os exercícios propostos devem ser utilizados em testes e trabalhos práticos para avaliação da aprendizagem.

Este texto pode também ser utilizado como manual para programadores que já tenham familiaridade com o assunto, pois são apresentadas implementações de algoritmos de utilidade geral. Os algoritmos propostos são completamente implementados nas linguagens Pascal e C e as operações envolvidas são descritas através da apresentação de exemplos de execução.

Agradecimentos

Uma versão inicial deste texto foi escrita para ser usada no Curso Estruturas de Dados e Algoritmos da I Escola Brasileiro-Argentina de Informática em fevereiro de 1986, publicada pela Editora da Unicamp sob o título *Projeto de Algoritmos e Estruturas de Dados.* Gostaria de agradecer a Carlos José Pereira de Lucena e Routo Terada por lembrarem do meu nome para participar da I Escola Brasileiro-Argentina de Informática, o que motivou o desenvolvimento da semente deste texto. Gostaria de agradecer a Cilio Rosa Ziviani, Cleber Hostalácio de Melo, José Monteiro da Mata, Lilia Tavares Mascarenhas, Luiz Carlos de Abreu Albuquerque, Regina Helena Bastos Cabral e Rosângela Fernandes pelas contribuições para a primeira versão do texto.

Muitos amigos e colegas me auxiliaram na. elaboração deste livro. Agradeço a todos pela ajuda e pelas críticas construtivas. O Departamento de Ciência da Computação da Universidade Federal de Minas Gerais tem proporcionado um excelente ambiente de trabalho. Os meus alunos de extensão, graduação, especialização e pós-graduação, especialmente os alunos das disciplinas Técnicas de Programação, Algoritmos e Estruturas de Dados e Projeto e Análise de Algoritmos contribuíram significativamente.

Vários erros foram corrigidos como conseqüência da leitura cuidadosa de várias pessoas, em especial Alberto Henrique Frade Laender, Eduardo Fernandes Barbosa, José Nagib Cotrim Arabe, Márcio Luiz Bunte de Carvalho, Osvaldo Sérgio Farhat de Carvalho, Roberto Márcio Ferreira de Souza e Virgílio Augusto Fernandes Almeida, aos quais gostaria de registrar meus agradecimentos. Gostaria de agradecer a Cristina Duarte Murta pela leitura crítica de todo o texto, pelos testes dos programas Pascal e pela execução dos programas que permitiu o estudo comparativo dos algoritmos de ordenação.

A versão C dos algoritmos existe graças ao trabalho paciente de tradução dos programas Pascal conduzido -por Maurício Antônio de Castro Lima e Wagner Toledo Corrêa, realizado com o auxílio do programa *p2c* para tradução automática de programas em Pascal para programas em C, desenvolvido por Dave Gillespie, do California Institute of Technology, EUA. O livro foi formatado com LATEX, um conjunto de macros para o TEX. Um agradecimento todo especial para Márcio Luiz Bunte de Carvalho pela imensa ajuda durante todo o trabalho de formatação, incluindo a criação de ambientes especiais em LATEX para este texto, sendo que esta etapa contou também com a ajuda de Murilo Silva Monteiro.

Nivio Ziviani
Belo Horizonte
Dezembro de 1992
Endereço Internet: nivio@dcc.ufmg.br

Sumário

Lista de Figuras

Lista de Tabelas

Lista de Programas

Capítulo 1

Introdução

1.1 Algoritmos, Estruturas de Dados e Programas

Os algoritmos fazem parte do dia-a-dia das pessoas. As instruções para o uso de medicamentos, as indicações de como montar um aparelho qualquer, uma receita de culinária são alguns exemplos de algoritmos. Um algoritmo pode ser visto como uma seqüência de ações executáveis para a obtenção de uma solução para um determinado tipo de problema. Segundo Dijkstra (1971) um **algoritmo** corresponde a uma descrição de um padrão de comportamento, expresso em termos de um conjunto finito de ações. Ao executarmos a operação $a + b$ percebemos um mesmo padrão de comportamento, mesmo que a operação seja realizada para valores diferentes de a e *b.*

Estruturas de dados e algoritmos estão intimamente ligados. Não se pode estudar estruturas de dados sem considerar os algoritmos associados a elas, assim como a **escolha** dos **algoritmos** em geral depende da representação e da estrutura dos dados. Para resolver um problema é necessário escolher uma abstração da realidade, em geral através da definição de um conjunto de dados que representa a situação real. A seguir deve ser escolhida a forma de representar estes dados.

A **escolha** da representação dos dados é determinada, entre outras, pelas operações a serem realizadas sobre os dados. Considere a operação de adição. Para pequenos números uma boa representação é através de barras verticais, caso em que a operação de adição é bastante simples. Já a representação através de dígitos decimais requer regras relativamente complicadas, as quais devem ser memorizadas. Entretanto, a situação se inverte quando consideramos a adição de grandes números, sendo mais fácil a representação por dígitos decimais por causa do princípio baseado no peso relativo da posição de cada dígito.

Programar é basicamente estruturar dados e construir algoritmos. De acordo com Wirth (1976, p.XII), **programas são** formulações concretas de algoritmos abstratos, baseados em representações e estruturas específicas de dados. Em outras palavras, programas representam uma classe especial de algoritmos capazes de serem seguidos por computadores.

Entretanto, um computador só é capaz de seguir programas em linguagem de máquina, que correspondem a uma seqüência de instruções obscuras e desconfortáveis. Para contornar tal problema é necessário construir linguagens mais adequadas para facilitar a tarefa de programar um computador. Segundo Dijkstra (1976), uma linguagem de programação é uma técnica de notação para programar, com a intenção de servir de veículo tanto para a expressão do raciocínio algorítmico quanto para a execução automática de um algoritmo por um computador.

1.2 Tipos de Dados e Tipos Abstratos de Dados

Em linguagens de programação é importante classificar constantes, variáveis, expressões e funções de acordo com certas características, as quais indicam o seu **tipo** de **dados.** Este tipo deve caracterizar o conjunto de valores a que uma constante pertence, ou que podem ser assumidos por uma variável ou expressão, ou que podem ser gerados por uma função (Wirth, 1976, pp.4–40).

Tipos simples de dados são grupos de valores indivisíveis, como os tipos básicos *integer, boolean, char, e real* do Pascal. Por exemplo, uma variável do tipo *boolean* pode assumir ou o valor verdadeiro ou o valor falso, e nenhum outro valor. Os tipos estruturados em geral definem uma coleção de valores simples, ou um agregado de valores de tipos diferentes. A linguagem Pascal oferece uma grande variedade de tipos de dados, como será mostrado na Seção 1.5.

Um **tipo abstrato de dados** pode ser visto como um modelo matemático, acompanhado das operações definidas sobre o modelo. 0 conjunto dos inteiros acompanhado das operações de adição, subtração e multiplicação forma um exemplo de um tipo abstrato de dados. Aho, Hoperoft e Ullman (1983), utilizam extensivamente tipos abstratos de dados como base para o projeto de algoritmos. Nestes casos a implementação do algoritmo em uma linguagem de programação específica exige que se encontre alguma forma de representar o tipo abstrato de dados, em termos dos tipos de dados e dos operadores suportados pela linguagem considerada. A representação do modelo matemático por trás do tipo abstrato de dados é realizada através de uma estrutura de dados.

Tipos abstratos de dados podem ser considerados generalizações de tipos primitivos de dados, da mesma forma que procedimentos são generalizações de operações. primitivas tais como adição, subtração e multiplicação. Da

mesma forma que um procedimento é usado para encapsular partes de um algoritmo, o tipo abstrato de dados pode ser usado para encapsular tipos de dados. Neste caso a definição do tipo e todas as operações definidas sabre ele podem ser localizadas em uma única seção do programa.

Como exemplo, considere uma aplicação que utilize uma lista de inteiros. Poderíamos definir um tipo abstrato de dados Lista, com as seguintes operações sobre a lista:

1. faça a lista vazia,

2. obtenha o primeiro elemento da lista. Se a lista estiver vazia então retorne nulo,

3. insira um elemento na lista.

Existem várias opções de estruturas de dados que permitem uma implementação eficiente para listas. Uma possível implementação para o tipo abstrato de dados Lista é através do tipo estruturado arranjo. A seguir cada operação do tipo abstrato de dados é implementada como um procedimento na linguagem de programação escolhida. Se existe necessidade de alterar a implementação do tipo abstrato de dados, a alteração fica restrita à parte encapsulada, sem causar impactos em outras partes do código..

Cabe ressaltar que cada conjunto diferente de operações define um tipo abstrato de dados diferente, mesmo que todos os conjuntos de operações atuem sobre um mesmo modelo matemático. Uma razão forte para isto é que a escolha adequada de uma implementação depende fortemente das operações a serem realizadas sobre o modelo.

1.3 Medida do Tempo de Execução de um Programa

0 projeto de algoritmos é fortemente influenciado pelo estudo de seus comportamentos. Depois que um problema é analisado e decisões de projeto são finalizadas, o algoritmo tem que ser implementado em um computador. Neste momento o projetista tem que estudar as várias opções de algoritmos a serem utilizados, onde os aspectos de tempo de execução e espaço ocupado são considerações importantes. Muitos destes algoritmos são encontrados em áreas tais como pesquisa operacional, otimização, teoria dos grafos, estatística, probabilidades, entre outras.

Na área de análise de algoritmos, existem dois tipos de problemas bem distintos, conforme apontou Knuth (1971):

(i) **Análise de um algoritmo particular.** Qual é o custo de usar um dado algoritmo para resolver um problema específico? Neste caso, características

importantes do algoritmo em questão devem ser investigadas, geralmente uma análise do número de vezes que cada parte do algoritmo deve ser executada, seguida do estudo da quantidade de memória necessária.
(ii) **Análise de uma classe de algoritmos.** Qual é o algoritmo de menor custo possível para resolver um problema particular? Neste caso, toda uma família de algoritmos para resolver um problema específico é investigada com o objetivo de identificar um que seja o melhor possível. Isto significa colocar limites para a complexidade computacional dos algoritmos pertencentes à classe. Por exemplo, é possível estimar o número mínimo de comparações necessárias para ordenar n números através de comparações sucessivas, conforme veremos mais adiante no Capítulo 3.

Quando conseguimos determinar o menor custo possível para resolver problemas de uma determinada classe, como no caso de ordenação, temos a medida da dificuldade inerente para resolver tais problemas. Ainda mais, quando o custo de um algoritmo é igual ao menor custo possível, então podemos concluir que o algoritmo é ótimo para a medida de custo considerada.

Em muitas situações podem existir vários algoritmos para resolver o mesmo problema, sendo pois necessário escolher aquele que é o melhor. Se uma mesma medida de custo é aplicada a diferentes algoritmos então é possível compará-los e escolher o mais adequado para resolver o problema em questão.

O custo de utilização de um algoritmo pode ser medido de várias maneiras. Uma delas é através da execução do programa em um computador real, sendo o tempo de execução medido diretamente. As medidas de tempo obtidas desta forma são bastante inadequadas e os resultados jamais devem ser generalizados. As principais objeções são: (i) os resultados são dependentes do compilador que pode favorecer algumas construções em detrimento de outras; (ii) os resultados dependem do *hardware; (iii)* quando grandes quantidades de memória são utilizadas, as medidas de tempo podem depender deste aspecto. Apesar disso, Gonnet e Baeza-Yates (1991, p.7) apresentam argumentos a favor de se obter medidas reais de tempo para algumas situações particulares como, por exemplo, quando existem vários algoritmos distintos para resolver um mesmo tipo de problema, todos com um custo de execução dentro de uma mesma ordem de grandeza. Assim os custos reais das operações são todos considerados, assim como os custos não aparentes tais como alocação de memória, indexação, carga, etc.

Uma forma mais adequada de se medir o custo de utilização de um algoritmo é através do uso de um modelo matemático, baseado em um computador idealizado como, por exemplo, o computador MIX proposto por Knuth (1968). 0 conjunto de operações a serem executadas deve ser especificado, assim como o custo associado com a execução de cada operação. Mais usual ainda é ignorar o custo de algumas das operações envolvidas e considerar apenas as operações mais significativas. Por exemplo, para algoritmos de

```
function Max (var A: Vetor): integer;
var i, Temp: integer;
begin
    Temp := A[1];
    for i := 2 to n do
      if Temp < A[i] then Temp := A[i];
    Max := Temp;
end.
```

Programa 1.1: Função para obter o máximo de um conjunto

ordenação consideramos o número de comparações entre os elementos do conjunto a ser ordenado e ignoramos as operações aritméticas, de atribuição e manipulações de índices, caso existam.

Para medir o custo de execução de um algoritmo é comum definir uma função de custo ou **função de complexidade** f, onde $f(n)$ é a medida de tempo necessário para executar um algoritmo para um problema de tamanho n. Seguindo Stanat e McAllister (1977), se $f(n)$ é uma medida da quantidade de tempo necessário para executar um algoritmo em um problema de tamanho n, então f é chamada função de **complexidade de tempo** do algoritmo. Se $f(n)$ é uma medida da quantidade de memória necessária para executar um algoritmo de tamanho n, então f é chamada função de **complexidade de espaço** do algoritmo. A não ser que haja uma referência explícita, f denotará uma função de complexidade de tempo daqui para frente. É importante ressaltar que a complexidade de tempo na realidade não representa tempo diretamente, mas o número de vezes que determinada operação considerada relevante é executada.

Para ilustrar alguns dos conceitos acima, considere o algoritmo para encontrar o maior elemento de um vetor de inteiros $A[1..n], n \geq 1$, implementado em Pascal, conforme mostrado no Programa 1.1. Seja f uma função de complexidade tal que $f(n)$ é o número de comparações entre os elementos de A, se A contiver n elementos. Logo

$f(n) = n - 1$, para $n > 0$.

Vamos provar que o algoritmo acima é **ótimo**.

Teorema: Qualquer algoritmo para encontrar o maior elemento de um conjunto com n elementos, $n \geq 1$, faz pelo menos $n - 1$ comparações.

Prova: Cada um dos $n-1$ elementos tem que ser mostrado, através de comparações, que é menor do que algum outro elemento. Logo $n-1$ comparações são necessárias. □

O teorema acima nos diz que se o número de comparações for utilizado como medida de custo então a função Max é ótima.

A medida do custo de execução de um algoritmo depende principalmente do tamanho da entrada dos dados. Por isso é comum considerar-se o tempo de execução de um programa como uma função do tamanho da entrada. Entretanto, para alguns algoritmos, o custo de execução é uma função da entrada particular dos dados, não apenas do tamanho da entrada. No caso da função Max do Programa 1.1 o algoritmo possui a propriedade de que o custo é uniforme sobre todos os problemas de tamanho n. Já para um algoritmo de ordenação isto não ocorre: se os dados de entrada já estiverem quase ordenados então o algoritmo pode ter que trabalhar menos.

Temos então que distinguir três cenários: melhor caso, pior caso e caso médio. 0 melhor caso corresponde ao menor tempo de execução sobre todas as possíveis entradas de tamanho n. 0 **pior caso** corresponde ao maior tempo de execução sobre todas as entradas de tamanho n. Se f *é* uma função de complexidade baseada na análise de pior caso então o custo de aplicar o algoritmo nunca. é maior do que $f(n)$.

0 caso médio (ou caso esperado) corresponde à média dos tempos de execução de todas as entradas de tamanho n. Na análise do caso esperado, **uma distribuição de probabilidades** sobre o conjunto de entradas de tamanho n é suposta, e o custo médio é obtido com base nesta distribuição. Por esta razão, a análise do caso médio é geralmente muito mais difícil de obter do que as análises do melhor e do pior caso. É comum supor uma distribuição de probabilidades em que todas as entradas possíveis são igualmente prováveis. Entretanto, na prática isto nem sempre é verdade. Por isso a análise do caso esperado dos algoritmos a serem estudados só será apresentada quando esta fizer sentido.

Para ilustrar estes conceitos considere o problema de acessar os **registros** de um arquivo. Cada registro contém uma chave única que é utilizada para recuperar registros do arquivo. Dada uma chave qualquer o problema consiste em localizar o registro que contenha esta chave. 0 algoritmo de pesquisa mais simples que existe é o que faz uma **pesquisa seqüencial.**

Este algoritmo examina os registros na ordem em que eles aparecem no arquivo, até que o registro procurado seja encontrado ou fique determinado que o mesmo não se encontra no arquivo.

Seja f uma função de complexidade tal que $f(n)$ *é o* número de registros consultados no arquivo, isto é, o número de vezes que a chave de consulta é comparada com a chave de cada registro. Os casos a considerar são:

melhor caso : $f(n) = 1$
pior caso : $f(n) = n$
caso médio : $f(n) = (n + 1)/2$

0 melhor caso ocorre quando o registro procurado é o primeiro consultado. 0 pior caso ocorre quando o registro procurado é o último consultado,

```
procedure MaxMin1 (var A: Vetor; var Max, Min: integer);
var i: integer;
begin
  Max := A[1];
  Min := A[1];
  for i := 2 to n do
    begin
    if A[i] > Max then Max := A[i];
    if A[i] < Min then Min := A[i];
    end;
end;
```

Programa 1.2: Implementação direta para obter o máximo e o mínimo

ou então não está presente no arquivo; para tal é necessário realizar n comparações.

Para o estudo do caso médio vamos considerar que toda pesquisa recupera um registro, não existindo pois pesquisa sem sucesso. Se p_i for a probabilidade de que o i-ésimo registro seja procurado, e considerando que para recuperar o i-ésimo registro sao necessárias i comparações, então

$$f(n) = 1 \cdot p_1 + 2 \cdot p_2 + 3 \cdot p_3 + \cdots + n \cdot p_n$$

Para calcular $f(n)$ basta conhecer a distribuição de probabilidades p_i. Se cada registro tiver a mesma probabilidade de ser acessado que todos os outros, então $p_i = 1/n, 1 \leq i \leq n$. Neste caso

$$f(n) = \frac{1}{n}(1 + 2 + 3 + \cdots + n) = \frac{1}{n}(\frac{n(n+1)}{2}) = \frac{n+1}{2}$$

A análise do caso esperado para a situação acima descrita revela que uma pesquisa com sucesso examina aproximadamente metade dos registros.

Finalmente, considere o problema de encontrar o maior elemento e o menor elemento de um vetor de inteiros $A[1..n], n \geq 1$. Um algoritmo simples para resolver este problema pode ser derivado do algoritmo apresentado no Programa 1.1, conforme mostrado no Programa 1.2. Seja f uma função de complexidade tal que $f(n)$ é o número de comparações entre os elementos de A, se A contiver n elementos. Logo

$$f(n) = 2(n-1), \quad \text{para } n > 0,$$

para o melhor caso, pior caso e caso médio.

O Programa 1.2 pode ser facilmente melhorado. Basta observar que a comparação $A[i] < Min$ somente é necessária quando o resultado da comparação $A[i] > Max$ é falso. Uma nova versão do algoritmo pode ser vista no Programa 1.3. Para esta implementação os casos a considerar são:

```
procedure MaxMin2 (var A: Vetor; var Max, Min: integer);
var i: integer;
begin
  Max := A[1];
  Min := A[1];
  for i := 2 to n do
    if A[i] > Max
    then Max := A[i]
    else if A[i] < Min then Min := A[i];
end;
```

Programa 1.3: Implementação melhorada para obter o máximo e o mínimo

melhor caso : $f(n) = n - 1$
pior caso : $f(n) = 2(n-1)$
caso médio : $f(n) = 3n/2 - 3/2$

O melhor caso ocorre quando os elementos de A estão em ordem crescente. O pior caso ocorre quando os elementos de A estão em ordem decrescente. No caso médio, $A[i]$ é maior do que Max a metade das vezes. Logo

$$f(n) = n - 1 + \frac{n-1}{2} = \frac{3n}{2} - \frac{3}{2}, \quad \text{para } n > 0.$$

Considerando o número de comparações realizadas, existe a possibilidade de obter um algoritmo mais eficiente para este problema? A resposta é sim. Considere o seguinte algoritmo:

1) Compare os elementos de A aos pares, separando-os em dois subconjuntos de acordo com o resultado da comparação, colocando os maiores em um subconjunto e os menores no outro, conforme mostrado na Figura 1.1, a um custo de $\lceil n/2 \rceil$[1] comparações.

2) O máximo é obtido do subconjunto que contém os maiores elementos, a um custo de $\lceil n/2 \rceil - 1$ comparações.

3) O mínimo é obtido do subconjunto que contém os menores elementos, a um custo de $\lceil n/2 \rceil - 1$ comparações.

A implementação do algoritmo acima descrito é apresentada no Programa 1.4.

[1] A função $\lceil\ \rceil$ é chamada de função teto: se x é um número real qualquer, então $\lceil x \rceil$ corresponde ao menor inteiro maior ou igual a x. Da mesma forma, a função $\lfloor\ \rfloor$ é chamada de função piso: $\lfloor x \rfloor$ corresponde ao maior inteiro menor ou igual a x. Se $e = 2.71828\ldots$ então $\lceil e \rceil = 3$, $\lfloor e \rfloor = 2$, $\lceil -e \rceil = -2$, $\lfloor -e \rfloor = -3$.

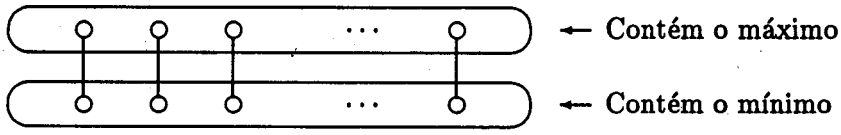


Figura 1.1: Partição de A em dois subconjuntos

```
procedure MaxMin3 (var A: Vetor; var Max, Min: integer);
var i, FimDoAnel: integer;
begin
  if (n mod 2) > 0
  then begin
         A[n+1] := A[n];
         FimDoAnel := n;
         end
  else FimDoAnel := n-1;
  if A[1] > A[2]
  then begin Max := A[1]; Min := A[2]; end
  else  begin Max := A[2]; Min := A[1]; end;
  i := 3;
  while i ≤ FimDoAnel do
    begin
    if A[i] > A[i+1]
    then begin
           if A[i] > Max then Max := A[i];
           if A[i+1] < Min then Min := A[i+1];
           end
     else begin
           if A[i] < Min then Min := A[i];
           if A[i+1] > Max then Max := A[i+1];
           end;
     i := i + 2;
     end;
end;
```

Programa 1.4: Outra implementação para obter o máximo e o mínimo

Os elementos de A são comparados dois a dois e os elementos maiores são comparados com *Max* e os elementos menores são comparados com *Min*. Quando n é ímpar o elemento que está na posição $A[n]$ é duplicado na posição $A[n+1]$ para evitar um tratamento de exceção. Para esta implementação,

$$f(n) = \frac{n}{2} + \frac{n-2}{2} + \frac{n-2}{2} = \frac{3n}{2} - 2, \quad \text{para } n > 0,$$

para o melhor caso, pior caso e caso médio.

A Tabela 1.1 apresenta uma comparação entre os algoritmos dos Programas 1.2, 1.3 e 1.4,. considerando o número de comparações como medida de complexidade. Os algoritmos MaxMin2 e MaxMin3 são superiores ao algoritmo MaxMinl de forma geral. O algoritmo MaxMin3 é superior ao algoritmo MaxMin2 com relação ao pior caso e bastante próximo quanto ao caso médio.

Os Três Algoritmos	*f (n)*		
	Melhor caso	Pior caso	Caso médio
MaxMin1	2(n-1)	2(n-1)	2(n-1)
MaxMin2	$n - 1$	2(n-1)	*3n/2-3/2*
MaxMin3	*3n/2-2*	*3n/2-2*	*3n/2-2*

Tabela 1.1: Comparação dos algoritmos para obter o máximo e o mínimo

Considerando novamente o número de comparações realizadas, existe possibilidade de obter um algoritmo mais eficiente para este problema? Para responder a esta questão é necessário conhecer o **limite inferior** para a classe de algoritmos para obter o maior e o menor elemento de um conjunto.

Uma técnica muito utilizada para obter o limite inferior para uma classe qualquer de algoritmos é através da utilização de um oráculo.[2] Dado um modelo de computação que expresse o comportamento do algoritmo o oráculo informa o resultado de cada passo possível, que no nosso caso seria o resultado de cada comparação. Para derivar o **limite inferior** o oráculo procura sempre fazer com que o algoritmo trabalhe o máximo, escolhendo como resultado da próxima comparação aquele que cause o maior trabalho possível que é necessário para determinar a resposta final.

O teorema abaixo, apresentado por Horowitz e Sahni (1978, p.476), utiliza um oráculo para derivar o limite inferior no número de comparações necessárias para obter o máximo e o mínimo de um conjunto com *n* elementos.

Teorema: Qualquer algoritmo para encontrar o maior elemento e o menor elemento de um conjunto com *n* elementos não ordenados, $n \geq 1$, faz pelo menos $[3n/2] - 2$ comparações.

Prova: A técnica utilizada define um oráculo que descreve o comportamento do algoritmo através de um conjunto de n–tuplas, mais um conjunto de regras associadas que mostram as tuplas possíveis (estados) que um algoritmo pode assumir a partir de uma dada tupla e uma única comparação.

[2]De acordo com o Novo Dicionário Aurélio da Lingua Portuguesa, um oráculo é: 1. Resposta de um deus a quem o consultava. 2. Divindade que responde consultas e orienta o crente: o oráculo de Delfos. 3. Fig. Palavra, sentença ou decisão inspirada, infalível ou que tem grande autoridade: os oráculos dos profetas, os oráculos da ciência.

O comportamento do algoritmo pode ser descrito por uma 4-tupla, representada por *(a, b, c, d),* onde a representa o número de elementos que nunca foram comparados; *b* representa o número de elementos que foram vencedores e nunca perderam em comparações realizadas; c representa o número de elementos que foram perdedores e nunca venceram em comparações realizadas; *d* representa o número de elementos que foram vencedores e perdedores em comparações realizadas. O algoritmo inicia no estado *(n,* 0, *0, 0)* e termina com (0, 1,1, *n — 2).* Desta forma, após cada comparação a tupla *(a, b, c, d)* consegue progredir apenas se ela assume um dentre os cinco estados possíveis, a saber:

$(a-2, b+1, c+1, d)$	se $a \geq 2$	{ dois elementos de a são comparados }
$(a-1, b+1, c, d)$ ou		
$(a-1, b, c+1, d)$	se $a \geq 1$	{ um elemento de a comparado com um de b ou um de c }
$(a, b-1, c, d+1)$	se $b \geq 2$	{ dois elementos de b são comparados }
$(a, b, c-1, d+1)$	se $c \geq 2$	{ dois elementos de c são comparados }

O primeiro passo requer necessariamente a manipulação do componente a. Observe que o caminho mais rápido para levar o componente a até zero requer $\lceil n/2 \rceil$ mudanças de estado e termina com a tupla $(0, n/2, n/2, 0)$, através da comparação dos elementos de a dois a dois. A seguir, para reduzir o componente b até um são necessárias $\lceil n/2 \rceil - 1$ mudanças de estado, correspondente ao número mínimo de comparações que é necessário para obter o maior elemento de b. Idem para c, com $\lceil n/2 \rceil - 1$ mudanças de estado. Logo, para obter o estado $(0, 1, 1, n-2)$ a partir do estado $(n, 0, 0, 0)$ são necessárias

$$\lceil n/2 \rceil + \lceil n/2 \rceil - 1 + \lceil n/2 \rceil - 1 = \lceil 3n/2 \rceil - 2$$

comparações. □

O teorema acima nos diz que se o número de comparações entre os elementos de um vetor for utilizado como medida de custo então o algoritmo MaxMin3 do Programa 1.4 é **ótimo**.

1.3.1 Comportamento Assintótico de Funções

Como já foi observado anteriormente, o custo para obter uma solução para um dado problema aumenta cem o tamanho *n* do problema. O número de comparações para encontrar o maior elemento de um conjunto de *n* inteiros, ou para ordenar os elementos de um conjunto com *n* elementos, aumenta com *n:* 0 parâmetro *n* fornece uma medida da dificuldade para se resolver

o problema, no sentido de que o tempo necessário para resolver o problema cresce quando *n* cresce.

Para valores suficientemente pequenos de *n*, qualquer algoritmo custa pouco para ser executado, mesmo os algoritmos ineficientes. Em outras palavras, a **escolha do algoritmo** não é um problema crítico para problemas de tamanho pequeno. Logo, a análise de algoritmos é realizada para valores grandes de *n*. Para tal considera-se o comportamento de suas funções de custo para valores grandes de n, isto é, estuda-se o comportamento assintótico das **funções de custo.** O comportamento assintótico de$f$(n) representa o limite do comportamento do custo quando n cresce.

A análise de um algoritmo geralmente conta apenas algumas operações elementares e, em muitos casos, apenas uma operação elementar. A medida de custo ou medida de complexidade relata o crescimento assintótico da operação considerada. A definição seguinte relaciona o comportamento assintótico de duas funções distintas.

Definição: Uma função $f(n)$ **domina assintoticamente** outra função $g(n)$ se existem duas constantes positivas c e m tais que, para $n \geq m$, temos $|g(n)| \leq c \cdot |f(n)|$.

O significado da definição acima pode ser expresso em termos gráficos, conforme ilustra a Figura 1.2.

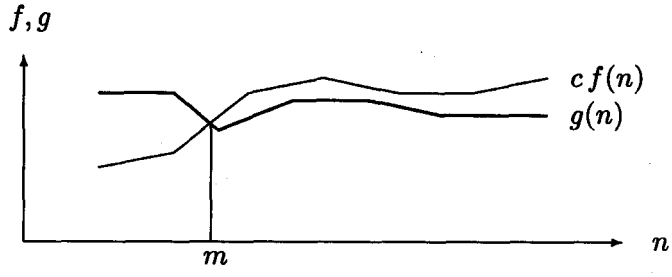


Figura 1.2: Dominação assintótica de $f(n)$ sobre $g(n)$

Exemplo: Seja $g(n) = n$ e $f(n) = -n^2$. Temos que $|n| \leq |-n^2|$ para todo n pertencente ao conjunto dos números naturais. Fazendo $c = 1$ e $m = 0$ a definição acima é satisfeita. Logo, $f(n)$ domina assintoticamente $g(n)$. Observe que $g(n)$ não domina assintoticamente $f(n)$ porque $|-n^2| > c|n|$ para todo $n > c$ e $n > 1$, qualquer que seja o valor de c.

Exemplo: Sejam $g(n) = (n+1)^2$ e $f(n) = n^2$. As funções $g(n)$ e $f(n)$ dominam assintoticamente uma a outra, desde que $|(n+1)^2| \leq 4|n^2|$ para $n \geq 1$ e $|n^2| \leq |(n+1)^2|$ para $n \geq 0$.

Knuth (1968, p.104) sugeriu uma notação para dominação assintótica. Para expressar que $f(n)$ domina assintoticamente $g(n)$ escrevemos $g(n) = O(f(n))$, onde se lê $g(n)$ é da ordem no máximo $f(n)$. Por exemplo, quando dizemos que o tempo de execução $T(n)$ de um programa é $O(n^2)$, isto significa que existem constantes c e m tais que, para valores de n maiores ou iguais a m, $T(n) \leq cn^2$.

As funções de complexidade de tempo são definidas sobre os inteiros não negativos, ainda que possam também ser não inteiros. A definição abaixo formaliza a notação de Knuth.

Definição Notação O: Uma função $g(n)$ é $O(f(n))$ se existem duas constantes positivas c e m tais que $g(n) \leq c \cdot f(n)$, para $n \geq m$.

Exemplo: Seja $g(n) = (n+1)^2$. Logo $g(n)$ é $O(n^2)$, quando $m = 1$ e $c = 4$. Isto porque $(n+1)^2 \leq 4n^2$ para $n \geq 1$.

Exemplo: A função $g(n) = 3n^3 + 2n^2 + n$ é $O(n^3)$. Basta mostrar que $3n^3 + 2n^2 + n \leq 6n^3$, para $n \geq 0$. A função $g(n) = 3n^3 + 2n^2 + n$ é também $O(n^4)$, entretanto esta afirmação é mais fraca do que dizer que $g(n)$ é $O(n^3)$.

Exemplo: Suponha $g(n) = n$ e $f(n) = n^2$. Sabemos pela definição acima que $g(n)$ é $O(n^2)$, pois para $n \geq 0$, $n \leq n^2$. Entretanto $f(n)$ não é $O(n)$. Suponha que existam constantes c e m tais que para todo $n \geq m$, $n^2 \leq cn$. Logo $c \geq n$ para qualquer $n \geq m$, e não existe uma constante c que possa ser maior ou igual a n para todo n.

Algumas **operações** que podem ser realizadas com a **notação** O são apresentadas na Figura 1.3. As provas das propriedades podem ser encontradas em Knuth (1968) ou em Aho, Hopcroft e Ullman (1983).

$$
\begin{aligned}
f(n) &= O(f(n)) \\
c \cdot O(f(n)) &= O(f(n)) \quad c = \textit{constante} \\
O(f(n)) + O(f(n)) &= O(f(n)) \\
O(O(f(n)) &= O(f(n)) \\
O(f(n)) + O(g(n)) &= O(max(f(n), g(n))) \\
O(f(n))O(g(n)) &= O(f(n)g(n)) \\
f(n)O(g(n)) &= O(f(n)g(n))
\end{aligned}
$$

Figura 1.3: Operações com a notação O

Exemplo: A regra da soma $O(f(n)) + O(g(n))$ pode ser usada para calcular o tempo de execução de uma seqüência de trechos de programas. Suponha três trechos de programas cujos tempos de execução são $O(n)$, $O(n^2)$ e $O(n \log n)$. O tempo de execução dos dois primeiros trechos é

$O(max(n, n^2))$, que é $O(n^2)$. O tempo de execução de todos os três trechos é então $O(max(n^2, n \log n))$, que é $O(n^2)$.

Exemplo: O produto de $[\log n + k + O(1/n)]$ por $[n + O(\sqrt{n})]$ é $n \log n + kn + O(\sqrt{n} \log n)$.

Dizer que $g(n)$ é $O(f(n))$ significa que $f(n)$ é um limite superior para a taxa de crescimento de $g(n)$. A definição abaixo especifica um limite inferior para $g(n)$.

Definição Notação Ω: Uma função $g(n)$ é $\Omega(f(n))$ se existir uma constante c tal que $g(n) \geq c \cdot f(n)$ para um número infinito de valores para n.

Exemplo: Para mostrar que $g(n) = 3n^3 + 2n^2$ é $\Omega(n^3)$ basta fazer $c = 1$, e então $3n^3 + 2n^2 \geq n^3$ para $n \geq 0$.

Exemplo: Seja $g(n) = n$ para n ímpar $(n \geq 1)$ e $g(n) = n^2/10$ para n par $(n \geq 0)$. Neste caso $g(n)$ é $\Omega(n^2)$, bastando considerar $c = 1/10$ e $n = 0, 2, 4, 6, \ldots$.

1.3.2 Classes de Comportamento Assintótico

Se f é uma **função de complexidade** para um algoritmo F, então $0(f)$ é considerada a **complexidade assintótica** ou o comportamento assintótico do algoritmo F. Igualmente, se g é uma função para um algoritmo G, então $0(g)$ é considerada a complexidade assintótica do algoritmo G. A relação de dominação assintótica permite comparar funções de complexidade. Entretanto, se as funções f e g dominam assintoticamente uma a outra então os algoritmos associados são equivalentes. Nestes casos, o comportamento assintótico não serve para comparar os algoritmos. Por exemplo, dois algoritmos F e G aplicados à mesma classe de problemas, sendo que F leva três vezes o tempo de G ao serem executados, isto é $f(n) = 3g(n)$, sendo que $0(f(n)) = 0(g(n))$. Logo o comportamento assintótico não serve para comparar os algoritmos F e G porque eles diferem apenas por uma constante.

Programas podem ser avaliados através da comparação de suas funções de complexidade, negligenciando as constantes de proporcionalidade. Um programa com tempo de execução $0(n)$ é melhor que um programa com tempo de execução $0(n^2)$. Entretanto, as constantes de proporcionalidade em cada caso podem alterar esta consideração. Por exemplo, é possível que um programa leve 100n unidades de tempo para ser executado enquanto um outro leve $2n^2$ unidades de tempo. Qual dos dois programas é melhor?

A resposta a esta pergunta depende do tamanho do problema a ser executado. Para problemas de tamanho $n < 50$, o programa com tempo de execução $2n^2$ é melhor do que o programa com tempo de execução 100n. Para problemas com entrada de dados pequena é preferível usar o programa cujo tempo de execução é $0(n^2)$. Entretanto, quando n cresce, o programa com tempo $0(n^2)$ leva muito mais tempo que o programa 0(n).

A maioria dos algoritmos possui um parâmetro que afeta o tempo de execução de forma mais significativa, usualmente o número de itens a ser processado. Este parâmetro pode ser o número de registros de um arquivo a ser ordenado, ou o número de nós de um grafo. As principais **classes de problemas** possuem as **funções de complexidade** descritas abaixo.

1. $f(n) = 0(1)$. Algoritmos de complexidade 0(1) são ditos de **complexidade constante**. O uso do algoritmo independe do tamanho de n. Neste caso as instruções do algoritmo são executadas um número fixo de vezes.

2. $f(n) = O(\log n)$. Um algoritmo de complexidade O(*logn)* é dito ter **complexidade logarítmica**. Este tempo de execução ocorre tipicamente em algoritmos que resolvem um problema transformando-o em problemas menores. Nestes casos, o tempo de execução pode ser considerado como sendo menor do que uma constante grande. Quando n é mil e a base do logaritmo é 2, $\log_2 n \approx 10$, quando n é um milhão, $\log_2 n \approx 20$. Para dobrar o valor de log n temos que considerar o quadrado de *n*. A base do logaritmo muda pouco estes valores: quando *n* é um milhão, o $\log_2 n$ é 20 e o $\log_{10} n$ é 6.

3. $f(n) = 0(n)$. Um algoritmo de complexidade *0(n)* é dito ter **complexidade linear.** Em geral um pequeno trabalho é realizado sobre cada elemento de entrada. Esta é a melhor situação possível para um algoritmo que tem que processar n elementos de entrada ou produzir *n* elementos de saída. Cada vez que *n* dobra de tamanho o tempo de execução dobra.

4. $f(n) = O(n \log n)$. Este tempo de execução ocorre tipicamente em algoritmos que resolvem um problema quebrando-o em problemas menores, resolvendo cada um deles independentemente e depois ajuntando as soluções. Quando n é um milhão e a base do logaritmo é 2, $n\log_2 n$ é cerca de 20 milhões. Quando *n é* dois milhões, $n\log_2 n$ *é* cerca de 42 milhões, pouco mais do que o dobro.

5. $f(n) = 0(n^2)$. Um algoritmo de complexidade $0(n^2)$ é dito ter **complexidade quadrática.** Algoritmos desta ordem de complexidade ocorrem quando os itens de dados são processados aos pares, muitas vezes em um anel dentro de outro. Quando n é mil, o número de operações é da ordem de 1 milhão. Sempre que n dobra o tempo de execução é multiplicado por 4. Algoritmos deste tipo são úteis para resolver problemas de tamanhos relativamente pequenos.

6. $f(n) = 0(n^3)$. Um algoritmo de complexidade $0(n^3)$ é dito ter **complexidade cúbica.** Algoritmos desta ordem de complexidade são úteis

apenas para resolver pequenos problemas. Quando *n* é cem, o número de operações é da ordem de 1 milhão. Sempre que *n* dobra o tempo de execução fica multiplicado por 8.

7. $f(n) = 0(2^n)$. Um algoritmo de complexidade $0(2^n)$ é dito ter **complexidade exponencial.** Algoritmos desta ordem de complexidade geralmente não são úteis sob o ponto de vista prático. Eles ocorrem na solução de problemas quando se usa **força bruta** para resolvê-los. Quando *n* é vinte, o tempo de execução é cerca de um milhão. Quando *n* dobra, o tempo de execução fica elevado ao quadrado.

Para ilustrar melhor a diferença entre as classes de comportamento assintótico Garey e Johnson (1979, p.7) apresentam o quadro mostrado na Ta-bela 1.2. Este quadro mostra a razão de crescimento de várias **funções** de **complexidade** para tamanhos diferentes de *n*, onde cada função expressa o tempo de execução em microsegundos. Um algoritmo linear executa em um segundo um milhão de operações.

Função de custo	Tamanho *n*					
	10	20	30	40	50	60
n	0,00001 seg.	0,00002 seg.	0,00003 seg.	0,00004 seg.	0,00005 seg.	0,00006 seg.
n^2	0,0001 seg.	0,0004 seg.	0,0009 seg.	0,0016 seg.	0,0025 seg.	0,0036 seg.
n^3	0,001 seg.	0,008 seg.	0,027 seg.	0,64 seg.	0,125 seg.	0,216 seg.
n^5	0,1 seg.	3,2 seg.	24,3 seg.	1,7 min.	5,2 min.	13 min.
2^n	0,001 seg.	1 seg.	17,9 min.	12,7 dias	35,7 anos	366 sec.
3^n	0,059 seg.	58 min.	6,5 anos	3855 séc.	10^8 séc.	10^{13} séc.

Tabela 1.2: Comparação de virias funções de complexidade

Um outro aspecto interessante é o efeito causado pelo aumento da velocidade dos computadores sobre os algoritmos com as funções de complexidade citadas acima. A Tabela 1.3 mostra como um aumento de 100 ou de 1000 vezes na velocidade de computação de um computador atual influi na solução do maior problema possível de ser resolvido em uma hora. Note que um aumento de 1000 vezes na velocidade de computação resolve um problema dez vezes maior para um algoritmo de complexidade $0(n^3)$, enquanto um algoritmo de complexidade $0(2^n)$ apenas adiciona dez ao tamanho do maior problema possível de ser resolvido em uma hora.

Função de custo de de tempo	Computador atual	Computador 100 vezes mais rápido	Computador 1000 vezes mais rápido
n	t_1	$100t_1$	$1000t_1$
n^2	t^2	$10t_2$	$31,6t_2$
n^3	t_3	$4,6t_3$	$10t_3$
2^n	t_4	$t_4+6,6$	t_4+10

Tabela 1.3: Influência do aumento de velocidade dos computadores no tamanho t do problema

Um algoritmo cuja função de complexidade *é* $O(c^n)$, $c > 1$, é chamado **de algoritmo exponencial** no tempo de execução. Um algoritmo cuja função de complexidade *é* $O(p(n))$, onde $p(n)$ é um polinômio, é chamado de **algoritmo polinomial** no tempo de execução. A distinção entre estes dois tipos de algoritmos torna-se significativa quando o tamanho do problema a ser resolvido cresce, conforme ilustra a Tabela 1.2. Esta é a razão por que algoritmos polinomiais são muito mais úteis na prática do que algoritmos exponenciais.

Os algoritmos exponenciais são geralmente simples variações de pesquisa exaustiva, enquanto algoritmos polinomiais são geralmente obtidos através de um entendimento mais profundo da estrutura do problema. Um problema é considerado intratável se ele é tão difícil que não existe um algoritmo polinomial para resolvê-lo, enquanto um problema é considerado bem resolvido quando existe um algoritmo polinomial para resolvê-lo.

Entretanto, a distinção entre algoritmos polinomiais eficientes e algoritmos exponenciais ineficientes possui várias exceções. Por exemplo, um algoritmo com função de complexidade $n) = 2^n$ é mais rápido que um algoritmo $g(n) = n^5$ para valores de n menores ou iguais a 20. Da mesma forma, existem algoritmos exponenciais que são muito úteis na prática. Por exemplo, o algoritmo Simplex para programação linear possui complexidade de tempo exponencial para o pior caso (Garey e Johnson, 1979), mas executa muito rápido na prática.

Infelizmente, exemplos como o do algoritmo Simplex não ocorrem com freqüência na prática, e muitos algoritmos exponenciais conhecidos não são muito úteis. Considere, como exemplo, o seguinte problema: um caixeiro viajante deseja visitar *n* cidades de tal forma que sua viagem inicie e ter-mine em uma mesma cidade, e cada cidade deve ser visitada uma única vez. Supondo que sempre exista uma estrada entre duas cidades quaisquer, o problema é encontrar a menor rota que o caixeiro viajante possa utilizar na sua viagem.

A Figura 1.4 ilustra o exemplo acima para quatro cidades c_1, c_2, c_3,c_4, onde os números nos arcos indicam a distância entre duas cidades. O per-

curso < c_1, c_3, c_4, c_2, c_1 > é uma solução para o problema, cujo percurso total tem distância 24.

Um algoritmo simples para o problema acima seria verificar todas as rotas e escolher a menor delas. Como existem $(n - 1)!$ rotas possíveis e a distância total percorrida em cada rota envolve n adições, então o número total de adições é $n!$. Para o exemplo da Figura 1.4 teriamos 24 adições. Suponha agora 50 cidades: o número de adições seria igual ao fatorial de 50, que é aproximadamente 10^{64}. Considerando um computador capaz de executar 10^9 adições por segundo, o tempo total para resolver o problema com 50 cidades seria maior do que 10^{45} séculos somente para executar as adições.

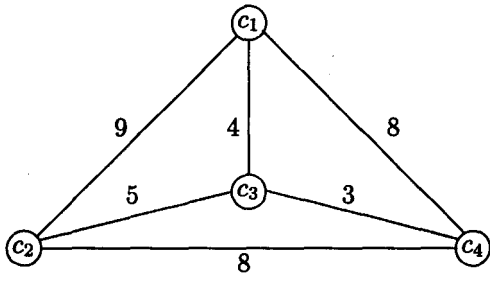


Figura 1.4: Problema do caixeiro viajante

1.4 Técnicas de Análise de Algoritmos

A determinação do tempo de execução de um programa qualquer pode se tornar um problema matemático complexo quando se deseja determinar o valor exato da função de complexidade. Entretanto, a determinação da ordem do tempo de execução de um programa, sem haver preocupação com o valor da constante envolvida, pode ser uma tarefa mais simples. E mais fácil determinar que o número esperado de comparações para recuperar um registro de um arquivo utilizando pesquisa seqüencial *é 0(n)* do que efetivamente determinar que este número é $(n + 1)/2$, quando cada registro tem a mesma probabilidade de ser procurado.

A análise de algoritmos ou programas utiliza técnicas de matemática discreta, envolvendo contagem ou enumeração dos elementos de um conjunto que possuam uma propriedade comum. Estas técnicas envolvem a manipulação de somas, produtos, permutações, fatoriais, coeficientes binomiais, solução de **equações de recorrência,** entre outras. Algumas destas técnicas serão ilustradas informalmente através de exemplos.

Infelizmente não existe um conjunto completo de regras para analisar programas. Aho, Hoperoft e Ullman (1983) enumeram alguns princípios a serem seguidos. Muitos destes princípios utilizam as propriedades sobre a notação O apresentadas na Figura 1.3. São eles:

1. 0 tempo de execução de um comando de atribuição, de leitura ou de escrita pode ser considerado como 0(1). Existem exceções para as linguagens que permitem a chamada de funções em comandos de atribuição, ou quando atribuições envolvem vetores de tamanho arbitrariamente grandes.

2. 0 tempo de execução de uma seqüência de comandos é determinado pelo maior tempo de execução de qualquer comando da seqüência.

3. 0 tempo de execução de um comando de decisão é composto pelo tempo de execução dos comandos executados dentro do comando condicional, mais o tempo para avaliar a condição, que é 0(1).

4. 0 tempo para executar um anel é a soma do tempo de execução do corpo do anel mais o tempo de avaliar a condição para terminação, multiplicado pelo número de iterações do anel. Geralmente o tempo para avaliar a condição para terminação é 0(1).

5. Quando o programa possui procedimentos não recursivos, o tempo de execução de cada procedimento deve ser computado separadamente um a um, iniciando com os procedimentos que não chamam outros procedimentos. A seguir devem ser avaliados os procedimentos que chamam os procedimentos que não chamam outros procedimentos, utilizando os tempos dos procedimentos já avaliados. Este processo é repetido até chegar no programa principal.

6. Quando o programa possui **procedimentos recursivos,** para cada procedimento é associada uma função de complexidade $f(n)$ desconhecida, onde n mede o tamanho dos argumentos para o procedimento, conforme será mostrado mais adiante.

Com o propósito de ilustrar os vários conceitos apresentados acima, vamos apresentar alguns programas e, para cada um deles, mostrar com detalhes os passos envolvidos em sua análise.

Exemplo: Considere o algoritmo para ordenar os n elementos de um conjunto A, cujo princípio é o seguinte:

1. Selecione o menor elemento do conjunto

2. Troque este elemento com o primeiro elemento A[1].

```
                procedure Ordena (var A : vetor );
                { ordena o vetor A em ordem ascendente }
                var i, j, min, x: integer;
                begin
(1)               for i := 1 to n-1 do
                  begin
(2)                 min := i;
(3)                 for j:= i+1 to n do
(4)                   if A[j] < A[min]
(5)                   then min := j;
                    {troca A[min] e A[i]}
(6)                 x:= A[min];
(7)                 A[min]:= A[i];
(8)                 A[i]:= x;
                  end;
                end;
```

Programa 1.5: Programa para ordenar

A seguir repita as duas operações acima com os $n-1$ elementos restantes, depois com os $n-2$ elementos, até que reste apenas um elemento. O Programa 1.5 mostra a implementação do algoritmo acima descrito, para um conjunto de inteiros implementado como um vetor $A[1..n]$.

O número n de elementos do conjunto representa o tamanho da entrada de dados. O programa contém dois anéis, um dentro do outro. O anel mais externo engloba os comandos de (2) a (8), sendo que o anel mais interno engloba os comandos (4) e (5). Neste caso devemos iniciar a análise pelo anel interno.

O anel mais interno contém um comando de decisão que, por sua vez, contém apenas um comando de atribuição. O comando de atribuição leva um tempo constante para ser executado, assim como a avaliação da condição do comando de decisão. Não sabemos se o corpo do comando de decisão será executado ou não: nestas situações devemos considerar o pior caso, isto é, assumir que a linha (5) será sempre executada.

O tempo para incrementar o índice do anel e avaliar sua condição de terminação também é $O(1)$, e o tempo combinado para executar uma vez o anel composto pelas linhas (3), (4) e (5) é $O(max(1,1,1)) = O(1)$, conforme a regra da soma para a notação O. Como o número de iterações do anel é $n-i$, então o tempo gasto no anel é $O((n-i) \times 1) = O(n-i)$, conforme a regra do produto para a notação O.

O corpo do anel mais externo contém, além do anel interno, os comandos de atribuição nas linhas (2), (6), (7) e (8). Logo o tempo de execução das linhas (2) a (8) é $O(max(1,(n-i),1,1,1)) = O(n-i)$. A linha (1)

```
          Pesquisa(n);
(1)       if n ≤ 1
(2)       then 'inspecione elemento' e termine
          else
          begin
(3)         para cada um dos n elementos 'inspecione elemento';
(4)         Pesquisa(n/3);
          end;
```

Programa 1.6: Algoritmo recursivo

é executada $n-1$ vezes, e o tempo total para executar o programa está limitado ao produto de uma constante pelo **somatório** de $(n-i)$, a saber

$$\begin{aligned}\sum_{1}^{n-1}(n-i) &= n(n-1)/2 \\ &= n^2/2 - n/2 \\ &= O(n^2)\end{aligned}$$

Se considerarmos o número de comparações como a medida de custo relevante (no caso representada pelo número de vezes que a linha (4) do Programa 1.5 é executada), então o programa faz $(n^2)/2 - n/2$ comparações para ordenar n elementos. Se considerarmos o número de trocas (linhas (6), (7) e (8) do Programa 1.5) como uma medida de custo relevante, então o programa realiza exatamente $n-1$ trocas.

Se existirem procedimentos recursivos então o problema deve ser tratado de forma diferente: para cada procedimento recursivo é associada uma função de complexidade $f(n)$ desconhecida, onde n mede o tamanho dos argumentos para o procedimento. A seguir obtemos uma equação de recorrência para $f(n)$, conforme será mostrado no exemplo a seguir.

Exemplo: Considere o algoritmo mostrado no Programa 1.6. O algoritmo inspeciona os n elementos de um conjunto e, de alguma forma, isso permite descartar $2/3$ dos elementos e então fazer uma chamada recursiva sobre os $n/3$ elementos restantes.

Seja $T(n)$ uma função de complexidade tal que $T(n)$ represente o número de inspeções nos elementos de um conjunto com n elementos. O custo de execução das linhas (1) e (2) é $O(1)$. O custo de execução da linha (3) é exatamente n. Quantas vezes a linha (4) é executada, isto é, quantas chamadas recursivas vão ocorrer?

Uma forma de descrever este comportamento é através de uma **relação de recorrência**. A principal característica de uma equação de recorrência

é que o termo $T(n)$ é especificado como uma função dos termos anteriores $T(1), T(2), \ldots, T(n-1)$.

No caso do algoritmo acima temos

$$T(n) = n + T(n/3) \tag{1.1}$$

onde $T(1) = 1$, isto é, para $n = 1$ fazemos uma inspeção.

Existem algumas técnicas para resolver equações de recorrência. Em alguns casos a solução de uma equação de recorrência pode ser difícil de obter. Um caminho possível para resolver a Eq.1.1 é procurar substituir os termos $T(k)$, $k < n$, no lado direito da equação até que todos os termos $T(k)$, $k > 1$, tenham sido substituídos por fórmulas contendo apenas $T(1)$. No caso da Eq.1.1 temos

$$\begin{aligned} T(n) &= n + T(n/3) \\ T(n/3) &= n/3 + T(n/3/3) \\ T(n/3/3) &= n/3/3 + T(n/3/3/3) \\ &\vdots \quad\quad \vdots \\ T(n/3/3\cdots/3) &= n/3/3\cdots/3 + T(n/3/3/3\cdots/3) \end{aligned}$$

Adicionando lado a lado, obtemos

$$T(n) = n + n\cdot(1/3) + n\cdot(1/3^2) + n\cdot(1/3^3) + \cdots + T(n/3/3\cdots/3)$$

A equação acima representa a soma de uma série geométrica de razão $1/3$, multiplicada por n, e adicionada de $T(n/3/3\cdots/3)$, que é menor ou igual a 1. Se desprezarmos o termo $T(n/3/3\cdots/3)$, quando n tende para infinito, então

$$\begin{aligned} T(n) &= n\sum_{i=0}^{\infty}(1/3)^i \\ &= n\left(\frac{1}{1-\frac{1}{3}}\right) \\ &= \frac{3n}{2} \end{aligned}$$

Se considerarmos o termo $T(n/3/3/3\cdots/3)$ e denominarmos x o número de subdivisões por 3 do tamanho do problema, então $n/3^x = 1$, e $n = 3^x$. Logo $x = log_3 n$. Lembrando que $T(1) = 1$ temos

```
procedure MaxMin4 (Linf, Lsup: integer; var Max, Min: integer);
var Max1, Max2, Min1, Min2, Meio: integer;
begin
  if Lsup - Linf ≤ 1
  then if A[Linf] < A[Lsup]
       then begin Max := A[Lsup]; Min := A[Linf]; end
       else begin Max := A[Linf]; Min := A[Lsup]; end
  else begin
       Meio := (Linf + Lsup) div 2;
       MaxMin4(Linf, Meio, Max1, Min1);
       MaxMin4(Meio+1, Lsup, Max2, Min2);
       if Max1 > Max2 then Max := Max1 else Max := Max2;
       if Min1 < Min2 then Min := Min1 else Min := Min2;
       end;
end;
```

Programa 1.7: Versão recursiva para obter o máximo e o mínimo

$$
\begin{aligned}
T(n) &= \sum_{i=0}^{x-1} \frac{n}{3^i} + T(\frac{n}{3^x}) \\
&= n \sum_{i=0}^{x-1} (1/3)^i + 1 \\
&= \frac{n(1-(\frac{1}{3})^x)}{(1-\frac{1}{3})} + 1 \\
&= \frac{3n}{2}\left(1 - \frac{1}{3^x}\right) + 1 \\
&= \frac{3n}{2} - \frac{1}{2}
\end{aligned}
$$

Logo, o Programa 1.6 é $O(n)$.

Exemplo: Considere o algoritmo para obter o maior e o menor elemento de um vetor de inteiros $A[1..n]$, $n \geq 1$, conforme mostrado no Programa 1.7. O vetor A é global ao procedimento MaxMin4. Os parâmetros Linf e Lsup são inteiros, $1 \leq \text{Linf} \leq \text{Lsup} \leq n$. O efeito produzido a cada chamada de MaxMin4 é o de atribuir às variáveis Max e a Min o maior elemento e o menor elemento em $A[\text{Linf}], A[\text{Linf}+1], \cdots, A[\text{Lsup}]$, respectivamente.

Seja f uma função de complexidade tal que $f(n)$ é o número de comparações entre os elementos de A, se A contiver n elementos. Logo

$$f(n) = 1, \qquad \text{para } n \leq 2,$$
$$f(n) = f(\lfloor n/2 \rfloor) + f(\lceil n/2 \rceil) + 2, \qquad \text{para } n > 2.$$

Quando $n = 2^i$, para algum inteiro positivo i, então

$$\begin{aligned} f(n) &= 2f(n/2) + 2 \\ 2f(n/2) &= 4f(n/4) + 2 \times 2 \\ 4f(n/4) &= 8f(n/8) + 2 \times 2 \times 2 \\ &\vdots \quad\quad \vdots \\ 2^{i-2}f(n/2^{i-2}) &= 2^{i-1}f(n/2^{i-1}) + 2^{i-1} \end{aligned}$$

Adicionando lado a lado, obtemos

$$\begin{aligned} f(n) &= 2^{i-1}f(n/2^{i-1}) + \sum_{k=1}^{i-1} 2^k \\ &= 2^{i-1}f(2) + 2^i - 2 \\ &= 2^{i-1} + 2^i - 2 \\ &= 3n/2 - 2 \end{aligned}$$

Logo,

$$f(n) = 3n/2 - 2$$

para o melhor caso, pior caso e caso médio.

De acordo com o Teorema da página 10 o algoritmo acima é **ótimo**, tendo o mesmo tipo de comportamento do algoritmo MaxMin3 do Programa 1.4. Entretanto, na prática, o algoritmo acima deve ser pior do que os algoritmos MaxMin2 e MaxMin3 dos Programas 1.3 e 1.4, respectivamente, podendo até ser pior do que o algoritmo MaxMin1 do Programa 1.2. Isto porque na implementação da versão recursiva, a cada chamada do procedimento MaxMin4 o compilador salva em uma estrutura de dados os valores de Linf, Lsup, Max e Min, além do endereço de retorno da chamada para o procedimento. Além disso, uma comparação adicional é necessária a cada chamada recursiva para verificar se Lsup − Linf ≤ 1. Outra observação relevante sobre a implementação acima é que $n + 1$ deve ser menor do que a metade do maior inteiro que pode ser representado pelo compilador a ser utilizado, porque o comando

```
Meio := (Linf + Lsup) div 2;
```

pode provocar *overflow* na operação Linf + Lsup.

Nos capítulos seguintes vamos estudar vários procedimentos recursivos.

1.5 Pascal

Os programas apresentados neste livro usam apenas as características básicas do Pascal, de acordo com a definição apresentada por Jensen e Wirth (1974). Sempre que possível são evitadas as facilidades mais avançadas disponíveis em algumas implementações do Pascal.

0 objetivo desta seção não é apresentar a linguagem Pascal na sua totalidade, mas apenas examinar algumas de suas características, facilitando assim a leitura deste livro para as pessoas pouco familiarizadas com a lingua -gem. Uma descrição clara e- concisa da linguagem é apresentada por Cooper (1983). Um bom texto introdutório sobre a linguagem é apresentado por Clancy e Cooper (1982).

As várias partes componentes de um programa Pascal podem ser vistas na Figura 1.5. Um programa Pascal começa com um cabeçalho que dá nome ao programa. Rótulos, constantes, tipos, variáveis, procedimentos, e funções são declaradas sempre na ordem indicada pela Figura 1.5. A parte ativa do programa é descrita como uma seqüência de comandos, os quais incluem chamadas de procedimentos e funções.

program	cabeçalho do programa
label	declaração de rótulo para goto
const	definição de constantes
type	definição de tipos de dados
var	declaração de variáveis
procedure ou function	declaração de subprogramas
begin	
.	
.	comandos do programa
.	
end	

Figura 1.5: Estrutura de um programa Pascal

A regra geral para a linguagem Pascal é tornar explícito o tipo associado quando se declara uma constante, variável ou função, o que. permite testes de consistência durante o tempo de compilação. A definição de tipos permite ao programador alterar o nome de tipos existentes, como também criar um número ilimitado de outros tipos. No caso do Pascal os tipos podem ser colocados em três categorias: *simples, estruturados, e apontadores.*

Tipos *simples* são grupos de valores indivisíveis, estando dentro desta categoria os tipos básicos integer, boolean, char, e real. Tipos simples adicionais podem ser enumerados através de uma listagem de novos grupos de

valores, ou através da indicação de subintervalos que restringem tipos a uma subseqüência dos valores de um outro tipo simples previamente definido. Exemplos de tipos enumerados

```
type cor     = (vermelho, azul, rosa);
type sexo    = (mas, fern);
type boolean = (false, true);
```

Se as variáveis *c, s, e d* são declaradas

```
var c :
cor;
var s :
sexo;
var b :
boolean;
```

então são possíveis as seguintes atribuições

```
c :
=rosa;
s :
=fern;
b :
=true;
```

Exemplos de tipos com subintervalos

```
type ano = 1900..
1999; type letra =
'A'..'Z';
```

Dadas as variáveis

```
vara :
ano; var b :
letra;
type cartão = array [1..80] of char;
type matriz = array [1..5, 1..5] of real;
type coluna = array [1..3] of real;
type linha = array [ano] of char;
type alfa = packed array [1..n] of char;
type vetor = array [1..n] of integer;
```

onde a constante *n* deve ser previamente declarada

```
const n = 20;
```

Dada a variável

```
var x : coluna;
```

as atribuições x[1]:=0.75, x[2]:=0.85 e x[3]:=1.5 são possíveis.

Um tipo estruturado **registro** é uma união de valores de tipos quaisquer, cujos campos podem ser acessados pelos seus nomes.
Exemplos:

```
type data = record
              dia : 1..31;
              mês : 1..12;
            end;
type pessoa = record
              sobrenome       :
              alfa;
              primeironome    :
              alfa;
              aniversário     :
              data;
              sexo            :    (mas, ferm);
            end;
```

Declarada a variável

```
var p: pessoa;
```

valores particulares podem ser atribuidos como se segue

```
p.sobrenome       := 'Ziviani';
p.primeironome    := 'Patricia';
p.aniversário.dia := 21;
p.aniversário.mês := 10;
p.sexo            := fern;
```

A Figura 1.6 ilustra este exemplo.

Um tipo estruturado conjunto define a coleção de todos os subconjuntos de algum tipo simples, com operadores especiais * (interseção), + (união), — (diferença) e *in* (pertence a) definidos para todos os tipos conjuntos. Exemplos:

```
type conjint  =  set of 1..9;
type conjcor  =  set of cor;
type conjchar =  set of char;
```

pessoa p

Figura 1.6: Registro do tipo pessoa

onde o tipo cor deve ser previamente definido como o tipo simples enumerado

```
type cor = (vermelho, azul, rosa);
```

Declaradas as variáveis

```
var ci : conjint;
var cc : array [1..5] of conjcor;
var ch : conjchar;
```

valores particulares do tipo conjunto podem ser construídos e atribuídos, como se segue:

```
ci    := [1,4,9];
cc[2] := [vermelho..rosa];
cc[4] := [ ];
cc[5] := [azul, rosa];
```

A ordem de prioridade para execução dos operadores é : "interseção" tem prioridade sobre os operadores de "união" e "diferença", que por sua vez têm prioridade sobre o operador "pertence a".
Exemplos:

```
[1..5,7] * [4,6,8]  é [4]
[1..3,5] + [4,6]    é [1..6]
[1..3,5] - [2,4]    é [1,3,5]
2 in [1..5]         é true
```

Um tipo estruturado **arquivo** define uma seqüência de valores homogêneos de qualquer tipo, e geralmente é associado com alguma unidade externa.
Exemplo:

```
type arquivopessoal = file of Pessoa;
```

```
program Copia(Velho, Novo);
{copia o arquivo Velho no arquivo Novo}
type Pessoa = record
                Sobrenome     : alfa;
                PrimeiroNome  : alfa;
                Aniversário   : data;
                Sexo          : (mas, fem);
              end;
var Velho, Novo  : file of Pessoa;
    Registro     : Pessoa;
begin
  reset(Velho); rewrite(Novo);
  while not eof(Velho) do
    begin
    read(Velho, Registro);
    write(Novo, Registro);
    end;
end.
```

Programa 1.8: Programa para copiar arquivo

O Programa 1.8 copia o conteúdo arquivo Velho no arquivo Novo. Observe que os nomes dos arquivos aparecem como parâmetros do programa. E importante observar que a atribuição de nomes de arquivos externos, ao programa varia de compilador para compilador. Por exempla, no caso do Turbo Pascal[3] a atribuição do nome externo de um arquivo a uma variável interna ao programa é realizada através do comando **assign** e não como parâmetros do programa.

Os tipos *apontadores* são úteis para criar estruturas de dados **encadeadas,** do tipo listas, árvores, e grafos. Um apontador é uma variável que referencia uma outra variável **alocada dinamicamente**. Em geral a variável referenciada é definida como um registro que inclui também um apontador para outro elemento do mesmo tipo.

Exemplo:

```
type Apontador = ^Nodo;
type Nodo = record
              Chave : integer
              Apont : Apontador;
            end;
```

Dada uma variável

[3]Turbo Pascal é marca registrada da Borland International

```
var Lista: Apontador;
```

é possível criar uma lista como ilustrada na Figura 1.7.

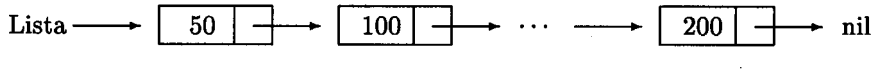


Figura 1.7: Lista encadeada

Notas Bibliográficas

Estudos básicos sobre os conceitos de algoritmos, estruturas de dados e programas podem ser encontrados em Dahl, Dijkstra e Hoare (1972), Dijkstra (1971), Dijkstra (1976), Hoare (1969), Wirth (1971), Wirth (1974), Wirth (1976). Mais recentemente Manber (1988) e Manber (1989) tratam da utilização de indução matemática para o projeto de algoritmos.

A análise assintótica de algoritmos é hoje a principal medida de eficiência para algoritmos. Existem muitos livros que apresentam técnicas para analisar algoritmos, tais como somatórios, equações de recorrência, árvores de decisão, oráculos, dentre outras. Knuth (1968), Knuth (1973), Knuth (1981), Graham, Knuth e Patashnik (1989), Aho, Hoperoft e Ullman (1974), Stanat e McAllister (1977), Cormem, Leiserson e Rivest (1990), Manber (1989), Horowitz e Sahni (1978), Greene e Knuth (1982), são alguns exemplos. Artigos gerais sobre o tópico incluem Knuth (1971), Knuth (1976), Weide (1977), Lueker (1980), Flajolet e Vitter (1987). Tarjan (1985) apresenta **custo amortizado:** se certa parte de um algoritmo é executada muitas vezes, cada vez com um tempo de execução diferente, ao invés de considerar o pior caso em cada execução, os diferentes custos são amortizados.

Existe uma enorme quantidade de livros sobre a linguagem **Pascal**. O Pascal padrão foi definido originalmente em Jensen e Wirth (1974). O livro de Cooper (1983) apresenta uma descrição precisa e ao mesmo tempo didática do Pascal padrão.

Exercícios

1) Dê o conceito de

- algoritmo

- tipo de dados
- tipo abstrato de dados

2) Avaliar as somas:

a) $\sum_{i=1}^{n} i$

b) $\sum_{i=1}^{n} a^i$

c) $\sum_{i=1}^{n} ia^i$

d) $\sum_{i=1}^{k} 2^{k-i} i^2$

e) $\sum_{i=0}^{n} \binom{n}{i}$

f) $\sum_{i=1}^{n} i\binom{n}{i}$

g) $\sum_{i=m}^{n} a_i - a_{i-1}$

h) $1 + 1/7 + 1/49 + \cdots + (1/7)^n$

i) $\sum_{i=1}^{n} log_2 i$

j) $\sum_{i=1}^{n} i2^{-i}$

3) Resolva as seguintes **relações de recorrência:**

a) $\begin{cases} T(n) = T(n-1) + c & c \text{ constante}, \ n > 1 \\ T(1) = 0 \end{cases}$

b) $\begin{cases} T(n) = T(n-1) + n - 1 & n > 1 \\ T(1) = 0 \end{cases}$

c) $\begin{cases} T(n) = 2T(n/2) + n - 1 & n > 1 \\ T(1) = 0 \end{cases}$

d) $\begin{cases} T(n) = T(n-1) + 2^n & n >= 1 \\ T(0) = 1 \end{cases}$

e) $\begin{cases} T(n) = cT(n-1) & c, k \text{ constantes}, \ n > 0 \\ T(0) = k \end{cases}$

f) $\begin{cases} T(n) = 3T(n/2) + n & n > 1 \\ T(1) = 1 \end{cases}$

g) $\begin{cases} T(n) = 3T(n-1) - 2T(n-2) & n > 1 \\ T(0) = 0 \\ T(1) = 1 \end{cases}$

h) $\begin{cases} T(n) = 1 + T(n-1) + T(n-2) & n >= 2 \\ T(0) = 0 \\ T(1) = 1 \end{cases}$

i) $\begin{cases} T(n) = \sum_{i=1}^{n-1} 2T(i) + 1 & n > 1 \\ T(1) = 1 \end{cases}$

j) $\begin{cases} T(n) = 2T(\lfloor n/2 \rfloor) + 2n\log_2 n \\ T(2) = 4 \\ \text{cuja solução satisfaz } T(n) = O(n \log^2 n). \\ \text{Prove este resultado usando indução matemática em } n \\ \text{(Manber, 1989, p. 56).} \end{cases}$

4) O que significa dizer que uma função $g(n)$ é $O(f(n))$?

5) Prove que $f(n) = 1^2 + 2^2 + \cdots + n^2$ é igual a $n^3/3 + O(n^2)$.

6) Indique se as afirmativas abaixo são verdadeiras ou falsas e justifique a sua resposta.

   a) $2^{n+1} = O(2^n)$
   b) $2^{2n} = O(2^n)$
   c) $f(n) = O(u(n))$ e $g(n) = O(v(n)) \Rightarrow f(n)+g(n) = O(u(n)+v(n))$
   d) $f(n) = O(u(n))$ e $g(n) = O(v(n)) \Rightarrow f(n)-g(n) = O(u(n)-v(n))$

7) Sejam duas funções não-negativas $f(n)$ e $g(n)$. Diz-se que $f(n) = \Theta(g(n))$ se $f(n) = O(g(n))$ e $g(n) = O(f(n))$.
   Mostre que $\max(f(n), g(n)) = \Theta(f(n) + g(n))$.

8) Suponha dois algoritmos A e B, com funções de complexidade de tempo $a(n) = n^2 - n + 549$ e $b(n) = 49n + 49$
   respectivamente. Determine quais valores de n pertencentes ao conjunto dos números naturais para os quais A leva menos tempo para executar do que B.

9) Considere o algoritmo abaixo. Suponha que a operação crucial é o fato de inspecionar um elemento. Neste caso o algoritmo inspeciona $1/3$ dos n elementos e então faz uma chamada recursiva sobre os $2/3n$ elementos restantes.

```
Pesquisa(n);
if n <= 1
then "inspecione elemento" e termine
else begin
     para cada um dos n elementos "inspecione elemento";
     {—de alguma forma isto permite descartar 1/3 dos
        elementos e fazer uma chamada recursiva no resto —}
     Pesquisa(2n/3);
     end;
```

a) Escreva uma **relação de recorrência** que descreva este comportamento.

b) Converta esta relação para um somatório.

c) Dê a fórmula fechada para este somatório.

10) É dada uma matriz $n \times n$ A, cada elemento é denominado A_{ij}, onde $1 \leq i, j \leq n$. Sabemos que a matriz foi ordenada de modo a (Carvalho, 1992):

$$A_{ij} < A_{ik}, \text{ para todo } i \text{ e } j < k.$$

$$A_{ij} < A_{kj}, \text{ para todo } i \text{ e } j < k.$$

Apresente um algoritmo que ache a localização de um determinado elemento x em A e analize o comportamento no pior caso. (Dica: Existe um algoritmo que resolve este problema em $O(n)$ comparações no pior caso).

11) Implemente os três algoritmos apresentados nos Programas 1.3, 1.4 e 1.7, para obter o máximo e o mínimo de um conjunto contendo n elementos. Execute os algoritmos para valores suficientemente grandes de n, gerando casos de teste para o melhor caso, pior caso e caso esperado. Meça o tempo de execução para cada algoritmo com relação aos três casos acima. Comente os resultados obtidos.

12) Apresente um algoritmo para obter o maior e o segundo maior elemento de um conjunto. Apresente também uma análise do algoritmo. Você acha o seu algoritmo eficiente? Por quê? Procure comprovar suas respostas.

13) São dados 2n números distintos distribuidos em dois arranjos com n elementos A e *B* ordenados de maneira tal que (Carvalho, 1992):

A[1] > A[2] > A[3] > • • • > *A [n] e*

B[1] > *B [2]* > *B [3]* > • • • > B[n].

O problema é achar o n-ésimo maior número dentre estes 2n elementos.

a) Obtenha um limite inferior para o número de comparações necessárias para resolver este problema.

b) Apresente um algoritmo cuja complexidade no pior caso seja iqual ao valor obtido na letra a), ou seja, um algoritmo ótimo.

Capítulo 2

Estruturas de Dados Básicas

2.1 Listas Lineares

Uma das formas mais simples de interligar os elementos de um conjunto é através de uma lista. Lista é uma estrutura onde as operações inserir, retirar e localizar são definidas. Listas são estruturas muito flexíveis porque podem crescer ou diminuir de tamanho durante a execução de um programa, de acordo com a demanda. Itens podem ser acessados, inseridos ou retirados de uma lista. Duas listas podem ser concatenadas para formar uma lista única, assim como uma lista pode ser partida em duas ou mais listas.

Listas são adequadas para aplicações onde não é possível prever a demanda por memória, permitindo a manipulação de quantidades imprevisíveis de dados, de formato também imprevisível. Listas são úteis em aplicações tais como manipulação simbólica, gerência de memória, simulação e compiladores. Na manipulação simbólica os termos de uma fórmula podem crescer sem limites. Em simulação dirigida por relógio pode ser criado um número imprevisível de processos, os quais têm que ser escalonados para execução de acordo com alguma ordem predefinida.

Uma **lista linear** é uma seqüência de zero ou mais itens $x_1, x_2, \bullet\bullet\bullet, x_n$, onde x_i é de um determinado tipo e n representa o tamanho da lista linear. Sua principal propriedade estrutural envolve as posições relativas dos itens em uma dimensão. Assumindo $n \geq 1$, xy_1 é o primeiro item da lista e x_n é o último item da lista. Em geral x_i precede x_i+_1 para $i = 1, 2, \bullet\bullet\bullet, n - 1$, e xi sucede xa_{-1} para i = *2, 3,* ••• , *n.* Em outras palavras, o elemento x_i é dito estar na i-ésima posição da lista.

Para criar um **tipo abstrato de dados** Lista, é necessário definir um conjunto de operações sobre os objetos do tipo Lista. 0 conjunto de operações a ser definido depende de cada aplicação, não existindo um conjunto de operações que seja adequado a todas as aplicações. Um conjunto

de operações necessário a uma maioria de aplicações é apresentado a seguir. Outras sugestões para o conjunto de operações podem ser encontradas em Knuth *(1968, p.235)* e Aho, Hoperoft e Ullman (1983, pp.38-39).

1. Criar uma lista linear vazia.
2. Inserir um novo item imediatamente após o i-ésimo item.
3. Retirar o i-ésimo item.
4. Localizar o i-ésimo item para examinar e/ou alterar o conteúdo de seus componentes.
5. Combinar duas ou mais listas lineares em uma lista única.
6. Partir uma lista linear em duas ou mais listas.
7. Fazer uma cópia da lista linear.
8. Ordenar os itens da lista em ordem ascendente ou descendente, de acordo com alguns de seus componentes.
9. Pesquisar a ocorrência de um item com um valor particular em algum componente.

0 item 8 acima é objeto de um estudo cuidadoso no Capítulo *3,* e o item *9* será tratado nos Capítulos 4 e 5.

Um conjunto de operações necessário para uma aplicação exemplo a ser apresentada mais adiante é apresentado a seguir.

1. FLVazia(Lista). Faz a lista ficar vazia.
2. Insere(x, Lista). Insere x após o último item da lista.
3. Retira(p, Lista, x). Retorna o item x que está na posição p da lista, retirando-o da lista e deslocando os itens a partir da posição p+l para as posições anteriores.
4. Vazia(Lista). Esta função retorna *true* se a lista está vazia; senão retorna *false.*
5. Imprime(Lista). Imprime os itens da lista na ordem de ocorrência.

Existem várias estruturas de dados que podem ser usadas para representar listas lineares, cada uma com vantagens e desvantagens particulares. As duas representações mais utilizadas são as implementações através de arranjos e de apontadores. A implementação através de cursores (Aho, Hoperoft e Ullman, 1983, pp. *48)* pode ser útil em algumas aplicações.

2.1.1 Implementação de Listas Através de Arranjos

Em um tipo estruturado arranjo, os itens da lista são armazenados em posições contíguas de memória, conforme ilustra a Figura 2.1. Neste caso a lista pode ser percorrida em qualquer direção. A inserção de um novo item pode ser realizada após o último item com custo constante. A inserção de um novo item no meio da lista requer um deslocamento de todos os itens localizados após o ponto de inserção. Da mesma forma, retirar um item do início da lista requer um deslocamento de itens para preencher o espaço deixado vazio.

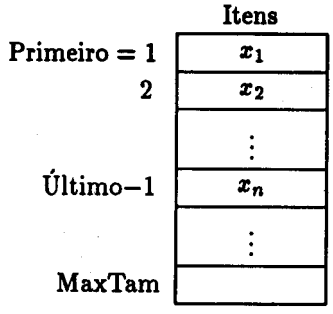


Figura 2.1: Implementação de uma lista através de arranjo

O campo Item é o principal componente do registro TipoLista mostrado no Programa 2.1. Os itens são armazenados em um **array** de tamanho suficiente para armazenar a lista. 0 campo Ultimo do registro TipoLista contém um apontador para a posição seguinte a do último elemento da lista. 0 i-ésimo item da lista está armazenado na i-ésima posição do **array**, 1 < i <Ultimo. A constante MaxTam define o tamanho máximo permitido para a lista.

Uma possível implementação para as cinco operações definidas anteriormente é mostrada no Programa 2.2. Observe que Lista é passada como **var** (por referência), mesmo nos procedimentos em que Lista não é modificada (como, por exemplo, a função Vazia) por razões de eficiência, porque desta forma a estrutura Lista não é copiada a cada chamada do procedimento.

A implementação de listas através de arranjos tem como vantagem a economia de memória, pois os apontadores são implícitos nesta estrutura. Como desvantagens citamos: (i) o custo para inserir ou retirar itens da lista, que pode causar um deslocamento de todos os itens, no pior caso; (ii) em aplicações em que não existe previsão sobre o crescimento da lista, a utilização de arranjos em linguagens como o Pascal pode ser problemática porque neste caso o tamanho máximo da lista tem que ser definido em tempo de compilação.

```
const
   InícioArranjo = 1;
   MaxTam        = 1000;
type
   Apontador  = integer;
   TipoItem   = record
                   Chave : TipoChave;
                   {outros componentes}
                end;
   TipoLista  = record
                   Item     : array [1..MaxTam] of TipoItem;
                   Primeiro: Apontador;
                   Último   : Apontador;
                end;
```

Programa 2.1: Estrutura da lista usando arranjo

2.1.2 Implementação de Listas Através de Apontadores

Em uma implementação de listas através de apontadores, cada item da lista é encadeado com o seguinte através de uma variável do tipo Apontador. Este tipo de implementação permite utilizar posições não contíguas de memória, sendo possível inserir e retirar elementos sem haver necessidade de deslocar os itens seguintes da lista.

A Figura 2.2 ilustra uma lista representada desta forma. Observe que existe uma célula cabeça que aponta para a célula que contém x_1. Apesar da célula cabeça não conter informação é conveniente fazê-la com a mesma estrutura que uma outra célula qualquer para simplificar as operações sobre a lista.

A lista é constituída de células, onde cada célula contém um item da lista e um apontador para a célula seguinte, de acordo com o registro Célula mostrado no Programa *2.3. O* registro TipoLista contém um apontador para a célula cabeça e um apontador para a última célula da lista. Uma possível implementação para as cinco operações definidas anteriormente é mostrada no Programa 2.4.

A implementação através de apontadores permite inserir ou retirar itens do meio da lista a um custo constante, aspecto importante quando a lista tem que ser mantida em ordem. Em aplicações em que não existe previsão sobre o crescimento da lista é conveniente usar **listas encadeadas** por apontadores, porque neste caso o tamanho máximo da lista não precisa ser definido a priori. A maior desvantagem deste tipo de implementação é a utilização de memória extra para armazenar os apontadores.

```
procedure FLVazia (var Lista : TipoLista);
begin
  Lista.Primeiro := InícioArranjo;
  Lista.Último := Lista.Primeiro;
end;
function Vazia (var Lista : TipoLista) : boolean;
begin
  Vazia := Lista.Primeiro = Lista.Último;
end;
procedure Insere (x : TipoItem; var Lista : TipoLista);
begin
  if Lista.Último > MaxTam
  then writeln('Lista está cheia')
  else begin
       Lista.Item[Lista.Último] := x;
       Lista.Último := Lista.Último +1;
       end;
end;
 procedure Retira(p : Apontador;
                  var Lista : TipoLista; var Item : TipoItem);
var Aux : integer;
begin
  if Vazia(Lista) or (p ≥ Lista.Último)
  then writeln (' Erro : Posição não existe')
  else begin
       Item := Lista.Item[p];
       Lista.Último := Lista.Último −1;
       for Aux := p to Lista.Último −1 do
         Lista.Item[Aux] := Lista.Item[Aux+1];
       end;
end;
procedure Imprime (var Lista : TipoLista);
var Aux : integer;
begin
  for Aux := Lista.Primeiro to Lista.Último −1 do
    writeln(Lista.Item[Aux].Chave);
end;
```

Programa 2.2: Operações sobre listas usando posições contíguas de memória

Figura 2.2: Implementação de uma lista através de apontadores

```
type
  Apontador = ^Célula;
  TipoItem  = record
                Chave : TipoChave;
                {outros componentes}
              end;
  Célula    = record
                Item : TipoItem;
                Prox : Apontador;
              end;
  TipoLista = record
                Primeiro: Apontador;
                Último  : Apontador;
              end;
```

Programa 2.3: Estrutura da lista usando apontadores

```
procedure FLVazia (var Lista : TipoLista);
begin
  new(Lista.Primeiro);
  Lista.Último := Lista.Primeiro;
  Lista.Primeiro^.Prox := nil;
end;
function Vazia (Lista : TipoLista) : boolean;
begin
  Vazia := Lista.Primeiro = Lista.Último;
end;
procedure Insere (x : TipoItem; var Lista : TipoLista);
begin
  new(Lista.Último^.Prox);
  Lista.Último := Lista.Último^.Prox;
  Lista.Último^.Item := x;
  Lista.Último^.Prox := nil;
end;
 procedure Retira(p : Apontador;
                  var Lista : TipoLista; var Item : TipoItem);
 {—Obs.: o item a ser retirado é o seguinte ao apontado por p — }
var q : Apontador;
begin
  if Vazia(Lista) or (p = nil) or (p^.Prox = nil)
  then writeln (' Erro : Lista vazia ou posição não existe')
  else begin
       q := p^.Prox; Item := q^.Item; p^.Prox := q^.Prox;
       if p^.Prox = nil then Lista.Último := p;
       dispose(q);
       end;
end;
procedure Imprime (Lista : TipoLista);
var Aux : Apontador;
begin
  Aux := Lista.Primeiro^.Prox;
  while Aux <> nil do
    begin
    writeln(Aux^.Item.Chave);
    Aux := Aux^.Prox;
    end;
end;
```

Programa 2.4: Operações sobre listas usando apontadores

```
Chave     : 1..999;
NotaFinal : 0..10;
Opção     : array [1..3] of 1..7;
```

Programa *2.5:* Campos do *registro de um* candidato

Exemplo: Considere o exemplo proposto por Furtado (1984), apresentado a seguir. Durante o exame vestibular de uma universidade, cada candidato tem direito a 3 opções para tentar uma vaga em um dos 7 cursos oferecidos. Para cada candidato é lido um registro contendo os campos mostrados no Programa *2.5*.

O campo Chave contém o número de inscrição do candidato (este campo identifica de forma única cada registro de entrada). O campo NotaFinal contém a média das notas do candidato. O campo Opção é um vetor contendo a primeira, a segunda e a terceira opções de curso do candidato (os cursos são numerados de 1 a 7).

O problema consiste em distribuir os candidatos entre os cursos, de acordo com a nota final e as opções apresentadas por cada candidato. No caso de empate serão atendidos primeiro os candidatos que se inscreveram mais cedo, isto é, os candidatos com mesma nota final serão atendidos na ordem de inscrição para os exames.

Um possível caminho para resolver o problema de distribuir os alunos entre os cursos contém duas etapas, a saber:

1. ordenar os registros pelo campo NotaFinal, respeitando-se a ordem de inscrição dos candidatos;

2. percorrer cada conjunto de registros com mesma NotaFinal, iniciando-se pelo conjunto de NotaFinal 10, seguido do conjunto da NotaFinal 9, e assim por diante. Para um conjunto de mesma NotaFinal tenta-se encaixar cada registro desse conjunto em um dos cursos, na primeira das três opções em que houver vaga (se houver).

Um primeiro refinamento do algoritmo pode ser visto no Programa 2.6.

Para prosseguirmos na descrição do algoritmo, nós somos forçados a tomar algumas decisões sobre representação de dados. Uma boa maneira de representar um conjunto de registros é através de listas. O **tipo abstrato de dados** Lista definido anteriormente, acompanhado do conjunto de operações definido sobre os objetos do tipo lista, mostra-se bastante adequado ao nosso problema.

O refinamento do comando "ordena os registros pelo campo NotaFinal" do Programa 2.6 pode ser realizado da seguinte forma: os registros ao se-rem lidos são armazenados em listas para cada nota, conforme ilustra a

```
program Vestibular;
begin
  ordena os registros pelo campo NotaFinal;
  for Nota := 10 downto 0 do
    while houver registro com mesma nota do
      if existe vaga em um dos cursos de opção do candidato
      then insere registro no conjunto de aprovados
      else insere registro no conjunto de reprovados;
  imprime aprovados por curso;
  imprime reprovados;
end.
```

Programa 2.6: Primeiro refinamento do programa Vestibular

Figura 2.3. Após a leitura do último registro os candidatos estão automaticamente ordenados por NotaFinal. Dentro de cada lista os registros estão ordenados por ordem de inscrição, desde que os registros sejam lidos na ordem de inscrição de cada candidato e inseridos nesta ordem.

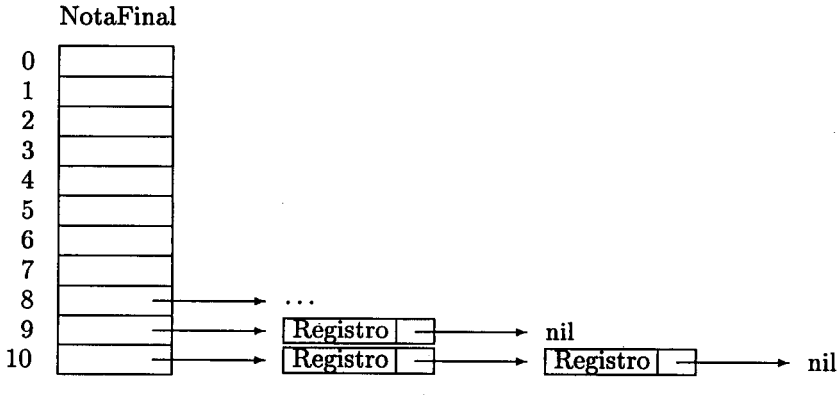


Figura 2.3: Classificação dos alunos por NotaFinal

Dessa estrutura passa-se para a estrutura apresentada na Figura 2.4. As listas de registros da Figura 2.3 são percorridas, inicialmente com a lista de NotaFinal 10, seguida da lista de NotaFinal 9, e assim sucessivamente. Ao percorrer uma lista, cada registro é retirado e colocado em uma das listas da Figura 2.4, na primeira das três opções em que houver vaga. Se não houver vaga o registro é colocado em uma lista de reprovados. Ao final a estrutura da Figura 2.4 conterá uma relação de candidatos aprovados em cada curso.

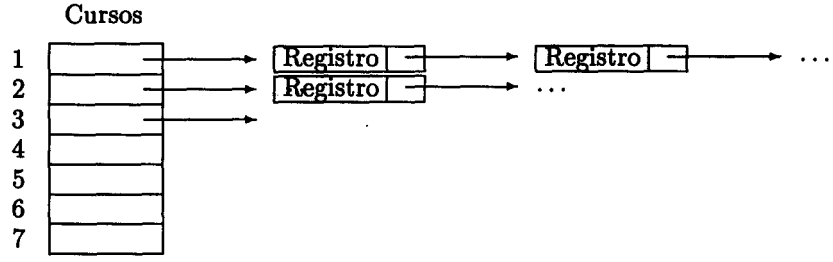


Figura 2.4: Lista de aprovados por Curso

Após as decisões mais importantes sobre a representação dos dados, é possível obter-se mais um refinamento, conforme mostra o Programa 2.7.

```
program Vestibular;
begin
  lê número de vagas para cada curso;
  inicializa listas de classificação, listas de aprovados e lista de reprovados;
  lê registro;          {—vide formato no Programa 2.5 —}
  while Chave <> 0 do
    begin
    insere registro em uma das listas de classificação, conforme nota final;
    lê registro;
    end;
  for Nota := 10 downto 0 do
    while houver próximo registro com mesma NotaFinal do
      begin
      retira registro da lista;
      if existe vaga em um dos cursos de opção do candidato
      then begin
             insere registro na lista de aprovados;
             decrementa o número de vagas para aquele curso;
             end
      else insere registro na lista de reprovados;
      obtém próximo registro;
      end;
  imprime aprovados por curso;
  imprime reprovados;
end.
```

Programa 2.7: Segundo refinamento do programa Vestibular

Neste momento nós somos forçados a tomar decisões sobre a implementação do **tipo abstrato de dados** Lista. Considerando-se que o tamanho das listas varia de forma totalmente imprevisível, a escolha deve cair

```
const
  NOpções= 3;
  NCursos= 7;
type
   TipoChave = 1..999;
   TipoItem  = record
                 Chave     : TipoChave;
                 NotaFinal : 0..10;
                 Opção     : array[1..NOpções] of 1..NCursos;
               end;
   Apontador = ^Célula;
   Célula    = record
                 Item : TipoItem;
                 Prox : Apontador;
               end;
   TipoLista = record
                 Primeiro : Apontador;
                 Último   : Apontador;
               end;
```

Programa 2.8: Estrutura da lista

sobre a implementação através de apontadores. O Programa 2.8 apresenta a definição dos tipos de dados a utilizar no último refinamento do algoritmo.

O refinamento final de algoritmo, descrito em Pascal, pode ser visto no Programa 2.9. Observe que o programa é completamente independente da implementação do tipo abstrato de dados Lista. Isto significa que podemos trocar a implementação do tipo abstrato de dados Lista de apontador para arranjo, bastando trocar a definição dos tipos apresentada no Programa 2.8 para uma definição similar à definição mostrada no Programa 2.1, acompanhada da troca dos operadores apresentados no Programa 2.4 pelos operadores apresentados no Programa 2.2. Esta substituição pode ser realizada sem causar impacto em nenhuma outra parte do código.

```
program Vestibular;
{—Entram aqui os tipos do Programa 2.8 —}
var  Registro     : TipoItem;
     Classificação : array[0..10] of TipoLista;
     Aprovados    : array[1..NCursos] of TipoLista;
     Reprovados   : TipoLista;
     Vagas        : array [1..NCursos] of integer;
     Passou       : boolean;
     i, Nota      : integer;
{—Entram aqui os operadores apresentados no Programa 2.4 —}
```

```
procedure LeRegistro (var Registro : TipoItem);
{—os valores lidos devem estar separados por brancos —}
var i : integer;
begin
  read(Registro.Chave, Registro.NotaFinal);
  for i := 1 to NOpções do read(Registro.Opção[i]);
  readln;
end;

begin {—Programa principal —}
  for i := 1 to NCursos do read(Vagas[i]); readln;
  for i := 0 to 10 do FLVazia(Classificação[i]);
  for i := 1 to NCursos do FLVazia(Aprovados[i]);
  FLVazia(Reprovados); LeRegistro(Registro);
  while Registro.Chave <> 0 do
    begin
    Insere(Registro, Classificação[Registro.NotaFinal]);
    LeRegistro(Registro);
    end;
  for Nota := 10 downto 0 do
    while not Vazia(Classificação[Nota]) do
      begin
      Retira(Classificação[Nota].Primeiro, Classificação[Nota], Registro);
      i := 1; Passou := false;
      while (i ≤ NOpções) and not Passou do
        begin
        if Vagas[Registro.Opção[i]] > 0
        then begin
             Insere(Registro, Aprovados[Registro.Opção[i]]);
             Vagas[Registro.Opção[i]] := Vagas[Registro.Opção[i]] - 1;
             Passou := true;
             end;
        i := i +1;
        end;
      if not Passou then Insere(Registro, Reprovados);
      end;
  for i := 1 to NCursos do
    begin
    writeln(' Relação dos aprovados no Curso', i:2);
    Imprime(Aprovados[i]);
    end;
  writeln(' Relação dos reprovados');
  Imprime(Reprovados);
end.
```

Programa 2.9: Refinamento final do programa Vestibular

Este exemplo mostra a importância de se escrever programas em função das operações para manipular **tipos abstratos de dados**, ao invés de utilizar detalhes particulares de implementação. Desta forma é possível alterar a implementação das operações rapidamente, sem haver necessidade de procurar por toda parte do código onde existe referência direta às estruturas de dados. Este aspecto é particularmente importante em programas de grande porte.

2.2 Pilhas

Existem aplicações para listas lineares nas quais inserções, retiradas e acessos a itens ocorrem sempre em um dos extremos da lista. Uma *pilha é* uma lista linear em que todas as inserções, retiradas e geralmente todos os acessos são feitos em apenas um extremo da lista..

Os itens em uma pilha estão colocados um sobre o outro, com o item inserido mais recentemente no topo e o item inserido menos recentemente no fundo. 0 modelo intuitivo de uma pilha é o de um monte de pratos em uma prateleira, sendo conveniente retirar pratos ou adicionar novos pratos na parte superior. Esta imagem está freqüentemente associada com a teoria de autômato, onde o topo de uma pilha é considerado como o receptáculo de uma cabeça de leitura/gravação que pode empilhar e desempilhar itens da pilha (Hoperoft e Ullman, 1969).

As pilhas possuem a seguinte propriedade: o último item 'inserido é o primeiro item que pode ser retirado da lista. Por esta razão as pilhas são chamadas de listas lifo, termo formado a partir de "last-in, first-out". Existe uma ordem linear para pilhas, que é a ordem do "mais recente para o menos recente". Esta propriedade torna a pilha uma ferramenta ideal para processamento de estruturas aninhadas de profundidade imprevisível, situação em que é necessário garantir que subestruturas mais internas sejam processadas antes da estrutura que as contenham. A qualquer instante uma pilha contém uma seqüência de obrigações adiadas, cuja ordem de remoção da pilha garante que as estruturas mais internas serão processadas antes das estruturas mais externas.

Estruturas aninhadas ocorrem freqüentemente na prática. Um exemplo simples é a situação em que é necessário caminhar em um conjunto de dados e guardar uma lista de coisas a fazer posteriormente (mais adi-ante veremos uma situação semelhante a esta em um programa editor de textos). 0 controle de seqüências de chamadas de subprogramas e sintaxe de expressões aritméticas são exemplos de estruturas aninhadas. As pilhas ocorrem também em conexão com algoritmos **recursivos** e estruturas de natureza recursiva, tais como as árvores.

Um tipo abstrato de dados Pilha, acompanhado de um conjunto de operações, é apresentado a seguir.

1. FPVazia(Pilha). Faz a pilha ficar vazia.

2. Vazia(Pilha). Esta função retorna *true* se a pilha está vazia; senão retorna *false.*
3. Empilha(x, Pilha). Insere o item x no topo da pilha.

4. Desempilha(Pilha, x). Retorna o item x no topo da pilha, retirando-o da pilha.

5. Tamanho(Pilha). Esta função retorna o número de itens da pilha.

Como no caso do tipo abstrato de dados Lista apresentado na Seção 2.1, existem várias opções de estruturas de dados que podem ser usadas para representar pilhas. As duas representações mais utilizadas são as implementações através de *arranjos e* de *apontadores.*

2.2.1 Implementação de Pilhas Através de Arranjos

Em uma implementação através de arranjos os itens da pilha são armazenados em posições contíguas de memória, conforme ilustra a Figura 2.5. Devido às características da pilha as operações de inserção e de retirada de itens devem ser implementadas de forma diferente das implementações usadas anteriormente para listas. Como as inserções e as retiradas ocorrem no topo da pilha, um cursor chamado topo é utilizado para controlar a posição do item no topo da pilha.

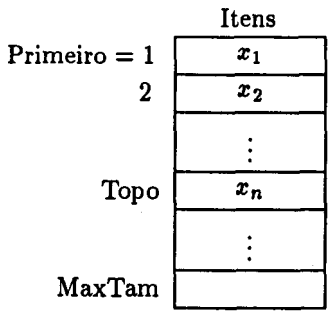


Figura 2.5: Implementação de uma pilha através de arranjo

O campo Item é o principal componente do registro TipoPilha mostrado no Programa 2.10. Os itens são armazenados em um array de tamanho

```
const
   MaxTam = 1000;
type
   Apontador = integer;
   TipoItem  = record
                  Chave : TipoChave;
                  {outros componentes}
               end;
   TipoPilha = record
                  Item : array[1..MaxTam] of TipoItem;
                  Topo : Apontador;
               end;
```

Programa 2.10: Estrutura da pilha usando arranjo

suficiente para armazenar a pilha. 0 outro campo do mesmo registro contém um apontador para o item no topo da pilha. A constante MaxTam define o tamanho máximo permitido para a pilha.

As cinco operações definidas sobre o TipoPilha podem ser implementadas conforme ilustra o Programa 2.11. Observe que Pilha é passada como var (por referência), mesmo nos procedimentos em que Pilha não é modificada (como, por exemplo, a função Vazia) por razões de eficiência, pois desta forma a estrutura Pilha não é copiada a cada chamada do procedimento ou função.

2.2.2 Implementação de Pilhas Através de Apontadores

Assim como na implementação de listas lineares através de apontadores, uma célula cabeça é mantida no topo da pilha para facilitar a implementação das operações empilha e desempilha quando a pilha está vazia, conforme ilustra a Figura 2.6. Para desempilhar o item x_n da Figura 2.6 basta desligar a célula cabeça da lista e a célula que contém x_n passa a ser a célula cabeça. Para empilhar um novo item basta fazer a operação contrária, criando uma nova célula cabeça e colocando o novo item na antiga célula cabeça. 0 campo Tamanho existe no registro TipoPilha por questão de eficiência, para evitar a contagem do número de itens da pilha na função Tamanho.

Cada célula de uma pilha contém um item da pilha e um apontador para outra célula, conforme ilustra o Programa 2.12. 0 registro TipoPilha contém um apontador para o topo da pilha (célula cabeça) e um apontador para o fundo da pilha.

As cinco operações definidas anteriormente podem ser implementadas através de apontadores, conforme ilustra o Programa 2.13.

```
procedure FPVazia(var Pilha : TipoPilha);
begin
  Pilha.Topo := 0;
end;
function Vazia(var Pilha : TipoPilha) : boolean;
begin
  Vazia := Pilha.Topo = 0;
end;
procedure Empilha(x : TipoItem; var Pilha : TipoPilha);
begin
  if Pilha.Topo = MaxTam
  then writeln(' Erro : pilha está cheia')
  else begin
       Pilha.Topo := Pilha.Topo +1;
       Pilha.Item[Pilha.Topo] := x;
       end;
end;
procedure Desempilha(var Pilha : TipoPilha; var Item : TipoItem);
begin
  if Vazia(Pilha)
  then writeln(' Erro : pilha está vazia')
  else begin
       Item := Pilha.Item[Pilha.Topo];
       Pilha.Topo := Pilha.Topo -1;
       end;
end;
function Tamanho(var Pilha : TipoPilha) : integer;
begin
  Tamanho := Pilha.Topo;
end;
```

Programa 2.11: Operações sobre pilhas usando arranjos

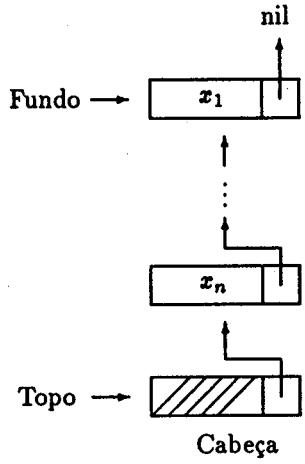


Figura 2.6: Implementação de uma pilha através de apontadores

```
type
  Apontador = ^Célula;
  TipoItem  = record
                Chave : TipoChave;
                {outros componentes}
              end;
  Célula    = record
                Item : TipoItem;
                Prox : Apontador;
              end;
  TipoPilha = record
                Fundo    : Apontador;
                Topo     : Apontador;
                Tamanho : integer;
              end;
```

Programa 2.12: Estrutura da pilha usando apontadores

```
procedure FPVazia(var Pilha : TipoPilha);
begin
  new(Pilha.Topo);
  Pilha.Fundo := Pilha.Topo;
  Pilha.Topo^.Prox := nil;
  Pilha.Tamanho := 0;
end;
function Vazia(var Pilha : TipoPilha) : boolean;
begin
  Vazia := Pilha.Topo = Pilha.Fundo;
end;
procedure Empilha(x : TipoItem; var Pilha : TipoPilha);
var Aux : Apontador;
begin
  new(Aux);
  Pilha.Topo^.Item := x;
  Aux^.Prox := Pilha.Topo;
  Pilha.Topo := Aux;
  Pilha.Tamanho := Pilha.Tamanho +1;
end;
procedure Desempilha(var Pilha : TipoPilha,var Item : TipoItem);
var q : Apontador;
begin
  if Vazia(Pilha)
  then writeln(' Erro : pilha vazia')
  else begin
       q := Pilha.Topo;
       Pilha.Topo := q^.Prox;
       Item := q^.Prox^.Item;
       dispose(q);
       Pilha.Tamanho := Pilha.Tamanho -1;
       end;
end;
function Tamanho(var Pilha : TipoPilha) : integer;
begin
  Tamanho := Pilha.Tamanho;
end;
```

Programa 2.13: Operações sobre pilhas usando apontadores

Exemplo: *Editor de Textos.*

Alguns editores de texto permitem que algum caractere funcione como um " cancela-caractere", cujo efeito é o de cancelar o caractere anterior na linha que está sendo editada. Por exemplo, se o cancela-caractere, então a seqüência de caracteres UEM##FMB#G corresponde à seqüência UFMG. Outro comando encontrado em editores de texto é o "cancela-linha", cujo efeito é o de cancelar todos os caracteres anteriores na linha que está sendo editada. Neste exemplo vamos considerar como o caractere cancela-linha. Finalmente, outro comando encontrado em editores de texto é o "salta-linha" , cujo efeito é o de causar a impressão dos caracteres que pertencem à linha que está sendo editada, iniciando uma nova linha de impressão a partir do caractere imediatamente seguinte ao caractere salta-linha. Por exemplo, se ` @' é o salta-linha, então a seqüência de caracteres DCC@UFMG.@ corresponde às duas linhas abaixo:

DCC
UFMG.

Vamos escrever um Editor de Texto (ET) que aceite os três comandos descritos acima. O ET deverá ler um caractere de cada vez do texto de entrada e produzir a impressão linha a linha, cada linha contendo no máximo 70 caracteres de impressão. O ET deverá utilizar o **tipo abstrato de dados** Pilha definido anteriormente, implementado através de arranjo.

A implementação do programa ET é apresentada no Programa 2.14. Da mesma forma que o programa Vestibular, apresentado na Seção 2.1, este programa utiliza um tipo abstrato de dados sem conhecer detalhes de sua implementação. Isto significa que a implementação do tipo abstrato de dados Pilha que utiliza arranjo pode ser substituída pela implementação que utiliza apontadores mostrada no Programa 2.13, sem causar impacto no programa. O procedimento Imprime, mostrado no Programa 2.15, é utilizado pelo programa ET.

A seguir é sugerido um texto para testar o programa ET, cujas características permite exercitar todas as partes importantes do programa.

```
Este et# um teste para o ET, o extraterrestre em
PASCAL.@Acabamos de testar a capacidade do ET saltar de linha,
utilizando seus poderes extras (cuidado pois agora vamos estourar
a capacidade máxima da linha de impressão, que é de setenta
caracteres.)@O k#cut#rso dh#e Estruturas de Dados et# h#um
cuu#rsh#o #x# x?*!#?!#+.@ Como et# bom
n#nt#ao### r#ess#tt#ar mb#aa#triz#cull#ado nn#x#ele!\ Sera
que este funciona\\\? O sinal? deve não### deve ficar! ~
```

```
program ET;
const MaxTam          = 70;
      CancelaCaractere = '#';
      CancelaLinha     = '\';
      SaltaLinha       = '@';
      MarcaEof         = '-';
type TipoChave = char;
{—Entram aqui os tipos do Programa 2.10 —}
var  Pilha : TipoPilha;
     x     : TipoItem;
{—Entram aqui os operadores do Programa 2.11 —}
{—Entra aqui o procedimento Imprime do Programa 2.15 —}
begin {—Programa principal —}
  FPVazia(Pilha); read(x.Chave);
  while x.Chave <> MarcaEof do
    begin
    if x.Chave = CancelaCaractere
    then begin
           if not Vazia(Pilha) then Desempilha(Pilha, x)
           end
    else if x.Chave = CancelaLinha
         then FPVazia(Pilha)
         else if x.Chave = SaltaLinha
              then Imprime(Pilha)
              else begin
                   if Tamanho(Pilha) = MaxTam then Imprime(Pilha);
                   Empilha(x, Pilha);
                   end;
    read(x.Chave);
    end;
  if not Vazia(Pilha) then Imprime(Pilha);
end. {—Programa principal —}
```

Programa 2.14: Implementação do ET

```
procedure Imprime(var Pilha : TipoPilha);
var Pilhaux : TipoPilha;
    x       : TipoItem;
begin
  FPVazia(Pilhaux);
  while not Vazia(Pilha) do
    begin
    Desempilha(Pilha, x);  Empilha(x, Pilhaux);
    end;
  while not Vazia(Pilhaux) do
    begin
    Desempilha(Pilhaux, x);  write(x.Chave);
    end;
  writeln;
end;
```

Programa 2.15: Procedimento Imprime utilizado no programa ET

2.3 Filas

Uma fila é uma lista linear em que todas as inserções são realizadas em um extremo da lista, e todas as retiradas e geralmente os acessos são realizados no outro extremo da lista. 0 modelo intuitivo de uma fila é o de uma fila de espera em que as pessoas no início da fila são servidas primeiro e as pessoas que chegam entram no fim da fila. Por esta razão as filas são chamadas de listas **fifo**, termo formado a partir de "first-in", "first-out". Existe uma ordem linear para filas que é a "ordem de chegada". Filas são utilizadas quando desejamos processar itens de acordo com a ordem "primeiro-quechega, primeiro-atendido". Sistemas operacionais utilizam filas para regular a ordem na qual tarefas devem receber processamento e recursos devem ser alocados a processos.

Um possível conjunto de operações, definido sobre um **tipo abstrato de dados** Fila, é definido a seguir.

1. FFVazia(Fila). Faz a fila ficar vazia.

2. Enfileira(x, Fila). Insere o item x no final da fila.

3. Desenfileira(Fila, x). Retorna o item x no início da fila, retirando-o da fila.

4. Vazia(Fila). Esta função retorna *true* se a fila está vazia; senão retorna *false*.

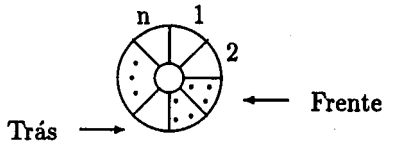


Figura 2.7: **Implementação *circular para* filas**

2.3.1 Implementação de Filas Através de Arranjos

Em uma implementação através de arranjos os itens são armazenados em posições contíguas de memória. Devido às características da fila, a operação enfileira faz a parte de trás da fila expandir-se, e a operação Desenfileira faz a parte da frente da fila contrair-se. Conseqüentemente, a fila tende a caminhar pela memória do computador, ocupando espaço na parte de trás e descartando espaço na parte da frente. Com poucas inserções e retiradas de itens a fila vai de encontro ao limite do espaço da memória alocado para ela.

A solução para o problema de caminhar pelo espaço alocado para uma fila é imaginar o array como um círculo, onde a primeira posição segue a última, conforme ilustra a Figura *2.7.* A fila se encontra em posições contíguas de memória, em alguma posição do círculo, delimitada pelos apontadores Frente e Trás. Para enfileirar um item basta mover o apontador Trás uma posição no sentido horário; para desenfileirar um item basta mover o apontador Frente uma posição no sentido horário.

O campo Item é o principal componente do registro TipoFila mostrado no Programa 2.16. 0 tamanho máximo do array circular é definido pela constante MaxTam. Os outros campos do registro TipoPilha contêm apontadores para a parte da frente e de trás da fila.

As quatro operações definidas sobre o TipoFila podem ser implementadas conforme ilustra o Programa 2.17. Observe que para a representação circular da Figura *2.7* existe um pequeno problema: não há uma forma de distinguir uma fila vazia de uma fila cheia, pois nos dois casos os apontadores Frente e trás apontam para a mesma posição do círculo. Uma possível saída para este problema, utilizada por Aho, Hoperoft e Ullman (1983, p.58), é não utilizar todo o espaço do array, deixando uma posição vazia. Neste caso a fila está cheia quando Trás+1 for igual a Frente, o que significa que existe uma célula vazia entre o fim e o início da fila.

Observe que a implementação do vetor circular é realizada através de aritmética modular. A aritmética modular é utilizada nos procedimentos Enfileira e Desenfileira do Programa *2.17,* através da função **mod** do Pascal.

```
const MaxTam = 1000;
type
  Apontador = integer;
  TipoItem  = record
                Chave : TipoChave;
                {outros componentes}
              end;
  TipoFila  = record
                Item  : array[1..MaxTam] of TipoItem;
                Frente: Apontador;
                Trás  : Apontador;
              end;
```

Programa 2.16: Estrutura da fila usando arranjo

```
procedure FFVazia (var Fila : TipoFila);
begin
  Fila.Frente := 1;
  Fila.Trás := Fila.Frente;
end;
function Vazia (var Fila : TipoFila) : boolean;
begin
  Vazia := Fila.Frente = Fila.Trás;
end;
procedure Enfileira (x : TipoItem; var Fila : TipoFila);
begin
  if Fila.Trás mod MaxTam +1 = Fila.Frente
  then writeln (' Erro : fila está cheia')
  else begin
       Fila.Item[Fila.Trás] := x;
       Fila.Trás := Fila.Trás mod MaxTam +1;
       end;
end;
procedure Desenfileira(var Fila : TipoFila; var Item : TipoItem);
begin
  if Vazia (Fila)
  then writeln (' Erro : fila está vazia')
  else begin
       Item := Fila.Item[Fila.Frente];
       Fila.Frente := Fila.Frente mod MaxTam +1;
       end;
end;
```

Programa 2.17: Operações sobre filas usando posições contíguas de memória

2.3.2 Implementação de Filas Através de Apontadores

Assim como em todas as outras implementações deste capítulo, uma célula cabeça é mantida para facilitar a implementação das operações Enfileira e Desenfileira quando a fila está vazia, conforme ilustra a Figura 2.8. Quando a fila está vazia os apontadores Frente e trás apontam para a célula cabeça. Para enfileirar um novo item basta criar uma célula nova, ligá-la após a célula que contém x_n e colocar nela o novo item. Para desenfileirar o item x_1 basta desligar a célula cabeça da lista e a célula que contém x_1 passa a ser a célula cabeça.

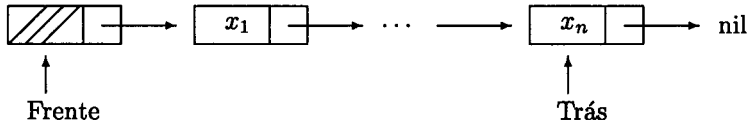


Figura 2.8: Implementação de uma fila através de apontadores

A fila é implementada através de células, onde cada célula contém um item da fila e um apontador para outra célula, conforme ilustra o Pro-grama 2.18. 0 registro TipoFila contém um apontador para a frente da fila (célula cabeça) e um apontador para a parte de trás da fila.

As quatro operações definidas sobre o TipoFila podem ser implementadas conforme ilustra o programa 2.19.

Notas Bibliográficas

Knuth (1968) apresenta um bom tratamento sobre listas lineares. Outras referências relevantes sobre estruturas de dados básicas são Aho, Hoperoft e Ullman (1983), Cormem, Leiserson e Rivest (1990) e Sedgewick (1988).

Exercícios

1) Considere a implementação de listas lineares utilizando apontadores e com célula-cabeça. Considere que um dos campos do TipoItem é uma Chave: TipoChave. Escreva uma função em Pascal
function EstaNaLista (Ch: TipoChave; var L: TipoLista): boolean;
que retorna true se Ch estiver na lista e retorna false se Ch não estiver na lista. Considere que não há ocorrências de chaves repetidas na lista. Determine a complexidade do seu algoritmo.

```
type
   Apontador = ^Célula;
   TipoItem  = record
                 Chave : TipoChave;
                 {outros componentes}
               end;
   Célula    = record
                 Item : TipoItem;
                 Prox : Apontador;
               end;
   TipoFila  = record
                 Frente: Apontador;
                 Trás  : Apontador;
               end;
```

Programa 2.18: Estrutura da fila usando apontadores

2) Um problema que pode surgir na manipulação de listas lineares simples é o de "voltar" atrás na lista, ou seja, percorrê-la no sentido inverso ao dos apontadores. A solução geralmente adotada é a incorporação à célula de um apontador para o seu antecessor. Listas deste tipo são chamadas de duplamente encadeadas. A Figura 2.9 mostra uma lista deste tipo com estrutura circular e a presença de uma célula cabeça.

   a) Declare os tipos necessários para a manipulação da lista.

   b) Escreva um procedimento em Pascal para retirar da lista a célula apontada por p:
   procedure Retira (p: Apontador; **var L**: TipoLista);
   Não deixe de considerar eventuais casos especiais.

3) Matrizes Esparsas – Utilização de Listas Através de Apontadores (Árabe, 1992)

   Matrizes esparsas são matrizes nas quais a maioria das posições são preenchidas por zeros. Para estas matrizes, podemos economizar um espaço significativo de memória se apenas os termos diferentes de zero forem armazenados. As operações usuais sobre estas matrizes (somar, multiplicar, inverter, pivotar) também podem ser feitas em tempo muito menor se não armazenarmos as posições que contém zeros.

   Uma maneira eficiente de representar estruturas com tamanho variável e/ou desconhecido é através de alocação encadeada, utilizando listas. Vamos usar esta representação para armazenar as matrizes esparsas.

```
procedure FFVazia (var Fila : TipoFila);
begin
  new (Fila.Frente);
  Fila.Trás := Fila.Frente;
  Fila.Frente^.Prox := nil;
end;
function Vazia (var Fila : TipoFila) : boolean;
begin
  Vazia := Fila.Frente = Fila.Trás;
end;
procedure Enfileira (x : TipoItem; var Fila : TipoFila);
begin
  new (Fila.Trás^.Prox);
  Fila.Trás := Fila.Trás^.Prox;
  Fila.Trás^.Item := x;
  Fila.Trás^.Prox := nil;
end;
procedure Desenfileira(var Fila : TipoFila; var Item : TipoItem);
var q : Apontador;
begin
  if Vazia (Fila)
  then writeln (' Erro : fila está vazia')
  else begin
       q := Fila.Frente;
       Fila.Frente := Fila.Frente^.Prox;
       Item := Fila.Frente^.Item;
       dispose(q);
       end;
end;
```

Programa 2.19: Operações sobre filas usando apontadores

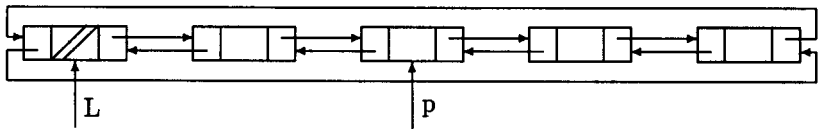


Figura 2.9: Lista circular duplamente encadeada

Cada coluna da matriz será representada por uma lista linear circular-Listas!circulares com uma célula cabeça. Da mesma maneira, cada linha da matriz também será representada por uma lista linear circular com uma célula cabeça. Cada célula da estrutura, além das células-cabeça, representará os termos diferentes de zero da matriz e devem ter o seguinte tipo:

```
type
   Apontador = ^ TipoCélula;

   TipoCélula = record
                   Direita,
                   Abaixo : Apontador;
                   Linha,
                   Coluna : integer;
                   Valor  : real;
                end;
```

O campo Abaixo deve ser usado para apontar o próximo elemento diferente de zero na mesma coluna. O campo Direita deve ser usado para apontar o próximo elemento diferente de zero na mesma linha. Dada uma matriz A, para um valor $A(i,j)$ diferente de zero, deverá haver uma célula com o campo Valor contendo $A(i,j)$, o campo Linha contendo i e o campo Coluna contendo j. Esta célula deverá pertencer à lista circular da linha i e também deverá pertencer à lista circular da coluna j. Ou seja, cada célula pertencerá a duas listas ao mesmo tempo. Para diferenciar as células cabeça, coloque -1 nos campos Linha e Coluna destas células.

Considere a matriz esparsa seguinte

$$A = \begin{pmatrix} 50 & 0 & 0 & 0 \\ 10 & 0 & 20 & 0 \\ 0 & 0 & 0 & 0 \\ -30 & 0 & -60 & 5 \end{pmatrix}$$

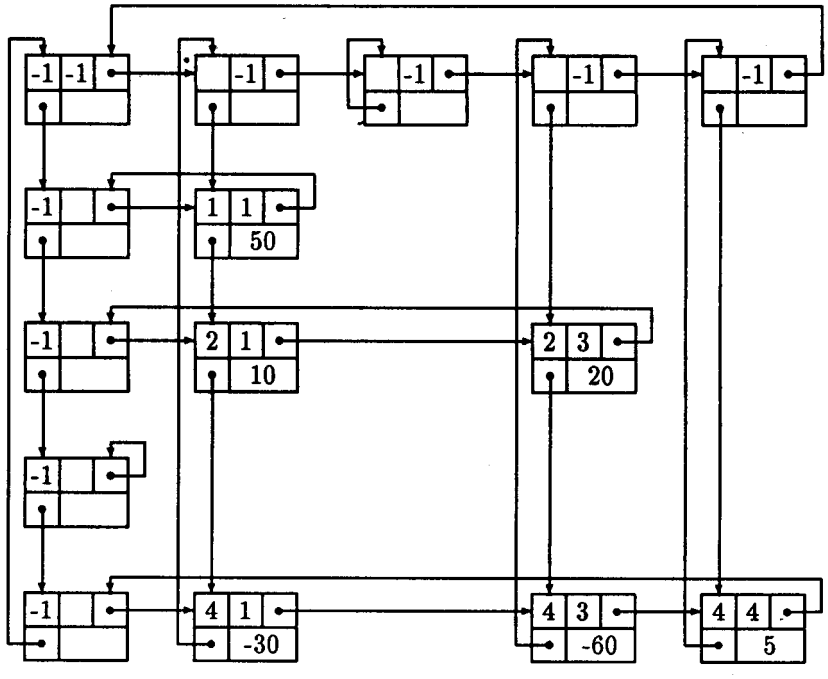


Figura 2.10: Exemplo de Matriz Esparsa

A representação da matriz A pode 'ser vista na Figura 2.10.

Com esta representação, uma matrix esparsa *m x n* com *r* elementos diferentes de zero gastará *(m+n+r)* células. E bem verdade que cada célula ocupa vários bytes na memória; no entanto, o total de memória usado será menor do que as *m x n* posições necessárias para representar a matriz toda, desde que *r* seja suficientemente pequeno.

Dada a representação acima, o trabalho consiste em desenvolver cinco procedimentos em Pascal, conforme especificação abaixo:

a) procedure LeMatriz (var A: Matriz);

Este procedimento lê de algum arquivo de entrada os elementos diferentes de zero de uma matriz e monta a estrutura especificada acima. Considere que a entrada consiste dos valores de *m* e *n* (número de linhas e de colunas da matriz) seguidos de triplas *(i, j, valor)* para os elementos diferentes de zero da matriz. Por

exemplo, para a matriz acima, a entrada seria:

```
4,  4
1,  1,    50.0
2,  1,    10.0
2,  3,    20.0
4,  1,   -30.0
4,  3,   -60.0
4,  4,     5.0
```

b) procedure ApagaMatriz (var A: Matriz);
Este procedimento devolve todas as células da matriz A para a área de memória disponível (use a **procedure** *dispose).*

c) procedure SomaMatriz (var A, B, C: Matriz);
Este procedimento recebe como parâmetros as matrizes A e B, devolvendo em C a soma de *A* com *B.*

d) procedure MultiplicaMatriz (var A, B, C: Matriz);
Este procedimento recebe como parâmetros as matrizes A e B, devolvendo em C o produto de *A* por *B.*

e) procedure ImprimeMatriz (var A: Matriz);
Este procedimento imprime (uma linha da matriz por linha da saída) a matriz A, *inclusive os* elementos iguais a zero.

Para inserir e retirar células das listas que formam a matriz, crie procedimentos especiais para este fim. Por exemplo, um procedimento

procedure Insere (i, j: integer; v: real; var A: Matriz);

para inserir o valor v na linha *i,* coluna *j* da matriz A será útil tanto na **procedure** LeMatriz quanto na **procedure SomaMatriz.**
As matrizes a serem lidas para testar os procedimentos são:

a) A mesma da Figura 2.10 deste enunciado;

b) $\begin{pmatrix} 50 & 30 & 0 & 0 \\ 10 & 0 & -20 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & -5 \end{pmatrix}$

c) $\begin{pmatrix} 3 & 0 & 0 \\ 0 & -1 & 0 \end{pmatrix}$

Os procedimentos deverão ser testados com o seguinte programa:

```
program TestaMatrizesEsparsas;
...
...
begin
    ...
    LeMatriz (A); ImprimeMatriz (A);
    LeMatriz (B); ImprimeMatriz (B);
    SomaMatriz (A, B, C); ImprimeMatriz (C); ApagaMatriz (C);
    MultiplicaMatriz (A, B, C); ImprimeMatriz (C);
    ApagaMatriz (B); ApagaMatriz (C);
    LeMatriz (B);
    ImprimeMatriz (A); ImprimeMatriz (B);
    SomaMatriz (A, B, C); ImprimeMatriz (C);
    MultiplicaMatriz (A, B, C); ImprimeMatriz (C);
    MultiplicaMatriz (B, B, C);
    ImprimeMatriz (B); ImprimeMatriz (B); ImprimeMatriz (C);
    ApagaMatriz (A); ApagaMatriz (B); ApagaMatriz (C);
    ...
end. { TestaMatrizesEsparsas }
```

O que deve ser apresentado:

a) Listagem do programa em Pascal.

b) Listagem dos testes executados.

c) Descrição sucinta (por exemplo, desenho), das estruturas de da-dos e as decisões tomadas relativas aos casos e detalhes de especificação que porventura estejam omissos no enunciado.

d) Estudo da complexidade do tempo de execução dos procedimentos implementados e do programa como um todo (notação 0).

É obrigatório o uso de alocação dinâmica de memória para implementar as listas de adjacência que representam as matrizes. A análise de complexidade deve ser feita em função de m, n (dimensões da matriz) e r (número de elementos diferente de zero).

4) Considere F uma fila não vazia e P uma pilha vazia. Usando apenas a variável temporária x, e as quatro operações

$$x \Leftarrow P\ ,\ P \Leftarrow x\ ,\ x \Leftarrow F\ ,\ F \Leftarrow x$$

e os dois testes P=vazio e F=vazio, escreva um algoritmo para reverter a ordem dos elementos em F.

5) Duas pilhas podem coexistir em um mesmo vetor, uma **crescendo** em um sentido, e a outra no outro. Duas filas, ou uma pilha e uma fila podem também ser alocadas no mesmo vetor com o mesmo grau de eficiência ? Por quê ?

6) Como resolver o problema da representação por alocação seqüencial (usando uma região fixa de memória) para mais de duas pilhas ?

7) Alteração do ET

Alterar a especificação do programa ET, apresentado na seção 2.2.2, para aceitar o comando cancela palavra, cujo efeito é cancelar a **palavra** anterior na linha que está sendo editada. Por exemplo, se `$' é o CancelaPalavra, então a seqüência de caracteres NÃO CANCELA$$ CANCELA LINHA$ PALAVRA corresponde à seqüência CANCELA PALAVRA.

Altere o programa de acordo com a nova especificação proposta. Para testar o programa utilize o texto da página 53, com as seguintes alterações:

a) insira o caractere $ após a palavra "extras" na terceira linha,

b) insira dois caracteres $$ após as palavras "Será que" na penúltima linha do texto.

8) Existem partes de sistemas operacionais que cuidam da ordem em que os programas devem ser executados. Por exemplo, em um sistema de computação de tempo-compartilhado ("time-shared") existe a necessidade de manter um conjunto de processos em uma fila, esperando para serem executados.

Escreva um programa em Pascal que seja capaz de ler uma série de solicitações para: (i) incluir novos processos na fila de processos; (ii) retirar da fila o processo com o maior tempo de espera; (iii) imprimir o conteúdo da lista de processos em um determinado momento. Assuma que cada processo é representado por um registro composto por um número identificador do processo. Utilize o tipo abstrato de dados Fila apresentado na Seção 2.3.

9) Se você tem que escolher entre uma representação por lista **encadeada** ou uma representação usando posições contíguas de memória para um vetor, quais informações são necessárias para você selecionar uma representação apropriada? Como estes fatores influenciam a escolha da representação?

10) Filas, Simulação (Árabe, 1992)

O objetivo deste exercício é simular os padrões de aterrissagem e decolagem em um aeroporto. Suponha um aeroporto que possui 3 pistas, numeradas 1, 2 e 3. Existem 4 (quatro) "prateleiras" de espera para aterrissagem, duas para cada uma das pistas 1 e 2. Aeronaves que se aproximam do aeroporto devem se integrar a uma das prateleiras (filas) de espera, sendo que estas filas devem procurar manter o mesmo tamanho. Assim que um avião entra em uma fila de aterrissagem, ele recebe um número de identificação ID e um outro número inteiro que indica o número de unidades de tempo que o avião pode permanecer na fila antes que ele tenha que descer (do contrário seu combustível termina e ele cai).

Existem também filas para decolagem, uma para cada pista. Os aviões que chegam nestas filas também recebem uma identificação ID. Estas filas também devem procurar manter o mesmo tamanho.

A cada unidade de tempo, de 0 a 3 aeronaves podem chegar nas filas de decolagem e de 0 a 3 aeronaves podem chegar nas prateleiras. A cada unidade de tempo, cada pista pode ser usada para um pouso ou uma decolagem. A pista 3 em geral só é usada para decolagens, a não ser que um dos aviões nas prateleiras fique sem combustível, quando então ela deve ser imediatamente usada para pouso. Se apenas uma aeronave está com falta de combustível, ela pousará na pista 3; se mais de um avião estiver nesta situação, as outras pistas poderão ser utilizadas (a cada unidade de tempo no máximo 3 aviões poderão estar nesta desagradável situação).

Utilize inteiros pares (ímpares) sucessivos para a ID dos aviões chegando nas filas de decolagem (aterrissagem). A cada unidade de tempo, assuma que os aviões entram nas filas antes que aterrissagem ou decolagens ocorram. Tente projetar um algoritmo que não permita o crescimento excessivo das filas de aterrissagem ou decolagem. Coloque os aviões sempre no final das filas, que não devem ser reordenadas.

A saída do programa deverá indicar o que ocorre a cada unidade de tempo. Periodicamente imprima:

a) o conteúdo de cada fila;
b) o tempo médio de espera para decolagem;
c) o tempo médio de espera para aterrissagem; e
d) o número de aviões que aterrissam sem reserva de combustível.

Os itens b e c acima devem ser calculados para os aviões que já decolaram ou pousaram, respectivamente. A saída do programa deve ser auto-explicativa e fácil de entender.

A entrada poderia ser criada manualmente, mas o melhor é utilizar um gerador de números aleatórios. Para cada unidade de tempo, a entrada deve ter as seguintes informações:

a) número de aviões (0-3) chegando nas filas de aterrissagem com respectivas reservas de combustível (de 1 a 20 em unidades de tempo);
b) número de aviões (0-3) chegando nas filas de decolagem.

0 que deve ser apresentado:

a) Listagem dos programas em Pascal.
b) Listagem dos testes executados.
c) Descrição sucinta (por exemplo, desenho), das estruturas de da-dos e as decisões tomadas relativas aos casos e detalhes de especificação que porventura estejam omissos no enunciado.
d) Estudo da complexidade do tempo de execução dos procedimentos implementados e dos programas como um todo (notação O).

Capítulo 3

Ordenação

Os algoritmos de ordenação constituem bons exemplos de como resolver problemas utilizando computadores. As técnicas de ordenação permitem apresentar um conjunto amplo de algoritmos distintos para resolver uma mesma tarefa. Dependendo da aplicação, cada algoritmo considerado possui uma vantagem particular sobre os outros. Além disso, os algoritmos ilustram muitas regras básicas para manipulação de estruturas de dados.

Ordenar corresponde ao processo de rearranjar um conjunto de objetos em uma ordem ascendente ou descendente. O objetivo principal da ordenação é facilitar a recuperação posterior de itens do conjunto ordenado. Imagine como seria difícil utilizar um catálogo telefônico se os nomes das pessoas não estivessem listados em ordem alfabética. A atividade de colocar as coisas em ordem está presente na maioria das aplicações onde os objetos armazenados têm que ser pesquisados e recuperados, tais como dicionários, índices de livros, tabelas e arquivos.

Antes de considerarmos os algoritmos propriamente ditos, é necessário apresentar alguma notação. Os algoritmos trabalham sobre os registros de um arquivo. Apenas uma parte do registro, chamada chave, é utilizada para controlar a ordenação. Além da chave podem existir outros componentes em um registro, os quais não têm influência no processo de ordenar, a não ser pelo fato de que permanecem com a mesma chave. A estrutura de dados registro é a indicada para representar os itens componentes de um arquivo, conforme ilustra o Programa 3.1.

A escolha do tipo para a chave é arbitrária. Qualquer tipo sobre o qual exista uma regra de ordenação bem-definida pode ser utilizado. As ordens numérica e alfabética são as usuais.

Um método de ordenação é dito estável se a ordem relativa dos itens com chaves iguais mantém-se inalterada pelo processo de ordenação. Por exemplo, se uma lista alfabética de nomes de funcionários de uma empresa é ordenada pelo campo salário, então um método estável produz uma lista

```
type Item =record
            Chave : ChaveTipo;
            {outros componentes}
          end;
```

Programa 3.1: Estrutura de um item do arquivo

em que os funcionários com mesmo salário aparecem em ordem alfabética. Alguns dos métodos de ordenação mais eficientes não são estáveis. Se a estabilidade é importante ela pode ser forçada quando o método é não-estável. Sedgewick (1988) sugere agregar um pequeno índice a cada chave antes de ordenar, ou então aumentar a chave de alguma outra forma.

Os métodos de ordenação são classificados em dois grandes grupos. Se o arquivo a ser ordenado cabe todo na memória principal, então o método de ordenação é chamado de **ordenação interna**. Neste caso o número de registros a ser ordenado é pequeno o bastante para caber em um **array** do Pascal. Se o arquivo a ser ordenado não cabe na memória principal e, por isso, tem que ser armazenado em **fita** ou **disco**, então o método de ordenação é chamado de **ordenação externa.** A principal diferença entre os dois métodos é que, em um método de ordenação interna, qualquer registro pode ser imediatamente acessado, enquanto, em um método de ordenação externa, os registros são acessados seqüencialmente ou em grandes blocos.

A grande maioria dos métodos de ordenação são baseados em **comparações** das chaves. Os métodos a serem apresentados a seguir são deste tipo. Entretanto, existem métodos de ordenação que utilizam o princípio da **distribuição.** Por exemplo, considere o problema de ordenar um baralho com 52 **cartas** não ordenadas. Suponha que ordenar o baralho implica em colocar as cartas de acordo com a ordem

$$A < 2 < 3 < \cdots < 10 < J < Q < K$$

e

$$\clubsuit < \diamondsuit < \heartsuit < \spadesuit.$$

Para ordenar por distribuição, basta seguir os passos abaixo:

1. Distribuir as **cartas** abertas em 13 montes, colocando em cada monte todos os ases, todos os dois, todos os três, ..., todos os reis.
2. Colete os montes na ordem acima (ás no fundo, depois os dois, etc.), até o rei ficar no topo).
3. Distribua novamente as cartas abertas em 4 montes, colocando em cada monte todas as cartas de paus, todas as cartas de ouros, todas as cartas de copas e todas as cartas de espadas.

4. Colete os montes na ordem indicada acima (paus, ouros, copas e espadas).

Métodos como o ilustrado acima são também conhecidos como **ordenação digital, radixsort** ou bucketsort. Neste caso não existe comparação entre chaves. As antigas **classificadoras de cartões** perfurados também utilizam o princípio da distribuição para ordenar uma massa de cartões. Uma das dificuldades de implementar este método está relacionada com o problema de lidar com cada monte. Se para cada monte nós reservarmos uma área, então a demanda por memória extra pode se tornar proibitiva. 0 custo para ordenar um arquivo com *n* elementos é da ordem de 0(n), *pois* cada elemento é manipulado algumas vezes.

0 principal objetivo deste capítulo é apresentar os métodos de ordenação mais importantes sob o ponto de vista prático. Cada método será apresentado através de um exemplo, e o algoritmo associado será refinado até o nível de um procedimento Pascal executável. A seguir vamos estudar os principais métodos de ordenação através de comparações de chaves.

3.1 Ordenação Interna

0 aspecto predominante na escolha de um algoritmo de ordenação é o tempo gasto para ordenar um arquivo. Para algoritmos de ordenação interna as medidas de complexidade relevantes contam o número de comparações entre chaves e o número de movimentações (ou trocas) de itens do arquivo. Seja C uma função de complexidade tal que *C(n)* é o número de comparações entre chaves, e seja M uma função de complexidade tal que *M(n)* é o número de movimentações de itens no arquivo, onde *n* é o número de itens do arquivo.

A quantidade extra de memória auxiliar utilizada pelo algoritmo é também um aspecto importante. 0 uso econômico da memória disponível é um requisito primordial na ordenação interna. Os métodos que utilizam a estrutura vetor e que executam a permutação dos itens no próprio vetor, exceto para a utilização de uma pequena tabela ou -pilha, são os preferidos. Os métodos que utilizam listas encadeadas necessitam *n* palavras extras de memória para os apontadores, e são utilizados apenas em algumas situações especiais. Os métodos que necessitam de uma quantidade extra de memória para armazenar uma outra cópia dos itens a serem ordenados possuem menor importância.

Os métodos de ordenação interna são classificados em *métodos simples e métodos eficientes. Os* métodos simples são adequados para pequenos arquivos e requerem $0(n^2)$ comparações, enquanto os métodos eficientes são adequados para arquivos maiores e requerem $0(n \log n)$ comparações. Os métodos simples produzem programas pequenos, fáceis de entender, ilustrando com simplicidade os princípios de ordenação por comparação. Além

```
type Índice  = 0..n;
     Vetor   = array [Índice] of Item;
var A : Vetor;
```

Programa 3.2: Tipos utilizados na implementação dos algoritmos

do mais, existe um grande número de situações em que é melhor usar os métodos simples do que usar os métodos mais sofisticados. Apesar de os métodos mais sofisticados usarem menos comparações, estas comparações são mais complexas nos detalhes, o que torna os métodos simples mais eficientes para pequenos arquivos.

Na implementação dos algoritmos de ordenação interna, serão utilizados o tipo de dados Índice, o tipo de dados Vetor e a variável A apresentados no Programa *3.2*. 0 tipo Vetor é do tipo estruturado arranjo, composto por uma repetição do tipo de dados Item apresentado anteriormente no Programa 3.1. Repare que o índice do vetor vai de 0 até *n*, para poder armazenar chaves especiais chamadas sentinelas.

3.1.1 Ordenação por Seleção

Um dos algoritmos mais simples de ordenação é o método já apresentado na Seção 1.4, cujo princípio de funcionamento é o seguinte: selecione o menor item do vetor e a seguir troque-o com o item que está na primeira posição do vetor. Repita estas duas operações com os *n* — 1 itens restantes, depois com os *n*— *2* itens, até que reste apenas um elemento. 0 método é ilustrado para o conjunto de seis chaves apresentado na Figura 3.1. As chaves em negrito sofreram uma troca entre si.

	1	2	3	4	5	6
Chaves iniciais:	*O*	*R*	*D*	*E*	*N*	*A*
i = 2	***A***	*R*	*D*	*E*	*N*	***O***
i = 3	*A*	**D**	***R***	*E*	*N*	*O*
i = 4	*A*	*D*	**E**	***R***	*N*	*O*
i = 5	*A*	*D*	*E*	**N**	**R**	O
i = 6	*A*	*D*	*E*	*N*	*O*	***R***

Figura 3.1: Exemplo de ordenação por seleção

O Programa *3.3* mostra a implementação do algoritmo, para um conjunto de itens implementado como um tipo Vetor.

```
procedure Seleção (var A : Vetor);
var  i, j, Min : Índice;
     x         : Item;
begin
for i := 1 to n-1 do
  begin
  Min := i;
  for j := i +1 to n do if A[j].Chave < A[Min].Chave then Min := j;
  x := A[Min]; A[Min] := A[i]; A[i] := x;
  end;
end;
```

Programa 3.3: Ordenação por seleção

Análise

$$C(n) = \frac{n^2}{2} - \frac{n}{2}$$

$$M(n) = 3(n-1)$$

O comando de atribuição $Min := j$ é executado em média cerca de n log n vezes, conforme pode ser verificado em Knuth (1973, exercícios 5.2.3.3-6) . Este valor é difícil de obter exatamente: ele depende do número de vezes que c, é menor do que todas as chaves anteriores c_1, c_2, ..., quando estamos percorrendo as chaves c_1, c_2, ... , c, .

0 algoritmo de ordenação por seleção é um dos métodos de ordenação mais simples que existem. Além disso, o método possui um comportamento espetacular quanto ao número de movimentos de registros, cujo tempo de execução é linear no tamanho da entrada, o que é muito difícil de ser batido por qualquer outro método. Conseqüentemente, este é o algoritmo a ser utilizado para arquivos com registros muito grandes. Em condições normais, com chaves do tamanho de uma palavra, este método é bastante interessante para arquivos com até 1000 registros.

Como aspectos negativos cabe registrar que: (i) o fato do arquivo já estar ordenado não ajuda em nada pois o custo continua quadrático; (ii) o algoritmo não é estável, pois ele nem sempre deixa os registros com chaves iguais na mesma posição relativa.

3.1.2 Ordenação por Inserção

Este é o método preferido dos jogadores de cartas. Em cada passo, a partir de i=2, o i-ésimo item da seqüência fonte é apanhado e transferido para a seqüência destino, sendo inserido no seu lugar apropriado. O método

é ilustrado para as mesmas seis chaves utilizadas anteriormente, conforme apresentado na Figura 3.2. As chaves em negrito representam a seqüência destino.

	1	2	3	4	5	6
Chaves iniciais:	**O**	*R*	*D*	*E*	*N*	*A*
i = 2	**O**	**R**	*D*	*E*	*N*	*A*
i = 3	**D**	**O**	**R**	*E*	*N*	*A*
i = 4	**D**	**E**	**O**	**R**	*N*	*A*
i = 5	**D**	**E**	**N**	**O**	**R**	*A*
i = 6	**A**	**D**	**E**	**N**	**O**	**R**

Figura 3.2: Exemplo de ordenação por inserção

0 Programa 3.4 mostra a implementação do algoritmo, para um conjunto de itens implementado como um tipo Vetor. A colocação do item no seu lugar apropriado na seqüência destino é realizada movendo-se itens com chaves maiores para a direita e então inserindo o item na posição deixada vazia. Neste processo de alternar comparações e movimentos de registros existem duas condições distintas que podem causar a terminação do processo: (i) um item com chave menor que o item em consideração é encontrado; (ii) o final da seqüência destino é atingido à esquerda. A melhor solução para a situação de um anel com duas condições de terminação é a utilização de um registro **sentinela:** na posição zero do vetor colocamos o próprio registro em consideração. Para tal o índice do vetor tem que ser estendido para 0..n.

Análise

No anel mais interno, na i-ésima iteração o valor de C_i é:

$$\text{melhor caso} : C_i(n) = 1$$
$$\text{pior caso} : C_i(n) = i$$
$$\text{caso médio} : C_i(n) = 1/i(1 + 2 + \cdots + i) = \frac{i+1}{2}$$

assumindo que todas as permutações de n são igualmente prováveis para o caso médio. Logo, o número de comparações é igual a

$$\text{melhor caso} : C(n) = (1 + 1 + \cdots + 1) = n - 1$$
$$\text{pior caso} : C(n) = (2 + 3 + \cdots + n) = \frac{n^2}{2} + \frac{n}{2} - 1$$
$$\text{caso médio} : C(n) = \frac{1}{2}(3 + 4 + \cdots + n + 1) = \frac{n^2}{4} + \frac{3n}{4} - 1$$

```
procedure Inserção (var A : Vetor);
var  i, j : Índice;
     x   : Item;
begin
for i := 2 to n do
  begin
  x := A[i];
  j := i-1;
  A[0] := x;          {— sentinela —}
  while x.Chave < A[j].Chave do
    begin
    A[j+1] := A[j];
    j := j-1;
    end;
  A[j+1] := x;
  end;
end;
```

Programa 3.4: Ordenação por inserção

O número de movimentações na i-ésima iteração é igual a

$$M_i(n) = C_i(n) - 1 + 3 = C_i(n) + 2$$

Logo, o número de movimentos é igual a

$$\text{melhor caso} : M(n) = (3 + 3 + \cdots + 3) = 3(n-1)$$

$$\text{pior caso} : M(n) = (4 + 5 + \cdots + n + 2) = \frac{n^2}{2} + \frac{5n}{2} - 3$$

$$\text{caso médio} : M(n) = (1/2)(5 + 6 + \cdots + n + 3) = \frac{n^2}{4} + \frac{11n}{4} - 3$$

O número mínimo de comparações e movimentos ocorre quando os itens estão originalmente em ordem, e o número máximo ocorre quando os itens estão originalmente na ordem reversa, o que indica um comportamento natural para o algoritmo. Para arquivos já ordenados o algoritmo descobre a um custo *0(n)* que cada item já está no seu lugar. Logo, o método da inserção é o método a ser utilizado quando o arquivo está "quase" ordenado. E também um bom método quando se deseja adicionar uns poucos itens a um arquivo já ordenado e depois obter um outro arquivo ordenado: neste caso o custo é linear. O algoritmo de ordenação por inserção é quase tão simples quanto o algoritmo de ordenação por seleção. Além disso, o método de ordenação por inserção é estável, pois ele deixa os registros com chaves iguais na mesma posição relativa.

3.1.3 Shellsort

Shell (1959) propôs uma extensão do algoritmo de ordenação por inserção. O método da inserção troca itens adjacentes quando está procurando o ponto de inserção na seqüência destino. Se o menor item estiver na posição mais a direita no vetor então o número de comparações e movimentações é igual a n — 1 para encontrar o seu ponto de inserção.

O método de Shell contorna este problema permitindo trocas de registros que estão distantes um do outro. Os itens que estão separados h posições são rearranjados de tal forma que todo h-ésimo item leva a uma seqüência ordenada. Tal seqüência é dita estar h-ordenada. A Figura *3.3* mostra como um arquivo de seis itens é ordenado usando os incrementos 4, *2,* e 1 para h.

	1	2	3	4	5	6
Chaves iniciais:	*O*	*R*	*D*	*E*	*N*	*A*
h = 4	*N*	*A*	*D*	*E*	*O*	*R*
h = 2	*D*	*A*	*N*	*E*	*O*	*R*
h = 1	*A*	*D*	*E*	*N*	*O*	*R*

Figura 3.3: Exemplo de ordenação usando Shellsort

Na primeira passada (h = 4), o O é comparado com o N (posições 1 e *5)* e trocados; a seguir o R é comparado com o A (posições 2 e *6)* e trocados. Na segunda passada (h = 2), N, D e O, nas posições 1, *3* e *5,* são rearranjados para resultar em D, N e O nestas mesmas posições; da mesma forma A, E e R, nas posições 2, 4 e *6,* são comparados e mantidos nos seus lugares. A última passada (h = 1) corresponde ao algoritmo de inserção: entretanto, nenhum item tem que se mover para posições muito distantes.

Várias seqüências para h têm sido experimentadas. Knuth (1973, *p.95)* mostrou experimentalmente que a escolha do incremento para h, mostrada abaixo, é difícil de ser batida por mais de 20% em eficiência no tempo de execução:

$$h(s) = 3h(s-1) + 1, \quad \text{para } s > 1$$
$$h(s) = 1, \quad \text{para } s = 1.$$

A seqüência para h corresponde a 1, 4, 13, *40,* 121, *364,* 1093, *3280,* .... O Programa *3.5* mostra a implementação do algoritmo, para a seqüência mostrada acima. Observe que não foram utilizados registros **sentinelas** porque teríamos que utilizar h sentinelas, uma para cada h-ordenações.

Análise

A razão pela qual este método é eficiente ainda não é conhecida, porque ninguém ainda foi capaz de analisar o algoritmo. A sua análise contém al-

```
procedure Shellsort (var A : Vetor);
label 999;
var  i, j, h : integer;
     x       : Item;
begin
  h := 1;
  repeat h := 3* h +1 until h > n;
  repeat
    h := h div 3;
    for i := h +1 to n do
      begin
      x := A[i];
      j := i;
      while A[j − h].Chave > x.Chave do
        begin
        A[j] := A[j − h];
        j := j − h;
        if j ≤ h then goto 999;
        end;
      999 : A[j] := x;
      end;
    until h = 1;
end;
```

Programa 3.5: Algoritmo Shellsort

guns problemas matemáticos muito difíceis, a começar pela própria seqüência de incrementos: o pouco que se sabe é que cada incremento não deve ser múltiplo do anterior. Para a seqüência de incrementos utilizada no Programa 3.5 existem duas conjeturas para o número de comparações, a saber:

$$\text{Conjetura 1} : C(n) = O(n^{1,25})$$

$$\text{Conjetura 2} : C(n) = O(n(\ln\, n)^2)$$

Shellsort é uma ótima opção para arquivos de tamanho moderado (da ordem de 5000 registros), mesmo porque sua implementação é simples e requer uma quantidade de código pequena. Existem métodos de ordenação mais eficientes, mas são também muito mais complicados para implementar. O tempo de execução do algoritmo é sensível à ordem inicial do arquivo. Além disso o método não é estável, pois ele nem sempre deixa registros com chaves iguais na mesma posição relativa.

3.1.4 *Quicksort*

Quicksort é o algoritmo de ordenação interna mais rápido que se conhece para uma ampla variedade de situações, sendo provavelmente mais utilizado do que qualquer outro algoritmo. O algoritmo foi inventado por C. A. R. Hoare em 1960, quando visitava a Universidade de Moscou como estudante. O algoritmo foi publicado mais tarde por Hoare (1962), após uma série de refinamentos.

A idéia básica é a de partir o problema de ordenar um conjunto com n itens em dois problemas menores. A seguir, os problemas menores são ordenados independentemente e depois os resultados são combinados para produzir a solução do problema maior.

A parte mais delicada deste método é relativa ao procedimento partição, o qual tem que rearranjar o vetor *A[Esq..Dir]* através da escolha arbitrária de um item *x* do vetor chamado pivô, de tal forma que ao final o vetor *A* está particionado em uma parte esquerda com chaves menores ou iguais a x e uma parte direita com chaves maiores ou iguais a x.

Este comportamento pode ser descrito pelo seguinte algoritmo:

1. escolha arbitrariamente um item do vetor e coloque-o em x;
2. percorra o vetor a partir da esquerda até que um item $A[i] \geq x$ é encontrado; da mesma forma percorra o vetor a partir da direita até que um item $A[j] \leq x$ é encontrado;
3. como os dois itens $A[i]$ e $A[j]$ estão fora de lugar no vetor final então troque-os de lugar;
4. continue este processo até que os apontadores i e j se cruzem em algum ponto do vetor.

Ao final o vetor $A[Esq..Dir]$ está particionado de tal forma que:

- os itens em $A[Esq], A[Esq+1], \ldots, A[j]$ são menores ou iguais a x,
- os itens em $A[i], A[i+1], \ldots, A[Dir]$ são maiores ou iguais a x.

O método é ilustrado para o conjunto de seis chaves apresentado na Figura 3.4. O item x é escolhido como sendo $A[(i+j)\mathbf{div}\,2]$. Como inicialmente $i = 1$ e $j = 6$, então $x = A[3] = D$, o qual aparece em negrito na segunda linha da mesma figura. A varredura a partir da posição 1 pára no item O e a varredura a partir da posição 6 pára no item A, sendo os dois itens trocados, como mostrado na terceira linha da Figura 3.4. A seguir a varredura a partir da posição 2 para no item R e a varredura a partir da posição 5 pára no item D, e então os dois itens são trocados, como mostrado na quarta linha.

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

Figura 3.4: Partição do vetor

```
procedure Partição (Esq, Dir : Índice; var i, j : Índice);
var x, w : Item;
begin
  i := Esq; j := Dir;
  x := A[(i+j) div 2]; {— obtém o pivô x —}
  repeat
    while x.Chave > A[i].Chave do i := i+1;
    while x.Chave < A[j].Chave do j := j−1;
    if i ≤ j
    then begin
           w := A[i]; A[i] := A[j]; A[j] := w;
           i := i+1; j := j−1;
           end;
  until i > j;
end;
```

Programa 3.6: Procedimento Partição

Neste momento i e j se cruzam ($i = 3$ e $j = 2$), o que encerra o processo de partição.

O Programa 3.6 mostra a implementação do procedimento Partição, onde Esq e Dir são apontadores para delimitar o subvetor dentro do vetor original A, o qual deve ser particionado. Os índices i e j retornam as posições finais das partições, onde $A[Esq], A[Esq + 1], \ldots A[j]$ são menores ou iguais ao **pivô** x, e $A[i], A[i + 1], \ldots, A[Dir]$ são maiores ou iguais a x. O vetor A é considerado global ao procedimento partição.

Observe que o anel interno do procedimento partição consiste apenas em incrementar um apontador e comparar um item do vetor contra um valor fixo em x. Este anel é extremamente simples, razão pela qual o algoritmo Quicksort é tão rápido.

Após obter os dois pedaços do vetor através do procedimento partição, cada pedaço é ordenado recursivamente. O refinamento final do procedimento Quicksort é mostrado no Programa 3.7. O procedimento Ordena é **recursivo**, e o vetor A é global aos procedimentos partição e ordena.

A Figura 3.5 ilustra o que acontece com o vetor exemplo em cada chamada recursiva do procedimento ordena. Cada linha mostra o resultado do

```
procedure Quicksort (var A : Vetor);
{— Entra aqui o procedimento partição (Programa 3.6) —}
  procedure Ordena (Esq, Dir : Índice);
  var i, j : Índice;
  begin
    partição (Esq, Dir, i, j);
    if Esq < j then Ordena (Esq, j);
    if i < Dir then Ordena (i, Dir);
  end;
begin
  Ordena (1, n);
end;
```

Programa 3.7: Procedimento Quicksort

procedimento partição, onde o **pivô** é mostrado em negrito.

Chaves iniciais:	*O*	*R*	*D*	*E*	*N*	*A*
1	*A*	**D**	*R*	*E*	*N*	*O*
2	**A**	*D*				
3			**E**	*R*	*N*	*O*
4				**N**	*R*	*O*
5					*O*	**R**
	A	*D*	*E*	*N*	*O*	*R*

Figura 3.5: Exemplo de ordenação usando Quicksort

Análise

Uma característica interessante do Quicksort é a sua ineficiência para arquivos já ordenados quando a escolha do pivô é inadequada. Por exemplo, a escolha sistemática dos extremos de um arquivo já ordenado leva ao seu pior caso. Neste caso, as partições serão extremamente desiguais, e o procedimento Ordena será chamado recursivamente n vezes, eliminando apenas um item em cada chamada. Esta situação é desastrosa pois o número de comparações passa a cerca de $n^2/2$, e o tamanho da pilha necessária para as chamadas recursivas é cerca de n. Entretanto, o pior caso pode ser evitado através de pequenas modificações no programa, conforme veremos mais adiante.

A melhor situação possível ocorre quando cada partição divide o arquivo em duas partes iguais. Logo,

$$C(n) = 2C(n/2) + n$$

onde $C(n/2)$ representa o custo de ordenar uma das metades e n é o custo de examinar cada item. A solução para esta recorrência é

$$C(n) \text{ —} 1,4n \log n$$

o que significa que em média o tempo de execução do Quicksort *é* $O(n \log n)$.

Quicksort é extremamente eficiente para ordenar arquivos de dados. O método necessita de apenas uma pequena pilha como memória auxiliar, e requer cerca de n log *n* operações em média para ordenar *n* itens. Como aspectos negativos cabe ressaltar que: (i) a versão recursiva do algoritmo tem um pior caso que *é* $0(n^2)$ operações; (ii) a implementação do algoritmo é muito delicada e :difícil: um pequeno engano pode levar a efeitos inesperados para algumas entradas de dados; (iii) o método não é estável.

Uma vez que se consiga uma implementação robusta para o Quicksort, este deve ser o algoritmo preferido para a maioria das aplicações. No caso de se necessitar de um programa utilitário para uso freqüente, então vale a pena investir na obtenção de uma implementação do algoritmo. Por exemplo, como evitar o pior caso do algoritmo? A melhor solução para este caso é escolher três itens quaisquer do arquivo e usar a mediana dos três como o item divisor na partição.

3.1.5 Heapsort

Heapsort é um método de ordenação cujo princípio de funcionamento é o mesmo princípio utilizado para a ordenação por seleção, a saber: selecione o menor item do vetor e a seguir troque-o com o item que está na primeira posição do vetor; repita estas duas operações com os n — *1* itens restantes, depois com os n — 2 itens, e assim sucessivamente.

0 custo para encontrar o menor (ou o maior) item entre *n* itens custa n — 1 comparações. Este custo pode ser reduzido através da utilização de uma estrutura de dados chamada fila de prioridades. Devido a enorme importância das filas de prioridades para muitas aplicações (inclusive ordenação), a próxima seção será dedicada ao seu estudo.

Filas de Prioridades

No estudo de listas lineares, no Capítulo 2, vimos que a operação de desempilhar um item de uma pilha retira o último item inserido (o mais novo), e a operação de desenfileirar um item de uma fila retira o primeiro item inserido

(o mais velho). Em muitas situações é necessário uma estrutura de dados que suporte as operações de inserir um novo item e retirar o item com a maior chave. Tal estrutura de dados é chamada fila de prioridades, porque a chave de cada item reflete sua habilidade relativa de abandonar o conjunto de itens rapidamente.

Filas de prioridades são utilizadas em um grande número de aplicações. Sistemas operacionais usam filas de prioridades, onde as chaves representam o tempo em que eventos devem ocorrer. Alguns métodos numéricos iterativos são baseados na seleção repetida de um item com maior (menor) valor. Sistemas de gerência de memória usam a técnica de substituir a página menos utilizada na memória principal do computador por uma nova página.

As operações mais comuns sobre o tipo fila de prioridades são: adicionar um novo item ao conjunto e extrair o item do conjunto que contenha o maior (menor) valor. Entretanto, filas de prioridades permitem a execução de um grande número de operações de forma eficiente. Um **tipo abstrato** de **dados** Fila de Prioridades, contendo registros com chaves numéricas (prioridades), deve suportar algumas das seguintes operações:

1. Constrói uma fila de prioridades a partir de um conjunto com *n* itens.
2. Insere um novo item.
3. Retira o item com maior chave.
4. Substitui o maior item por um novo item, a não ser que o novo item seja maior.
5. Altera a prioridade de um item.
6. Remove um item qualquer.
7. Ajunta duas filas de prioridades em uma única.

A única diferença entre a operação substitui e as operações encadeadas insere/retira é que as operações encadeadas fazem com que a fila de prioridades aumente temporariamente de tamanho. A operação constrói é equivalente ao uso repetido da operação insere, e a operação altera é equivalente a operação remove seguida de insere.

Uma representação óbvia para uma fila de prioridades é uma lista linear ordenada. Neste caso Constrói leva tempo $O(n \log n)$, insere é $0(n)$ e retira é 0(1). Outra representação é através de uma lista linear não ordenada, na qual a operação constrói tem custo linear, insere é 0(1), retira é $0(n)$ e ajunta é 0(1) para implementações através de apontadores *e* $0(n)$ para implementações através de arranjos, onde *n* representa o tamanho da menor fila de prioridades.

Filas de prioridades podem ser melhor representadas por estruturas de dados chamadas *heaps*. *A* operação constrói tem custo linear, e insere, retira, substitui e altera tem custo logarítmico. Para implementar a operação ajunta de forma eficiente e ainda preservar um custo logarítmico para as operações insere, retira, substitui e altera é necessário utilizar estruturas de dados mais sofisticados, tais como árvores binomiais (Vuillemin, 1978).

Qualquer algoritmo para filas de prioridades pode ser transformado em um algoritmo de ordenação, através do uso repetido da operação insere para construir a fila de prioridades, seguido do uso repetido da operação retira para receber os itens na ordem reversa. Neste esquema, o uso de listas lineares não ordenadas corresponde ao método da seleção; o uso de listas lineares **ordenadas** corresponde ao método da inserção; o uso de *heaps* corresponde ao método Heapsort.

Heaps

Uma estrutura de dados eficiente para suportar as operações constrói, insere, retira, substitui e altera é o *heap*, proposta por Williams (1964). Um *heap é* definido como uma seqüência de itens com chaves

c[1], c[2], ... , *c[*

n] tal que

$c[i] > c[2i]$

$c[i] > c[2i + 1]$

para todo i = *1, 2, ... , n/2*. Esta ordem pode ser facilmente visualizada se a seqüência de chaves for desenhada em uma árvore binária, onde as linhas que saem de uma chave levam a duas chaves menores no nível inferior, conforme ilustra a Figura. 3.6. Esta estrutura é conhecida como uma árvore binária completa[1]: o primeiro nodo é chamado raiz, os nodos abaixo de cada nodo são chamados nodos filhos, e o nodo acima de cada nodo é chamado nodo pai. Um estudo mais detalhado de árvores será apresentado no Capítulo 4.

Observe que as chaves na árvore da Figura 3.6 satisfazem a condição do *heap:* a chave em cada nodo é maior do que as chaves em seus filhos, se eles existirem. Conseqüentemente, a chave no nodo raiz é a maior chave do conjunto.

[1]Uma **árvore binária completa** é uma árvore binária com os nodos numerados de 1 a n, onde o nodo *Lk/2* j é o pai do nodo *k*, para 1 < *k* < n. Em outras palavras, em uma árvore binária completa os nodos externos aparecem em dois níveis adjacentes e os nodos no nível mais baixo estão posicionados mais à esquerda.

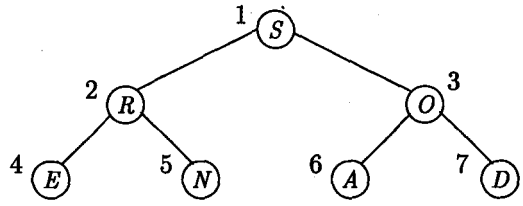


Figura 3.6: Árvore binária completa

Uma árvore binária completa pode ser **representada** por um **array,** conforme ilustra a Figura 3.7. Esta representação é extremamente compacta e, além disso, permite caminhar pelos nodos da árvore facilmente: os filhos de um nodo *i* estão nas posições *2i e 2i* + 1 (caso existam), e o pai de um nodo i está na posição i div 2.

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

Figura 3.7: Árvore binária completa representada por um arranjo

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

Heapsort

O primeiro passo é construir o *heap.* Um método elegante e que não necessita de memória auxiliar alguma foi apresentado por Floyd (1964). Dado um vetor $A[1], A[2], \ldots, A[n]$, os itens $A[n/2+1], A[n/2+2], \ldots, A[n]$ formam um *heap*, porque neste intervalo do vetor não existem dois índices i e j tais que $j = 2i$ ou $j = 2i+1$.

No caso das chaves iniciais da Figura 3.8, os itens de $A[4]$ a $A[7]$ formam a parte inferior da árvore binária associada, onde nenhuma relação de ordem é necessária para formarem um *heap.* A seguir o *heap* é estendido para a esquerda ($Esq = 3$), englobando o item $A[3]$, pai dos itens $A[6]$ e $A[7]$. Neste momento a condição do *heap* é violada, e os itens D e S são trocados, conforme ilustra a segunda linha da Figura 3.8. A seguir, o *heap* é novamente estendido para a esquerda ($Esq = 2$) incluindo o item R, passo que não viola a condição do *heap.* Finalmente, o *heap* é estendido para a esquerda ($Esq = 1$), incluindo o item O, e os itens O e S são trocados, encerrando o processo.

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

Figura 3.8: Construção do *heap*

O Programa 3.8 mostra a implementação do algoritmo para construir o *heap*. O procedimento Refaz reconstrói o *heap* conforme descrito acima.

```
procedure Constrói (var A : Vetor);
var Esq : Índice;

  procedure Refaz (Esq, Dir : Índice; var A : Vetor);
  label 999;
  var  i, j : Índice;
       x   : Item;
  begin
    i := Esq; j := 2 * i;
    x := A[i];
    while j ≤ Dir do
      begin
      if j < Dir
      then if A[j].Chave < A[j+1].Chave then j := j+1;
      if x.Chave ≥ A[j].Chave then goto 999;
      A[i] := A[j];
      i := j; j := 2 * i;
      end;
    999 : A[i] := x;
  end;

begin {— Constrói —}
  Esq := (n div 2) +1;
  while Esq > 1 do
    begin
    Esq := Esq −1;
    Refaz (Esq, n, A);
    end;
end;
```

Programa 3.8: Procedimento para construir o *heap*

A partir do *heap* obtido pelo procedimento Constrói, pega-se o item na posição 1 do vetor (raiz do *heap)* e troca-se com o item que está na posição n do vetor. A seguir, basta usar o procedimento Refaz para reconstituir o *heap* para os itens A[1], A[2], ..., *A[n -1].* Repita estas duas operações com os *n* — 1 itens restantes, depois com *os n — 2* itens, até que reste apenas um item. Este método é exatamente o que o Heapsort faz, conforme ilustra a Figura 3. 9. 0 caminho seguido pelo procedimento Refaz para reconstituir a condição do *heap* está em negrito. Por exemplo, após a troca dos itens S e D na segunda linha da Figura 3.9, o item D volta para a posição 5 após passar pelas posições 1 e *2*.

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

Figura 3.9: Exemplo de ordenação usando Heapsort

O Programa 3.9 mostra a implementação do algoritmo, para um conjunto de itens implementado como um tipo Vetor.

Análise

A primeira vista o algoritmo não parece eficiente, pois as chaves são movimentadas várias vezes. Entretanto, o procedimento Refaz gasta cerca de log *n* operações, no pior caso. Logo, Heapsort gasta um tempo de execução proporcional a *n* log *n* no pior caso!

Heapsort não é recomendado para arquivos com poucos registros, por causa do tempo necessário para construir o *heap,* bem como porque o anel interno do algoritmo é bastante complexo, se comparado com o anel interno do Quicksort. De fato, Quicksort é, em média, cerca de duas vezes mais rápido que o Heapsort. Entretanto, Heapsort é melhor que o Shellsort para grandes arquivos. Um aspecto importante a favor do Heapsort é que o seu comportamento é *O(n log n),* qualquer que seja a entrada. Aplicações que não podem tolerar eventualmente um caso desfavorável devem usar o Heapsort. Um aspecto negativo sobre o Heapsort é que o método não é estável.

```
procedure Heapsort (var A : Vetor);
var  Esq, Dir : Índice;
     x        : Item;

{— Entra aqui o procedimento Refaz do Programa 3.8 —}

begin
  {— constrói o heap —}
  Esq := (n div 2) +1;
  Dir := n;
  while Esq > 1 do
    begin
    Esq := Esq −1;
    Refaz (Esq, Dir, A);
    end;
  {— ordena o vetor —}
  while Dir > 1 do
    begin
    x := A[1];
    A[1] := A[Dir];
    A[Dir] := x;
    Dir := Dir −1;
    Refaz (Esq, Dir, A);
    end;
end;
```

Programa 3.9: Procedimento Heapsort

3.1.6 Comparação Entre os Métodos

A ordenação interna é utilizada quando todos os registros do arquivo ca-bem na memória principal. Neste capítulo apresentamos cinco métodos de ordenação interna através de comparação de chaves. Foram estudados dois métodos simples (Seleção e Inserção) que requerem $0(n^2)$ comparações e três métodos eficientes (Shellsort, Quicksort e Heapsort) que requerem $0(n \log n)$ comparações (apesar de não se conhecer analiticamente o comportamento do Shellsort, ele é considerado um método eficiente).

As Tabelas 3.1, 3.2 e 3.3 apresentam quadros comparativos do tempo total real para ordenar arranjos com 500, 5.000, 10.000 e 30.000 registros na ordem aleatória, na ordem ascendente e na ordem descendente, respectivamente. Em cada tabela, o método que levou menos tempo real para executar recebeu o valor 1 e os outros receberam valores relativos a ele. Assim, na Tabela 3.1, o Shellsort levou o dobro do tempo do Quicksort para ordenar 30.000 registros.

	500	5000	10000	30000
Inserção	11.3	87	161	-
Seleção	16.2	124	228	-
Shellsort	1.2	1.6	1.7	2
Quicksort	1	1	1	1
Heapsort	1.5	1.6	1.6	1.6

Tabela 3.1: Ordem aleatória dos registros

	500	15000	110000	30000
Inserção	1	1	1	1
Seleção	128	1524	3066	-
Shellsort	3.9	6.8	7.3	8.1
Quicksort	4.1	6.3	6.8	7.1
Heapsort	12.2	20.8	22.4	24.6

Tabela 3.2: Ordem ascendente dos registros

	500	5000	10000	30000
Inserção	40.3	305	575	-
Seleção	29.3	221	417	-
Shellsort	1.5	1.5	1.6	1.6
Quicksort	1	1	1	1
Heapsort	2.5	2.7	2.7	2.9

Tabela 3.3: Ordem descendente dos registros

A seguir, apresentamos algumas observações sobre cada um dos métodos.

1. Shellsort, Quicksort e Heapsort têm a mesma ordem de grandeza.

2. O Quicksort é o mais rápido para todos os tamanhos aleatórios experimentados.

3. A relação Heapsort/Quicksort se mantém constante para todos os tamanhos, sendo o Heapsort mais lento.

4. A relação Shellsort/Quicksort aumenta à medida que o número de elementos aumenta; para arquivos pequenos (500 elementos) o Shellsort é mais rápido que o Heapsort, porém quando o tamanho da entrada cresce, esta relação se inverte.

5. O Inserção é o mais rápido para qualquer tamanho se os elementos estão ordenados; este é seu melhor caso, que é 0(n). Ele é o mais lento para qualquer tamanho se os elementos estão em ordem descendente; este é o seu pior caso.

6. Entre os algoritmos de custo $0(n^2)$ o Inserção é melhor para todos os tamanhos aleatórios experimentados.

A Tabela 3.4 mostra a influência da ordem inicial do arquivo sobre cada um dos três métodos mais eficientes. Ao observar a tabela podemos notar que:

	Shellsort			Quicksort			Heapsort		
	5000	10000	30000	5000	10000	30000	5000	10000	30000
Asc	1	1	1	1	1	1	1.1	1.1	1.1
Des	1.5	1.6	1.5	1.1	1.1	1.1	1	1	1
Ale	2.9	3.1	3.7	1.9	2.0	2.0	1.1	1	1

Tabela 3.4: Influência da ordem inicial

1. O Shellsort é bastante sensível à ordenação ascendente ou descendente da entrada; para arquivos do mesmo tamanho executa mais rápido se o arquivo estiver ordenado do que se os elementos forem aleatórios.

2. O Quicksort é sensível à ordenação ascendente ou descendente da entrada; para arquivos do mesmo tamanho executa mais rápido se o arquivo estiver ordenado do que se os elementos forem aleatórios. Ele é o mais rápido para qualquer tamanho se os elementos estão em ordem ascendente.

3. O Heapsort praticamente não é sensível à ordenação da entrada; para arquivos do mesmo tamanho executa 10% mais rápido se o arquivo estiver ordenado do que se os elementos forem aleatórios.

O método da Inserção é o mais interessante para arquivos com menos do que 20 elementos, podendo ser mais eficiente do que algoritmos que tenham comportamento assintótico mais eficiente. O método é estável e seu comportamento é melhor do que outro método estável muito citado na literatura, o Bubblesort ou método da bolha. Além disso, sua implementção é tão simples quanto 'as implementações do Bubblesort e Seleção. Para arquivos já ordenados o método é 0(n): quando se deseja adicionar alguns elementos a um arquivo já ordenado e depois obter um outro arquivo ordenado o custo é linear.

O método da Seleção somente é vantajoso quanto ao número de movimentos de registros, que é 0(n). Logo, ele deve ser usado para arquivos com registros muito grandes, desde que o tamanho do arquivo não seja maior do que 1.000 elementos.

O Shellsort é o método a ser escolhido para a maioria das aplicações por ser muito eficiente para arquivos de até 10.000 elementos. Mesmo para

arquivos grandes o método é cerca de apenas duas vezes mais lento do que o Quicksort. Sua implementação é simples e fácil de colocar funcionando corretamente e geralmente resulta em um programa pequeno. Ele não possui um pior caso ruim e quando encontra um arquivo parcialmente ordenado trabalha menos.

O **Quicksort** é o algoritmo mais eficiente que existe para uma grande variedade de situações. Entretanto, é um método bastante frágil no sentido de que qualquer erro de implementação pode ser difícil de ser detectado. O algoritmo é recursivo, o que demanda uma pequena quantidade de memória adicional. Além disso, seu desempenho é da ordem de $O(n^2)$ operações no pior caso.

Uma vez que se consiga uma implementação robusta, o Quicksort deve ser o método a ser utilizado. 0 principal cuidado a ser tomado é com relação escolha do **pivô**. A escolha do elemento do meio do arranjo melhora muito o desempenho quando o arquivo está total ou parcialmente ordenado, e o pior caso nestas condições tem uma probabilidade muito remota de ocorrer quando os elementos forem aleatórios. A melhor solução para tornar o pior caso mais improvável ainda é escolher ao acaso uma pequena amostra do arranjo e usar a mediana da amostra como pivô na partição. Geralmente se usa a **mediana** de uma amostra **de três elementos.** Além de tornar o pior caso muito mais improvável esta solução melhora o caso médio ligeiramente.

Outra importante melhoria para o desempenho do Quicksort é evitar chamadas recursivas para **pequenos subarquivos,** através da chamada de um método de ordenação simples, como o método da Inserção. Para tal, basta colocar um teste no início do procedimento recursivo Ordena do Pro-grama 3.7 para verificar o tamanho do subarquivo a ser ordenado: para arquivos com menos do que 25 elementos o algoritmo da Inserção deve ser chamado (a implementação do algoritmo da Inserção deve ser alterada para aceitar parâmetros indicando os limites do subarquivo a ser ordenado). A melhoria no desempenho é significativa, podendo chegar a *20%* para a maio-ria das aplicações (Sedgewick, 1988).

O **Heapsort** é um método de ordenação elegante e eficiente. Apesar de possuir um anel interno relativamente complexo, que o torna cerca de duas vezes mais lento do que o Quicksort, ele não necessita nenhuma memória adicional. Além disso, ele executa sempre em tempo proporcional a $n \log n$, qualquer que seja a ordem inicial dos elementos do arquivo de entrada. Aplicações que não podem tolerar eventuais variações no tempo esperado de execução devem usar o Heapsort.

Finalmente, quando os registros do arquivo A[1], A[2], ..., *A[n]* são muito grandes é desejável que o método de ordenação realize apenas n movimentos dos registros, através do uso de uma **ordenação indireta.** Isto pode ser realizado através da utilização de um arranjo P[1], P[2], ..., *P[n]* de aponta-dores, um apontador para cada registro: os registros somente são acessados

para fins de comparações e toda movimentação é realizada apenas sobre os apontadores. Ao final P[1] contém o índice do menor elemento de A, P[2] contém o índice do segundo menor elemento de A, e assim sucessivamente. Esta estratégia pode ser utilizada para qualquer dos métodos de ordenação interna vistos anteriormente. Sedgewick (1988) mostra como implementar esta estratégia para o método da **Inserção** e para aplicações de **filas de prioridades** usando **Heaps.**

3.2 Ordenação Externa

A ordenação externa envolve arquivos compostos por um número de registros que é maior do que a memória interna do computador pode armazenar. Os métodos de ordenação externa são muito diferentes dos métodos de ordenação interna. Em ambos os casos o problema é o mesmo: rearranjar os registros de um arquivo em ordem ascendente ou descendente. Entre-tanto, na ordenação externa as estruturas de dados têm que levar em conta o fato de que os dados estão armazenados em unidades de memória externa, relativamente muito mais lentas do que a memória principal.

Nas memórias externas, tais como **fitas**, discos e tambores magnéticos, os dados são armazenados como um arquivo seqüencial, onde apenas um registro pode ser acessado em um dado momento. Esta é uma restrição forte se comparada com as possibilidades de acesso da estrutura de dados do tipo vetor. Conseqüentemente, os métodos de ordenação interna apresentados na Seção 3.1 são inadequados para ordenação externa, e então ténicas de ordenação completamente diferentes têm que ser usadas. Existem três importantes fatores que fazem os algoritmos para ordenação externa diferentes dos algoritmos para ordenação interna, a saber:

1. O custo para acessar um item é algumas ordens de grandeza maior do que os custos de processamento na memória interna. 0 custo principal na ordenação externa está relacionado com o custo de transferir dados entre a memória interna e a memória externa.

2. Existem restrições severas de acesso aos dados. Por exemplo, os itens armazenados em fita magnética só podem ser acessados de forma seq uencial. Os itens armazenados em disco magnético podem ser acessados diretamente, mas a um custo maior do que o custo para acessar seqüencialmente, o que contra-indica o uso do acesso direto.

3. O desenvolvimento de métodos de ordenação externa é muito dependente do estado atual da tecnologia. A grande variedade de tipos de unidades de memória externa pode tornar os métodos de ordenação externa dependentes de vários parâmetros que afetam seus desempenhos.

Por esta razão, apenas métodos gerais serão apresentados nesta seção, ao invés de apresentarmos refinamentos de algoritmos até o nível de um programa Pascal executável.

Para desenvolver um método de ordenação externa eficiente o aspecto sistema de computação deve ser considerado no mesmo nível do aspecto algorítmico. A grande ênfase deve ser na minimização do número **de vezes que** cada item é transferido entre a memória interna e a memória externa. Mais ainda, cada transferência deve ser realizada de forma tão eficiente quanto as características dos equipamentos disponíveis permitam.

O método de ordenação externa mais importante é o método de ordenação por intercalação. Intercalar significa combinar dois ou mais **blocos ordenados** em um único bloco ordenado através de seleções repetidas entre os itens disponíveis em cada momento. A intercalação é utilizada como uma operação auxiliar no processo de ordenar.

A maioria dos métodos de ordenação externa utilizam a seguinte estratégia geral:

1. É realizada uma primeira passada sobre o arquivo, quebrando-o em blocos do tamanho da memória interna disponível. Cada bloco é então ordenado na memória interna.

2. Os blocos ordenados são intercalados, fazendo várias passadas sobre o arquivo. A cada passada são criados blocos ordenados cada vez maiores, até que todo o arquivo esteja ordenado.

Os algoritmos para ordenação externa devem procurar reduzir o número de passadas sobre o arquivo. Como a maior parte do custo é para as operações de entrada e saída de dados da memória interna, uma boa medida de complexidade de um algoritmo de ordenação por intercalação é o número de vezes que um item é lido ou escrito na memória auxiliar. Os bons métodos de ordenação geralmente envolvem no total menos do que 10 passadas sobre o arquivo.

3.2.1 Intercalação Balanceada de Vários Caminhos

Vamos considerar o processo de ordenação externa quando o arquivo está armazenado em fita magnética. Para apresentar os vários passos envolvidos em um algoritmo de ordenação por intercalação balanceada vamos utilizar um arquivo exemplo. Considere um arquivo armazenado em uma fita de entrada, composto pelos registros com as chaves mostradas na Figura 3.10. Os *22* registros devem ser ordenados de acordo com as chaves e colocados em uma fita de saída. Neste caso os registros são lidos um após o outro.

I N T E R C A L A C A O B A L A N C E A D A

Figura 3.10: Arquivo exemplo com 22 registros

Assuma que a memória interna do computador a ser utilizado só tem espaço para três registros, e o número de unidades de fita **magnética** é seis.

Na primeira etapa o arquivo é lido de três em três registros. Cada **bloco** de três registros é ordenado e escrito em uma das fitas de saída. No exemplo da Figura 3.10 são lidos os registros INT e escrito o bloco INT na fita 1, a seguir são lidos os registros ERC e escrito o bloco CER na fita 2, e assim por diante, conforme ilustra a Figura 3.11. Três fitas são utilizadas em uma intercalação-de-3-caminhos.

fita 1: *I N T A C O A D E*
fita 2: *C E R A B L A*
fita 3: *A A L A C N*

Figura 3.11: Formação dos blocos ordenados iniciais

Na segunda etapa os blocos ordenados devem ser intercalados. 0 primeiro registro de cada uma das três fitas é lido para a memória interna, ocupando toda a memória interna. A seguir o registro contendo a menor chave dentre as três é retirado e o próximo registro da mesma fita é lido para a memória interna, repetindo-se o processo. Quando o terceiro registro de um dos blocos é lido aquela fita fica inativa até que o terceiro registro das outras fitas também sejam lidos e escritos na fita de saída, formando um bloco de 9 registros ordenados. A seguir o segundo bloco de 3 registros de cada fita é lido para formar outro bloco ordenado de nove registros, o qual é escrito em uma outra fita. Ao final três novos blocos ordenados são obtidos, conforme mostra a Figura 3.12.

fita 4: *A A C E I L N R T*
fita 5: *A A A B C C L N O*
fita 6: *A A D E*

Figura 3.12: Intercalação-de-3-caminhos

A seguir mais uma intercalação-de-3-caminhos das fitas 4, 5 e 6 para as fitas 1, 2 e 3 completa a ordenação. Se o arquivo exemplo tivesse um número maior de registros, então vários blocos ordenados de 9 registros seriam formados nas fitas 4, 5 e 6. Neste caso, a segunda passada produziria blocos ordenados de 27 registros nas fitas 1, 2 e 3; a terceira passada produziria blocos ordenados de 81 registros nas fitas 4, 5 e 6, e assim sucessivamente, até obter-se um único bloco ordenado. Neste ponto cabe a seguinte pergunta:

quantas passadas são necessárias para ordenar um arquivo de tamanho arbitrário?

Considere um arquivo contendo *n* registros (neste caso cada registro contém apenas uma palavra) e uma memória interna de *m* palavras. A passada inicial sobre o arquivo produz *n/m* blocos ordenados (se cada registro contiver *k* palavras, *k > 1*, então teríamos *n/m/k* blocos ordenados.) Seja P uma função de complexidade tal que *P(n)* é o número de passadas para a fase de intercalação dos blocos ordenados, e seja f o número de fitas utilizadas em cada passada. Para uma intercalação-de- *f* -caminhos o número de passadas é

$$P(n) = \log_f \frac{n}{m}$$

No exemplo acima, n=22, m=3 e f=3. Logo

$$P(n) = \log_3 \frac{22}{3} = 2.$$

Considere um exemplo de um arquivo de tamanho muito grande, tal como 1 bilhão de palavras. Considere uma memória interna disponível de *2* milhões de palavras e 4 unidades de fitas magnéticas. Neste caso *P(n) = 5*, e o número total de passadas, incluindo a primeira passada para obter os n/m blocos ordenados, é 6. Uma estimativa do tempo total gasto para ordenar este arquivo pode ser obtido multiplicando-se por 6 o tempo gasto para transferir o arquivo de uma fita para outra.

Para uma intercalação-de- *f* -caminhos foram utilizadas *2f* fitas nos exemplos acima. Para usar apenas *f + 1* fitas basta encaminhar todos os blocos para uma única fita e, com mais uma passada, redistribuir estes blocos entre as fitas de onde eles foram lidos. No caso do exemplo de *22* registros apenas 4 fitas seriam suficientes: a intercalação dos blocos a partir das fitas 1, *2* e 3 seria toda dirigida para a fita 4; ao final, o segundo e o terceiro blocos ordenados de 9 registros seriam transferidos, de volta para as fitas 1 e *2,* e assim por diante. O custo envolvido é uma passada a mais em cada intercalação.

3.2.2 Implementação Através de Seleção por Substituição

A implementação do método de intercalação balanceada pode ser realizada utilizando-se filas de **prioridades.** Tanto a passada inicial para quebrar o arquivo em blocos ordenados quanto a fase de intercalação podem ser implementadas de forma eficiente e elegante utilizando-se filas de prioridades. A operação básica necessária para formar os blocos ordenados iniciais corresponde a obter o menor dentre os registros presentes na memória interna, o qual deve ser substituído pelo próximo registro da fita de entrada. A operação de substituição do menor item de uma fila de prioridades implementada através de um *heap é* a operação ideal para resolver o problema.

A operação de substituição corresponde a retirar o menor item da fila de prioridades, colocando no seu lugar um novo item, seguido da reconstituição da propriedade do *heap*.

Para cumprir esta primeira passada nós iniciamos o processo fazendo m inserções na fila de prioridades inicialmente vazia. A seguir o menor item da fila de prioridades é substituído pelo próximo item do arquivo de entrada, com o seguinte passo adicional: se o próximo item é menor que o que está saindo (o que significa que este item não pode fazer parte do bloco ordenado corrente), então ele deve ser marcado como membro do próximo bloco e tratado como maior do que todos os itens do bloco corrente. Quando um item marcado vai para o topo da **fila de prioridades,** o bloco corrente é encerrado e um novo bloco ordenado é iniciado. A Figura 3.13 mostra o resultado da primeira passada sobre o arquivo da Figura 3.10. Os asteriscos indicam quais chaves na fila de prioridades pertencem a blocos diferentes.

Entra	1	2	3
E	*I*	*N*	*T*
R	*N*	*E**	*T*
C	*R*	*E**	*T*
A	*T*	*E**	*C**
L	*A**	*E**	*C**
A	*C**	*E**	*L**
C	*E**	*A*	*L**
A	*L**	*A*	*C*
O	*A*	*A*	*C*
B	*A*	*O*	*C*
A	*B*	*O*	*C*
L	*C*	*O*	*A**
A	*L*	*O*	*A**
N	*O*	*A**	*A**
C	*A**	*N**	*A**
E	*A**	*N**	*C**
A	*C**	*N**	*E**
D	*E**	*N**	*A*
A	*N**	*D*	*A*
	A	*D*	*A*
	A	*D*	
	D		

Figura 3.13: Resultado da primeira passada usando seleção por substituição

Cada linha da Figura 3.13 representa o conteúdo de um *heap* de tamanho três. A condição do *heap é* que a primeira chave tem que ser menor do que a segunda e a terceira chaves. Nós iniciamos com as três primeiras chaves do arquivo, as quais já formam um *heap*. *A* seguir, o registro I sai e é substituído

pelo registro E, que é menor do que a chave I. Neste caso, o registro E não pode ser incluído no bloco corrente: ele é marcado e considerado maior do que os outros registros do *heap.* Isto viola a condição do *heap,* e o registro E* é trocado com o registro N para reconstituir o *heap.* A seguir, o registro N sai e é substituído pelo registro R, o que não viola a condição do *heap.* Ao final do processo vários blocos **ordenados** são obtidos. Esta forma de utilizar filas de **prioridades** é chamada seleção por substituição (vide Knuth, *1973,* Seção 5.4.1; Sedgewick, 1988, p.180).

Para uma memória interna capaz de reter apenas *3* registros é possível produzir os blocos ordenados INRT, ACEL, AABCLO, AACEN e AAD, de tamanhos 4, 4, *6, 5* e *3,* respectivamente. Knuth (1973, pp. *254-256)* mostra que, se as chaves são randômicas, os blocos ordenados produzidos são cerca de duas vezes o tamanho dos blocos produzidos por ordenação interna. Assim, a fase de intercalação inicia com blocos ordenados em média duas vezes maiores do que o tamanho da memória interna, o que pode salvar uma passada na fase de intercalação. Se houver alguma ordem nas chaves então os blocos ordenados podem ser ainda maiores. Ainda mais, se nenhuma chave possui mais do que m chaves maiores do que ela, antes dela, então o arquivo é ordenado já na primeira passada. Por exemplo, o conjunto de registros RAPAZ é ordenado pela primeira passada, conforme ilustra a Figura 3.14.

Entra	1	2	3
A	*A*	*R*	*P*
Z	*A*	*R*	*P*
	P	*R*	*Z*
	R	*Z*	
	Z		

Figura 3.14: Conjunto ordenado na primeira passada

A fase de intercalação dos blocos ordenados obtidos na primeira fase também pode ser implementada utilizando-se uma **fila de prioridades. A** operação básica para fazer a intercalação-de- *f* -caminhos é obter o menor item dentre os itens ainda não retirados dos *f* blocos a serem intercalados. Para tal basta montar uma fila de prioridades de tamanho *f* a partir de cada uma das *f* entradas. Repetidamente, substitua o item no topo da fila de prioridades (no caso o menor item) pelo próximo item do mesmo bloco do item que está sendo substituído, e imprima em outra fita o elemento substituído. A Figura *3.15* mostra o resultado da intercalação de INT com CER com AAL, os quais correspondem aos blocos iniciais das fitas 1, 2 e *3* mostrados na Figura 3.11.

Quando *f* não é muito grande não há vantagem em utilizar seleção por substituição para intercalar os blocos, pois é possível obter o menor item

Entra	1	2	3
A	A	C	I
L	A	C	I
E	C	L	I
R	E	L	I
N	I	L	R
	L	N	R
T	N	R	
	R	T	
	T		

Figura 3.15: Intercalação usando seleção por substituição

fazendo $f - 1$ comparações. Quando f é 8 ou mais, é possível ganhar tempo usando um *heap* como mostrado acima. Neste caso cerca de $\log_2 f$ comparações são necessárias para obter o menor item.

3.2.3 Considerações Práticas

Para implementar o método de ordenação externa descrito anteriormente é muito importante implementar de forma eficiente as operações de entrada saída de dados. Estas operações compreendem a transferência dos dados entre a memória interna e as unidades externas onde estão armazenadas os registros a serem ordenados. Deve-se procurar realizar a leitura, a escrita o processamento interno dos dados de forma simultânea. Os computa-dores de maior porte possuem uma ou mais unidades independentes para processamento de entrada e saída que permitem realizar simultaneamente as operações de entrada, saída e processamento interno.

Knuth (1973) discute várias técnicas para obter superposição de entrada saída com processamento interno. Uma técnica comum é a de utilizar $2f$ áreas de entrada e $2f$ áreas de saída. Para cada unidade de entrada ou saída são mantidas duas áreas de armazenamento: uma para uso do processador central e outra para uso do processador de entrada ou saída. Para entrada, o processador central usa uma das duas áreas enquanto a unidade de entrada está preenchendo a outra área. No momento que o processador central termina a leitura de uma área, ele espera que a unidade de entrada acabe de preencher a outra área e então passa a ler dela, enquanto a unidade de entrada passa a preencher a outra. Para saída, a mesma técnica é utilizada.

Existem dois problemas relacionados com a técnica de utilização de duas áreas de. armazenamento. Primeiro, apenas metade da memória disponível é utilizada, o que pode levar a uma ineficiência se o número de áreas for grande, como no caso de uma intercalação-de-f-caminhos para f grande. Segundo,

em uma intercalação-de- f -caminhos existem f áreas correntes de entrada; se todas as áreas se tornarem vazias aproximadamente ao mesmo tempo, muita leitura será necessária antes de podermos continuar o processamento, a não ser que haja uma previsão de que esta eventualidade possa ocorrer.

Os dois problemas podem ser resolvidos com a utilização de uma técnica chamada **previsão,** a qual requer a utilização de apenas uma área extra de armazenamento (e não f áreas) durante o processo de intercalação. A melhor forma de superpor a entrada com processamento interno durante o processo de seleção por substituição é superpor a entrada da próxima área que precisa ser preenchida a seguir com a parte de processamento interno do algoritmo. Felizmente, é fácil saber qual área ficará vazia primeiro, simplesmente olhando para o último registro de cada área. A área cujo último registro é o menor, será a primeira a se esvaziar; assim nós sempre sabemos qual conjunto de registros deve ser o próximo a ser transferido para a área. Por exemplo, na intercalação de INT com CER com AAL nós sabemos que a terceira área será a primeira a se esvaziar.

Uma forma simples de superpor processamento com entrada na intercalação de vários caminhos é manter uma área extra de armazenamento, a qual é preenchida de acordo com a regra descrita acima. Enquanto os blocos INT, CER e AAL da Figura 3.11 estão sendo intercalados o processador de entrada está preenchendo a área extra com o bloco ACN. Quando o processador central encontrar uma área vazia, ele espera até que a área de entrada seja preenchida caso isto ainda não tenha ocorrido, e então aciona o processador de entrada para começar a preencher a área vazia com o próximo bloco, no caso ABL.

Outra consideração prática importante está na escolha do valor de f, que é a ordem da intercalação. No caso de fita magnética a escolha do valor de f deve ser igual ao número de unidades de fita disponíveis menos um. A fase de intercalação usa f fitas de entrada e uma fita de saída. 0 número de fitas de entrada deve ser no mínimo dois pois não faz sentido fazer intercalação com menos de duas fitas de entrada.

No caso de disco magnético o mesmo raciocínio acima é válido. Apesar do disco magnético permitir acesso direto a posições arbitrárias do arquivo, o acesso seqüencial é mais eficiente. Logo, o valor de f deve ser igual ao número de unidades de disco disponíveis menos um, para evitar o maior custo envolvido se dois arquivos diferentes estiverem em um mesmo disco.

Sedegwick (1983) apresenta outra alternativa: considerar f grande o suficiente para completar a ordenação em um número pequeno de passadas. Uma intercalação de duas passadas em geral pode ser realizada com um número razoável para f. A primeira passada no arquivo utilizando seleção por substituição produz cerca de $n / 2 m$ blocos ordenados. Na fase de intercalação cada etapa divide o número de passadas por f. Logo, f deve ser escolhido tal que

$f^2 > \frac{n}{2m}$.

Para *n* igual a 200 milhões e m igual a 1 milhão então $f = 11$ é suficiente para garantir a ordenação em duas passadas. Entretanto, a melhor escolha para *f* entre estas duas alternativas é muito dependente de vários parâmetros relacionados com o sistema de computação disponível.

Notas Bibliográficas

Knuth (1973) é a referência mais completa que existe sobre ordenação em geral. Outros livros interessantes sobre o assunto são Sedgewick (1988), Wirth (1976), Cormem, Leiserson e Rivest (1990), Aho, Hoperoft e Ullman (1983). 0 livro de Gonnet e Baeza-Yates (1991) apresenta uma relação bibliográfica extensa e atualizada sobre o assunto.

Shellsort foi proposto por Shell (1959). Quicksort foi proposto por Hoare (1962). Um estudo analítico detalhado, bem como um estudo exaustivo dos efeitos práticos de muitas modificações e melhorias sugeridas para o Quicksort pode ser encontrado em Sedgewick (1975) e Sedgewick (1978a). Heapsort foi proposto por Williams (1964) e melhorado por Floyd (1964).

Exercícios

1) Dado que existe necessidade de ordenar arquivos de tamanhos diversos, podendo também variar o tamanho dos registros de um arquivo para outro, apresente uma discussão sobre quais algoritmos de ordenação você escolheria diante das diversas situações colocadas acima.
   Que observações adicionais você apresentaria caso haja

   a) restrições de estabilidade ou

   b) de intolerância para o pior caso (isto é, a aplicação exige um algoritmo eficiente mas não permite que o mesmo eventualmente leve muito tempo para executar).

2) Invente um vetor-exemplo de entrada para demonstrar que Ordenação por Seleção é um método instável. Mostre os passos da execução do algoritmo até que a estabilidade é violada. Note que quanto menor for o vetor que você inventar, mais rápido você vai resolver a questão.

3) Considere uma matriz retangular. Ordene em ordem crescente os elementos de cada linha. A seguir ordene em ordem crescente os elementos de cada coluna. Prove que os elementos de cada linha continuam em ordem.

4) Ordenação Pós-Catástrofe (Árabe, 1992)

Imagine que você estava trabalhando na Universidade da California, em Berkeley. Logo depois que você tinha acabado de ordenar um conjunto muito grande de n números inteiros, cada número de grande magnitude (com muitos dígitos), usando o seu método *O(n* log *n)* favorito, aconteceu um terremoto de grandes proporções. Milagrosamente, o computador não é destruído (nem você), mas, por algum motivo, cada um dos 4 bits menos significativos de cada número inteiro é aleatóriamente alterado. Você quer, agora, ordenar os novos números inteiros. Escolha um algoritmo capaz de ordenar os novos números em 0(n). Justifique.

5) Algoritmos de Ordenação: Estudos Comparativos

Considere os seguintes algoritmos de ordenação interna: Inserção, Seleção, Shellsort, Quicksort, Heapsort.

a) Determine experimentalmente o número esperado de (i) comparações e (ii) movimento-de-registros para cada um dos cinco métodos de ordenação indicados acima.

Use um gerador de números aleatórios para gerar as chaves. Utilize arquivos de diferentes tamanhos com chaves geradas randomicamente. Repita cada experimento algumas vezes e obtenha a média para cada medida de complexidade. Utilize o tipo de da-dos array do Pascal. Dê a sua interpretação dos dados obtidos, comparando-os com resultados analíticos.

b) Uma opção alternativa, para o caso do uso de máquinas que permitem medir o tempo de execução de um programa, é considerar os mesmos algoritmos propostos e determinar experimentalmente o tempo de execução de cada um dos cinco métodos de ordenação indicados acima.

Use um gerador de números aleatórios para gerar arquivos de tamanhos 500, 2000 e 10.000 registros. Para cada tamanho de arquivo utilize dois tamanhos de registros, a saber: um registro contendo apenas a chave e outro registro de tamanho 11 vezes o tamanho da chave (isto é, a chave acompanhada de "outros componentes" cujo tamanho seja equivalente a 10 chaves). Repita cada experimento algumas vezes e obtenha a média dos tempos de execução. Utilize o tipo de dados array do Pascal. Dê a sua interpretação dos dados obtidos.

6) Considere a estrutura de dados *heap.* Determine empiricamente o número esperado de trocas para:

a) construir um *heap* através de *n* inserções sucessivas a partir de um *heap* vazio;
b) inserir um novo elemento em um *heap* contendo *n* elementos;
c) extrair o elemento maior de um *heap* contendo *n* + 1 elementos.

Use um gerador de números aleatórios para gerar as chaves. Repita o experimento para diferentes tamanhos de *n*. *A* estrutura de dados utilizada deve usar o mínimo possível de memória para armazenar o *heap*. Utilize o Pascal para implementar o algoritmo. Finalmente, dê a sua interpretação dos resultados obtidos. Como estes resultados se comparam com os resultados analíticos?

7) Quicksort

a) Mostre como o vetor *A* B *A* B *A* B *A* é particionado quando se escolhe o elemento do meio, *A[(esq + dir)* div 2], como pivô.
b) Mostre as etapas de funcionamento do Quicksort para ordenar as chaves Q U I C K S O R T . Considere que o pivô escolhido é o elemento do meio, *A[(esq + dir)* div 2].

8) Quicksort

a) Descreva uma maneira para manter o tamanho da pilha de recursão o menor possível na implementação do Quicksort.
b) Se você não se preocupasse com o uso deste artifício, até que tamanho a pilha poderia crescer, no pior caso? Por quê?

9) O objetivo deste trabalho é fazer um estudo comparativo de diversas implementações do algoritmo Quicksort. Para tanto, você deverá implementar as seguintes versões do algoritmo:

a) Quicksort recursivo;
b) Quicksort recursivo com interrupção da partição para a ordenação de sub-vetores menores que M elementos. Determine empiricamente o melhor valor de M para um arquivo gerado aleatoriamente com 1000 (mil) elementos;
c) Melhore a versão 2 utilizando a técnica de *mediana-de-três* elementos para escolha do pivô.

Gere massas de testes para testar e comparar cada uma das implementações. Use sua criatividade para criar arquivos de teste interessantes. Faça tabelas e/ou gráficos para mostrar e explicar os resultados obtidos. .

O que deve ser apresentado:

a) Listagem dos programas em Pascal. Vão valer pontos clareza, indentação e comentários no programa.
b) Listagem dos testes executados.
c) Descrição sucinta dos arquivos de teste utilizados, relatório comparativo dos testes executados e as decisões tomadas relativas aos casos e detalhes de especificação que porventura estejam omissos no enunciado.
d) Estudo da complexidade do tempo de execução de cada uma das implementações para diversos cenários de teste.

Algumas sugestões:

a) determine o tempo de processamento necessário na fase de classificação utilizando o relógio da máquina;
b) mantenha contadores (que devem ser atualizados pelos procedimentos de ordenação) para armazenar o número de comparações e de trocas de elementos executados pelos algoritmos;
c) execute o programa algumas vezes, com cada algoritmo, com massas de dados diferentes, para obter a média dos tempos de execução e dos números de comparações e trocas;
d) outro experimento interessante é executar o programa uma vez com uma massa de dados que force o pior caso do algoritmo.

10) Considere o seguinte vetor de entrada: H E A P S O R T

a) Utilizando o algoritmo Heapsort rearranje os elementos do vetor para formar a representação de um *heap* utilizando o próprio vetor de entrada. *0 heap* deve conter o máximo do conjunto na raiz.
b) A partir do *heap* criado execute três iterações do anel principal do Heapsort para extrair os três maiores elementos. Mostre os desenhos *heap-vetor.*

11) Suponha que você tenha que ordenar vários arquivos de 100, 5.000 e *20.000* números inteiros. Para todos os três tamanhos de arquivos é necessário realizar a ordenação no menor tempo possível.

a) Que algoritmo de ordenação você usaria para cada tamanho de **arquivo? Justifique.**
b) Se for necessário manter a ordem relativa dos itens com chaves iguais (isto é, manter estabilidade), que algoritmo você usaria para cada tamanho de arquivo? Justifique.

c) Suponha que as ordenações tenham que ser realizadas em um ambiente em que uma configuração dos dados de entrada que leve a um pior caso não possa ser tolerada, mesmo que este pior caso possa acontecer muito raramente (em outras palavras, você não quer que o pior tempo de execução seja muito maior que o caso médio). Ainda assim, continua sendo importante realizar a ordenação no menor tempo possível. Que algoritmo de ordenação você usaria em cada tamanho de arquivo? Justifique.

12) 0 objetivo deste problema é projetar uma estrutura de dados para um conjunto S (Arabe, 1992). S conterá elementos retirados de algum universo U; S pode conter elementos duplicados. A estrutura de dados projetada deve implementar eficientemente as seguintes operações:

a) Insere (b, S): Insere o elemento *b* em S (isto vai adicionar uma nova cópia de *b* em S se já existia alguma).

b) RetiraMin (x, S): retira de S o menor elemento (pode haver mais de um), retornando seu valor em x.

Descreva uma estrutura de dados e como implementar as operações *Insere e RetiraMin* de modo que estas operações sejam executadas, no pior caso, em O(log *n).*

13) A operação de unir dois arquivos ordenados gerando um terceiro arquivo ordenado é denominada intercalação *(merge).* Esta operação consiste em colocar no terceiro arquivo a cada passo sempre o menor elemento entre os menores dos dois arquivos iniciais, desconsiderando este mesmo elemento nos passos posteriores. Este processo deve ser repetido até que todos os elementos dos arquivos de entrada sejam escolhidos.

Esta idéia pode ser utilizada para construir um algoritmo de ordenação. 0 processo é o seguinte: dividir recursivamente o vetor a ser ordenado em dois vetores até obter *n* vetores de 1 único elemento. Aplicar o algoritmo de merge tendo como entrada 2 vetores de um elemento e formando um vetor ordenado de dois elementos. Repetir este processo formando vetores ordenados cada vez maiores até que todo o vetor esteja ordenado. Escrever um algoritmo para implementar este método, conhecido na literatura como **mergesort.**

14) Considere o arquivo de 10 registros: B A L A N C E A D A

a) Vamos considerar o método de ordenação externa usando intercalação-balanceada-de-2-caminhos, utilizando apenas *3* fitas magnéticas e uma memória interna com capacidade para armazenar

três registros. Mostre todas as etapas para ordenar o **arquivo** exemplo acima utilizando intercalação balanceada simples (sem utilizar seleção por substituição).

b) Quantas passadas foram realizadas?

15) **Ordenação Externa**

O objetivo deste trabalho é projetar e implementar um sistema de programas para ordenar arquivos que não cabem na memória primária, o que nos obriga a utilizar um algoritmo de ordenação externa.

Existem muitos métodos para ordenação externa. Entretanto, a gran-de maioria utiliza a seguinte estratégia geral: blocos de entrada tão grandes quanto possível são ordenados internamente e copiados em arquivos intermediários de trabalho. A seguir os arquivos intermediários são intercalados e copiados em outros arquivos de trabalho, até que todos os registros são intercalados em um único bloco final representando o arquivo ordenado.

Um procedimento simples e eficiente para realizar esta tarefa é o de colocar cada bloco ordenado em um arquivo separado até que a entrada é toda lida. A seguir os N primeiros arquivos são intercalados em um novo arquivo e esses *N* arquivos removidos. *N* é uma constante que pode ter valores entre 2 e 10, chamada Ordem de Intercalação. Este processo é repetido até que fique apenas um arquivo, o arquivo final ordenado. A cada passo o procedimento de intercalação nunca tem que lidar com mais do que N arquivos de intercalação mais um único arquivo de saída.

Para tornar mais claro o que cada aluno tem que realizar apresentamos abaixo um primeiro refinamento do procedimento que permite implementar a estratégia descrita acima. Pode-se observar que grande parte do procedimento lida com criação, abertura, fechamento e remoção de arquivos em momentos adequados.

A fase de intercalação utiliza dois índices, *Low e High*, para indicar o intervalo de arquivos ativos. 0 índice *High* é incrementado de 1, OrdemIntercalConst arquivos são intercalados a partir de *Low e ar*mazenados no arquivo *High e*, finalmente, *Low* é incrementado de OrdemIntercalConst. Quando *Low* fica igual ou maior do que *High a* intercalação termina com o último bloco resultante *High* totalmente ordenado.

É importante observar que as interfaces dos vários procedimentos presentes no código abaixo não estão completamente especificadas.

```
procedure OrdeneExterno;
const OrdemIntercalConst = 2;
var NBlocos      : integer;
    ArqEntrada   : ArqEntradaTipo;
    ArqSaída     : ArqEntradaTipo;
    ArrArqEnt    : array [1..OrdemIntercalConst] of ArqEntradaTipo;
    Fim          : boolean;
    Low,
    High,
    Lim          : integer;
begin {OrdeneExterno}
  NBlocos := 0;
  ArqEntrada := Arquivo a ser ordenado;
  repeat {Formação inicial dos NBlocos ordenados}
    NBlocos := NBlocos + 1;
    Fim := EnchePaginas(NBlocos, ArqEntrada);
    OrdeneInterno;
    ArqSaída := AbreArqSaída (NBlocos);
    DescarregaPaginas(ArqSaída);
    Close(ArqSaída);
  until Fim;
  Close(ArqEntrada);
  Low := 1;
  High := NBlocos;
  while Low < High do {Intercalação dos NBlocos ordenados}
    begin
    Lim := Minimo(Low + OrdemIntercalConst -1, High);
    AbreArqEntrada(ArrArqEnt, Low, Lim);
    High := High + 1;
    ArqSaída := AbreArqSaída(High);
    Intercale(ArrArqEnt, Lim-Low+1, ArqSaída);
    Close(ArqSaída);
    for i := 1 to Lim-Low+1 do
      begin
      Close(ArrArqEnt[i]);
      Erase(ArrArqEnt[i]);
      end;
    Low := Low + OrdemIntercalConst;
  end;
  Dar Rename no arquivo High para nome fornecido pelo usuário;
end; {OrdeneExterno}
```

Para mostrar o funcionamento dos módulos do programa OrdeneExterno você deve proceder da seguinte forma:

a) Utilizar um arquivo contendo 22 registros, onde a chave de cada registro é uma letra maiúscula, conforme mostrado a seguir:

INTERCALACAOBALANCEADA

a) Faça a impressão dos blocos ordenados obtidos na primeira fase do programa.

b) Na segunda fase do programa, para cada iteração do anel mostre o conteúdo de: *Low*, *Lim*, *High*, nome dos arquivos de entrada abertos, nome do arquivo de saída aberto, conteúdo do arquivo de saída.

c) Gere um arquivo contendo um grande número de registros de 16 bytes de tamanho (digamos 30.000 registros), cada registro contendo um campo chave constituído por um número inteiro obtido com o auxílio de um gerador de números pseudo-aleatórios. Faça a medida do tempo necessário para ordenar este arquivo.

Capítulo 4

Pesquisa em Memória Primária

Este capítulo é dedicado ao estudo de como recuperar informação a partir de uma massa grande de informação previamente armazenada. A informação é dividida em **registros,** onde cada registro possui uma chave para ser usada na pesquisa. 0 objetivo da pesquisa é encontrar uma ou mais ocorrências de registros com chaves iguais à **chave de pesquisa.** Neste caso ocorreu uma **pesquisa com sucesso;** caso contrário a pesquisa é **sem sucesso. Um** conjunto de registros é chamado de tabela ou arquivo. Geralmente o termo tabela é associado a entidades de vida curta, criadas na memória interna durante a execução de um programa. Já o termo arquivo é geralmente associado a entidades de vida mais longa, armazenadas em memória externa. Entretanto, esta distinção não é precisa: um arquivo de índices pode ser tratado como uma tabela, enquanto uma tabela de valores de funções pode ser tratada mais como um arquivo.

Existe uma variedade enorme de métodos de pesquisa. A **escolha** do método de pesquisa mais adequado a uma determinada aplicação depende principalmente: (i) da quantidade dos dados envolvidos, (ii) do arquivo estar sujeito a inserções e retiradas freqüentes, ou do conteúdo do arquivo ser praticamente estável (neste caso é importante minimizar o tempo de pesquisa, sem preocupação com o tempo necessário para estruturar o arquivo).

E importante considerar os algoritmos de **pesquisa** como **tipos abstratos de dados,** com um conjunto de operações associado a uma estrutura de dados, de tal forma que haja uma independência de implementação para as operações. Algumas das operações mais comuns incluem:

1. Inicializar a estrutura de dados.

2. Pesquisar um ou mais registros com determinada chave.

3. Inserir um novo registro.
4. Retirar um registro específico.
5. Ordenar um arquivo para obter todos os registros em ordem de acordo com a chave.
6. Ajuntar dois arquivos para formar um arquivo maior.

A operação 5 foi objeto de estudo no Capítulo 3. A operação 6 demanda a utilização de técnicas sofisticadas e não será tratada neste texto.

Um nome comumente utilizado para descrever uma estrutura de dados para pesquisa é dicionário. Um **dicionário** é um **tipo abstrato de dados** com as operações Inicializa, Pesquisa, Insere e Retira. Em uma analogia com um dicionário da língua portuguesa, as chaves são as palavras e os registros são as entradas associadas com cada palavra, onde cada entrada contém a pronúncia, definição, sinônimos e outras informações associadas com a palavra.

Para alguns dos métodos de pesquisa a serem estudados a seguir, vamos implementar o método como um dicionário, como é o caso das Arvores de Pesquisa (Seção 4.3) e Hashing (Seção 4.5). Para os métodos Pesquisa Seqüencial e Pesquisa Digital vamos implementar as operações Inicializa, Pesquisa e Insere e para o método Pesquisa Binária vamos implementar apenas a operação Pesquisa.

4.1 Pesquisa Seqüencial

O método de pesquisa mais simples que existe funciona da seguinte forma: a partir do primeiro registro, pesquise seqüencialmente até encontrar a chave procurada; então pare. Apesar de sua simplicidade, a pesquisa seqüencial envolve algumas idéias interessantes, servindo para ilustrar vários aspectos e convenções a serem utilizadas nos outros métodos de pesquisa a serem apresentados.

Uma forma possível de armazenar um conjunto de registros é através do tipo estruturado arranjo, conforme ilustra o Programa 4.1. Cada registro contém um campo chave que identifica o registro. Além da chave, podem existir outros componentes em um registro, os quais não têm influência muito grande nos algoritmos. Por exemplo, os outros componentes de um registro podem ser substituídos por um apontador contendo o endereço de um outro local que os contenha.

Uma possível implementação para as operações Inicializa, Pesquisa e Insere é mostrada no Programa 4.2.

A função Pesquisa retorna o índice do registro que contém a chave x; caso não esteja presente o valor retornado é zero. Observe que esta implementação

```
const Maxn = 1000;
type Registro =record
                 Chave : TipoChave;
                 {outros componentes}
               end;
     Índice = 0..Maxn;
     Tabela = record
                Item : array [Índice] of Registro;
                n    : Índice;
              end;
```

Programa 4.1: Estrutura do tipo dicionário implementado como arranjo

```
procedure Inicializa (var T : Tabela);
begin
  T.n := 0;
end;

function Pesquisa (x : TipoChave; var T : Tabela) : Índice;
var i : integer;
begin
  T.Item[0].Chave := x;
  i := T.n + 1;
  repeat
    i := i - 1;
  until T.Item[i].Chave = x;
  Pesquisa := i;
end;

procedure Insere (Reg : Registro; var T : Tabela);
begin
  if T.n = Maxn
  then writeln (' Erro : tabela cheia')
  else begin
       T.n := T.n + 1;
       T.Item[T.n] := Reg;
       end;
end;
```

Programa 4.2: Implementação das operações usando arranjo

não suporta mais de um registro com uma mesma chave. Para aplicações com esta característica é necessário incluir um argumento a mais na função Pesquisa para conter o índice a partir do qual se quer pesquisar, e alterar a implementação de acordo.

Um registro sentinela contendo a chave de pesquisa é colocado na posição zero do array. Esta técnica garante que a pesquisa sempre termina. Após a chamada da função Pesquisa, se o índice é zero, significa que a pesquisa foi sem sucesso.

Análise

Para uma pesquisa com sucesso, conforme mostrado no final da Seção 1.3, temos

melhor caso : $C(n) = 1$
pior caso : $C(n) = n$
caso médio : $C(n) = (n + 1)/2$

Para uma pesquisa sem sucesso temos

C'(n)=n+1.

Observe que o anel interno da função Pesquisa, no Programa *4.2*, é extremamente simples: o índice i é decrementado e a chave de pesquisa é comparada com a chave que está no registro. Por esta razão, esta técnica usando sentinela é conhecida por pesquisa seqüencial rápida. Este algoritmo é a melhor solução para o problema de pesquisa em tabelas com *25* registros ou menos.

4.2 Pesquisa Binária

A pesquisa em uma tabela pode ser muito mais eficiente se os registros forem mantidos em ordem. Para saber se uma chave está presente na tabela, compare a chave com o registro que está na posição do meio da tabela. Se a chave é menor, então o registro procurado está na primeira metade da tabela; se a chave é maior, então o registro procurado está na segunda metade da tabela. Repita o processo até que a chave seja encontrada, ou fique apenas um registro cuja chave é diferente da **procurada, significando uma pesquisa** sem sucesso. A Figura *4.1* mostra os subconjuntos **pesquisados** para. recuperar o índice da chave G .

O Programa 4.3 mostra a implementação do algoritmo para um conjunto de registros implementado como uma tabela.

	1	2	3	4	5	6	7	8
Chaves iniciais:	*A*	*B*	*C*	*D*	*E*	*F*	*G*	*H*
	A	*B*	*C*	**D**	*E*	*F*	*G*	*H*
					E	**F**	*G*	*H*
							G	*H*

Figura 4.1: Exemplo de pesquisa binária para a chave G

```
function Binária (x : TipoChave; var T : Tabela) : Índice;
var i, Esq, Dir : Índice;
begin
  if T.n = 0
  then Binária := 0
  else begin
       Esq := 1; Dir := T.n;
       repeat
         i := (Esq + Dir) div 2;
         if x > T.Item[i].Chave then Esq:=i+1 else Dir:=i-1;
       until (x = T.Item[i].Chave) or (Esq > Dir);
       if x=T.Item[i].Chave then Binária:=i else Binária:=0;
       end;
end;
```

Programa 4.3: Pesquisa binária

Análise

A cada iteração do algoritmo, o tamanho da tabela é dividido ao meio. Logo, o número de vezes que o tamanho da' tabela é dividido ao meio é cerca de log n. Entretanto, o custo para manter a tabela ordenada é alto: a cada inserção na posição p da tabela implica no deslocamento dos registros a partir da posição p para as posições seguintes. Conseqüentemente, a pesquisa binária não deve ser usada em aplicações muito dinâmicas.

4.3 Árvores de Pesquisa

A árvore de pesquisa é uma estrutura de dados muito eficiente para armazenar informação. Ela é particularmente adequada quando existe necessidade de **considerar** todos ou alguma combinação de requisitos tais como: (i) acesso direto e seqüencial eficientes; (ii) facilidade de inserção e retirada de registros; (iii) boa taxa de utilização de memória; (iv) utilização de memória primária e secundária.

Se alguém considerar separadamente qualquer um dos requisitos acima, é possível encontrar uma estrutura de dados que seja superior à árvore de pesquisa. Por exemplo, tabelas *hashing* possuem tempos médios de pesquisa melhores e tabelas usando posições contíguas de memória possuem melhores taxas de utilização de memória. Entretanto, uma tabela usando *hashing* precisa ser ordenada se existir necessidade de processar os registros seqüencialmente em ordem lexicográfica, e a inserção/retirada de registros em tabelas usando posições contíguas de memória tem custo alto. As árvores de pesquisa representam um compromisso entre estes requisitos conflitantes.

4.3.1 Árvores Binárias de Pesquisa Sem Balanceamento

De acordo com Knuth (1968, p. 315), uma **árvore** binária é formada a partir de um conjunto finito de nodos, consistindo de um nodo chamado raiz mais 0 ou 2 subárvores binárias distintas. Em outras palavras, uma árvore binária é um conjunto de nodos onde cada nodo tem exatamente 0 ou 2 filhos; quando um nodo tem 2 filhos, eles são Chamadas filhos à esquerda e à direita do nodo.

Cada nodo contém apontadores para subárvores esquerda e direita. 0 número de subárvores de um nodo é chamado grau daquele nodo. Um nodo de grau zero é chamado de nodo externo ou folha (de agora em diante não haverá distinção entre estes dois termos). Os outros nodos são chamados nodos internos.

Uma **árvore** binária de **pesquisa** é uma árvore binária em que todo nodo interno contém um registro, e, para cada nodo, a seguinte propriedade é verdadeira: todos os registros com chaves menores estão na subárvore esquerda e todos os registros com chaves maiores estão na subárvore direita.

O **nível** do nodo raiz é 0; se um nodo está no nível i então a raiz de suas subárvores estão no nível i + 1. A altura de um nodo é o comprimento do caminho mais longo deste nodo até um nodo folha. A altura de uma árvore é a altura do nodo raiz. A Figura 4.2 mostra uma árvore binária de pesquisa de altura 4.

A estrutura de dados árvore binária de pesquisa será utilizada para implementar o tipo abstrato de dados Dicionário (lembre-se que o tipo abstrato Dicionário contém as operações Inicializa, Pesquisa, Insere e Retira). A estrutura e a representação do Dicionário é apresentada no Programa 4.4.

Um procedimento Pesquisa para uma árvore binária de pesquisa é bastante simples, conforme ilustra a implementação do Programa 4.5. Para encontrar um registro com uma chave x, primeiro compare-a com a chave que está na raiz. Se é menor, vá para a subárvore esquerda; se x é maior, vá para a subárvore direita. Repita o processo recursivamente, até que a chave procurada seja encontrada ou então um nodo folha é atingido. Se a pesquisa for com sucesso então o conteúdo do registro retorna no próprio registro x.

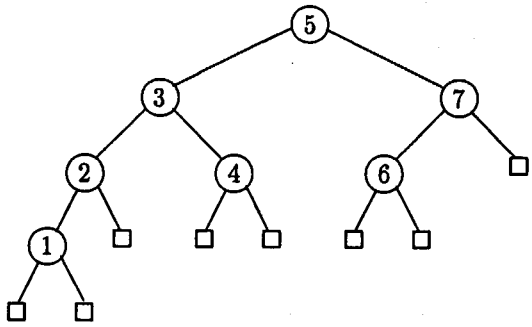


Figura 4.2: Árvore binária de pesquisa

```
type Registro =record
                 Chave : TipoChave;
                 {outros componentes}
               end;
     Apontador = ^Nodo;
     Nodo =record
             Reg      : Registro;
             Esq, Dir : Apontador;
           end;
     TipoDicionário = Apontador;
```

Programa 4.4: Estrutura do dicionário para árvores sem balanceamento

Atingir um apontador nulo em um processo de pesquisa significa uma pesquisa sem sucesso (o registro procurado não está na árvore). Caso se queira inseri-lo na árvore, o apontador nulo atingido é justamente o ponto de inserção, conforme ilustra a implementação do procedimento **Insere** do Programa 4.6.

0 procedimento Inicializa é extremamente simples, conforme ilustra o Programa 4.7. A árvore de pesquisa mostrada na Figura 4.2 pode ser obtida quando as chaves são lidas pelo Programa 4.8, na ordem 5, 3, 2, 7, 6, 4, 1, 0, sendo 0 a marca de fim de arquivo.

A última operação a ser estudada é Retira. Se o nodo que contém o registro a ser retirado possui no máximo um descendente então a operação é simples. No caso do nodo conter dois descendentes, o registro a ser retirado deve ser primeiro substituído pelo registro mais à direita na subárvore esquerda, ou pelo registro mais à esquerda na subárvore direita. Assim, para retirar o registro com chave 5 na árvore da Figura 4.2, basta trocá-lo pelo registro com chave 4 ou pelo registro com chave 6, e então retirar o nodo que recebeu o registro com chave 5.

```
procedure Pesquisa (var x : Registro; var p : Apontador);
begin
  if p = nil
  then writeln (' Erro : Registro não está presente na árvore')
  else if x.Chave < p^.Reg.Chave
       then Pesquisa (x, p^.Esq)
       else if x.Chave > p^.Reg.Chave
            then Pesquisa (x, p^.Dir)
            else x := p^.Reg;
end;
```

Programa 4.5: Procedimento para pesquisar na árvore

```
procedure Insere ( x : Registro; var p : Apontador);
begin
  if p = nil
  then begin
       new (p);
       p^.Reg := x;
       p^.Esq := nil; p^.Dir := nil;
       end
  else if x.Chave < p^.Reg.Chave
       then Insere (x, p^.Esq)
       else if x.Chave > p^.Reg.Chave
            then Insere (x, p^.Dir)
            else writeln (' Erro : Registro já existe na árvore');
end;
```

Programa 4.6: Procedimento para inserir na árvore

```
procedure Inicializa (var Dicionário : TipoDicionário);
begin
  Dicionário := nil;
end;
```

Programa 4.7: Procedimento para inicializar

```
program CriaÁrvore;
type TipoChave = integer;
{— Entra aqui a definição dos tipos do Programa 4.4 —}
var  Dicionário : TipoDicionário;
     x          : Registro;
{— Entram aqui os Programas 4.7 e 4.6 —}
begin
  Inicializa (Dicionário);
  read (x.Chave);
  while x.Chave > 0 do
    begin
    Insere (x, Dicionário);
    read (x.Chave);
    end;
end.
```

Programa 4.8: Programa para criar a árvore

O Programa 4.9 mostra a implementação da operação Retira. 0 procedimento recursivo Antecessor somente é ativado quando o nodo que contém o registro a ser retirado possui dois descendentes. Esta solução elegante é utilizada por Wirth(1976, p.211).

Após construída a árvore pode ser necessário percorrer todos os registros que compõem à tabela ou arquivo. Existe mais de uma ordem de **caminhamento em** árvores, mas a mais útil é a chamada ordem de **caminhamento central.** Assim como a estrutura da árvore, o caminhamento central é melhor expresso em termos recursivos, a saber:

1. caminha na subárvore esquerda na ordem central;
2. visita a raiz;
3. caminha na subárvore direita na ordem central.

Uma característica importante do caminhamento central é que os nodos são visitados em ordem lexicográfica das chaves. Percorrer a árvore da Figura 4.2 usando caminhamento central recupera as chaves na ordem 1, 2, 3, 4, 5, 6 e 7. ' 0 procedimento. Central, mostrado no Programa 4.10, faz exatamente isto. Observe que este procedimento representa um método de ordenação similar ao Quicksort, onde a chave na raiz faz o papel do item que particiona o vetor.

```
procedure Retira (x : Registro; var p : Apontador);
var Aux : Apontador;
  procedure Antecessor (q : Apontador; var r : Apontador);
  begin
    if r^.Dir <> nil
    then Antecessor (q, r^.Dir)
    else begin
         q^.Reg := r^.Reg;
         q := r;
         r := r^.Esq;
         dispose (q);
         end;
  end;

begin {— Retira —}
  if p = nil
  then writeln (' Erro : Registro não está na árvore')
  else if x.Chave < p^.Reg.Chave
       then Retira (x, p^.Esq)
       else if x.Chave > p^.Reg.Chave
            then Retira (x, p^.Dir)
            else if p^.Dir = nil
                 then begin
                      Aux := p;
                      p := p^.Esq;
                      dispose (Aux);
                      end
                 else if p^.Esq = nil
                      then begin
                           Aux := p;
                           p := p^.Dir;
                           dispose (Aux);
                           end
                      else Antecessor (p, p^.Esq);
end;
```

Programa 4.9: Procedimento para retirar x da árvore

```
procedure Central (p : Apontador);
begin
  if p <> nil
  then begin
       Central (p^.Esq);
       writeln (p^.Reg.Chave);
       Central (p^.Dir);
       end;
end;
```

Programa 4.10: Caminhamento central

Análise

O número de comparações em uma pesquisa com sucesso é:

melhor caso : $C(n) = 0(1)$ pior caso : $C(n) = 0(n)$ caso médio : $C(n) = 0(\log n)$

O tempo de execução dos algoritmos para árvores binárias de pesquisa dependem muito do formato das árvores. Para obter o pior caso basta que as chaves sejam inseridas em ordem crescente (ou decrescente). Neste caso a árvore resultante é uma lista linear, cujo número médio de comparações é $(n + 1)/2$.

Para uma **árvore de pesquisa randômica**[1] é possível mostrar que o número esperado de comparações para recuperar um registro qualquer é cerca de 1, 39 log *n*, apenas 39% pior que a árvore completamente balanceada (vide seção seguinte).

4.3.2 Árvores Binárias de Pesquisa Com Balanceamento

Para uma distribuição uniforme das chaves, onde cada chave é igualmente provável de ser usada em uma pesquisa, a **árvore completamente balanceada**[2] minimiza o tempo médio de pesquisa. Entretanto, o custo para manter a árvore completamente balanceada após cada inserção tem um custo muito alto. Por exemplo, para inserir a chave 1 na árvore à esquerda na

[1]Uma árvore A com *n* chaves possui n + 1 nodos externos e estas n chaves dividem todos os valores possíveis em n+1 intervalos. Uma inserção em A é considerada *randômica* se ela tem probabilidade igual de acontecer em qualquer um dos n + 1 intervalos. Uma *árvore de pesquisa randômica* com n chaves é uma árvore construída através de *n* inserções randômicas sucessivas em uma árvore inicialmente vazia

[2]Em uma árvore completamente balanceada os nodos externos aparecem em no máximo dois níveis adjacentes.

Figura 4.3 e obter a árvore à direita na mesma figura é necessário movimentar todos os nodos da árvore original.

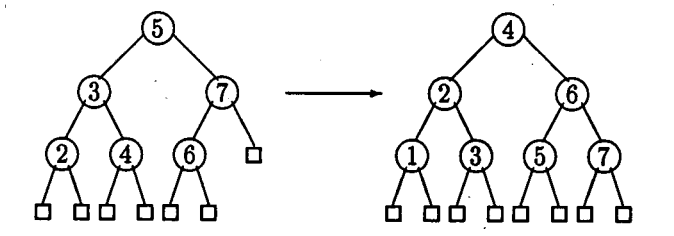


Figura 4.3: Arvore binária de pesquisa completamente balanceada

Uma forma de contornar este problema é procurar uma solução intermediária que possa manter a árvore "quase-balanceada", ao invés de tentar manter a árvore completamente balanceada. 0 objetivo é procurar obter bons tempos de pesquisa, próximos do tempo ótimo da árvore completa-mente balanceada, mas sem pagar muito para inserir ou retirar da árvore.

Existem inúmeras heurísticas baseadas no princípio acima. Gonnet e Baeza-Yates (1991) apresentam algoritmos que utilizam vários critérios de balanceamento para árvores de pesquisa, tais como restrições impostas na diferença das alturas de subárvores de cada nodo da árvore, na redução do **comprimento** do caminho interno[3] da árvore, ou que todos os nodos externos apareçam no mesmo nível. Na seção seguinte vamos apresentar uma árvore binária de pesquisa com balanceamento em que todos os nodos externos aparecem no mesmo nível.

Árvores SBB

As árvores B foram introduzidas por Bayer e McCreight (1972) como uma estrutura para memória secundária, conforme mostrado em detalhes na Seção 5.3.1. Um caso especial da árvore B, mais apropriada para memória primária, é a árvore 2-3, na qual cada nodo tem duas ou três subárvores. Bayer (1971) mostrou que as árvores 2-3 podem ser representadas por árvores binárias, conforme mostrado na Figura 4.4.

Quando a árvore 2-3 é vista como uma árvore B binária, existe uma assimetria inerente no sentido de que os apontadores à esquerda têm que ser verticais (isto é, apontam para um nodo no nível abaixo), enquanto os apontadores à direita podem ser verticais ou horizontais. A eliminação da assimetria nas árvores B binárias leva às árvores B binárias simétricas,

[3]O comprimento do caminho interno corresponde à soma dos comprimentos dos caminhos entre a raiz e cada um dos nodos internos da árvore. Por exemplo, o comprimento do caminho interno da árvore à esquerda na Figura 4.3 é 8 = (0 + 1 + 1 + 2 + 2 + 2).

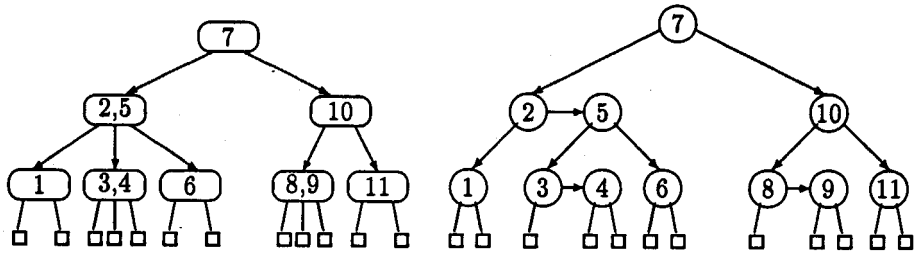


Figura 4.4: Uma árvore 2-3 e a árvore B binária correspondente

cujo nome foi abreviado para árvores SBB *(Symmetric Binary B-trees)* por Bayer (1972). A Figura 4.5 apresenta uma árvore SBB.

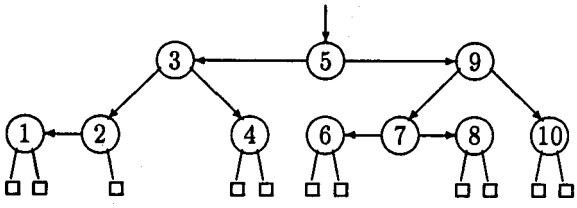


Figura 4.5: Árvore SBB

A **árvore SBB** é uma árvore binária com dois tipos de apontadores, chamados apontadores verticais e apontadores horizontais, tal que:

1. todos os caminhos da raiz até cada nodo externo possuem o mesmo número de apontadores verticais, e
2. não podem existir dois apontadores horizontais sucessivos.

Uma árvore SBB pode também ser vista como uma representação binária da **árvore 2-3-4** apresentada por Guibas e Sedgewick (1978) e mostrada em detalhes em Sedgewick (1988), na qual "supernodos" podem conter até três chaves e quatro filhos. Por exemplo, tal "supernodo", com chaves 3, 5 e 9, pode ser visto na árvore SBB da Figura 4.5.

Transformações para Manutenção da Propriedade SBB

0 algoritmo para árvores SBB usa transformações locais no caminho de inserção (retirada) para preservar o balanceamento. A chave a ser inserida (retirada) é sempre inserida (retirada) após o apontador vertical mais baixo na árvore. Dependendo da situação anterior à inserção (retirada), podem aparecer dois apontadores horizontais sucessivos e, neste caso, é necessário

realizar uma transformação. Se uma transformação é realizada, a altura da subárvore transformada é um mais do que a altura da subárvore original, o que pode provocar outras transformações ao longo do caminho de pesquisa, até a raiz da árvore. A Figura 4.6 mostra as transformações propostas por Bayer (1972), onde transformações simétricas podem ocorrer.

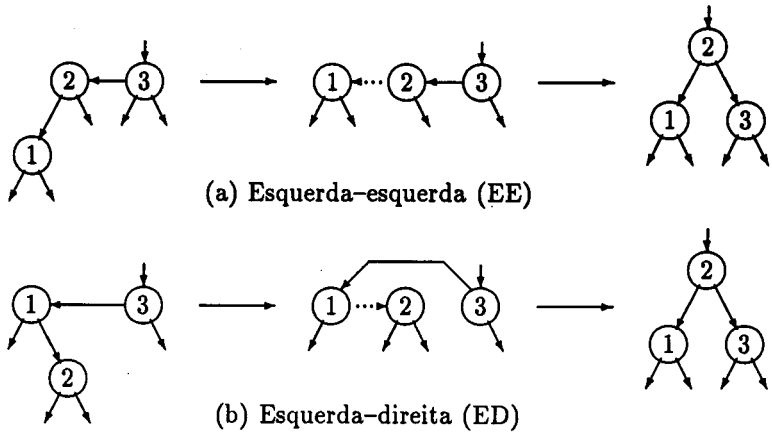


Figura 4.6: Transformações propostas por Bayer (1972)

A estrutura de dados árvore SBB será utilizada para implementar o tipo abstrato de dados Dicionário. A estrutura do Dicionário é apresentada no Programa 4.11. A única diferença da estrutura utilizada para implementar a árvore de pesquisa sem balanceamento (vide Programa 4.4) está nos campos BitE e BitD dentro do registro Nodo, usados para indicar o tipo de apontador (horizontal ou vertical) que sai do nodo.

```
type Registro =record
                  Chave : TipoChave;
                  {outros componentes}
               end;
     Inclinação = (Vertical, Horizontal);
     Apontador = ^Nodo;
     Nodo =record
              Reg         : Registro;
              Esq, Dir    : Apontador;
              BitE, BitD : Inclinação;
           end;
     TipoDicionário = Apontador;
```

Programa 4.11: Estrutura do dicionário para árvores SBB

O procedimento Pesquisa para árvores SBB é idêntico ao procedimento Pesquisa para árvores sem balanceamento mostrado no Programa 4.5, porque o procedimento Pesquisa ignora completamente os campos BitE e BitD. Logo, nenhum tempo adicional é necessário para pesquisar na árvore SBB.

Os quatro procedimentos EE, ED, DD e DE são utilizados nos procedimentos Insere e Retira, com o objetivo de eliminar dois apontadores horizontais sucessivos. 0 Programa 4.12 mostra a implementação destes procedimentos.

```
procedure EE (var Ap: Apontador);
var Ap1 : Apontador;
begin
  Ap1 := Ap^.Esq;  Ap^.Esq := Ap1^.Dir;  Ap1^.Dir := Ap;
  Ap1^.BitE := Vertical;  Ap^.BitE := Vertical;
  Ap := Ap1;
end;

procedure ED (var Ap: Apontador);
var Ap1, Ap2 : Apontador;
  begin
  Ap1 := Ap^.Esq;  Ap2 := Ap1^.Dir;
  Ap1^.BitD := Vertical;  Ap^.BitE := Vertical;
  Ap1^.Dir := Ap2^.Esq;  Ap2^.Esq := Ap1;
  Ap^.Esq := Ap2^.Dir;  Ap2^.Dir := Ap;
  Ap:= Ap2;
end;

procedure DD (var Ap: Apontador);
var Ap1 : Apontador;
  begin
  Ap1 := Ap^.Dir;  Ap^.Dir := Ap1^.Esq;  Ap1^.Esq := Ap;
  Ap1^.BitD := Vertical;  Ap^.BitD := Vertical;
  Ap := Ap1;
end;

procedure DE (var Ap: Apontador);
var Ap1, Ap2 : Apontador;
  begin
  Ap1 := Ap^.Dir;  Ap2 := Ap1^.Esq;
  Ap1^.BitE := Vertical;  Ap^.BitD := Vertical;
  Ap1^.Esq := Ap2^.Dir;  Ap2^.Dir := Ap1;
  Ap^.Dir := Ap2^.Esq;  Ap2^.Esq := Ap;
  Ap:= Ap2;
end;
```

Programa 4.12: Procedimentos auxiliares para árvores SBB

O procedimento **Insere** tem uma interface idêntica à interface do procedimento Insere para árvores sem balanceamento, conforme pode ser visto no Programa 4.13. Para que isso seja possível o procedimento Insere simplesmente chama um outro procedimento interno de nome IInsere, cuja interface contém dois parâmetros a mais que o procedimento Insere, a saber: o parâmetro IAp indica que a inclinação do apontador toma o valor horizontal sempre que um nodo é elevado para o nível seguinte durante uma inserção, e o parâmetro Fim toma o valor `true` quando a propriedade SBB é reestabelecida e nada mais é necessário fazer.

```
procedure Insere ( x : Registro; var Ap : Apontador);
var Fim : boolean; IAp : Inclinacao;

procedure IInsere (x : Registro; var Ap : Apontador;
                   var IAp : Inclinacao; var Fim : boolean);
begin
  if Ap = nil
  then begin
       new(Ap); IAp := Horizontal; Ap^.Reg := x;
       Ap^.BitE := Vertical; Ap^.BitD := Vertical;
       Ap^.Esq := nil; Ap^.Dir := nil;
       Fim := false;
       end
  else
  if x.Chave < Ap^.Reg.Chave
  then begin
       IInsere (x, Ap^.Esq, Ap^.BitE, Fim);
       if not Fim
       then if Ap^.BitE = Horizontal
            then begin
                 if Ap^.Esq^.BitE = Horizontal
                 then begin EE (Ap); IAp := Horizontal; end
                 else if Ap^.Esq^.BitD = Horizontal
                      then begin ED (Ap); IAp := Horizontal; end;
                 end
            else Fim := true;
       end
  else
  if x.Chave > Ap^.Reg.Chave
  then begin
       IInsere (x, Ap^.Dir, Ap^.BitD, Fim);
       if not Fim
       then if Ap^.BitD = Horizontal
            then begin
                 if Ap^.Dir^.BitD = Horizontal
                 then begin DD (Ap); IAp := Horizontal; end
                 else if Ap^.Dir^.BitE = Horizontal
                      then begin DE (Ap); IAp := Horizontal; end;
```

```
                    end
                else Fim := true;
            end
    else begin
            writeln (' Erro: Chave já está na árvore');
            Fim := true
            end;
end; { IInsere }

begin { Insere }
   IInsere (x, Ap, IAp, Fim);
end; { Insere }
```

Programa 4.13: Procedimento para inserir na árvore SBB

A Figura 4.7 mostra o resultado obtido quando se insere uma seqüência de chaves em uma árvore SBB inicialmente vazia: a árvore à esquerda é obtida após a inserção das chaves 7, 10, 5; a árvore do meio é obtida após a inserção das chaves 2, 4 na árvore anterior; a árvore à direita é obtida após a inserção das chaves 9, 3, 6 na árvore anterior. A árvore de pesquisa mostrada na Figura 4.5 pode ser obtida quando as chaves 1, 8 são inseridas na árvore à direita na Figura 4.7.

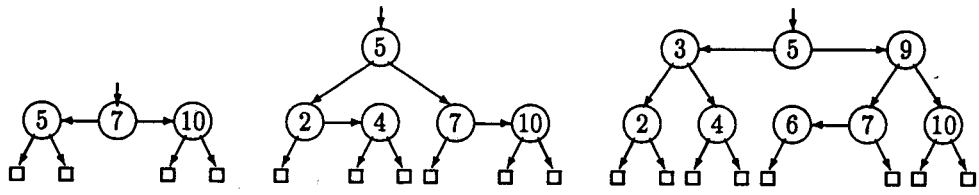


Figura 4.7: Crescimento de uma árvore SBB

O procedimento Inicializa é extremamente simples, conforme ilustra o Programa 4.14.

```
procedure Inicializa (var Dicionário : TipoDicionário);
begin
   Dicionário := nil;
end;
```

Programa 4.14: Procedimento para inicializar a árvore SBB

O procedimento Retira pode ser visto no Programa 4.15. Assim como o procedimento Insere mostrado acima, o procedimento Retira contém um outro procedimento interno de nome IRetira, cuja interface interface contém um parâmetro a mais que o procedimento Retira, a saber: o parâmetro Fim toma o valor true quando a propriedade SBB é reestabelecida e nada mais é necessário fazer.

Por sua vez, o procedimento IRetira utiliza três procedimentos internos, a saber:

- EsqCurto (DirCurto) é chamado quando um nodo folha (que é referendado por um apontador vertical) é retirado da subárvore à **esquerda** (direita), tornando-a menor na altura após a retirada;
- Quando o nodo a ser retirado possui dois descendentes, o procedimento Antecessor localiza o nodo antecessor para ser trocado com o nodo a ser retirado.

```
procedure Retira ( x : Registro; var Ap : Apontador);
var Fim : boolean;

procedure IRetira ( x: Registro; var Ap: Apontador; var Fim: boolean);
var Aux : Apontador;

procedure EsqCurto (var Ap: Apontador; var Fim: boolean);
var Ap1 : Apontador;
begin { Folha esquerda retirada => árvore curta na altura esquerda }
  if Ap^.BitE = Horizontal
  then begin Ap^.BitE := Vertical; Fim := true; end
  else if Ap^.BitD = Horizontal
       then begin
            Ap1 := Ap^.Dir; Ap^.Dir := Ap1^.Esq;
            Ap1^.Esq := Ap; Ap := Ap1;
            if Ap^.Esq^.Dir^.BitE = Horizontal
            then begin DE(Ap^.Esq); Ap^.BitE := Vertical; end
            else if Ap^.Esq^.Dir^.BitD = Horizontal
                 then begin DD(Ap^.Esq); Ap^.BitE := Vertical; end;
            Fim := true;
            end
       else begin
            Ap^.BitD := Horizontal;
            if Ap^.Dir^.BitE = Horizontal
            then begin DE(Ap); Fim := true; end
            else if Ap^.Dir^.BitD = Horizontal
                 then begin DD(Ap); Fim := true; end;
            end;
end; { EsqCurto }
```

```
procedure DirCurto (var Ap: Apontador; var Fim: boolean);
var Ap1 : Apontador;
begin { Folha direita retirada => árvore curta na altura direita }
  if Ap^.BitD = Horizontal
  then begin Ap^.BitD := Vertical; Fim := true; end
  else if Ap^.BitE = Horizontal
       then begin
            Ap1 := Ap^.Esq; Ap^.Esq := Ap1^.Dir;
            Ap1^.Dir := Ap; Ap := Ap1;
            if Ap^.Dir^.Esq^.BitD = Horizontal
            then begin ED(Ap^.Dir); Ap^.BitD := Vertical; end
            else if Ap^.Dir^.Esq^.BitE = Horizontal
                 then begin EE(Ap^.Dir); Ap^.BitD := Vertical; end;
            Fim := true;
            end
       else begin
            Ap^.BitE := Horizontal;
            if Ap^.Esq^.BitD = Horizontal
            then begin ED(Ap); Fim := true; end
            else if Ap^.Esq^.BitE = Horizontal
                 then begin EE(Ap); Fim := true; end;
            end;
end; { DirCurto }

procedure Antecessor(q:Apontador; var r:Apontador; var Fim:boolean);
begin
  if r^.Dir <> nil
  then begin
       Antecessor(q, r^.Dir, Fim);
       if not Fim then DirCurto(r, Fim);
       end
  else begin
       q^.Reg := r^.Reg;
       q := r;   r := r^.Esq;
       dispose (q);
       if r <> nil then Fim := true;
       end;
end; { Antecessor }

begin { IRetira }
  if Ap = nil
  then begin writeln(' Chave não está na árvore'); Fim := true; end
  else if x.Chave < Ap^.Reg.Chave
       then begin
            IRetira(x, Ap^.Esq, Fim);
            if not Fim then EsqCurto(Ap, Fim);
            end
       else if x.Chave > Ap^.Reg.Chave
```

```
            then begin
                 IRetira(x, Ap^.Dir, Fim);
                 if not Fim then DirCurto(Ap, Fim);
                 end
            else begin { Encontrou chave }
                 Fim := false;
                 Aux := Ap;
                 if Aux^.Dir = nil
                 then begin
                      Ap := Aux^.Esq;
                      if Ap <> nil then Fim := true;
                      end
                 else if Aux^.Esq = nil
                      then begin
                           Ap := Aux^.Dir;
                           if Ap <> nil then Fim := true;
                           end
                      else begin
                           Antecessor (Aux, Aux^.Esq, Fim);
                           if not Fim then EsqCurto(Ap, Fim);
                           end;
                 end;
end; { IRetira }

begin { Retira }
  IRetira(x, Ap, Fim);
end; { Retira }
```

Programa 4.15: Procedimento para retirar da árvore SBB

A Figura 4.8 mostra o resultado obtido quando se retira uma seqüência de chaves da árvore SBB: a árvore à esquerda é obtida após a retirada da chave 7 da árvore à direita na Figura 4.7 acima; a árvore do meio é obtida após a retirada da chave 5 da árvore anterior; a árvore à direita é obtida após a retirada da chave 9 da árvore anterior.

Análise

Para as árvores SBB é necessário distinguir dois tipos de **alturas.** Uma delas é a altura vertical h, necessária para manter a altura uniforme e obtida através da contagem do número de apontadores verticais em qualquer caminho entre a raiz e um nodo externo. A outra é a altura k, que representa o número máximo de comparações de chaves obtida através da contagem do número total de apontadores no maior caminho entre a raiz e um nodo ex-

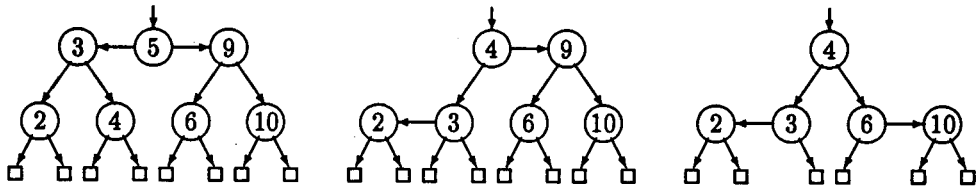


Figura 4.8: Decomposição de uma árvore SBB

terno. A altura k é maior que a altura h sempre que existirem apontadores horizontais na árvore. Para uma árvore SBB com n nodos internos, temos que

$$h \leq k \leq 2h.$$

De fato Bayer (1972) mostrou que

$$\log(n+1) \leq k \leq 2\log(n+2) - 2.$$

O custo para manter a propriedade SBB é exatamente o custo para percorrer o caminho de pesquisa para encontrar a chave, seja para inserí-la ou para retirá-la. Logo, este custo é $O(\log n)$.

O número de comparações em uma pesquisa com sucesso na árvore SBB é

melhor caso : $C(n) = O(1)$
pior caso : $C(n) = O(\log n)$
caso médio : $C(n) = O(\log n)$

Na prática o caso médio para C_n é apenas cerca de 2% pior que o C_n para uma árvore completamente balanceada, conforme mostrado em Ziviani e Tompa (1982).

4.4 Pesquisa Digital

A pesquisa digital é baseada na representação das chaves como uma seqüência de caracteres ou de dígitos. Grosso modo, o método de pesquisa digital é realizado da mesma forma que uma pesquisa em dicionários que possuem aqueles "índices de dedo". Com a primeira letra da palavra são determinadas todas as páginas que contêm as palavras iniciadas por aquela letra.

Os métodos de pesquisa digital são particularmente vantajosos quando as chaves são grandes e de tamanho variável. No problema de casamento

de cadeias, trabalha-se com **chaves semi-infinitas[4],** isto é, sem limitação explicita quanto ao tamanho delas. Um aspecto interessante quanto aos métodos de pesquisa digital é a possibilidade de localizar todas as ocorrências de uma determinada cadeia em um texto, com tempo de resposta logarítmico em relação ao tamanho do texto.

Trie

Uma trie é uma árvore M-ária cujos nodos são vetores de M componentes com campos correspondentes ao dígitos ou caracteres que formam as chaves. Cada nodo no nível i representa o conjunto de todas as chaves que começam com a mesma seqüência de i dígitos ou caracteres. Este nodo especifica uma ramificação com *M* caminhos dependendo do *(i* + 1)-ésimo dígito ou caractere de uma chave. Considerando as chaves como seqüência de bits (isto é, M = 2), o algoritmo de pesquisa digital é semelhante ao de pesquisa em árvore, exceto que, ao invés de se caminhar na árvore de acordo com o resultado de comparação entre chaves, caminha-se de acordo com os bits de chave. A Figura 4.9 mostra uma trie construída a partir das seguintes chaves de 6 bits:

B = 010010
C = 010011
H = 011000
J = 100001
Q = 101000

Para construir .uma trie, faz-se uma pesquisa na árvore com a chave a ser inserida. Se o nodo externo em que a pesquisa teminar for vazio, cria-se um novo nodo externo nesse ponto contendo a nova chave, como ilustra a inserção da chave W = 110110 na Figura 4.10. Se o nodo externo contiver uma chave, cria-se um ou mais nodos internos cujos descendentes conterão a chave já existente e a nova chave. A Figura 4.10 ilustra a inserção da chave K = 100010 que envolve repor J por um novo nodo interno cuja subárvore esquerda é outro novo nodo interno cujos filhos são J e K, porque estas chaves possuem os mesmos bits até a quinta posição.

O formato das tries, diferentemente das árvores binárias comuns, não depende da ordem em que as chaves são inseridas e sim da estrutura das chaves através da distribuição de seus bits. Uma grande desvantagem das

[4]Uma chave semi-infinita é uma seqüência de caracteres em que somente a sua extremidade inicial é definida. Logo, cada posição no texto representa uma chave semi-infinita, constituída pela seqüência que inicia naquela posição e se estende direita tanto quanto for necessário ou até o final do texto. Por exemplo, um banco de dados constituído de n palavras (as posições de interesse nesse caso são os endereços de início das palavras) possui n chaves semi-infinitas.

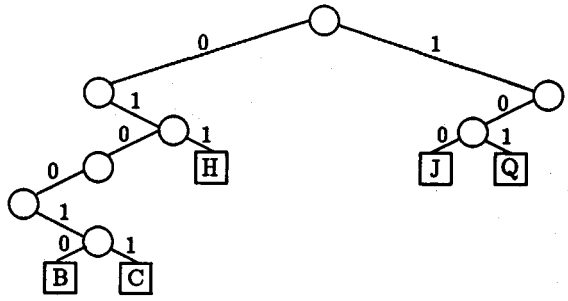


Figura 4.9: Trie binária

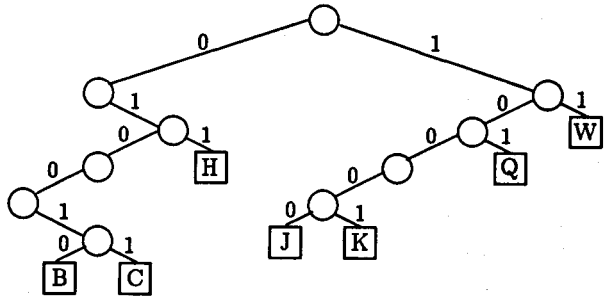


Figura 4.10: Inserção das chaves W e K

tries é a formação de caminhos de uma só direção para chaves com um grande número de bits em comum. Por exemplo, se duas chaves diferirem somente no último bit, elas formarão um caminho cujo comprimento é igual ao tamanho delas, não importando quantas chaves existem na árvore. Veja o caminho gerado pelas chaves B e C na Figura 4.10.

Patricia

PATRICIA é a abreviatura de Practical Algorithm To Retrieve Information Coded In Alphanumeric (Algoritmo Prático para Recuperar Informação Codificada em Alfanumérico). Este algoritmo foi originalmente criado por Morrison (1968) num trabalho de **casamento de cadeias,** aplicado à recuperação de informação em arquivos de grande porte. Knuth (1973) deu um novo tratamento ao algoritmo, reapresentando-o de fôrma mais clara como um caso particular de pesquisa digital, essencialmente, um caso de árvore trio binária. Sedgewick(1988) apresentou novos algoritmos de pesquisa e de inserção baseados nos algoritmos propostos por Knuth. Gonnet e Baeza-Yates (1991) **propuzeram** também outros algoritmos.

O algoritmo para construção da árvore Patricia é baseado no método de pesquisa digital, mas sem apresentar o inconveniente citado para o caso das tries. 0 problema de caminhos de uma só direção é eliminado por meio de uma solução simples e elegante: cada nodo interno da árvore contém o índice do bit a ser testado para decidir qual ramo tomar. A Figura 4.11 apresenta a árvore Patricia gerada a partir das chaves B, C, H, J e Q apresentadas acima.

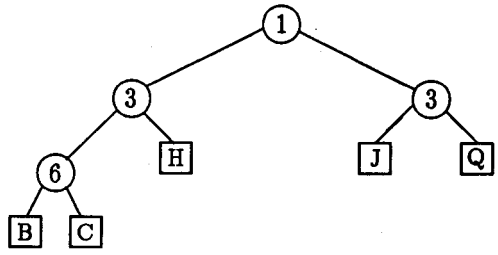


Figura 4.11: Árvore Patricia

Para inserir a chave K = 100010 na árvore da Figura 4.11, a pesquisa inicia pela raiz e termina quando se chega ao nodo externo contendo J. Os índices dos bits nas chaves estão ordenados da esquerda para a direita. Assim, o bit de índice 1 de K é 1, indicando a subárvore direita, e o bit de índice 3 indica a subárvore esquerda que, neste caso, é um nodo externo. Isto significa que as chaves J e K mantêm o padrão de bits 1xOxxx, assim como qualquer outra chave que seguir este caminho de pesquisa. Um novo nodo interno repõe o nodo J, e este juntamente com o nodo K serão os nodos externos descendentes. 0 índice do novo nodo interno é dado pelo primeiro bit diferente das duas chaves em questão, que é o bit de índice 5. Para determinar qual será o descendente esquerdo e o direito, é só verificar o valor do bit 5 de ambas as chaves, conforme mostrado na Figura 4.12.

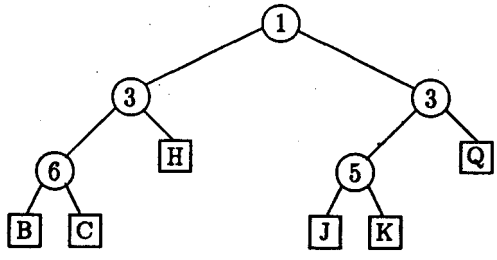


Figura 4.12: Inserção da chave K

A inserção da chave W = 110110 ilustra um outro aspecto. A pesquisa sem sucesso na árvore da Figura 4.13 é realizada de maneira análoga. Os bits das chaves K e W são comparados a partir do primeiro para determinar em qual índice eles diferem, sendo, neste caso, os de índice 2. Portanto o ponto de inserção agora será no caminho de pesquisa entre os nodos internos de índice 1 e 3. Cria-se aí um novo nodo interno de índice 2, cujo descendente direito é um nodo externo contendo W e cujo descendente esquerdo é a subárvore de raiz de índice 3, conforme ilustra a Figura 4.13.

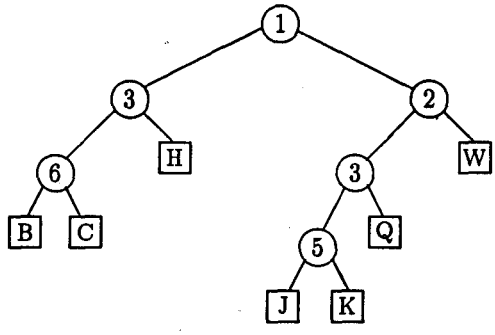


Figura 4.13: Inserção da chave W

A implementação apresentada a seguir é derivada de Albuquerque e Ziviani (1985). O Programa 4.16 apresenta a definição da estrutura de dados utilizada na implementação do algoritmo. Os Programas 4.17 e 4.18 apresentam algumas funções e procedimentos utilizados pelos algoritmos de pesquisa e inserção. O Programa 4.19 apresenta a implementação do algoritmo de pesquisa. O Programa 4.20 apresenta a inicialização da árvore. O Programa 4.21 apresenta a implementação do algoritmo de inserção.

```
const D = "no. de bits de cada chave"; {depende de ChaveTipo}
type ChaveTipo = "a definir, dependendo da aplicação";
     IndexAmp = 0..D;
     NodoTipo = (Interno, Externo);
     Árvore = ^PatNodo;
     PatNodo =record
                case nt : NodoTipo of
                 Interno : (Index : IndexAmp; Esq, Dir : Árvore);
                 Externo : (Chave : ChaveTipo)
                end;
              end;
     Dib = 0..1;
```

Programa 4.16: Estrutura de dados

```
function Bit (i : IndexAmp; k : ChaveTipo) : Dib;
{— Retorna o i-ésimo bit da chave k a partir da esquerda —}
var c, j : integer;
begin
  if i = 0
  then Bit := 0
  else begin
       c := ord(k);
       for j := 1 to D-i do c := c div 2;
       Bit := c mod 2;
       end;
end;

function EData (p : Árvore) : boolean;
{— Verifica se p^ é nodo externo —}
begin
  EData := p^.nt = Externo;
end;
```

Programa 4.17: Funções auxiliares

```
  procedure CrieNodos ( k, ka : ChaveTipo; i : integer;
                        var v : Árvore; var h : boolean);
  {— O nodo interno contém o valor do índice do bit em que
      as chaves k e ka diferem. O nodo externo conterá a
      chave k a ser inserida. O endereço do nodo interno
      retorna em v; h indica se a inserção já foi feita —}
  var p, q  : Árvore;
      b     : boolean;
  begin
    b := true;
    while b and (i ≤ D) do
      if Bit(i,k) = Bit(i,ka) then i := i + 1 else b := false;
    if i > D then h := true {k já se encontra na árvore}
    else begin
         h := false;
         new (p, Interno); new (q, Externo);
         with p^ do
           begin
           nt := Interno; Index := i;
           if Bit(i,k) = 0 then Esq := q else Dir := q;
           end;
         with q^ do
           begin
           nt := Externo; Chave := k;
```

```
            end;
          v := p;
          end;
end;
```

Programa 4.18: Procedimento CrieNodos

```
procedure Pesquise (k : ChaveTipo; t : Árvore);
begin
  if EData(t)
  then if k = t^.Chave
         then Encontrado(t)
         else NãoEncontrado(k)
  else if Bit (t^.Index, k) = 0
         then Pesquise (k, t^.Esq)
         else Pesquise (k, t^.Dir)
end;
```

Programa 4.19: Algoritmo de pesquisa

```
procedure Inicialize (var r : Árvore);
begin
  new(r, Interno);
  with r^ do
    begin Index := 0; Esq := nil; end;
end;
```

Programa 4.20: Inicialização da árvore

Cada chave *k é* inserida de acordo com os passos abaixo, partindo da raiz:

1. Se a subárvore corrente for vazia, então é criado um nodo externo contendo a chave *k* (isto ocorre somente na inserção da primeira chave) e o algoritmo termina.

2. Se a subárvore corrente for simplesmente um nodo externo, os bits da chave k são comparados, a partir do bit de índice imediatamente após o último índice da seqüência de índices consecutivos do caminho de pesquisa, com os bits correspondentes da chave k' deste nodo externo até encontrar um índice i cujos bits difiram. A comparação dos bits

```
procedure Insira ( k : ChaveTipo; var t, u : Árvore;
                   i : integer; var h : boolean);
begin
  if t = nil
  then begin {insere a primeira chave}
       h := true; new (t, Externo);
       with t^ do begin nt := Externo; Chave := k end;
       end
  else if EData(t)
       then CrieNodos (k, t^.Chave, i+1, u, h)
       else begin
            if t^.Index = i + 1 then i := i + 1;
            if Bit (t^.Index, k) = 0
            then begin
                 Insira (k, t^.Esq, u, i, h);
                 if not h
                 then if t^.Index < u^.Index
                      then begin {insere par de nodos}
                           h := true;
                           if Bit (u^.Index, k) = 0
                           then u^.Dir := t^.Esq
                           else u^.Esq := t^.Esq;
                           t^.Esq := u;
                           end;
                 end
                 else begin
                      Insira (k, t^.Dir, u, i, h);
                      if not h
                      then if t^.Index < u^.Index
                           then begin
                                h := true;
                                if Bit (u^.Index, k) = 0
                                then u^.Dir := t^.Dir
                                else u^.Esq := t^.Dir;
                                t^.Dir := u;
                                end;
                      end;
                 end;
            end;
end;
```

Programa 4.21: Algoritmo de inserção

a partir do último índice consecutivo melhora consideravelmente o desempenho do algoritmo. Se todos forem iguais, a chave já se encontra na árvore e o algoritmo termina, senão vai-se para o Passo 4.

3. Caso contrário, ou seja, se a raiz da subárvore corrente for um nodo interno, vai-se para a subárvore indicada pelo bit da chave k de índice dado pelo nodo corrente, de forma recursiva.

4. Depois são criados um nodo interno e um nodo externo: o primeiro contendo o índice i e o segundo, a chave k. A seguir, o nodo interno é ligado ao externo pelo apontador de subárvore esquerda ou direita, dependendo se o bit de índice i da chave k seja 0 ou 1, respectivamente.

5. 0 caminho de inserção é percorrido novamente de baixo para cima, subindo com o par de nodos criados no Passo 4 até chegar a um nodo interno cujo índice seja menor que o índice i determinado no Passo 2. Este é o ponto de inserção e o par de nodos é inserido.

4.5 Transformação de Chave (Hashing)

Os métodos de pesquisa apresentados anteriormente são baseados na comparação da chave de pesquisa com as chaves armazenadas na tabela, ou na utilização dos bits da chave de pesquisa para escolher o caminho a seguir. 0 método de transformação de chave (ou *hashing)* é completamente diferente: os registros armazenados em uma tabela são diretamente endereçados a partir de uma transformação aritmética sobre a chave de pesquisa. De acordo com o Webster's New World Dictionary, a palavra ***hash* significa:** (i) fazer picadinho de carne e vegetais para cozinhar; (ii) fazer uma bagunça. Como veremos a seguir, o termo *hashing é um* nome apropriado pata o método.

Um método de pesquisa através da transformação de chave é constituído de duas etapas principais:

1. Computar o valor da **função de transformação** (também conhecida por **função hashing),** a qual transforma a chave de pesquisa em um endereço da tabela;

2. Considerando que duas ou mais chaves podem ser transformadas em um mesmo endereço de tabela, é necessário existir um método para lidar com colisões.

Se porventura as chaves fossem inteiros de 1 a n, então poderíamos armazenar o registro com chave i na posição i da tabela, e qualquer registro poderia ser imediatamente acessado a partir do valor da chave. Por **outro**

lado, vamos supor uma tabela capaz de armazenar M = 97 chaves, onde cada chave pode ser um número decimal de 4 dígitos. Neste caso existem N = 10000 chaves possíveis, e a **função de transformação não pode** ser um para um: mesmo que o número de registros a serem **armazenados** seja muito menor do que 97, **qualquer** que seja a função de transformação, algumas colisões irão ocorrer fatalmente, e tais colisões têm que ser **resolvidas** de alguma forma.

Mesmo que se obtenha uma 'função de transformação que **distribua** os registros de forma uniforme entre as entradas da tabela, existe uma alta **probabilidade** de haver colisões. 0 **paradoxo do aniversário** (Feller, 1968, p. 33), diz que em um grupo de 23 ou mais pessoas, juntas ao acaso, existe uma chance maior do que 50% de que 2 pessoas comemorem aniversário no mesmo dia. Isto significa que, se for utilizada uma função de **transformação** uniforme que enderece 23 chaves randômicas em uma tabela de tamanho 365, a probabilidade de que haja **colisões** é maior do que 50%.

4.5.1 Funções de Transformação

Uma função de transformação deve mapear chaves em inteiros dentro do intervalo [0..M — *1],* onde M é o tamanho da tabela. A função de transformação ideal é aquela que: (i) seja simples de ser computada; (ii) para cada chave de entrada, qualquer uma das saídas possíveis é igualmente provável de ocorrer.

Considerando que as transformações sobre as chaves são aritméticas, o primeiro passo é transformar as chaves não numéricas em números. No caso do Pascal, basta utilizar a função ord que recebe um argumento de um tipo escalar qualquer e retorna o número ordinal dentro do tipo (por exemplo, ord (**true**) é 1 desde que o tipo boolean é definido como **(false, true**)).

Várias funções de transformação têm sido estudadas (Knott, 1975; Knuth, 1973). Um dos métodos que funciona muito bem usa o resto da divisão por *M* [5]:

$$h(K) = K \bmod M$$

onde *K* é um inteiro correspondente à chave. Este é um método muito simples de ser implementado, conforme ilustra o Programa 4.22. 0 único cuidado a tomar é na escolha do valor de *M*. Por exemplo, se *M* é par, então *h(K)* é par quando *K* é par, e *h(K)* é ímpar quando *K é* ímpar. Resumindo, *M* deve ser um número primo, mas não qualquer primo: 'devem ser evitados os números primos obtidos a partir de

[5] Para números reais x e y a operação binária mod é definida como $x \bmod y = x - y\lfloor x/y \rfloor$, se $y \neq 0$. Quando x e y são inteiros então $5 \bmod 3 = 2$, $6 \bmod 3 = 0$.

```
type TipoChave = packed array [1..n] of char;
     Índice = 0..M-1;
function h(Chave : TipoChave) : Índice;
var i, Soma : integer;
begin
  Soma := 0;
  for i := 1 to n do Soma := Soma + ord (Chave[i]);
  h := Soma mod M;
end;
```

Programa 4.22: Implementação de função de transformação

$$b^i \pm j$$

onde b é a base do conjunto de caracteres (geralmente $b = 64$ para BCD, 128 para ASCII, 256 para EBCDIC, ou 100 para alguns códigos decimais), e i e j são pequenos inteiros (Knuth, 1973, p.509).

4.5.2 Listas Encadeadas

Uma das formas de resolver as **colisões** é simplesmente construir uma lista linear encadeada para cada endereço da tabela. Assim, todas as chaves com mesmo endereço são encadeadas em uma lista linear.

Se a i-ésima letra do alfabeto é representada pelo número i e a função de transformação $h(Chave) = Chave \bmod M$ é utilizada para $M = 7$, então a Figura 4.14 mostra o resultado da inserção das chaves $P\ E\ S\ Q\ U\ I\ S\ A$ na tabela. Por exemplo, $h(A) = h(1) = 1$, $h(E) = h(5) = 5$, $h(S) = h(19) = 5$, e assim por diante.

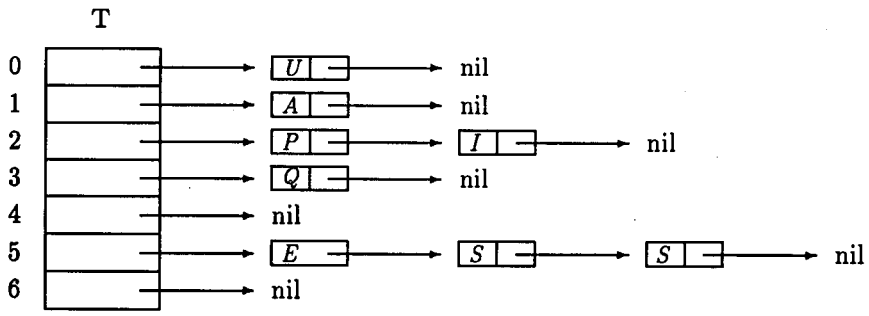


Figura 4.14: Lista encadeada em separado

A estrutura de dados lista encadeada em separado será utilizada para implementar o tipo abstrato de dados Dicionário, com as operações Inicia-

```
type TipoChave = packed array [1..n] of char;
     TipoItem =record
                 Chave : TipoChave;
                 {outros componentes}
               end;
     Apontador = ^Célula;
     Célula =record
               Item : TipoItem;
               Prox : Apontador;
             end;
     TipoLista = record
                   Primeiro : Apontador;
                   Último   : Apontador;
                 end;
     TipoDicionário = array [Índice] of TipoLista;
```

Programa 4.23: Estrutura do dicionário para listas encadeadas

liza, Pesquisa, Insere, Retira. A estrutura do dicionário é apresentada no Programa 4.23.

A implementação das operações sobre o Dicionário são mostradas no Programa 4.24. As operações FLVazia, Insere e Retira, definidas sobre o TipoLista, mostradas no Programa 2.4 do Capítulo 2, podem ser utilizadas para manipular as listas encadeadas. Entretanto, será necessário alterar os nomes do procedimentos Insere e Retira do Programa 2.4 para Ins e Ret respectivamente, para não haver conflito com os nomes dos procedimentos Insere e Retira do Dicionário (vide procedimentos Insere e Retira no Programa 4.24).

Análise

Assumindo que qualquer item do conjunto tem igual probabilidade de ser endereçado para qualquer entrada de T, então o comprimento esperado de cada lista encadeada *é* N/M, onde N representa o número de registros na tabela e M o tamanho da tabela.

Logo, as operações Pesquisa, Insere e Retira custam $0(1 + N/M)$ operações em média, onde a constante 1 representa o tempo para encontrar a entrada na tabela *e* N/M o tempo para percorrer a lista. Para valores de M próximos de N, o tempo se torna constante, isto é, independente de N.

```
procedure Inicializa (var T : TipoDicionário);
var i : integer;
begin
  for i := 0 to M−1 do FLVazia(T[i]);
end;
function Pesquisa (Ch: TipoChave; var T: TipoDicionário): Apontador;
{—Obs.: o Apontador de retorno aponta para o item anterior da lista— }
var i : Índice;
    p : Apontador;
begin
  i := h (Ch);
  if Vazia (T[i])
  then Pesquisa := nil { Pesquisa sem sucesso }
  else begin
       p := T[i].Primeiro;
       while (p^.Prox^.Prox <> nil) and
         (Ch <> p^.Prox^.Item.Chave) do p:= p^.Prox;
       if Ch = p^.Prox^.Item.Chave
       then Pesquisa := p
       else Pesquisa := nil; { Pesquisa sem sucesso }
       end;
end;
procedure Insere (x: TipoItem; var T: TipoDicionário);
begin
  if Pesquisa (x.Chave, T) = nil
  then Ins (x, T[h(x.Chave)])
  else writeln (' Registro já está presente');
end;
procedure Retira (x: TipoItem; var T : TipoDicionário);
var p : Apontador;
begin
  p := Pesquisa (x.Chave, T);
  if p = nil
  then writeln (' Registro não está presente')
  else Ret (p, T[h(x.Chave)], x);
end;
```

Programa 4.24: Operações do Dicionário usando listas encadeadas

4.5.3 Open Addressing

Quando o número de registros a serem armazenados na tabela puder ser previamente estimado, então não haverá necessidade de usar apontadores para armazenar os registros. Existem vários métodos para armazenar N registros em uma tabela de tamanho $M > N$, os quais utilizam os **lugares** vazios na própria tabela para resolver as **colisões.** Tais métodos **são chamados Open-addressing** (Knuth, 1973, p.518).

Em outras palavras, todas as chaves são armazenadas na própria tabela, sem o uso de apontadores explícitos. Quando uma chave x é endereçada para uma entrada da tabela que já esteja ocupada, uma seqüência de localizações alternativas $h_1(x)$, $h_2(x)$, ... é escolhida dentro da tabela. Se nenhuma das $h_1(x)$, $h_2(x)$, ... posições está vazia então a tabela está cheia e não podemos inserir x.

Existem várias propostas para a escolha de localizações alternativas. A mais simples é chamada de hashing linear, onde a posição h_i na tabela é dada por:

$$h_j = (h(x) + j) \bmod M, \quad \text{para } 1 \leq j \leq M - 1$$

Se a i-ésima letra do alfabeto é representada pelo número i e a função de transformação $h(Chave) = Chave \bmod M$ é utilizada para $M = 7$, então a Figura 4.15 mostra o resultado da inserção das chaves $L\ U\ N\ E\ S$ na tabela, usando *hashing linear* para resolver colisões. Por exemplo, $h(L) = h(12) = 5$, $h(U) = h(21) = 0$, $h(N) = h(14) = 0$, $h(E) = h(5) = 5$, e $h(S) = h(19) = 5$.

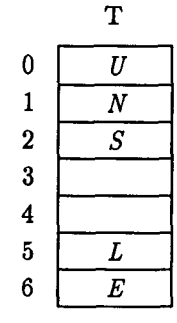


Figura 4.15: Open addressing

A estrutura de dados *open addressing* será utilizada para implementar o tipo abstrato de dados Dicionário, com as operações Inicializa, Pesquisa, Insere, Retira. A estrutura do dicionário é apresentada no Programa 4.25.

A implementação das operações sobre o Dicionário são mostradas no Programa 4.26.

```
const Vazio     = '          ';
      Retirado  = '**********';
type Apontador = integer;
     TipoChave = packed array [1..n] of char;
     TipoItem = record
                  Chave : TipoChave;
                  {outros componentes}
                end;
     TipoDicionário = array [Índice] of TipoItem;
```

Programa 4.25: Estrutura do dicionário usando *open addressing*

Análise

Seja $\alpha = N/M$ o fator de carga da tabela. Conforme demonstrado por Knuth (1973), o custo de uma pesquisa com sucesso é

$$C(n) = \frac{1}{2}\left(1 + \frac{1}{1-\alpha}\right)$$

O hashing linear sofre de um mal chamado **agrupamento (clustering)** (Knuth, 1973, pp.520–521). Este fenômeno ocorre na medida em que a tabela começa a ficar cheia, pois a inserção de uma nova chave tende a ocupar uma posição na tabela que esteja contígua a outras posições já ocupadas, o que deteriora o tempo necessário para novas pesquisas. Entretanto, apesar do *hashing linear* ser um método relativamente pobre para resolver colisões os resultados apresentados são bons. A tabela 4.1 mostra alguns valores para *C(n)* para diferentes valores de a.

α	C(n)
0.10	1.06
0.25	1.17
0.50	1.50
0.75	2.50
0.90	5.50
0.95	10.50

Tabela 4.1: Número de comparações em uma pesquisa com sucesso para *hashing linear*

0 aspecto negativo do método, seja listas encadeadas ou *open addresstng,* está relacionado com o pior caso, que *é O(N).* Se a função de transformação não conseguir espalhar os registros de forma razoável pelas entradas da tabela, então uma longa lista linear pode ser formada, deteriorando o tempo médio de pesquisa. 0 melhor caso, assim como o caso médio, é 0(1).

```
procedure Inicializa (var T : TipoDicionário);
var i : integer;
begin
  for i := 0 to M−1 do T[i].Chave := Vazio;
end;
function Pesquisa (Ch: TipoChave; var T: TipoDicionário): Apontador;
var i       : integer;
    Inicial : integer;
begin
  Inicial := h (Ch);
  i := 0;
  while  (T[(Inicial + i) mod M].Chave <> Vazio) and
         (T[(Inicial + i) mod M].Chave <> Ch) and
         (i < M) do i := i + 1;
  if T[(Inicial + i) mod M].Chave = Ch
  then Pesquisa := (Inicial + i) mod M
  else Pesquisa := M; { Pesquisa sem sucesso }
end;
procedure Insere ( x : TipoItem; var T: TipoDicionário);
var i       : integer;
    Inicial : integer;
begin
  Inicial := h (Ch);
  i := 0;
  while  ((T[(Inicial + i) mod M].Chave <> Vazio) and
          (T[(Inicial + i) mod M].Chave <> Retirado))
         and (i < M) do i := i + 1;
  if i < M
  then T[(Inicial + i) mod M] := x
  else writeln (' Tabela cheia');
end;
procedure Retira (Ch : TipoChave; var T : TipoDicionário);
var i : Índice;
begin
  i := Pesquisa (Ch, T);
  if i < M
  then T[i].Chave := Retirado
  else writeln (' Registro não está presente');
end;
```

Programa 4.26: Operações do dicionário usando *open addressing*

Como vantagens na utilização do método de transformação da chave citamos: (i) alta eficiência no custo de pesquisa, que é 0(1) para o caso médio e, (ii) simplicidade de implementação. Como aspectos negativos citamos: (i) o custo para recuperar os registros na ordem lexicográfica das chaves é alto, sendo necessário ordenar o arquivo e, (ii) o pior caso é O(N).

Notas Bibliográficas

As principais referências para pesquisa em memória interna são Gonnet e Baeza-Yates (1991), Knuth (1973), e Mehlhorn (1984). Outros livros incluem Standish (1980), Wirth (1976), Wirth (1986), Aho, Hoperoft e Ullman (1983), Terada (1991). Um estudo mais avançado sobre estruturas de dados e algoritmos pode ser encontrado em Tarjan (1983).

Um dos primeiros estudos sobre inserção e retirada em árvores de pesquisa foi realizado por Hibbard (1962), tendo provado que o comprimento médio do **caminho interno** após *n* inserções randônicas é 21n *n.* A definição de árvore binária foi extraída de Knuth (1968, p.315).

A primeira árvore binária de pesquisa com balanceamento foi proposta por Adel'son-Vel'skii e Landis (1962), dois matemáticos russos, a qual recebeu o nome de árvore AVL. Uma árvore binária de pesquisa é uma árvore AVL se a altura da subárvore à esquerda de cada nodo nunca difere de ±l da altura da subárvore à direita. A Figura 4.16 apresenta uma árvore com esta propriedade.

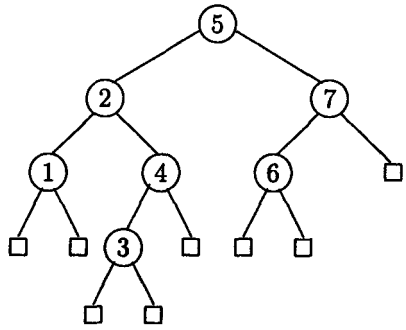


Figura 4.16: Arvore AVL

A forma de manter a propriedade AVL é através de transformações localizadas no caminho de pesquisa. Como a altura das árvores AVL fica sempre entre $\log_2(n + 1)$ e $1.4404 \log_2(n + 2) — 0.328$ (Adel'son-Vel'skii e Landis, 1962), o custo para inserir ou retirar é O(log n), que é exatamente o custo

para percorrer o caminho de pesquisa. Wirth (1976, *1986)* apresenta implementações dos algoritmos de inserção e de retirada para as árvores AVL.

O material utilizado na Seção 4.3.2 veio de Bayer (1971), Bayer (1972), Olivié (1980), Ziviani e Tompa (1982) e Ziviani, Olivié e Gonnet *(1985)*. Os trabalhos de Bayer apresentam as árvores SBB, o de Olivié **sugere uma** melhoria para o algoritmo de inserção e o de Ziviani e Tompa apresentam implementações para os algoritmos de inserção e retirada. A árvore SBB pode ser vista como uma representação binária da **árvore 2-3-4, apresentada** por Guibas e Sedgewick (1978). Este mesmo trabalho mostra como adaptar vários algoritmos clássicos para árvores de pesquisa balanceadas dentro do **esquema árvores red-black.**

Sleator e Tarjan (1983) apresentam vários métodos para manutenção de **árvores auto-ajustáveis.** A idéia é mover os nodos mais freqüentemente acessados em direção à raiz após cada acesso: embora cada operação isolada possa ter custo mais alto, ao longo de um período maior o tempo médio de cada operação é menor, isto é, o custo **amortizado** diminui ao longo do tempo. Em outras palavras, uma operação particular pode ser lenta, mas qualquer seqüência de operações é rápida.

Exercícios

1) Considere as técnicas de pesquisa seqüencial, pesquisa binária e a pesquisa baseada em *hashing.*

   a) Descreva as vantagens e desvantagens de cada uma das técnicas acima, colocando em que situações você usaria cada uma delas.

   b) Dê a ordem do pior caso e do caso esperado de tempo de execução para cada método.

   c) Qual é a eficiência de utilização de memória (relação entre o espaço necessário para dados e o espaço total necessário) para cada método?

2) Suponha uma lista ordenada contendo n itens e um item x que não está presente na lista. O problema consiste em determinar entre qual par de itens na lista está o item x, isto é, encontrar $a[i]$ e $a[i+1]$ de tal forma que $a[i] < x < a[i+1]$, para $1 \leq i < n$, ou que $x < a[1]$ ou que $x > a[n]$.

   a) Encontre o limite inferior para esta classe de problemas quanto ao número de comparações.

   b) Apresente uma prova informal para o limite inferior.

c) Você conhece algum algoritmo que seja ótimo para resolver o problema?

3) Qual é a principal propriedade de uma árvore binária de pesquisa?

4) Árvore Binária de Pesquisa

a) Desenhe a árvore binária de pesquisa que resulta da inserção sucessiva das chaves Q U E S T A O F C I L numa árvore inicialmente vazia.

b) Desenhe as árvores resultantes das retiradas dos elementos E e depois U da árvore obtida no item anterior.

5) Arvores Binárias

Suponha que você tenha uma árvore binária na qual estão armazenadas uma chave em cada nodo. Suponha também que a árvore foi construída de tal maneira que, ao caminhar nela na ordem central, as chaves são visitadas em *ordem crescente.*

a) Qual propriedade entre as chaves deve ser satisfeita para que isso seja possível?

b) Dada uma chave *k,* descreva sucintamente um algoritmo que procure por *k* em uma árvore com essa estrutura.

c) Qual é a complexidade do seu algoritmo no melhor caso e no pior caso? Justifique.

6) Árvore SBB

a) Desenhe a árvore SBB que resulta da inserção sucessiva das chaves Q U E S T A O F C I L numa árvore inicialmente vazia.

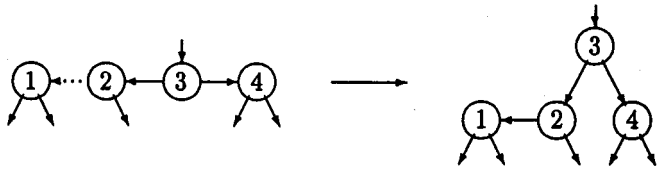
b) Desenhe as árvores resultantes das retiradas dos elementos E e depois U da árvore obtida no item anterior.

7) Árvore SBB

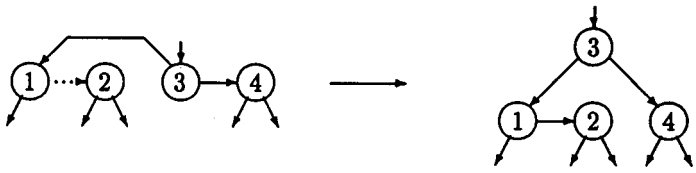
Um novo conjunto de transformações para a árvore SBB foi proposto por Olivié (1980). 0 algoritmo de inserção usando as novas transformações produz árvores SBB com menor altura e demanda um número menor de transformações de divisão de nodos para construir a árvore, conforme comprovado em Ziviani e Tompa (1982) e Ziviani, Olivié e Gonnet (1985). A Figura 4.17 mostra as novas transformações. 0 operação divide esquerda-esquerda requer modificação de três apontadores, a operação divide esquerda-direita requer a alteração de cinco

(a) Rearranjo esquerda–esquerda

(b) Rearranjo esquerda–direita



(c) Divisão esquerda–esquerda



(d) Divisão esquerda–direita

Figura 4.17: Transformações propostas por Olivié (1980)

apontadores, e a operação aumenta altura requer apenas a modificação de dois bits. Transformações simétricas também podem ocorrer.

Quando ocorre uma transformação do tipo aumenta altura, a altura da subárvore transformada é um mais do que a altura da subárvore original, o que pode provocar outras transformações ao longo do caminho de pesquisa até a raiz da árvore. Usualmente, o retorno ao longo do caminho de pesquisa termina quando um apontador vertical é encontrado ou uma transformação do tipo divide é realizada. Como **a** altura da subárvore que sofreu a divisão é a mesma que a altura da subárvore original, apenas uma transformação do tipo divide é suficiente para restaurar a propriedade SBB da árvore.

Bayer (1972), Olivié (1980) e também Wirth (1976) usaram dois bits por nodo em suas implementações para indicarem se os apontadores à direita e à esquerda são horizontais ou verticais. Entretanto, apenas um bit é necessário: a informação indicando se o apontador à direita (esquerda) é horizontal ou vertical pode ser armazenada no filho à direita (esquerda). Além do fato de demandar menos espaço em cada nodo, o retorno ao longo do caminho de pesquisa para procurar por dois apontadores horizontais pode ser terminado mais éedo, porque a informação sobre o tipo de apontador que leva a um nodo é disponível sem a necessidade de retornar até seu pai.

Implemente as novas transformações mostradas na Figura 4.17. Utilize apenas 1 bit por nodo para manter a informação sobre a inclinação dos apontadores.

8) Quais as características de uma boa função *hash?*

9) Um dos métodos utilizados para se organizar dados é através de tabelas *hash*.

   a) Em que situações a tabela *hash* deve ser utilizada?

   b) Descreva dois mecanismos diferentes para resolver o problema de **colisões** de várias chaves em uma mesma posição da tabela. Quais são as vantagens e desvantagens de cada mecanismo?

10) Em uma tabela *hash cam* 100 entradas, as colisões são resolvidas usando listas encadeadas. Para reduzir o tempo de pesquisa, decidiu-se que cada lista seria organizada como uma árvore binária de pesquisa. A função utilizada *é h(k) = k* mod 100. Infelizmente, as chaves inseridas seguem o padrão $k_i = 50i$, onde k_i corresponde à i-ésima chave inserida.

   a) Mostre a situação da tabela após a inserção de k_i, com i = 1,2,••., 13. (Faça desenho.)

   b) Depois que 1000 chaves são inseridas de acordo com o padrão acima, inicia-se a inserção de chaves escolhidas de forma randômica (isto é, não seguem o padrão das chaves já inseridas). Assim responda:

      i) Qual é a ordem do pior caso (isto é, o maior número de comparações) para inserir uma chave?

      ii) Qual é o número esperado de comparações para inserir uma chave? (Assuma que cada uma das 100 entradas da tabela é igualmente provável de ser endereçada pela função h.)

11) *Hashing*

Substitua XXXXXXXXXXXX pelas 12 primeiras letras do seu nome, desprezando brancos e letras repetidas, nas duas partes desta questão. Para quem não tiver doze letras diferentes no nome, completar com as letras PQRSTUVWXYZ, nesta ordem, até completar 12 letras. Por exemplo, eu deveria escolher

N I V O Z A P Q R S T U

A segunda letra I de NIVIO não entra porque ela já apareceu antes, e assim por diante (Arabe, 1992).

a) Desenhe o conteúdo da tabela *hash* resultante da inserção de registros com as chaves XXXXXXXXXXXX, nesta ordem, numa tabela inicialmente vazia de tamanho 7 (sete), usando listas encadeadas. Use a função *hash h(k) = k* mod 7 para a k-ésima letra do alfabeto.

b) Desenhe o conteúdo da tabela *hash* resultante da inserção de registros com as chaves XXXXXXXXXXXX, nesta ordem, numa tabela inicialmente vazia de tamanho 13 (treze), usando *open addressing e hashing linear* para resolver as colisões. Use a função *hash h(k) = k* mod 13 para a k-ésima letra do alfabeto.

12) *Hashing — Open addressing*

a) Hashing Linear. Desenhe o conteúdo da tabela hash resultante da inserção de registros com as chaves Q U E S T A O F C I L, nesta ordem, numa tabela inicialmente vazia de tamanho 13 (treze) usando *open addressing* com *hashing linear* para a escolha de localizações alternativas. Use a função hash *h(k) = **k*** **mod** 13 para a k-ésima letra do alfabeto.

b) Hash Duplo. Desenhe o conteúdo da tabela hash resultante da inserção de registros com as chaves Q U E S T A O F C I L, nesta ordem, numa tabela inicialmente vazia de tamanho 13 (treze) usando *open addressing* com **hash duplo**. Use a função hash $h_1(k) = k$ mod 13 para calcular o endereço primário e j = 1 + (k mod 11) para resolver as colisões, ou seja, para a escolha de localizações alternativas. Logo $h_i(k) = (h_{i-1}(k) + j)$ **mod** 13, para $2 \text{“ } i \bullet M$ (Sedgewick, 1988).

13) Considere as seguintes estruturas de dados:

a) heap

b) árvore binária de pesquisa

c) vetor ordenado

d) tabela *hash* com solução para colisões usando "open addressing"

e) tabela *hash* com solução para colisões usando "listas encadeadas".

Para cada um dos problemas abaixo, sugira a estrutura de dados mais apropriada dentre as listadas acima, de forma a minimizar tempo esperado e espaço necessário. Indique o tempo e o espaço necessário em cada escolha e por que é superior aos outros.

i) inserir/retirar/encontrar um elemento dado;

ii) inserir/retirar/encontrar o elemento de valor mais próximo ao solicitado;

iii) coletar um conjunto de registros, processar o maior elemento, coletar mais registros, processar o maior elemento, e assim por diante.

iv) mesma situação descrita no item anterior adicionada da operação extra de ajuntar ("merge") duas estruturas.

14) 0 objetivo deste trabalho é o de projetar e implementar um sistema de programas, incluindo as estruturas de dados e os algoritmos. Neste trabalho, o aluno terá a oportunidade de exercitar parcialmente o conceito de independência de implementação, através da utilização de duas estruturas de dados distintas para implementar o mesmo problema. Neste caso, o módulo que implementa cada uma das estruturas de dados deverá permitir o intercâmbio entre uma estrutura e outra, causando o menor impacto possível em outras partes do programa.

Problema: Criação de índice remissivo

Várias aplicações necessitam de um relatório de referências cruzadas. Por exemplo, a maioria dos livros apresentam um índice remissivo que corresponde a uma lista alfabética de palavras chave ou palavras relevantes do texto com a indicação dos locais no texto onde cada palavra chave ocorre.

Como exemplo, suponha um arquivo contendo um texto constituído como abaixo:

Linha 1: Good programming is not learned from
Linha 2: generalities, but by seeing how significant
Linha 3: programs can be made clean, easy to *Linha 4:* read, easy to maintain and modify,

Linha 5: human-engineered, efficient, and reliable,
Linha 6: by the application of common sense and
Linha 7: by the use of good programming practices.

Assumindo que o índice remissivo seja constituído das palavras chave:

programming, programs, easy, by, human-engineered, and, be, to, o

programa para criação do índice deve produzir a seguinte saída:

and	4	5	6
be	3		
by	2	6	7
easy	3	4	
human-engineered	5		
programming	1	7	
programs	3		
to	3	4	

Note que a lista de palavras chave está em ordem alfabética. Adjacente a cada palavra está uma lista de números de linhas, um para cada vez que a palavra ocorre no texto.

Projete um sistema para produzir um índice remissivo. O sistema deverá ler um número arbitrário de palavras chave que deverão constituir o índice remissivo, seguido da leitura de um texto de tamanho arbitrário, o qual deverá ser esquadrinhado à procura de palavras que pertençam ao índice remissivo.

Cabe ressaltar que:

a) Uma palavra é considerada como uma seqüência de letras e dígitos, começando com uma letra;
b) Apenas os primeiros c_1 caracteres devem ser retidos nas chaves. Assim, duas palavras que não diferem nos primeiros c_1 caracteres são consideradas idênticas;
c) Palavras constituídas por menos do que c_1 caracteres devem ser preenchidas por um número apropriado de brancos.

Utilize um método eficiente para verificar se uma palavra lida do texto pertence ao índice. Para resolver este problema, você deve utilizar duas estruturas de dados distintas:

a) Implementar o índice como uma árvore de pesquisa;
b) Implementar o índice como uma tabela *hash*, usando o método hashing linear para resolver **colisões.**

Observe que, apesar do *hashing* ser mais eficiente do que árvores de pesquisa, existe uma desvantagem na sua utilização: após atualizado todo o índice remissivo, é necessário imprimir suas palavras em ordem alfabética. Isto é imediato em árvores de pesquisa, mas, quando **se usa** *hashing*, isto é problemático, sendo necessário ordenar a tabela *hash* que contém o índice remissivo.

Utilize o exemplo acima para testar seu programa. Comece a pensar tão logo seja possível, enquanto o problema está fresco na memória e o prazo para terminá-lo está tão longe quanto jamais poderá estar.

15) Considere duas listas ordenadas de números. Determine para cada elemento da lista menor se ele está presente também na lista maior. (Pode assumir que não existem duplicações em nenhuma das duas listas.) Considere os seguintes casos:

- uma lista contém apenas 1 elemento, a outra n
- as duas listas contêm n elementos
- uma lista contém $\sqrt{n}$ elementos, a outra n

a) Sugira algoritmos eficientes para resolver o problema
b) Apresente o número de comparações necessário
c) Mostre que cada algoritmo minimiza o número de comparações.

16) Árvore Patricia Desenhe a árvore Patricia que resulta da inserção sucessiva das chaves

Q U E S T A O F C I L

numa árvore inicialmente vazia.

17) Árvore Patricia

a) Desenhe a árvore Patricia que resulta da inserção sucessiva das chaves

M U L T I C S

numa árvore inicialmente vazia.
b) Qual é o custo para pesquisar em uma árvore Patricia construída através de n inserções randômicas? Explique.

c) Qual é o custo para construir uma árvore Patricia através de *n* inserções randômicas? Explique.

i) Sob o ponto de vista prático, quando *n é* muito grande (digamos 100 milhões), qual é a maior dificuldade para construir a árvore Patricia?

ii) Como a dificuldade apontada no item anterior pode ser superada?

18) Árvore Patricia (Murta, 1992)

Considere o seguinte trecho do poema "Quadrilha" de Carlos Drummond de Andrade:

> "João amava Teresa que amava Raimundo que amava Maria que amava Joaquim que amava Lill que não amava ninguém."

Construa uma árvore Patricia para indexar o texto acima. Considere a seguinte codificação para as palavras do texto:

João	01001011	Maria	01100101
amava	00011101	Joaquim	00101110
Teresa	11101011	Lili	01010011
que	10100101	não	10011100
Raimundo	11011010	ninguém	10110010

a) Faça uma pesquisa pelas chaves "amava", "que amava" e "Lill". Mostre o caminho percorrido para cada pesquisa e as ocorrências do termo pesquisado.

b) Aponte a maior sequência de palavras que se repete no banco de dados e mostre como localizar, em qualquer árvore Patricia, este tipo de ocorrência.

19) Árvore Patricia

Construa, passo a passo, a árvore Patricia para as seis primeiras chaves semi-infinitas do texto abaixo, representado como uma seqüência de bits:

0 1 1 0 0 1 1 0 1 1 0 0 1 •••Texto

1 2 3 4 5 6 7 8 9 ••••••••••Posição

20) Árvore Patricia

Projete e implemente um sistema de programas para recuperação eficiente de informação em bancos de dados constituídos de textos. Tais bancos de dados geralmente recebem adições periódicas, mas nenhuma atualização do que já existe é realizada. Além disso, o tipo de consulta aos dados é totalmente imprevisível. Estes conjuntos de dados aparecem em sistemas legislativos, judiciários, bibliotecas, jornalismo, automação de escritório, dentre outros.

Neste trabalho você deve utilizar um método que cria um índice cuja estrutura é uma árvore Patricia, construída a partir de uma sequência de **chaves** semi-infinitas.

O sistema de programas deverá ser capaz de:

a) construir a árvore Patricia sobre um texto de tamanho arbitrário, representado como um conjunto de palavras;
b) ler um conjunto de palavras de tamanho arbitrário;
c) encontrar todas as ocorrências do conjunto de palavras no texto, imprimindo junto com o conjunto algumas palavras anteriores e posteriores no texto;
d) informar o número de ocorrências do conjunto de palavras no texto;
e) encontrar o maior conjunto de palavras que se repete pelo menos uma vez no texto e informar o seu tamanho;
f) dado um inteiro encontrar, se houver, todas as ocorrências de conjuntos de palavras no texto cujo tamanho seja igual ao inteiro dado.

21) *Pat Array*

Projete e implemente um sistema de programas para recuperação eficiente de informação em bancos de dados constituídos de textos. Tais bancos de dados geralmente recebem adições periódicas, mas nenhuma atualização do que já existe é realizada. Além disso, o tipo de consulta aos dados é totalmente imprevisível. Estes conjuntos de dados aparecem em sistemas legislativos, judiciários, bibliotecas, jornalismo, automação de escritório, dentre outros.

Neste trabalho você deve utilizar uma estrutura de dados chamada **PAT array,** (Gonnet e Baeza-Yates, 1991) construída a partir de uma seqüência de **chaves semi-infinitas.** *0 PAT array é* uma representação compacta da árvore Patricia (Seção 4.4), por armazenar

apenas os nodos externos da árvore. 0 arranjo é constituído de apontadores para o início de cada palavra de um arquivo de texto. Logo, é necessário apenas um apontador para cada ponto de indexação no texto. Este arranjo deverá estar indiretamente ordenado pela ordem lexicográfica das chaves semi-infinitas, conforme mostrado na Figura 4.18.

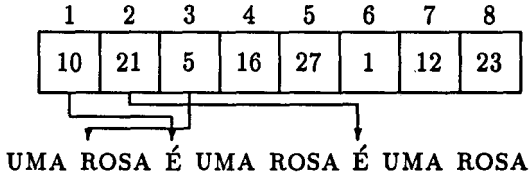


Figura 4.18: *Pat array*

A construção de um *Pat array é* equivalente a ordenação de registros de tamanhos variáveis, representados pelas chaves semi-infinitas. Qualquer operação sobre a árvore Patricia poderá ser simulada sobre o *Pat array* a um custo adicional de O(log *n).* Mais ainda, para a operação de pesquisa de prefixo a árvore Patricia não precisa de fato ser simulada, sendo possível obter algoritmos de custo O(log *n)* ao invés de O($\log^2$ n) para esta operação. Esta operação pode ser implementada através de uma pesquisa binária indireta sobre o arranjo, com o resultado de cada comparação sendo menor que, igual ou menor que. *Pat arrays* são também chamados **Suffix arrays** (Manber e Myers, 1990).

O sistema de programas deverá ser capaz de:

a) Construir o *PAT array* sobre um texto de tamanho arbitrário, representado como um conjunto de palavras;

b) Ler um conjunto de caracteres de tamanho arbitrário. Este conjunto poderá ser uma palavra ou um prefixo de palavra;

c) Informar o número de ocorrências do conjunto de caracteres no texto;

d) Encontrar todas as ocorrências do conjunto de caracteres no texto, imprimindo junto com o conjunto algumas palavras anteriores e posteriores no texto;

e) Apresente a complexidade de pior caso para a letra c);

f) Mostre a relação entre o *PAT array e* a árvore Patricia.

Capítulo 5

Pesquisa em Memória Secundária

A pesquisa em memória secundária envolve arquivos contendo um número de registros que é maior do que o número que a memória interna pode armazenar. Os algoritmos e as estruturas de dados para processamento em memória secundária têm que levar em consideração os seguintes aspectos:

1. O custo para acessar um registro é algumas ordens de grandeza maior do que o custo de processamento na memória primária. Logo, a medida de complexidade principal está relacionada com o custo para transferir dados entre a memória principal e a memória secundária. A ênfase deve ser na minimização do número de vezes que cada registro é transferido entre a memória interna e a memória externa. Por exemplo, o tempo necessário para a localização e a leitura de um número inteiro em disco magnético pode ser suficiente para obter a média aritmética de algumas poucas centenas de números inteiros ou mesmo ordená-los na memória principal.

2. Em memórias secundárias apenas um registro pode ser acessado em um dado momento, ao contrário das memórias primárias que permitem o acesso a qualquer registro de um arquivo a um custo uniforme. Os registros armazenados em fita magnética somente podem ser acessados de forma seqüencial. Os registros armazenados em disco magnético ou disco ótico podem ser acessados diretamente, mas a um custo maior do que o custo para acessá-los seqüencialmente. Os sistemas operacionais levam esse aspecto em consideração e dividem o arquivo em blocos, onde cada bloco é constituído de vários registros. A operação básica sobre arquivos é trazer um bloco da memória secundária para uma **área de armazenamento** na memória principal. Assim, a leitura de

um único registro implica na transferência de todos os registros de um bloco para a memória principal.

A escrita de registros em um arquivo segue caminho contrário. Na medida em que registros são escritos no arquivo eles vão sendo colocados em posições contíguas de memória na área de armazenamento. Quando a área de armazenamento não possui espaço suficiente para armazenar mais um registro o bloco é copiado para a memória secundária, deixando a área de armazenamento vazia e pronta para receber novos registros.

A técnica de utilização de áreas de armazenamento evita que um processo que esteja realizando múltiplas transferências de dados de forma seqüencial tenha que ficar esperando que as transferências se realizem para prosseguir o processamento. As transferências são realizadas em blocos pelo sistema operacional diretamente para uma área de armazenamento. 0 processo usuário pega o dado nesta área e somente é obrigado a esperar quando a área se esvazia. Quando isto ocorre, o sistema operacional enche novamente a área e o processo continua. Esta técnica pode ser aprimorada com o uso de duas ou mais áreas de armazenamento. Neste caso, enquanto um processo está operando em uma área o sistema operacional enche a outra.

3. Para desenvolver um método eficiente de pesquisa o aspecto sistema de computação é da maior importância. As características da arquitetura e do sistema operacional da máquina tornam os métodos de pesquisa dependentes de parâmetros que afetam seus desempenhos. Assim, a transferência de blocos entre as memórias primária e secundária deve ser tão eficiente quanto as características dos equipamentos disponíveis o permitam. Tipicamente, a transferência se torna mais eficiente quando o tamanho dos blocos é de 512 bytes ou múltiplos deste valor, até 4096 bytes.

Na próxima seção apresentamos um modelo de computação para memória secundária que transforma o endereço usado pelo programador no endereço físico alocado para o dado a ser acessado. Este mecanismo é utilizado pela maioria dos sistemas atuais para controlar o trânsito de dados entre o disco e a memória principal. A seguir, apresentamos o método de acesso seqüencial indexado e mostramos sua utilização para manipular grandes arquivos em discos óticos de apenas leitura. Finalmente, apresentamos um método eficiente para manipular grandes arquivos em discos magnéticos que é a árvore n-ária de pesquisa.

5.1 Modelo de Computação para Memória Secundária

Esta seção apresenta um modelo de computação para memória secundária conhecido como memória virtual. Este modelo é normalmente implementado como uma função do sistema operacional. Uma exceção é o sistema operacional DOS para microcomputadores do tipo IBM-PC, que, apesar de muito vendido no mundo inteiro, não oferece um sistema de memória virtual. Por essa razão, vamos apresentar o conceito e mostrar uma das formas possíveis de se implementar um sistema de memória virtual. Além disso, o conhecimento de seu funcionamento facilita a implementação eficiente dos algoritmos para pesquisa em memória secundária também em ambientes que já ofereçam esta facilidade. Maiores detalhes sobre este tópico podem ser obtidos em livros da área de sistemas operacionais, tais como Lister (1975), Peterson e Silberschatz (1983) e Tanenbaum (1987).

Memória Virtual

A necessidade de grandes quantidades de memória e o alto custo da memória principal têm levado ao modelo de sistemas de armazenamento em dois níveis. O compromisso entre velocidade e custo é encontrado através do uso de uma pequena quantidade de memória principal (até 640 kbytes em microcomputadores do tipo IBM-PC usando sistema operacional DOS) e de uma memória secundária muito maior (vários milhões de bytes).

Como apenas a informação que está na memória principal pode ser acessada diretamente, a organização do fluxo de informação entre as memórias primária e secundária é extremamente importante. A organização desse fluxo pode ser realizada utilizando-se um mecanismo simples e elegante para transformar o endereço usado pelo programador na correspondente localização física de memória. 0 ponto crucial é a distinção entre *espaço de endereçamento* — endereços usados pelo programador — e *espaço de memória* — localizações de memória no computador. O espaço de endereçamento N e o espaço de memória M pode ser visto como um mapeamento de endereços do tipo

$$f: N \acute{I} M.$$

O mapeamento de endereços permite ao programador usar um espaço de endereçamento que pode ser maior que o espaço de memória primária disponível. Em outras palavras, o programador enxerga uma memória virtual cujas características diferem das características da memória primária.

Existem várias formas de implementar sistemas de memória virtual. Um dos meios mais utilizados é o sistema de paginação no qual o espaço de endereçamento é dividido em páginas de igual tamanho, em geral múltiplos

de 512 bytes, e a memória principal é dividida de forma semelhante em Molduras_de_Páginas de igual tamanho. As Molduras de_Páginas contêm algumas páginas ativas enquanto o restante das páginas estão residentes em memória secundária (páginas inativas). O mecanismo de paginação possui duas funções, a saber:

a) realizar o mapeamento de endereços, isto é, determinar qual página um programa está endereçando, e encontrar a moldura, se existir, que contenha a página;

b) transferir páginas da memória secundária para a memória primária quando necessário, e transferi-las de volta para a memória secundária quando não estão mais sendo utilizadas.

Para determinar a qual página um programa está se referindo, uma parte dos bits que compõe o endereço é interpretada como um número de página e a outra parte como o número do byte dentro da página. Por exemplo, se o espaço de endereçamento possui 24 bits então a memória virtual é de $_2 24$ bytes; se o tamanho da página é de 512 bytes (2^9) então 9 bits são utilizados para representar o número do byte dentro da página e os restantes 15 bits são utilizados para representar o número da página.

O mapeamento de endereços a partir do espaço de endereçamento (número da página mais número do byte) para o espaço de memória (localização física da memória) é realizado através de uma Tabela_de_Páginas, cuja p-ésima entrada contém a localização p/ da Moldura_de_Página contendo a página número p, desde que esteja na memória principal (a possibilidade de que p não esteja na memória principal será tratada logo à frente). Logo, o mapeamento de endereços é

$$f(e)=f(p,b)=p/+b$$

onde o endereço de programa *e* (número da página *p e* número do byte *b)* pode ser visto na Figura 5.1.

A Tabela_de_Páginas pode ser um arranjo do tamanho do número de páginas possíveis. Quando acontecer do programa endereçar um número de página que não esteja na memória principal, a entrada correspondente na Tabela_de_Páginas estará vazia (p/ = nil) e a página correspondente terá que ser trazida da memória secundária para a memória primária, atualizando a Tabela_de_Páginas.

Se não existir uma Moldura_de_Página vazia no momento de trazer uma nova página do disco então alguma outra página tem que ser removida da memória principal para abrir espaço para a nova página. O ideal é remover a página que não será referenciada pelo período de tempo mais longo no futuro. Entretanto, não há meios de se prever o futuro. O que normalmente

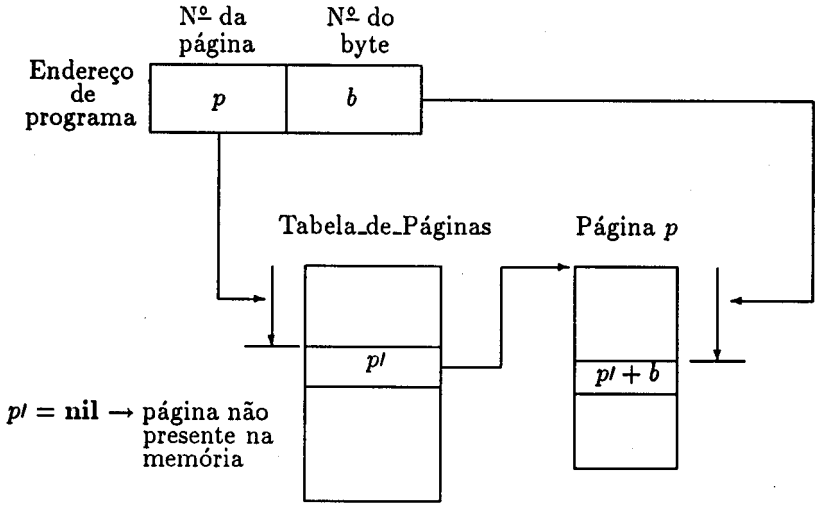


Figura 5.1: Mapeamento de endereços para paginação

é feito é tentar inferir o futuro a partir do comportamento passado. Existem vários algoritmos propostos na literatura para a escolha da página a ser removida. Os mais comuns são:

- Menos Recentemente Utilizada (LRU). Um dos algoritmos mais utilizados é o LRU *(Least Recently Used), o* qual remove a página menos recentemente utilizada, partindo do princípio que o comportamento futuro deve seguir o passado recente. Neste caso, temos que registrar a seqüência de acesso a todas as páginas.

  Uma forma possível de implementar a política LRU para sistemas paginados é através do uso de uma fila de Molduras_de_Páginas, conforme ilustrado na Figura 5.2. Toda vez que uma página é utilizada (para leitura apenas, para leitura e escrita ou para escrita apenas), ela é removida para o fim da fila (o que implica na alteração de cinco apontadores). A página que está na moldura do início da fila é a página LRU. Quando uma nova página tem que ser trazida da memória secundária ela deve ser colocada na moldura que contém a página LRU.

- Menos Freqüentemente Utilizada (LFU). 0 algoritmo LFU *(Least Frequently Used)* remove a página menos freqüentemente utilizada. A justificativa é semelhante ao caso anterior, e o custo é o de registrar o número de acessos a todas as páginas. Um inconveniente é que uma página recentemente trazida da memória secundária tem um baixo número de acessos registrados e, por isso, pode ser removida.

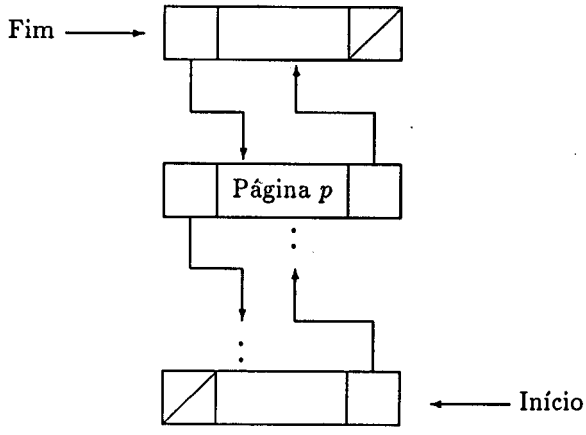


Figura 5.2: Fila de Molduras_de_Páginas

Ordem de Chegada (FIFO). 0 algoritmo FIFO *(First In First Out)* remove a página que está residente há mais tempo. Este algoritmo é o mais simples e o mais barato de se manter. A desvantagem é que ele ignora o fato de que a página mais antiga pode ser a mais referenciada.

Toda informação necessária ao algoritmo escolhido para remoção de páginas pode ser armazenada em cada Moldura_de_Página. Para registrar o fato de que uma página sofreu alteração no seu conteúdo (para sabermos se ela terá que ser reescrita na memória secundária) basta manter um bit na Moldura_de_Página correspondente.

Resumindo, em um sistema de memória virtual o programador pode endereçar grandes quantidades de dados, deixando para o sistema a responsabilidade de transferir o dado endereçado da memória secundária para a memória principal. Esta estratégia funciona muito bem para os algoritmos que possuam uma localidade de referência pequena, isto é, cada referência a uma localidade de memória tem grande chance de ocorrer em uma área que é relativamente perto de outras áreas que foram recentemente referenciadas. Isto faz com que o número de transferências de páginas entre a memória principal e a memória secundária diminua muito. Por exemplo, a maioria de referências a dados no Quicksort ocorre perto de um dos dois apontadores que realizam a partição do conjunto, o que pode fazer com que este algoritmo de ordenação interna funcione muito bem em um ambiente de memória virtual para uma ordenação externa.

```
const TamanhodaPágina = 512;
      ItensPorPágina = 64; { TamanhodaPágina / TamanhodoItem }
type Registro =record
                 Chave : TipoChave;
                 {outros componentes}
               end;
     EndereçoTipo = record
                      p : integer;
                      b : 1..ItensPorPágina;
                    end;
     ItemTipo = record
                  Reg      : Registro;
                  Esq, Dir : EndereçoTipo;
                end;
     PáginaTipo = array[1..ItensPorPágina] of ItemTipo;
```

Programa 5.1: Estrutura de dados para o sistema de paginação

Implementação de um Sistema de Paginação

A seguir vamos mostrar uma das formas possíveis de se implementar um sistema de paginação. A estrutura de dados é apresentada no Programa 5.1. O Programa apresenta também a estrutura de dados para representar uma árvore binária de pesquisa, onde um apontador para um nodo da árvore é representado pelo par número_da_página (p) e posição_dentro_da_página (b). Assumindo que a chave é constituída por um inteiro de 2 bytes e o endereço ocupa 2 bytes para p e 1 byte para b, o total ocupado por cada nodo da árvore é de 8 bytes. Como o tamanho da página é de 512 bytes então o número de itens (nodos) por página é 64.

Em algums casos pode ser necessário manipular mais de um arquivo ao mesmo tempo. Neste caso, uma página pode ser definida como no Programa 5.2, onde o usuário pode declarar até três tipos diferentes de páginas. Se o tipo PáginaTipoA for declarado

```
type PáginaTipoA = array[1..ItensPorPágina] of ItemTipo;
```

e a variável Página for declarada

```
var Página : PáginaTipo;
```

então é possível a seguinte atribuição

```
Página.Pa[1].Reg.Chave := 10;
```

```
type PáginaTipo = record
                    case byte of
                    0 : (Pa : PáginaTipoA);
                    1 : (Pb : PáginaTipoB);
                    2 : (Pc : PáginaTipoC);
                  end;
```

Programa 5.2: Diferentes tipos de páginas para o sistema de paginação

A Tabela_de_Páginas para cada arquivo poderá ser declarada separadamente, mas a Fila_de_Molduras é única, bastando para isso ter em cada moldura a indicação do arquivo a que se refere aquela página.

A comunicação com o sistema de paginação poderá ser realizada através dos seguintes procedimentos:

1. ObtémRegistro: Torna disponível um registro de um arquivo. O parâmetro de entrada é o endereço virtual $< p, b >$ e o parâmetro de saída é o apontador para a Moldura_de_Página ($< p', b >$ na Figura 5.1).

2. EscreveRegistro: Permite criar ou alterar o conteúdo de um registro. Possui dois parâmetros de entrada: o registro e seu endereço virtual $< p, b >$.

3. DescarregaPáginas: Permite varrer a Fila_de_Molduras para atualizar na memória secundária todas as páginas que porventura tenham sofrido qualquer alteração no seu conteúdo (bit de alteração = **true**).

O diagrama da Figura 5.3 mostra a transformação do endereço virtual para o endereço real de memória do sistema de paginação, tornando disponível na memória principal o registro endereçado pelo programador. Os quadrados representam resultados de processos ou arquivos, e os retângulos representam os processos transformadores de informação.

A partir do endereço p o processo P1 verifica se a página que contém o registro solicitado se encontra na memória principal. Caso a página esteja na memória principal o processo P2 simplesmente retorna esta informação para o programa usuário. Se a página está ausente o processo P3 determina uma moldura para receber a página solicitada que deverá ser trazida da memória secundária. Caso não haja nenhuma moldura disponível, alguma página deverá ser removida da memória principal para ceder lugar para a nova página, de acordo com o algoritmo adotado para remoção de páginas. Neste caso estamos assumindo o algoritmo mais simples de ser implementado, o FIFO, onde a página a ser removida é a página que está na cabeça da fila de Molduras_de_Páginas. Se a página a ser substituída sofreu algum tipo

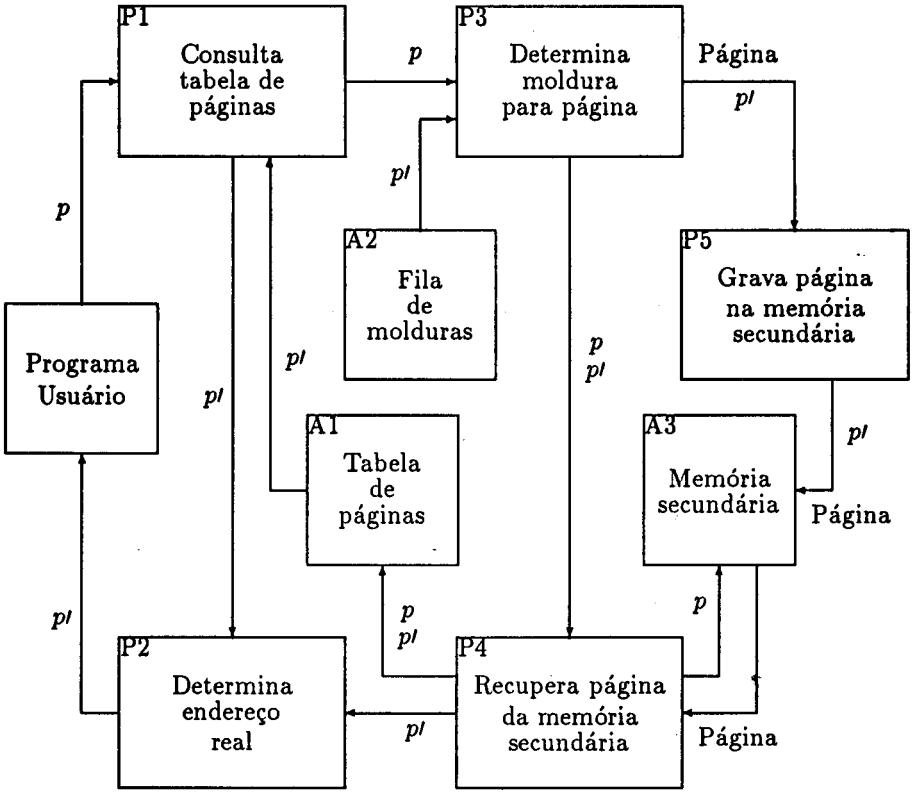


Figura 5.3: Endereçamento no sistema de paginação

de alteração no seu conteúdo ela deverá ser gravada de volta na memória secundária pelo processo P5. 0 processo P4 lê da memória secundária a página solicitada, coloca-a na moldura determinada pelo processo P3 e atualiza a Tabela_de_Páginas.

5.2 Acesso Seqüencial Indexado

O método de acesso seqüencial indexado utiliza o princípio da pesquisa seqüencial: a partir do primeiro, cada registro é lido seqüencialmente até encontrar uma chave maior ou igual a chave de pesquisa. Para aumentar a eficiência, evitando que todos os registros tenham que ser lidos seqüencialmente do disco, duas providências são necessárias: (i) o arquivo deve mantido ordenado pelo campo chave do registro, (ii) um arquivo de *índices* contendo

pares de valores < *x*, *p* > deve ser criado, onde *x* representa uma chave e p representa o endereço da página na qual o primeiro registro contém a chave x.

A Figura 5.4 mostra um exemplo da estrutura de um arquivo seqüencial indexado para um conjunto de 15 registros. No exemplo, cada página tem capacidade para armazenar 4 registros do arquivo de dados e cada entrada do índice de páginas armazena a chave do primeiro registro de cada página e o endereço da página no disco. Por exemplo, o índice relativo à primeira página informa que ela contém registros com chaves entre 3 e 14 exclusive, o índice relativo a segunda página informa que ela contém registros com chaves entre 14 e 25 exclusive, e assim por diante.

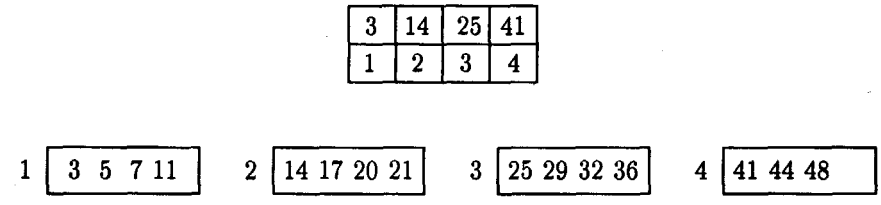


Figura 5.4: Estrutura de um arquivo seqüencial indexado

Em um **disco magnético** várias superfícies de gravação são utilizadas, conforme ilustra a Figura 5.5. 0 disco magnético é dividido em círculos concêntricos chamados **trilhas**. Quando o mecanismo de acesso está posicionado em uma determinada trilha, todas as trilhas que estão verticalmente alinhadas e possuem mesmo diâmetro formam um cilindro. Neste caso, uma referência a um registro que se encontre em uma página de qualquer trilha do cilindro não requer o deslocamento do mecanismo de acesso e o único tempo necessário é o de **latência rotacional**, que é o tempo necessário para que o início do bloco que contenha o registro a ser lido passe pela cabeça de leitura/gravação. A necessidade de deslocamento do mecanismo de acesso de uma trilha para outra é responsável pela parte maior do custo para acessar os dados e é chamado de **tempo de busca (seek time).**

Pelo fato de combinar acesso indexado com a organização seqüencial o método é chamado de acesso seqüencial indexado. Para aproveitar as características do disco magnético e procurar minimizar o número de deslocamentos do mecanismo de acesso utiliza-se um esquema de índices de cilindros e de páginas. Dependendo do tamanho do arquivo e da capacidade da memória principal disponível é possível acessar qualquer registro do arquivo de dados realizando apenas um deslocamento do mecanismo de acesso. Para tal, um índice de cilindros contendo o valor de chave mais alto dentre os registros

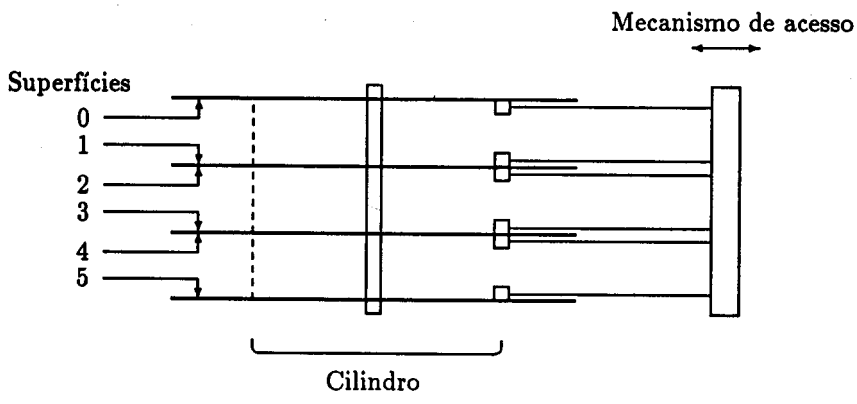


Figura 5.5: Disco magnético

de cada cilindro é mantido na memória principal. Por sua vez, cada cilindro contém um índice de blocos ou índice de páginas, conforme mostrado na Figura 5.4. Para localizar o registro que contenha uma chave de pesquisa são necessários os seguintes passos:

1. Localize o cilindro correspondente à chave de pesquisa no índice de cilindros;
2. Desloque o mecanismo de acesso até o cilindro correspondente;
3. Leia a página que contém o índice de páginas daquele cilindro;
4. Leia a página de dados que contém o registro desejado.

Desta forma, o método de acesso seqüencial indexado possibilita tanto o acesso seqüencial quanto o acesso randômico. Entretanto, este método é adequado apenas para aplicações nas quais as operações de inserção e de retirada ocorrem com baixa freqüência. Sua grande vantagem é a garantia de acesso aos dados com apenas um deslocamento do mecanismo de acesso do disco magnético. Sua grande desvantagem é a inflexibilidade: em um ambiente muito dinâmico, com muitas operações de inserção e retirada, os dados têm que sofrer reorganizações freqüentes. Por exemplo, a adição de um registro com a chave 6 provoca um rearranjo em todos os registros do arquivo da Figura 5.4.

Para contornar este problema é necessário criar **áreas de armazenamento** (ou áreas de *overflow)* para receber estes registros adicionais até que a próxima reorganização de todo o arquivo seja realizada. Normalmente, uma área de armazenamento é reservada em cada cilindro, além de uma

grande área comum para ser utilizada quando alguma área de algum cilindro também transborde. Assim, em ambientes realmente dinâmicos, os tempos de acesso se deterioram rapidamente. Entretanto, em ambientes onde apenas a leitura de dados é necessária, como no caso dos discos óticos de apenas-leitura, o método de acesso indexado seqüencial é bastante eficiente e adequado, conforme veremos na seção seguinte.

Discos áticos de Apenas-Leitura

Os discos óticos de apenas-leitura, conhecidos como CD-ROM *(Compact Disk Read Only Memory),* têm sido largamente utilizados para distribuição de grandes arquivos de dados. 0 interesse crescente sobre os discos CD-ROM é devido a sua capacidade de armazenamento (600 Megabytes) e baixo custo para o usuário final. As principais diferenças entre o disco CD-ROM e o disco magnético são:

1. o CD-ROM é um meio de apenas-leitura e, portanto, a estrutura da informação armazenada é estática;

2. a eficiência na recuperação dos dados é afetada pela localização dos dados no disco e pela seqüência com que são acessados;

3. devido à velocidade linear constante as trilhas possuem capacidade variável e o tempo de latência rotacional varia de trilha para trilha.

Ao contrário dos discos magnéticos, a trilha no disco CD-ROM tem a forma de uma espiral contínua, embora para efeito de estudo analítico cada volta da espiral possa ser considerada como uma trilha. Ele possui cerca de 300.000 setores de tamanho fixo de 2 Kbytes, distribuídos em aproximadamente 20.000 trilhas. Como a velocidade linear de leitura é constante, o tempo de latência rotacional varia de cerca de 60 milissegundos para as trilhas mais internas até 138 milissegundos para as trilhas mais externas. Por outro lado, o número de setores por trilha aumenta de 9 para a trilha mais interna até 20 para a trilha mais externa.

No CD-ROM o tempo de busca (seek time) para acesso a trilhas mais distantes demanda mais tempo que no disco magnético devido à necessidade de deslocamento do mecanismo de acesso e mudanças na rotação do disco. Entretanto, é possível acessar um conjunto de trilhas vizinhas sem nenhum deslocamento do mecanismo de leitura. Esta característica dos discos CD-ROM é denominada varredura estática. Nos discos atuais a amplitude de varredura estática pode atingir até 60 trilhas (± 30 trilhas a partir da trilha corrente). 0 acesso a trilhas dentro da amplitude da varredura estática consome 1 milissegundo por trilha adicional, sendo realizado por um pequeno deslocamento angular do feixe de laser a partir da trilha corrente, chamada

de ponto de âncora. Neste caso, o tempo de procura é desprezível se comparado com o tempo de latência rotacional. Para acessar trilhas fora da varredura estática o tempo de procura varia de 200 até 600 milissegundos.

Conforme mostrado na seção imediatamente anterior, a estrutura seqüencial indexada permite tanto o acesso seqüencial quanto o acesso randômico aos dados. Nos discos magnéticos a estrutura seqüencial indexada é implementada mantendo-se um índice de cilindros na memória principal. Neste caso, cada acesso demanda apenas um deslocamento do mecanismo de acesso, desde que cada cilindro contém um índice de páginas com o maior valor de chave em cada página daquele cilindro. Entretanto, para aplicações dinâmicas esta condição não pode ser mantida se um número grande de registros tem que ser adicionado ao arquivo. No caso dos discos CD-ROM esta organização é particularmente interessante devido à natureza estática da informação.

O conceito de **cilindro em discos magnéticos** pode ser estendido para os discos CD-ROM. Barbosa e Ziviani (1992) denominaram o conjunto de trilhas cobertas.por uma varredura estática de **cilindro ótico**. 0 cilindro ótico difere do cilindro de um disco magnético em dois aspectos: (i) as trilhas de uma varredura estática que compõem um cilindro ótico podem se sobrepor a trilhas de outro cilindro ótico com ponto de âncora próximo; (ii) como as trilhas têm capacidade variável, os cilindros óticos com ponto de âncora em trilhas mais internas têm capacidade menor do que cilindros óticos com ponto de âncora em trilhas mais externas.

Em um trabalho analítico sobre discos óticos Christodoulakis e Ford (1988) demonstraram que o número de deslocamentos e a distância total percorrida pela cabeça ótica de leitura são minimizados quando (i) as trilhas de duas varreduras estáticas consecutivas não se sobrepõem, (ii) a cabeça de leitura se movimenta apenas em uma direção durante a recuperação de um conjunto de dados.

A estrutura seqüencial indexada pode ser implementada eficientemente no CD-ROM considerando a natureza estática da informação e a capacidade de varredura estática do mecanismo de leitura. A partir destas observações, Barbosa e Ziviani (1992) propuseram uma estrutura seqüencial indexada para discos CD-ROM na qual o mecanismo de leitura é posicionado em cilindros óticos pré-selecionados, com o objetivo de evitar sobreposição de varreduras e minimizar o número de deslocamentos da cabeça de leitura. Para tal, a estrutura de índices é construída de maneira que cada página de índices faça referência ao maior número possível de páginas de dados de um cilindro ótico.

A Figura 5.6 mostra esta organização para um arquivo exemplo de 3 Megabytes, alocado a partir da trilha 1940 do disco, no qual cada página ocupa 2 Kbytes (equivalente a 1 setor do disco). Supondo que o mecanismo de leitura tenha uma amplitude de varredura estática de 8 trilhas, na posição

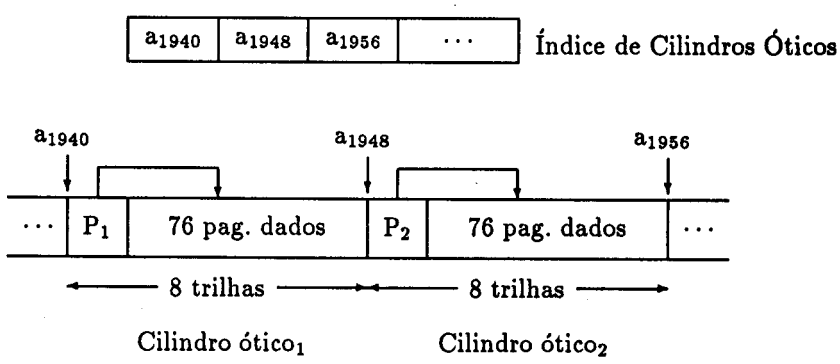


Figura 5.6: Organização de urn arquivo indexado seqüencial para o CD-ROM

de trilha número 1940 é possível acessar aproximadamente 78 setores sem deslocamento da cabeça de leitura. Assim sendo, para obter uma organização seqüencial indexada para este arquivo são necessários os seguintes passos:

1. Alocar o arquivo no disco determinando a trilha inicial e calculando a trilha final que ele deve ocupar;
2. Computar o número total de cilindros óticos para cobrir todas as trilhas do arquivo sem que haja sobreposição de trilhas. Determine os respectivos pontos de âncora;
3. Construir um índice de cilindros óticos, o qual deverá conter o valor de chave mais alto associado aos registros que estão dentro de cada cilindro ótico. 0 índice de cilindros óticos deve ser mantido na memória principal;
4. Construir um índice de páginas para cada cilindro ótico. Este índice deverá conter o valor de chave mais alto de cada página e deve ser armazenado na trilha central ou ponto de âncora de cada cilindro ótico.

Para recuperar uma dada chave de pesquisa o primeiro passo é obter o endereço do cilindro ótico que contém a chave consultando o índice de cilindros óticos na memória principal. 0 mecanismo de leitura é então deslocado para o ponto de âncora selecionado na única operação de busca necessária. A seguir, o índice de páginas é lido e a página de dados contendo a chave de pesquisa poderá ser encontrada dentro dos limites da varredura estática. Os detalhes para obtenção do número de trilhas que um arquivo deve ocupar a partir de determinada posição no disco, os pontos de âncora dos cilindros óticos, ou quaisquer outros, podem ser obtidos em Barbosa e Ziviani (1992).

5.3 Árvores de Pesquisa

As árvores binárias de pesquisa introduzidas na Seção 4.3 são estruturas de dados muito eficientes quando se deseja trabalhar com tabelas que caibam inteiramente na memória principal do computador. Elas satisfazem condições e requisitos diversificados e conflitantes, tais como acesso direto e seqüencial, facilidade de inserção e retirada de registros e boa utilização de memória.

Vamos agora considerar o problema de recuperar informação em grandes arquivos de dados que estejam armazenados em memória secundária do tipo disco magnético. Uma forma simplista de resolver este problema utilizando árvores binárias de pesquisa é armazenar os nodos da árvore no disco e os apontadores à esquerda e à direita de cada nodo se tornam endereços de disco ao invés de endereços de memória principal. Se a pesquisa for realizada utilizando o algoritmo de pesquisa para memória principal visto anteriormente serão necessários da ordem de $\log_2 n$ acessos a disco, significando que um arquivo com $n = 10^6$ registros necessita aproximadamente $\log_2 10^6$ —20 buscas no disco.

Para diminuir o número de acessos a disco, os nodos da árvore podem ser agrupados em páginas, conforme ilustra a Figura 5.7. Neste exemplo, o formato da árvore muda de binário para quaternário, com quatro filhos por página, onde o número de acessos a páginas cai para metade no pior caso. Para arquivos divididos em páginas de 127 registros, é possível recuperar qualquer registro do arquivo com três acessos a disco no pior caso.

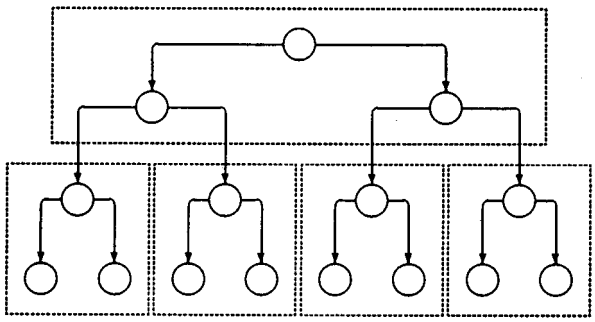


Figura 5.7: Árvore binária dividida em páginas

A forma de organizar os nodos da árvore dentro de páginas é muito importante sob o ponto de vista do número esperado de páginas lidas quando se realiza uma pesquisa na árvore. A árvore da Figura 5.7 é ótima sob este aspecto. Entretanto, a organização ótima é difícil de ser obtida durante a construção da árvore, tornando-se um problema de otimização muito com-

plexo. Um algoritmo bem simples, o da alocação seqüencial, armazena os nodos em posições consecutivas na página à medida em que vão chegando, sem considerar o formato físico da árvore. Este algoritmo utiliza todo o espaço disponível na página, mas os nodos dentro da página estão relacionados pela localidade da ordem de entrada das chaves e não pela localidade dentro da árvore, o que torna o tempo de pesquisa muito pior do que o tempo da árvore ótima.

Um método de alocação de nodos em páginas que leva em consideração a relação de proximidade dos nodos dentro da estrutura da árvore foi proposto por Muntz e Uzgalis (1970). No método proposto, o novo nodo a ser inserido é sempre colocado na mesma página do nodo pai. Se o nodo pai estiver em uma página cheia, então uma nova página é criada e o novo nodo é colocado no início da nova página. Knuth (1973) mostrou que o número esperado de acessos a páginas em uma pesquisa na árvore é muito próxih (1973) mostrou que o número esperado de acessos a páginas em uma pesquisa na árvore é muito próximo do ótimo. Entretanto, a ocupação média das páginas é extremamente baixa, da ordem de 10%, o que torna o algoritmo inviável para aplicações práticas.

Uma solução brilhante para este problema, simultaneamente com uma proposta para manter equilibrado o crescimento da árvore e permitir inserções e retiradas a vontade, é o assunto da próxima seção.

5.3.1 Árvores B

O objetivo desta seção é o de apresentar uma técnica de organização e manu tenção de arquivos através do uso de árvores B (Bayer e McCreight, 1972). A origem do nome árvores B nunca foi explicada pelos autores, R. Bayer e E. McCreight, cujo trabalho foi desenvolvido no Boeing Scientific Research Labs. Alguns autores sugerem que o "B" se refere a "Boeing", enquanto Comer (1979) acha apropriado pensar em "B-trees" como "Bayer-trees", por causa das contribuições de R. Bayer ao assunto. Outras introduções ao assunto podem ser encontradas em Comer (1979), Wirth (1976), e Knuth (1973).

Definição e Algoritmos

Quando uma árvore de pesquisa possui mais de um registro por nodo ela deixa de ser binária. Estas árvores são chamadas n-árias, pelo fato de possuírem mais de dois descendentes por nodo. Nestes casos os nodos são mais comumente chamados de páginas.

A árvore B é n-ária. Em uma árvore B de ordem m temos que:

1. cada página contém no mínimo m registros (e m + 1 descendentes) e

no máximo 2m registros (e 2m + 1 descendentes), exceto a página raiz que pode conter entre 1 e 2m registros;

2. todas as páginas folha aparecem no mesmo nível.

Uma árvore B de ordem m = 2 com 3 níveis pode ser vista na Figura 5.8. Todas as páginas contêm 2, 3, ou 4 registros, exceto a raiz que pode conter um registro apenas. Os registros aparecem em ordem crescente da esquerda para a direita. Este esquema representa uma extensão natural da organização da árvore binária de pesquisa. A Figura 5.9 apresenta a forma geral de uma página de uma árvore B de ordem m.

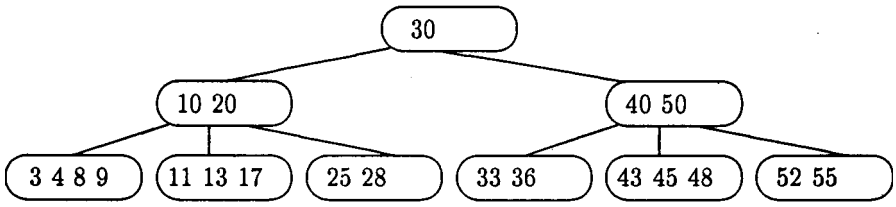


Figura 5.8: Árvore B de ordem 2 com 3 níveis

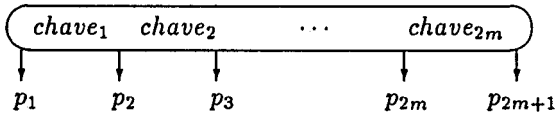


Figura 5.9: Nodo de uma árvore B de ordem m com 2m registros

A estrutura de dados árvore B será utilizada para implementar o tipo abstrato de dados Dicionário, com as operações Inicializa, Pesquisa, Insere e Retira. A estrutura e a representação do Dicionário é apresentada no Programa 5.3, onde *mm* significa *2m*. 0 procedimento Inicializa é extremamente simples, conforme ilustra o Programa 5.4.

Um procedimento Pesquisa para árvore B é semelhante ao algoritmo Pesquisa para árvore binária de pesquisa, conforme pode ser visto no Programa 5.5. Para encontrar um registro x, primeiro compare a chave que rotula o registro com as chaves que estão na página raiz, até encontrar o intervalo onde ela se encaixa. Depois, siga o apontador para a subárvore correspondente ao intervalo citado e repita o processo recursivamente, até que a chave procurada seja encontrada ou então uma página folha seja atingida (no caso um apontador nulo). Na implementação do Programa 5.5 a

```
type Registro =record
                 Chave : TipoChave;
                 {outros componentes}
               end;
     Apontador = ^Página;
     Página = record
                n : 0..mm;
                r : array[1..mm] of Registro;
                p : array[0..mm] of Apontador;
              end;
     TipoDicionário = Apontador;
```

Programa 5.3: Estrutura do dicionário para árvore B

```
procedure Inicializa (var Dicionário : TipoDicionário);
begin
  Dicionário := nil;
end;
```

Programa 5.4: Procedimento para inicializar uma árvore B

localização do intervalo onde a chave se encaixa é obtida através de uma pesquisa seqüencial. Entretanto, esta etapa pode ser realizada de forma mais eficiente através do uso de pesquisa binária (vide Seção 4.2).

Vamos ver agora como inserir novos registros em uma árvore B. Em primeiro lugar é preciso localizar a página apropriada onde o novo registro deve ser inserido. Se o registro a ser inserido encontra seu lugar em uma página com menos de 2m registros, o processo de inserção fica limitado àquela página. Entretanto, quando um registro precisa ser inserido em uma página já cheia (com 2m registros), o processo de inserção pode provocar a criação de uma nova página. A Figura 5.10, parte (b), ilustra o que acontece quando o registro contendo a chave 14 é inserido na árvore da parte (a). 0 processo é composto pelas seguintes etapas:

1. O registro contendo a chave 14 não é encontrado na árvore, e a página 3 (onde o registro contendo a chave 14 deve ser inserido) está cheia.

2. A página 3 é dividida em 2 páginas, o que significa que uma nova página 4 é criada.

3. Os 2m+1 registros (no caso são 5 registros) são distribuídos igualmente entre as páginas 3 e 4, e o registro do meio (no caso o registro contendo a chave 20) é movido para a página pai no nível acima.

```
procedure Pesquisa (var x : Registro; var Ap : Apontador);
var i : integer;
begin
  if Ap = nil
  then writeln (' Registro não está presente na árvore' )
  else with Ap^ do
       begin
       i := 1;
       while (i < n) and (x.Chave > r[i].Chave) do i := i + 1;
       if x.Chave = r[i].Chave
       then x := r[i]
       else  if x.Chave < r[i].Chave
             then Pesquisa (x, p[i-1])
             else Pesquisa (x, p[i])
       end
end;
```

Programa 5.5: Procedimento para pesquisar na árvore B

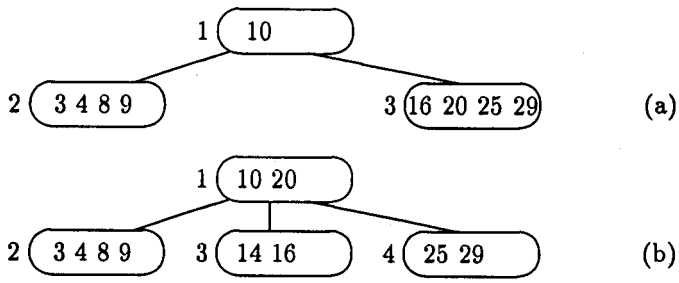


Figura 5.10: Inserção em uma árvore B de ordem 2

No esquema de inserção apresentado acima, a página pai tem que acomodar um novo registro. Se a página pai também estiver cheia, então o mesmo processo de divisão tem que ser aplicado de novo. No pior caso, o processo de divisão pode se propagar até a raiz da árvore e, neste caso, a árvore aumenta sua altura de um nível. E interessante observar que uma árvore B somente aumenta sua altura através da divisão da raiz.

Um primeiro refinamento do procedimento Insere pode ser visto no Pro-. grama 5.6. 0 procedimento contém um outro procedimento interno recursivo, de nome Ins, de estrutura semelhante ao Programa 5.5 acima. Quando um apontador nulo é encontrado, significa que o ponto de inserção foi localizado. Neste momento o parâmetro Cresceu passa a indicar este fato informando que um registro vai ser passado para cima através do parâmetro

RegRetorno para ser inserido na próxima página que contenha espaço para acomodá-lo. Se Cresceu = true no momento do retorno do procedimento Ins para o procedimento Insere significa que a página raiz foi dividida e então uma nova página raiz deve ser criada para acomodar o registro emergente, fazendo com que a árvore cresca na altura.

```
procedure Insere (Reg: Registro; var Ap: Apontador);

  procedure Ins (Reg:Registro; Ap:Apontador; var Cresceu:Boolean;
                 var RegRetorno:Registro; var ApRetorno:Apontador);
  var i : integer;
  begin
    if Ap = nil
    then begin
         Cresceu := true;
         Atribui Reg a RegRetorno;  Atribui nil a ApRetorno;
         end
    else with Ap^ do
         begin
         i := 1;
         while (i < n) and (x.Chave > r[i].Chave) do i := i + 1;
         if x.Chave = r[i].Chave
         then writeln (' Erro : Registro já está presente na árvore')
         else if x.Chave < r[i].Chave
              then Ins (x, p[i-1], Cresceu, RegRetorno, ApRetorno)
              else Ins (x, p[i], Cresceu, RegRetorno, ApRetorno);
         if Cresceu
         then if (Nº de registros em Ap) < mm
              then Insere na página Ap e Cresceu := false
              else begin { Overflow: página tem que ser dividida }
                   Cria nova página ApTemp;
                   Transfere metade dos registros de Ap para ApTemp;
                   Atribui registro do meio a RegRetorno;
                   Atribui ApTemp a ApRetorno;
                   end;
         end
  end;

begin {Insere}
  Ins (Reg, Ap, Cresceu, RegRetorno, ApRetorno);
  if Cresceu then Cria nova página raiz para RegRetorno e ApRetorno;
end;
```

Programa 5.G: Primeiro refinamento do algoritmo Insere na árvore B

O procedimento Insere utiliza o procedimento auxiliar InsereNaPágina mostrado no Programa 5.7.

```
procedure InsereNaPágina (Ap : Apontador;
                          Reg : Registro; ApDir : Apontador);
var NãoAchouPosição : Boolean;
    k               : integer;
begin
with Ap^ do
  begin
  k := n;
  NãoAchouPosição := k > 0;
  while NãoAchouPosição do
    if Reg.Chave < r[k].Chave
    then begin
           r[k+1] := r[k]; p[k+1] := p[k]; k := k - 1;
           if k < 1 then NãoAchouPosição := false;
           end
    else NãoAchouPosição := false;
  r[k+1] := Reg; p[k+1] := ApDir; n := n + 1;
  end;
end;
```

Programa 5.7: Procedimento InsereNaPágina

O Programa 5.8 apresenta o refinamento final do procedimento Insere.

```
procedure Insere (Reg: Registro; var Ap: Apontador);
var Cresceu      : Boolean;
    RegRetorno : Registro;
    ApRetorno  : Apontador;
    ApTemp     : Apontador;

  procedure Ins (Reg : Registro; Ap : Apontador; var Cresceu : Boolean;
                 var RegRetorno : Registro; var ApRetorno : Apontador);
  var i, j : integer; ApTemp : Apontador;
  begin
    if Ap = nil
    then begin
           Cresceu := true;  RegRetorno := Reg;  ApRetorno := nil;
           end
    else with Ap^ do
         begin
         i := 1;
         while (i < n) and (Reg.Chave > r[i].Chave) do i := i + 1;
         if Reg.Chave = r[i].Chave
         then begin
                writeln ('Erro:Registro já está presente'); Cresceu := false;
                end
         else begin
```

```
            if Reg.Chave < r[i].Chave
            then Ins (Reg, p[i-1], Cresceu, RegRetorno, ApRetorno)
            else Ins (Reg, p[i], Cresceu, RegRetorno, ApRetorno);
            if Cresceu
            then if n < mm
            then begin { Página tem espaço }
                 InsereNaPágina(Ap, RegRetorno, ApRetorno);
                 Cresceu := false;
                 end
            else begin { Overflow: página tem que ser dividida }
                 new (ApTemp);
                 ApTemp^.n := 0;  ApTemp^.p[0] := nil;
                 if i <= m+1
                 then begin
                      InsereNaPágina (ApTemp, r[mm], p[mm]);
                      n := n - 1;
                      InsereNaPágina (Ap, RegRetorno, ApRetorno);
                      end
                 else InsereNaPágina (ApTemp,RegRetorno,ApRetorno);
                 for j:=m+2 to mm do
                   InsereNaPágina (ApTemp, r[j], p[j]);
                 n := m;  ApTemp^.p[0] := p[m+1];
                 RegRetorno := r[m+1]; ApRetorno := ApTemp;
                 end;
            end;
       end;
  end; { Ins }

begin { Insere }
  Ins (Reg, Ap, Cresceu, RegRetorno, ApRetorno);
  if Cresceu
  then begin { Árvore cresce na altura pela raiz }
       new (ApTemp);
       ApTemp^.n := 1;
       ApTemp^.r[1] := RegRetorno;
       ApTemp^.p[1] := ApRetorno;
       ApTemp^.p[0] := Ap;  Ap := ApTemp;
       end;
end; { Insere }
```

Programa 5.8: Refinamento final do algoritmo Insere

A Figura 5.11 mostra o resultado obtido quando se insere uma seqüência de chaves em uma árvore B de ordem 2: A árvore da Figura 5.11, parte (a), é obtida após a inserção da chave 20, a árvore da parte (b) é obtida após a inserção das chaves 10, 40, 50 e 30 na árvore da parte (a), a árvore da parte

(c) é obtida após a inserção das chaves 55, 3, 11, 4, 28, 36, 33, 52, *17*, 25 e 13 na árvore da parte (b) e, finalmente, a árvore da parte (d) é obtida após a inserção das chaves 45, 9, 43, 8 e 48.

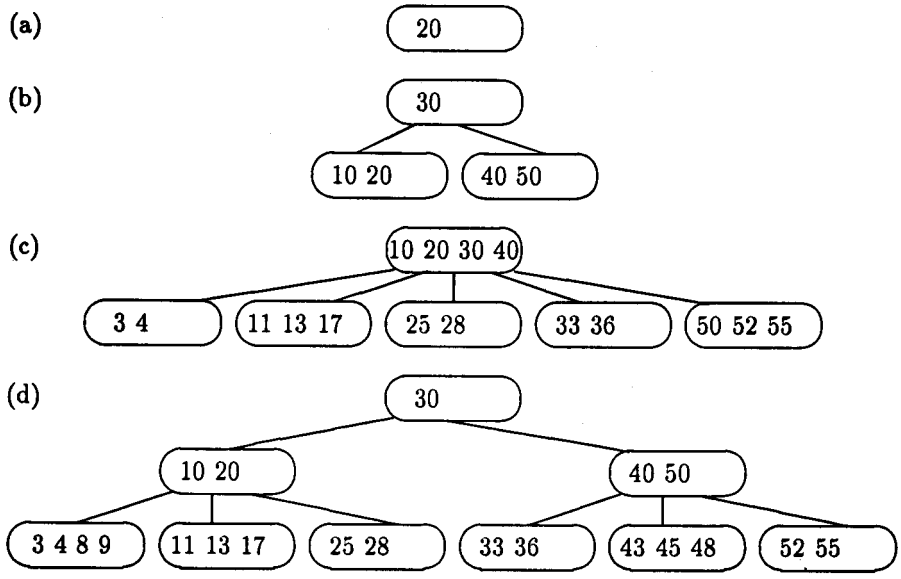


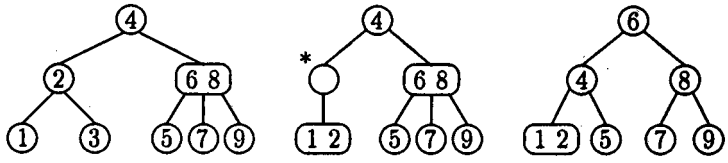
Figura 5.11: Crescimento de uma árvore B de ordem 2

A última operação a ser estudada é de retirada. Quando a página que contém o registro a ser retirado é uma página folha a operação é simples. No caso de não ser uma página folha, o registro a ser retirado deve ser primeiro substituído por um registro contendo uma chave adjacente (antecessora ou sucessora), como no caso da operação de retirada de registros em árvores binárias de pesquisa, conforme mostrado na Seção 4.3. Para localizar uma chave lexicograficamente antecessora, basta procurar pela página folha mais à direita na subárvore à esquerda. Por exemplo, a antecessora da chave 30 na árvore da Figura 5.11 (d) é a chave 28.

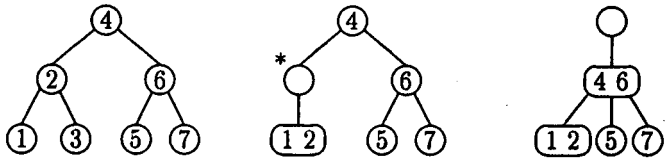
Tão logo o registro seja retirado da página folha, é necessário verificar se pelo menos m registros passam a ocupar a página. Quando menos do que m registros passam a ocupar a página significa que a propriedade árvore B é violada. Para reconstituir a propriedade árvore B é necessário tomar emprestado um registro da página vizinha. Conforme pode ser verificado na Figura 5.12, existem duas possibilidades:

1. O número de registros na página vizinha é maior do que m: basta tomar um registro emprestado e trazê-lo para a página em questão via página pai. A Figura 5.12 (a) mostra a retirada da chave 3.

2. Não existe um número suficiente de registros na página vizinha (a página vizinha possui exatamente m registros): neste caso o número total de registros nas duas páginas é 2m — 1 e, conseqüentemente, as duas páginas têm que ser fundidas em uma só, tomando emprestado da página pai o registro do meio, o que permite liberar uma das páginas. Este processo pode se propagar até a página raiz e no caso em que o número de registros da página raiz fica reduzido a zero ela é eliminada, causando redução na altura da árvore. A Figura 5.12 (b) mostra a retirada da chave 3.



(a) Página vizinha possui mais do que *m* registros



(b) Página vizinha possui exatamente m registros

Figura 5.12: Retirada da chave 3 na árvore B de ordem m = 1

O procedimento Retira é apresentado no Programa 5.9. 0 procedimento contém um outro procedimento interno recursivo, de nome Ret. No procedimento Ret, quando a página que contém o registro a ser retirado é uma página folha a operação é simples. No caso de não ser uma página folha, a tarefa de localizar o registro antecessor é realizada pelo procedimento Antecessor. A condição de que menos do que m registros passam a ocupar a página é sinalizada pelo parâmetro Diminuiu, fazendo com que o procedimento Reconstitui seja ativado.

```
procedure Retira (Ch : TipoChave; var Ap : Apontador);
var  Diminuiu  : Boolean;
     Aux       : Apontador;

procedure Ret(Ch:TipoChave; var Ap:Apontador; var Diminuiu:Boolean);
var Ind, j : integer;
```

```
procedure Reconstitui (ApPag : Apontador; ApPai : Apontador;
                       PosPai : integer; var Diminuiu : Boolean);
var Aux       : Apontador;
    DispAux, j : integer;
begin
  if PosPai < ApPai^.n
  then begin { Aux = Página à direita de ApPag }
       Aux := ApPai^.p[PosPai+1];
       DispAux := (Aux^.n - m +1) div 2;
       ApPag^.r[ApPag^.n+1] := ApPai^.r[PosPai+1];
       ApPag^.p[ApPag^.n+1] := Aux^.p[0];
       ApPag^.n := ApPag^.n + 1;
       if DispAux > 0
       then begin { Existe folga: transfere de Aux para ApPag }
            for j := 1 to DispAux-1 do
               InsereNaPágina (ApPag, Aux^.r[j], Aux^.p[j]);
            ApPai^.r[PosPai+1] := Aux^.r[DispAux];
            Aux^.n := Aux^.n-DispAux;
            for j := 1 to Aux^.n do Aux^.r[j] := Aux^.r[j+DispAux];
            for j := 0 to Aux^.n do Aux^.p[j] := Aux^.p[j+DispAux];
            Diminuiu := false;
            end
       else begin { Fusão: intercala Aux em ApPag e libera Aux }
            for j := 1 to m do
               InsereNaPágina (ApPag, Aux^.r[j], Aux^.p[j]);
            dispose (Aux);
            for j := PosPai+1 to ApPai^.n-1 do with ApPai^ do
               begin r[j] := r[j+1]; p[j] := p[j+1]; end;
            ApPai^.n := ApPai^.n - 1;
            if ApPai^.n >= m then Diminuiu := false;
            end;
       end
  else begin { Aux = Página à esquerda de ApPag }
       Aux := ApPai^.p[PosPai-1];
       DispAux := (Aux^.n - m + 1) div 2;
       for j := ApPag^.n downto 1 do ApPag^.r[j+1] := ApPag^.r[j];
       ApPag^.r[1] := ApPai^.r[PosPai];
       for j := ApPag^.n downto 0 do ApPag^.p[j+1] := ApPag^.p[j];
       ApPag^.n := ApPag^.n + 1;
       if DispAux > 0
       then begin { Existe folga: transfere de Aux para ApPag }
            for j := 1 to DispAux-1 do with Aux^ do
               InsereNaPágina(ApPag, r[n+1-j], p[n+1-j]);
            ApPag^.p[0] := Aux^.p[Aux^.n+1-DispAux];
            ApPai^.r[PosPai] := Aux^.r[Aux^.n+1-DispAux];
            Aux^.n := Aux^.n - DispAux;
            Diminuiu := false;
            end
```

```
        else begin { Fusão: intercala ApPag em Aux e libera ApPag }
             for j := 1 to m do
                InsereNaPágina (Aux, ApPag^.r[j], ApPag^.p[j]);
             dispose (ApPag);
             ApPai^.n := ApPai^.n - 1;
             if ApPai^.n >= m then Diminuiu := false;
             end;
        end;
end; { Reconstitui }


procedure Antecessor (Ap : Apontador; Ind : integer;
                      ApPai : Apontador; var Diminuiu : Boolean);
begin
with ApPai^ do
  begin
  if p[n] <> nil
  then begin
        Antecessor (Ap, Ind, p[n], Diminuiu);
        if Diminuiu then Reconstitui (p[n], ApPai, n, Diminuiu);
        end
  else begin Ap^.r[Ind] := r[n];  n := n-1;  Diminuiu := n<m; end;
  end;
end; { Antecessor }

begin { Ret }
  if Ap = nil
  then begin
        writeln ( 'Erro: registro não está na árvore' );  Diminuiu := false;
        end
  else with Ap^ do
       begin
       Ind := 1;
       while (Ind < n) and (Ch > r[Ind].Chave) do Ind := Ind + 1;
       if Ch = r[Ind].Chave
       then if p[Ind-1] = nil
            then begin { Página folha }
                 n := n-1;  Diminuiu := n<m;
                 for j := Ind to n do
                    begin r[j] := r[j+1]; p[j] := p[j+1]; end;
                 end
            else begin { Página não é folha: trocar com antecessor }
                 Antecessor (Ap, Ind, p[Ind-1], Diminuiu);
                 if Diminuiu
                 then Reconstitui (p[Ind-1], Ap, Ind-1, Diminuiu);
                 end
       else begin
            if Ch > r[Ind].Chave then Ind := Ind + 1;
```

```
            Ret (Ch, p[Ind-1], Diminuiu);
            if Diminuiu
            then Reconstitui(p[Ind-1], Ap, Ind-1, Diminuiu);
            end;
        end;
end; { Ret }

begin { Retira }
  Ret (Ch, Ap, Diminuiu);
  if Diminuiu and (Ap^.n = 0)
  then begin { Árvore diminui na altura }
         Aux := Ap;  Ap := Aux^.p[0];  dispose (Aux);
       end;
end; { Retira }
```

Programa 5.9: Procedimento Retira

A Figura 5.13 mostra o resultado obtido quando se retira a seguinte seqüência de chaves da árvore B: 45 30 28; 50 8 10 4 20 40 55 17 33 11 36; 3 11 52. Cada ponto-e-vírgula corresponde a um salto de uma árvore para outra no desenho da Figura 5.13.

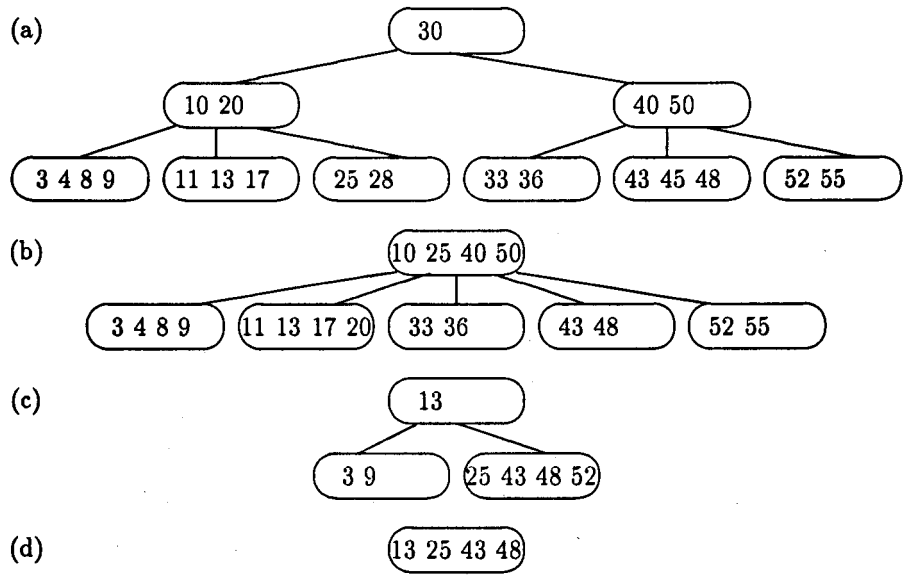


Figura 5.13: Decomposição de uma árvore B de ordem 2

5.3.2 Árvores B*

Existem várias alternativas para implementação da árvore B original. Uma delas é a árvore B*. Em uma árvore B*, todos os registros são **armazenados** no último nível (páginas folhas). Os níveis acima do último nível constituem um índice cuja organização é a organização de uma árvore B.

A Figura 5.14 mostra a separação lógica entre o índice e os registros que constituem o arquivo propriamente dito. No índice só aparecem as chaves, sem nenhuma informação associada, enquanto nas páginas folha estão todos os registros do arquivo. As páginas folha são conectadas da **esquerda** para a direita, o que permite um acesso seqüencial mais eficiente do que o acesso através do índice. Além do acesso seqüencial mais eficiente, as árvores B* apresentam outras vantagens sobre as árvores B, como a de facilitar o acesso concorrente ao arquivo, conforme veremos adiante.

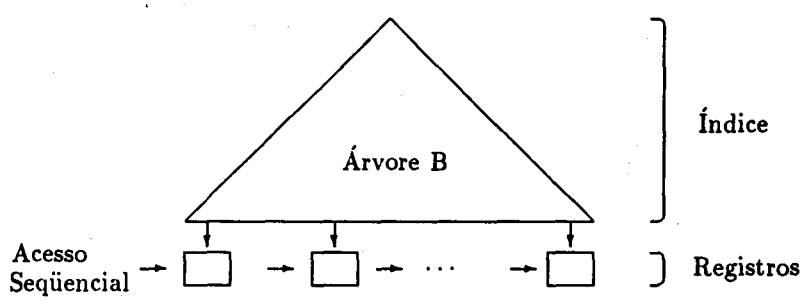


Figura 5.14: Estrutura de uma árvore B*

Para recuperar um registro, o processo de pesquisa inicia-se na raiz da árvore e continua até uma página folha. Como todos os registros residem nas folhas, a pesquisa não pára se a chave procurada for encontrada em uma página do índice. Neste caso o apontador à direita é seguido até que uma página folha seja encontrada. Esta característica pode ser vista na árvore B* da Figura 5.15 onde as chaves 29, 60, e 75 aparecem no índice e em registros do arquivo. Os valores encontrados ao longo do caminho são irrelevantes desde que eles conduzam à página folha correta.

Como não há necessidade do uso de apontadores nas páginas folha é possível utilizar este espaço para armazenar uma quantidade maior de registros em cada página folha. Para tal devemos utilizar um valor de m diferente para para as páginas folha. Isto não cria nenhum problema para os algoritmos de inserção pois as metades de uma página que está sendo particionada permanecem no mesmo nível da página original antes da partição (algo semelhante acontece com a retirada de registros).

A estrutura de dados árvore B* apresentada no Programa 5.10.

O procedimento Pesquisa deve ser implementado como no Programa 5.11.

```
type Registro =record
                 Chave : TipoChave;
                 {outros componentes}
               end;
     Apontador = ^Página;
     PáginaTipo = (Interna, Externa);
     Página = record
              case Pt: PáginaTipo of
              Interna : ( n : 0..mm;
                          r : array[1..mm] of TipoChave;
                          p : array[0..mm] of Apontador; );
              Externa :( n : 0..mm2;
                          r : array[1..mm2] of Registro; )
            end;
     TipoDicionário = Apontador;
```

Programa 5.10: Estrutura do dicionário para árvore B*

```
procedure Pesquisa (var x : Registro; var Ap : Apontador);
var i : integer;
begin
  if Ap^.Pt = Interna
  then with Ap^ do
       begin
       i := 1;
       while (i < n) and (x.Chave > r[i]) do i := i + 1;
       if x.Chave < r[i]
       then Pesquisa (x, p[i-1])
       else Pesquisa (x, p[i])
       end
  else with Ap^ do
       begin
       i := 1;
       while (i < n) and (x.Chave > r[i].Chave) do i := i + 1;
       if x.Chave = r[i].Chave
       then x := r[i]
       else  writeln (' Registro não está presente na árvore' );
       end;
end;
```

Programa 5.11: Procedimento para pesquisar na árvore B*

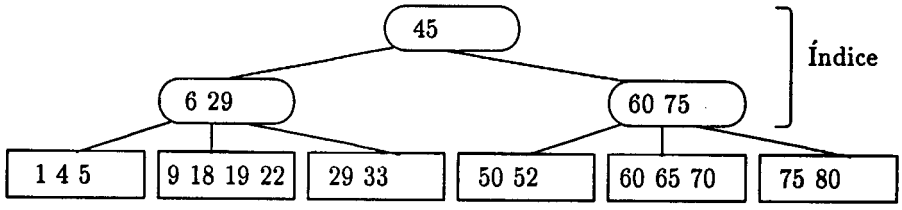


***Figura* 5.15: Exemplo de uma árvore B"**

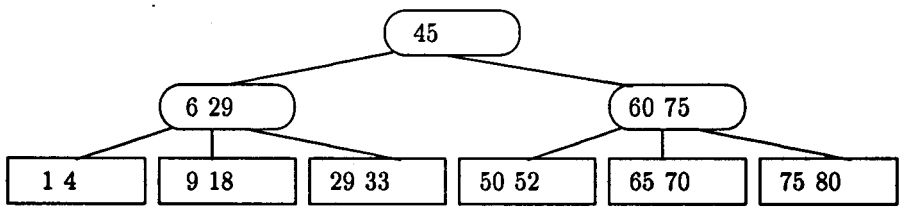
A operação de inserção de um registro em uma árvore B* é essencialmente igual à inserção de um registro em uma árvore B. A única diferença é que quando uma folha é dividida em duas, o algoritmo promove uma cópia da chave que pertence ao registro do meio para a página pai no nível anterior, retendo o registro do meio na página folha da direita.

A operação de retirada em uma árvore B* é relativamente mais simples do que em uma árvore B. 0 registro a ser retirado reside sempre em uma página folha, o que torna sua remoção simples, não havendo necessidade de utilização do procedimento para localizar a chave antecessora (vide procedimento Antecessor do Programa 5.9). Desde que a página folha fique pelo menos com metade dos registros, as páginas do índice não precisam ser modificadas, mesmo se uma cópia da chave que pertence ao registro a ser retirado esteja no índice. A Figura 5.16 mostra a árvore B* resultante quando a seguinte seqüência de chaves é retirada da árvore B* da Figura 5.15: 5 19 22 60; 9. Observe que a retirada da chave 9 da árvore da Figura 5.16, parte (a), provoca a redução da árvore.

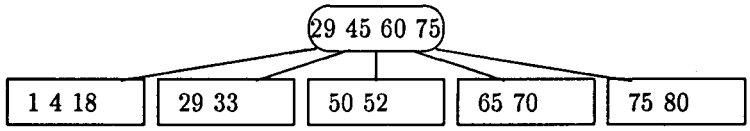
5.3.3 Acesso Concorrente em Árvores B*

Em muitas aplicações o acesso simultâneo ao banco de dados por mais de um usuário é fator importante. Nestes casos, permitir acesso para apenas um processo de cada vez pode criar um gargalo inaceitável para o sistema de banco de dados. Concorrência é então introduzida para aumentar a utilização e melhorar o tempo de resposta do sistema. Deste modo o uso de árvores B* em tais sistemas deve permitir o processamento simultâneo de várias solicitações diferentes.

Entretanto, existe a necessidade de se criar mecanismos chamados **protocolos** para garantir a integridade tanto dos dados quanto da estrutura. Considere a situação em que dois processos estejam simultaneamente acessando o banco de dados. Em um determinado momento, um dos processos está percorrendo uma página para localizar o intervalo onde a chave de pesquisa se encaixa e seguir o apontador para a subárvore correspondente, enquanto o outro processo está inserindo um novo registro que provoca divisões



(a) Retirada dos registros 5, 19, 22, 60 da árvore da Figura 5.15



(b) Retirada do registro 9 em (a): a estrutura do índice é modificada

Figura 5.16: Retirada de registros em árvores B*

de páginas no mesmo caminho da árvore. Pode acontecer que o processo que está percorrendo a página pode obter um apontador que fica apontando para uma subárvore errada ou para um endereço inexistente.

Uma página é chamada segura quando se sabe que não existe possibilidade de modificações na estrutura da árvore, como conseqüência de uma operação de inserção ou de retirada naquela página. Cabe lembrar que a operação de recuperação não altera a estrutura da árvore, ao contrário das operações de inserção ou retirada, que podem provocar modificações na estrutura da árvore. No caso de operações de inserção, uma página é considerada segura se o número atual de chaves naquela página é menor do que 2m. No caso de operações de retirada, uma página é considerada segura quando o número de chaves na página é maior do que m. Os algoritmos para acesso concorrente fazem uso destes fatos para aumentar o nível de concorrência em uma árvore B*.

Bayer e Schkolnick (1977) apresentam um conjunto de três diferentes alternativas de protocolos para travamentos[1] *(lock protocols)* que asseguram a integridade dos caminhos de acesso aos dados da árvore B*, e, ao mesmo tempo, permitem acesso concorrente. Em uma das alternativas pro-postas a operação de recuperação trava (ou retém) uma página tão logo ela seja lida, de modo que outros processos não possam interferir com a página. Na medida em que a pesquisa continua em direção ao nível seguinte

[1]Um protocolo para travamento é um mecanismo que assegura a modificação de apenas uma página de cada vez na árvore.

da árvore, a trava aplicada na página antecessora é liberada, permitindo a leitura das páginas por outros processos.

Um processo executando uma operação de recuperação é chamado **processo leitor,** enquanto um processo executando uma operação de inserção ou de retirada é chamado **processo modificador.** A operação de modificação requer protocolos mais sofisticados, porque tal operação pode modificar as páginas antecessoras nos níveis acima. Em uma das alternativas apresentadas por Bayer e Schkolnick (1977), o processo modificador coloca um travamento exclusivo em cada página acessada, podendo mais tarde liberar o travamento caso a página seja segura.

Vamos apresentar a seguir o **protocolo para processos leitores e** o **protocolo para processos modificadores** relativos à alternativa mais simples dentre as três alternativas apresentadas por Bayer e Schkolnick (1977). Estes protocolos utilizam dois tipos de travamento:

1. *o travamento-para-leitura,* que permite um ou mais leitores acessarem os dados, mas não permite inserção ou retirada de chaves;
2. *o travamento-exclusivo,* que permite qualquer tipo de operação na página (quando um processo recebe este tipo de travamento, nenhum outro processo pode operar na página).

O protocolo para processos leitores é:

(0) Coloque um travamento-para-leitura na raiz;
(1) Leia a página raiz e faça-a página corrente;
(2) Enquanto a página corrente não é uma página folha faça {o n° de travamentos-para-leitura mantidos pelo processo é = 1 }
(3) Coloque um travamento-para-leitura no descendente apropriado;
(4) Libere o travamento-para-leitura na página corrente;
(5) Leia a descendente cia página corrente e faça-a página corrente.

O protocolo para processos modificadores é:

(0) Coloque um travamento-exclusivo na raiz;
(1) Leia a página raiz e faça-a página corrente;
(2) Enquanto a página corrente não é uma página folha faça {o némero de travamentos-exclusivos mantidos pelo processo á " 1 }
(3) Coloque um travamento-exclusivo no descendente apropriado;
(4) Leia a descendente da põgina corrente e faãa-a põgina corrente;
(5) Se a põgina corrente á segura entüo libere todos os travamentos mantidos sobre as põginas antecessoras da põgina corrente.

Para exemplificar o funcionamento do modelo do protocolo para processos modificadores considere a modificação da página y da árvore B* apresentada na Figura 5.17. Assuma que as põginas ", •, e – süo seguras, e a põgina y nüo á segura. Antes da execuãüo do anel principal (passos 2 a 5 do algoritmo), um travamento-exclusivo á colocado na põgina raiz, e a põgina á lida e examinada. A seguir a seguinte seq*f*óncia de eventos ocorre:

- Passo 3: Um travamento-exclusivo sobre a põgina • á solicitado;
- Passo 4: Apês receber o travamento-exclusivo a põgina • á lida;
- Passo 5: Desde que a põgina • á segura, o travamento-exclusivo sobre a põgina a e liberado, permitindo o acesso a põgina a para outros processos;
- Passo 3: Um travamento-exclusivo sobre a põgina —á solicitado;
- Passo 4: Apês receber o travamento-exclusivo a põgina —á lida;
- Passo 5: Desde que a põgina —nüo á segura, o travamento-exclusivo sobre a põgina • á mantido;
- Passo 3: Um travamento-exclusivo sobre a põgina – á solicitado;
- Passo 4: Apês receber o travamento-exclusivo, a põgina b á lida;
- Passo 5: Desde que a põgina – á segura, os travamentos-exclusivos sobre as põginas *(* e • podem ser liberados.

A solução apresentada acima requer um protocolo bem simples e ainda assim permite um nível razoável de concorrência. Essa solução pode ser melhorada em relação ao nível de concorrência com a utilização de protocolos mais sofisticados. Por exemplo, o processo modificador pode fazer uma " reserva" em cada página acessada e mais tarde modificar a reserva para travamento-exclusivo caso o processo modificador verifique que as modificações a serem realizadas na estrutura da árvore deverão se propagar até a página com reserva. Essa solução aumenta o nível de concorrência, desde que as páginas com reserva possam ser lidas por outros processos.

Os tipos de travamentos referidos aqui são aplicados ao nível físico do banco de dados. Em um banco de dados cujo acesso aos dados é realizado através de uma árvore B*, a unidade de transferência dos dados da memória secundária para a memória principal é a página. Deste modo, os protocolos de travamento são aplicados neste nível.

A implementação dos travamentos descritos acima pode ser obtida usando **semáforos.** De acordo com Dijkstra (1965), um semáforo é um inteiro

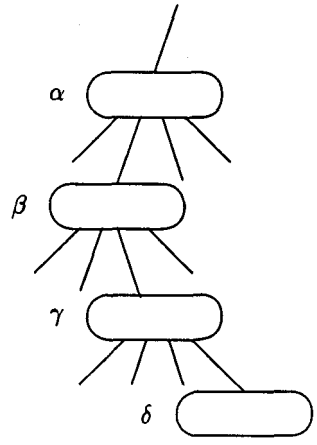


Figura 5.17: Parte de uma árvore B*

não negativo que pode ser modificado somente pelas operações *wait e signal,* assim descritas: *wait (s):* when s > 0 do s := s — *1; e signal (s): s :* = *s + 1.* A operação s := s + *1* é indivisível, isto é, somente um processo consegue realizá-la de cada vez. Por exemplo, se dois processos A e B querem realizar *signal (s)* ao mesmo tempo para s = 3, ao final s = 5. Se a operação s := s+l não é indivisível e as duas operações atribuem o resultado 4 a s, o resultado final pode ser 4 (e não 5). Outra referência sobre semáforos, bem como sua aplicação para sincronizar processos concorrentes, pode ser encontrada em Lister (1975).

Outro importante aspecto a ser considerado em um ambiente de processamento concorrente é o problema de deadlock. *0 deadlock* ocorre quando dois processos estão inserindo um registro cada um em páginas adjacentes que estejam cheias. Neste caso cada um dos processos fica esperando pelo outro eternamente. Lister (1975) mostra que o *deadlock* pode ser evitado através da eliminação de dependências circulares entre processos e recursos. Esta condição pode ser satisfeita através do uso da estrutura em árvore para ordenar todas as solicitações para acessar o banco de dados. Basta que os algoritmos usem as operações de travamento de cima para baixo, isto é, da página raiz para as páginas folha. Bayer e Schkolnick (1977) provaram que as soluções apresentadas são livres de *deadlock*.

5.3.4 Considerações Praticas

A árvore B é simples, de fácil manutenção, eficiente e versátil. A árvore B permite acesso seqüencial eficiente, e o custo para recuperar, inserir, e retirar registros do arquivo é logarítmico. O espaço utilizado pelos dados é, no mínimo, 50% do espaço reservado para o arquivo. Oespaço **utilizado** varia com a aquisição e liberação da área utilizada, à medida em que o arquivo cresce ou diminui de tamanho. As árvores B crescem e diminuem automaticamente, e nunca existe necessidade de uma reorganização completa do banco de dados. O emprego de árvores B em ambientes onde o acesso concorrente ao banco de dados é necessário é viável e relativamente simples de ser implementado.

Um bom exemplo de utilização prática de árvores B* é o método de acesso **a** arquivos da IBM, chamado VSAM (Keehn e Lacy, *1974;* Wagner, 1973). VSAM é um método de acesso a arquivos de aplicação geral que permite tanto o acesso seqüencial eficiente, bem como as operações de inserção, retirada e recuperação em tempo logarítmico. Comparado com a organização indexado seqüencial, o método VSAM oferece as vantagens da **alocação** dinâmica de memória, garantia de utilização de no mínimo 50% da memória reservada ao arquivo, e nenhuma necessidade de reorganização periódica de todo o arquivo. O VSAM é considerado uma evolução do antigo ISAM, o qual utiliza o método indexado seqüencial (vicie Seção 5.2).

Análise

Pelo que foi visto acima, as operações de inserção e retirada de registros em uma árvore B sempre deixam a árvore balanceada. Além do mais, o caminho mais longo em uma árvore B de ordem *m* com *N* registros contém no máximo cerca de $log_{,m+1}N$ páginas. De fato, Bayer e McCreight (1972) provaram que os limites para a altura máxima e altura mínima de uma árvore B de ordem *m* contendo *N* registros são

$$\log_{2m+1}(N+1) \leq altura \leq 1 + \log_{m+1}\left(\frac{N+1}{2}\right)$$

O custo para processar uma operação de recuperação de um registro cresce com o logaritmo base m do tamanho do arquivo. Para se ter uma idéia do significado da fórmula acima, considere a Tabela 5.1. Uma árvore B de ordem 50, representando um índice de um arquivo de um milhão de registros, permite a recuperação de qualquer registro com 4 acessos ao disco, no pior caso. Na realidade o número de acessos no caso médio e ***3.***

A altura esperada de uma árvore B não é conhecida analiticamente, pois ninguém foi capaz de apresentar um resultado analítico. A partir do cálculo analítico do número esperado de páginas para os quatro primeiros níveis

Tamanho da página	Tamanho do arquivo				
	1 . 0 0 0	10.000	100.000	1.000.000	10.000.000
10	3	4	5	6	7
50	2	3	3	4	4
100	2	2	3	3	4
150	2	2	3	3	4

Tabela 5.1: Número de acessos a disco, no pior caso, para tamanhos variados de páginas e arquivos usando árvore B

contados das páginas folha em direção à página raiz 'de uma **árvore 2-3** (ou árvore B de ordem m = 1) obtido por Eisenbarth, Ziviani, Gonnet, Mehlhorn e Wood (1982, p. 159), eles propuseram a seguinte conjetura: a altura esperada de uma árvore 2-3 randômica (vide definição de árvore de pesquisa randômica na Seção 4.3) com N chaves é

$$\overline{h}(N) \approx \log_{7/3}(N+1)$$

Outras medidas de complexidade relevantes para as árvores B randômicas **são:**

1. A utilização de memória é cerca de ln 2 para o algoritmo original proposto por Bayer e McCreight (1972). Isto significa que as páginas têm uma ocupação de aproximadamente 69% da área reservada após N inserções randômicas em uma árvore B inicialmente vazia.

2. No momento da inserção, a operação mais cara é a partição da página quando ela passa a ter mais do que 2m chaves, desde que a operação envolve a criação de uma nova página, o rearranjo das chaves e **a** inserção da chave do meio na página pai localizada no nível acima. Uma medida de complexidade de interesse *é Pr{j* partições}, **que corres**ponde à probabilidade de que j partições ocorram durante a N-ésima inserção randômica. No caso da árvore 2-3

$$Pr\{0 \text{ partições}\} = \frac{4}{7}$$

$$Pr\{1 \text{ ou mais partições}\} = \frac{3}{7}$$

No caso da árvore B de ordem m

$$Pr\{0 \text{ partições}\} = 1 - \frac{1}{(2\ln 2)m} + O(m^{-2})$$

$$Pr\{\text{1 ou mais partições}\} = \frac{1}{(2 \ln 2)m} + O(m^{-2})$$

No caso de uma árvore B de ordem $m = 70$, $Pr\{1$ ou mais partições$\}$ 0, 01. Em outras palavras, em 99% das vezes nada acontece em termos de partições durante uma inserção.

3. Considere o acesso concorrente em árvores B. Bayer e Schkolnick (1977) propuseram a técnica de aplicar um travamento na *página segura mais profunda* (Psmp) no caminho de inserção. De acordo com o que foi mostrado na Seção 5.3.3, uma página é segura se ela contém menos do que 2m chaves. Uma página segura é a mais profunda de um caminho de inserção se não existir outra página segura abaixo dela.

   Desde que o travamento da página impede o acesso de outros processos é interessante saber qual é a probabilidade de que a página segura mais profunda esteja no primeiro nível. Estas medidas estão relacionadas com as do item anterior. No caso da árvore 2-3

$$Pr\{\text{Psmp esteja no 1º nível}\} = \frac{4}{7}$$

$$Pr\{\text{Psmp esteja acima do 1º nível}\} = \frac{3}{7}$$

No caso da árvore B de ordem m

$$Pr\{\text{Psmp esteja no 1º nível}\} = 1 - \frac{1}{(2 \ln 2)m} + O(m^{-2})$$

$$Pr\{\text{Psmp esteja acima do 1º nível}\} = \frac{3}{7} = \frac{1}{(2 \ln 2)m} + O(m^{-2})$$

Novamente, no caso de uma árvore B de ordem m = 70, em 99% das vezes a página segura mais profunda (Psmp) está localizada em uma página folha, o que permite um alto grau de concorrência para processos modificadores. Estes resultados mostram que soluções muito complicadas para permitir o uso de concorrência de operações em árvores B não trazem grandes benefícios porque, na maioria das vezes, o travamento ocorrerá em páginas folha, o que permite alto grau de concorrência mesmo para os protocolos mais simples.

Maiores detalhes sobre os resultados analíticos apresentados acima podem ser obtidos em Eisenbarth *et al.* (1982).

Observações Finais

Existem inúmeras variações sobre o algoritmo original da árvore B. Uma delas é a árvore B*, tratada na Seção 5.3.2.

Outra importante modificação é **a técnica de transbordamento (ou** técnica de **overflow)** proposta por Bayer e McCreight (1972) e Knuth (1973). A idéia é a seguinte: assuma que um registro tenha que ser inserido em uma página cheia que contenha 2m registros. Ao invés de particioná-la, nós olhamos primeiro para a página irmã à direita. Se a página irmã possui menos do que 2m registros, um simples rearranjo de chaves torna a partição desnecessária. Se a página à direita também estiver cheia ou não existir, nós olhamos para a página irmã à esquerda. Se ambas estiverem cheias, então a partição terá que ser realizada. 0 efeito desta modificação é o de produzir uma árvore com melhor utilização de memória e uma altura esperada menor. Esta alteração produz uma utilização de memória de cerca de 83% para uma árvore B randômica.

Qual é a influência de um sistema de paginação no comportamento de uma árvore B? Desde que o número de níveis de uma árvore B é muito pequeno (apenas 3 ou 4) se comparado com o número de molduras de páginas, o sistema de paginação garante que a página raiz esteja sempre presente na memória principal, desde que a política de reposição de páginas adotada seja a política LRU. 0 esquema LRU faz também com que as páginas a serem particionadas em uma inserção estejam automaticamente disponíveis na memória principal.

Finalmente, é importante observar que a escolha do tamanho adequado da ordem m da árvore B é geralmente feita em função das características de cada computador. Por exemplo, em um computador com memória virtual paginada, o tamanho ideal da página da árvore corresponde ao tamanho da página do sistema, e a transferência de dados da memória secundária para a memória principal e vice-versa é realizada pelo sistema operacional. Estes tamanhos variam de 512 bytes até 4096 bytes, em múltiplos de 512 bytes.

Notas Bibliográficas

O material utilizado na Seção 5.1 sobre um modelo de computação para memória secundária veio de Lister (1975). As árvores B foram introduzi-das por Bayer e McCreight (1972). Comer (1979) discute árvores B sob um ponto de vista mais prático. Wirth (1976) apresenta uma implementação dos algoritmos de inserção e de retirada; Gonnet e Baeza-Yates (1991) apresentam uma implementação do algoritmo de inserção. A principal referência utilizada no item concorrência em árvores B veio de Bayer e Schkolnik (1977).

Exercícios

1) a) Construa uma árvore B de ordem m = 1 para as seguintes chaves: 15, 10, 30, 40, 5, 20, 12.

b) Retire a chave 15 e depois a chave 20 da árvore obtida acima.

2) Neste trabalho você deve apresentar uma implementação do conjunto de procedimentos para criação de um ambiente de memória virtual paginada em Pascal, conforme descrito na Secao 5.1.

Ao implementar o sistema você deve ter em mente que o conjunto de procedimentos deverá permitir a o usuário incorporar facilmente um ambiente de memória virtual ao seu programa para poder organizar o fluxo de dados entre a memória primária e a memória secundária. Para tal procure colocar todos os procedimentos e declarações de tipos em um único arquivo chamado SMV.PAS. Este arquivo poderá ser incorporado a qualquer programa escrito em Pascal, o qual deverá aparecer antes das declarações de variáveis do usuário.

O tamanho máximo de cada estrutura de dados utilizada pelo sistema deverá ser definido através de constantes que poderão facilmente ser ajustadas pelos usuários diretamente n o arquivo SMV.PAS, de acordo com suas conveniências.
O que cada aluno deve fornecer:

a) Uma listagem completa do conjunto de procedimentos precedida de documentação pertinente. A descrição de cada procedimento deverá conter pelo menos a sua função e a de seus parâmetros. Dependendo da complexidade de cada procedimento pode ser interessante descrever sucintamente a lógica do módulo obtido (evite descrever o que é óbvio).

b) Uma listagem de um programa de demonstração (DEMO) que mostre claramente ao usuário como utilizar o pacote SMV.PAS. O programa DEMO deve servir também para mostrar toda a flexibilidade e potencial do SMV.PAS.

c) Teste do Sistema
Para testar os vários módulos do sistema de paginação você deve gerar um arquivo em disco contendo algumas páginas (para fins de teste você pode utilizar uma página de tamanho pequeno, digamos 32 bytes). 0 arquivo de teste em disco deverá conter as páginas de uma árvore binária de pesquisa sem balanceamento, conforme mostrado no Programa 5.1.

Para mostrar o pleno funcionamento do sistema é importante criar procedimentos para mostrar o conteúdo de todas as páginas em disco, da fila de Moldura_de_Páginas e da Tabela_de_Páginas. Estes procedimentos devem ser invocados pelo programa de teste nos momentos mais interessantes para se ver o comportamento do sistema (talvez seja interessante realizar uma adaptação do programa DEMO para fins de testar o SMV conforme descrito acima). A impressão de todos estes momentos deve ser fornecida junto com a listagem do programa de teste.

3) O objetivo deste trabalho é projetar e implementar um sistema de programas para recuperar, inserir e retirar registros de um arquivo que pode conter até alguns poucos milhões de registros. A aplicação que utiliza o arquivo é bastante dinâmica, existindo um grande número de consultas e atualizações (inserções, retiradas e alterações de registros). Além do mais a aplicação requer periodicamente a recuperação total ou parcial dos registros na ordem lexicográfica das chaves. Es-tas características sugerem fortemente a utilização de **árvore B** como estrutura de dados. 0 que fazer:

   a) Para implementar os algoritmos de pesquisa, inserção e retirada em uma árvore B de ordem m utilize o pacote SMV.PAS proposto em exercício acima, para criar um ambiente de memória virtual e resolver o problema de fluxo de dados entre as memórias primária e secundária.

   b) Para testar o seu sistema de programas para uma árvore B de ordem m = 2, utilize a seguinte seqüência de chaves: Inserção:
   20; 10 40 50 30; 55 3 11 4 28; 36; 33 52 17 25 13; 45 9 43 8 48;
   A cada ponto-e-vírgula você deverá imprimir a árvore. Retirada:
   45 30 28; 50 8; 10 4 20 40 55; 17 33 11 36; 3 11 52; A cada ponto-e-vírgula você deverá imprimir a árvore.

   c) Para termos uma idéia da eficiência do método de acesso construído faça a medida do tempo de execução para:

      a) Construir árvores B de tamanhos 1.000, 10.000 e 50.000 chaves geradas randomicamente através de um gerador de números aleatórios. A medida de tempo deve ser tomada de forma independente para cada uma das três árvores. A ordem m das árvores deve ser tal que o tamanho da página seja igual ou menor do que 512 bytes.

b) Para a maior das três árvores que você conseguiu construir gere aleatoriamente um conjunto de 200 chaves e realize uma pesquisa na árvore para cada chave. Caso haja chaves que não estejam presentes no arquivo informe quantas pesquisas foram com sucesso e quantas foram sem sucesso. Com esta medida podemos ter uma idéia do tempo aproximado para pesquisar um registro no arquivo.

Atenção: procure interpretar os resultados obtidos. Por exemplo, você deve informar qual foi o número de Molduras_de_Páginas utilizadas para os experimentos acima (com 256 Kbytes de memória real disponível é possível manter cerca de 120 molduras na memória principal). Quaisquer outras observações relevantes devem ser relatadas.

Observações:

a) A pesquisa da chave de um registro dentro de uma página da árvore B pode ser feita através de uma pesquisa seqüencial.

b) A decisão sobre a documentação a ser apresentada fica por conta do aluno.

4) Modifique a implementação do procedimento original apresentado no Programa 5.5 para que a pesquisa da chave dentro de uma página da árvore B seja realizada através de um pesquisa binária.

Apêndice A

Programas C do Capítulo 1

```
int Max(Vetor A)
{
  int i, Temp;

  Temp = A[0];
  for (i = 1; i < n; i++) {
    if (Temp < A[i]) Temp = A[i];
  }
  return Temp;
} /* Max */
```

Programa A.1: Função para obter o maior elemento de um vetor

```
void MaxMin1(Vetor A, int *Max, int *Min)
{
  int i;

  *Max = A[0];
  *Min = A[0];
  for (i = 1; i < n; i++) {
    if (A[i] > *Max) *Max = A[i];
    if (A[i] < *Min) *Min = A[i];
  }
} /* MaxMin1 */
```

Programa A.2: Implementação direta para obter o máximo e o mínimo

```
void MaxMin2(Vetor A, int *Max, int *Min)
{
 int i;

 *Max = A[0];
 *Min = A[0];
 for (i = 1; i < n; i++) {
  if (A[i] > *Max)
    *Max = A[i];
  else if (A[i] < *Min)
    *Min = A[i];
 }
} /* MaxMin2 */
```

Programa A.3: Implementação melhorada para obter o máximo e o mínimo

```
void MaxMin3(Vetor A, int *Max, int *Min)
{
 int i, FimDoAnel;

 if ((n & 1) > 0) {
  A[n] = A[n - 1];
  FimDoAnel = n;
 } else
  FimDoAnel = n - 1;
 if (A[0] > A[1]) {
  *Max = A[0];
  *Min = A[1];
 } else {
  *Max = A[1];
  *Min = A[0];
 }
 i = 3;
 while (i <= FimDoAnel) {
  if (A[i - 1] > A[i]) {
   if (A[i - 1] > *Max)
        *Max = A[i - 1];
   if (A[i] < *Min)
        *Min = A[i];
  } else {
   if (A[i - 1] < *Min)
        *Min = A[i - 1];
   if (A[i] > *Max)
        *Max = A[i];
  }
  i += 2;
 }
```

```
} /* MaxMin3 */
```

Programa A.4: Outra implementação para obter o máximo e o mínimo

```
void Ordena(Vetor A)
{
  /* ordena o vetor A em ordem ascendente */
  int i, j, min, x;

  for (i = 1; i < n; i++) {
    min = i;
    for (j = i + 1; j <= n; j++) {
      if (A[j - 1] < A[min - 1])
          min = j;
    }
    /* troca A[min] e A[i] */
    x = A[min - 1];
    A[min - 1] = A[i - 1];
    A[i - 1] = x;
  }
} /* Ordena */
```

Programa A.5: Programa para ordenar

```
  Pesquisa(n);
  if (n <= 1)
    /* inspecione elemento e termine */
  else {
    /* para cada um dos n elementos inspecione elemento */
    Pesquisa(n / 3);
  }
```

Programa A.6: Algoritmo recursivo

```
void MaxMin4(int Linf, int Lsup, int *Max, int *Min)
{
  int Max1, Max2, Min1, Min2, Meio;

  if (Lsup - Linf <= 1) {
    if (A[Linf - 1] < A[Lsup - 1]) {
      *Max = A[Lsup - 1];
      *Min = A[Linf - 1];
    } else {
      *Max = A[Linf - 1];
      *Min = A[Lsup - 1];
```

```
    }
    return;
  }
  Meio = (Linf + Lsup) / 2;
  MaxMin4(Linf, Meio, &Max1, &Min1);
  MaxMin4(Meio + 1, Lsup, &Max2, &Min2);
  if (Max1 > Max2)
    *Max = Max1;
  else
    *Max = Max2;
  if (Min1 < Min2)
    *Min = Min1;
  else
    *Min = Min2;
} /* MaxMin4 */
```

Programa A.7: Versão recursiva para obter o máximo e o mínimo

```
#include <stdio.h>

/* copia o arquivo Velho no arquivo Novo */

#define n           15

typedef char alfa[n];

typedef struct {
  int dia;
  int mês;
} data;

typedef struct {
  alfa Sobrenome, PrimeiroNome;
  data Aniversário;
  enum {
    mas, fem
  } Sexo;
} Pessoa;

void main()
{
  FILE *Velho, *Novo;
  Pessoa Registro;

  Novo = fopen("novo.arq","w");
  Velho = fopen("velho.arq","r");
```

```
  while (fread(&Registro, sizeof(Pessoa), 1, Velho) > 0) {
    fwrite(&Registro, sizeof(Pessoa), 1, Novo);
  }

  fclose(Velho);
  fclose(Novo);
} /* Copia */
```

Programa A.8: Programa para copiar arquivo

Apêndice B

Programas C do Capítulo 2

```
#define InícioArranjo   1
#define MaxTam          1000

typedef int Apontador;

typedef struct {
  TipoChave Chave;
  /* — outros componentes — */
} TipoItem;

typedef struct {
  TipoItem Item[MaxTam];
  Apontador Primeiro, Último;
} TipoLista;
```

Programa B.1: Estrutura da lista usando arranjo

```
void FLVazia(TipoLista *Lista)
{
  Lista->Primeiro = InícioArranjo;
  Lista->Último = Lista->Primeiro;
} /* FLVazia */

int Vazia(TipoLista Lista)
{
  return (Lista.Primeiro == Lista.Último);
} /* Vazia */

void Insere(TipoItem x, TipoLista *Lista)
{
```

```
  if (Lista->Último > MaxTam)
    printf("Lista está cheia\n");
  else {
    Lista->Item[Lista->Último - 1] = x;
    Lista->Último++;
  }
} /* Insere */

void Retira(Apontador p, TipoLista *Lista, TipoItem *Item)
{
  int Aux;

  if (Vazia(*Lista) || p >= Lista->Último) {
    printf(" Erro:  Posição  não existe\n");
    return;
  }
  *Item = Lista->Item[p - 1];
  Lista->Último--;
  for (Aux = p; Aux < Lista->Último; Aux++)
    Lista->Item[Aux - 1] = Lista->Item[Aux];
} /* Retira */

void Imprime(TipoLista Lista)
{
  int Aux;

  for (Aux = Lista.Primeiro - 1; Aux <= (Lista.Último - 2); Aux++)
    printf("%12d\n", Lista.Item[Aux].Chave);
} /* Imprime */
```

Programa B.2: Operações sobre listas usando posições contíguas de memória

```
typedef struct Célula_str *Apontador;

typedef struct {
  int Chave;
  /* outros componentes */
} TipoItem;

typedef struct Célula_str {
  TipoItem Item;
  Apontador Prox;
} Célula;

typedef struct {
```

```
  Apontador Primeiro, Último;
} TipoLista;
```

Programa B.3: Estrutura da lista usando apontadores

```
void FLVazia(TipoLista *Lista)
{
  Lista->Primeiro = (Apontador) malloc(sizeof(Célula));
  Lista->Último = Lista->Primeiro;
  Lista->Primeiro->Prox = NULL;
} /* FLVazia */

int Vazia(TipoLista Lista)
{
  return (Lista.Primeiro == Lista.Último);
} /* Vazia */

void Insere(TipoItem x, TipoLista *Lista)
{
  Lista->Último->Prox = (Apontador) malloc(sizeof(Célula));
  Lista->Último = Lista->Último->Prox;
  Lista->Último->Item = x;
  Lista->Último->Prox = NULL;
} /* Insere */

void Retira(Apontador p, TipoLista *Lista, TipoItem *Item)
{
  /* Obs.: o item a ser retirado é o seguinte ao apontado por p */
  Apontador q;

  if (Vazia(*Lista) || p == NULL || p->Prox == NULL) {
    printf(" Erro:  Lista vazia ou posição não existe\n");
    return;
  }
  q = p->Prox;
  *Item = q->Item;
  p->Prox = q->Prox;
  if (p->Prox == NULL)
    Lista->Último = p;
  free(q);
} /* Retira */

void Imprime(TipoLista Lista)
{
  Apontador Aux;
```

```
  Aux = Lista.Primeiro->Prox;
  while (Aux != NULL) {
   printf("%12d\n", Aux->Item.Chave);
   Aux = Aux->Prox;
  }
} /* Imprime */
```

Programa B.4: Operações sobre listas usando apontadores

```
int Chave;     /* variando de 1 a 999 */
int NotaFinal; /* variando de 0 a 10 */
int Opcao[3];  /* variando de 1 a 7 */
```

Programa B.5: Campos do registro de um candidato

```
/* programa Vestibular */
void main()
{
 int Nota;

 ordena os registros pelo campo NotaFinal;
 for (Nota = 10; Nota >= 0; Nota--)
 {
  while houver registro com mesma nota
  {
   if existe vaga em um dos cursos de opção do candidato {
    insere registro no conjunto de aprovados }
   else {
    insere registro no conjunto de reprovados; }
  }
 }
 imprime aprovados por curso;
 imprime reprovados;

}
```

Programa B.6: Primeiro refinamento do programa Vestibular

```
/* programa Vestibular */
void main()
{
 int Nota;
 TipoChave Chave;
```

```
    lê número de vagas para cada curso;
    inicializa listas de classificação, listas de aprovados e lista de reprovados;
    lê registro; /* — vide formato no programa B.5 — */
    while (Chave != 0)
    {
        insere registro em uma das listas de classificação, conforme nota final;
        lê registro
    }
    for (Nota = 10; Nota >= 0; Nota--)
    {
      while houver próximo registro com mesma NotaFinal
      {
        retira registro da lista;
        if existe vaga em um dos cursos de opção do candidato {
           insere registro na lista de aprovados
           decrementa o número de vagas para aquele curso; }
        else {
           insere registro na lista de reprovados; }
        obtém próximo registro;
      }
    }
    imprime aprovados por curso;
    imprime reprovados;

}
```

Programa B.7: Segundo refinamento do programa Vestibular

```
#define NOpções      9
#define NCursos      7

typedef int TipoChave;

typedef struct {
  TipoChave Chave;
  int NotaFinal;
  int Opção[NOpções];
} TipoItem;

typedef struct Célula_str *Apontador;

typedef struct Célula_str {
  TipoItem Item;
  Apontador Prox;
} Célula;

typedef struct {
```

```
  Apontador Primeiro, Último;
} TipoLista;
```

Programa B.8: Estrutura da lista

```
/* — Entram aqui os tipos do programa B.8 — */
/* — Entram aqui os operadores apresentados do programa B.4 — */
void LêRegistro(TipoItem *Registro)
{
  /* — os valores lidos devem estar separados por brancos — */
  int i;

  scanf("%d %d", &Registro->Chave, &Registro->NotaFinal);
  for (i = 0; i < NOpções; i++) {
    scanf("%d", &Registro->Opção[i]);
  }
} /* LêRegistro */

void main()
{
  TipoItem Registro;
  TipoLista Classificação[11], Aprovados[NCursos], Reprovados;
  int Vagas[NCursos], Passou, i, Nota;

  for (i = 1; i <= NCursos; i++)
    scanf("%d", &Vagas[i - 1]);
  for (i = 0; i <= 10; i++)
    FLVazia(&Classificação[i]);
  for (i = 1; i <= NCursos; i++)
    FLVazia(&Aprovados[i - 1]);
  FLVazia(&Reprovados);
  LêRegistro(&Registro);
  while (Registro.Chave != 0) {
    Insere(Registro, &Classificação[Registro.NotaFinal]);
    LêRegistro(&Registro);
  }
  for (Nota = 10; Nota >= 0; Nota--) {
    while (!Vazia(Classificação[Nota])) {
      Retira(Classificação[Nota].Primeiro, &Classificação[Nota], &Registro);
      i = 1;
      Passou = 0;
      while (i <= (NOpções & (!Passou))) {
          if (Vagas[Registro.Opção[i - 1] - 1] > 0) {
            Insere(Registro, &Aprovados[Registro.Opção[i - 1] - 1]);
            Vagas[Registro.Opção[i - 1] - 1]--;
            Passou = 1;
          }
```

```
            i++;
        }
        if (!Passou)
            Insere(Registro, &Reprovados);
    }
  }
  for (i = 1; i <= NCursos; i++) {
    printf(" Relação dos aprovados no Curso%2d\n", i);
    Imprime(Aprovados[i - 1]);
  }
  printf(" Relação dos reprovados\n");
  Imprime(Reprovados);
} /* Vestibular */
```

Programa B.9: Refinamento final do programa Vestibular

```
#define MaxTam          1000

typedef int Apontador;

typedef struct {
  TipoChave Chave;
  /* — outros componentes — */
} TipoItem;

typedef struct {
  TipoItem Item[MaxTam];
  Apontador Topo;
} TipoPilha;
```

Programa B.10: Estrutura da pilha usando arranjo

```
void FPVazia(TipoPilha *Pilha)
{
  Pilha->Topo = 0;
} /* FPVazia */

int Vazia(TipoPilha Pilha)
{
  return (Pilha.Topo == 0);
} /* Vazia */

void Empilha(TipoItem x, TipoPilha *Pilha)
{
  if (Pilha->Topo == MaxTam)
    printf(" Erro:  pilha está cheia\n");
```

```
  else {
    Pilha->Topo++;
    Pilha->Item[Pilha->Topo - 1] = x;
  }
} /* Empilha */

void Desempilha(TipoPilha *Pilha, TipoItem *Item)
{
  if (Vazia(*Pilha))
    printf(" Erro:  pilha está vazia\n");
  else {
    *Item = Pilha->Item[Pilha->Topo - 1];
    Pilha->Topo--;
  }
} /* Desempilha */

int Tamanho(TipoPilha Pilha)
{
  return (Pilha.Topo);
} /* Tamanho */
```

Programa B.11: Operações sobre pilhas usando arranjos

```
typedef struct Célula_str *Apontador;

typedef struct {
  int Chave;
  /* — outros componentes — */
} TipoItem;

typedef struct Célula_str {
  TipoItem Item;
  Apontador Prox;
} Célula;

typedef struct {
  Apontador Fundo, Topo;
  int Tamanho;
} TipoPilha;
```

Programa B.12: Estrutura da pilha usando apontadores

```
void FPVazia(TipoPilha *Pilha)
{
  Pilha->Topo = (Apontador) malloc(sizeof(Célula));
```

```
  Pilha->Fundo = Pilha->Topo;
  Pilha->Topo->Prox = NULL;
  Pilha->Tamanho = 0;
} /* FPVazia */

int Vazia(TipoPilha Pilha)
{
  return (Pilha.Topo == Pilha.Fundo);
} /* Vazia */

void Empilha(TipoItem x, TipoPilha *Pilha)
{
  Apontador Aux;

  Aux = (Apontador) malloc(sizeof(Célula));
  Pilha->Topo->Item = x;
  Aux->Prox = Pilha->Topo;
  Pilha->Topo = Aux;
  Pilha->Tamanho++;
} /* Empilha */

void Desempilha(TipoPilha *Pilha, TipoItem *Item)
{
  Apntador q;

  if (Vazia(*Pilha)) {
    printf(" Erro:  lista vazia\n");
    return;
  }
  q = Pilha->Topo;
  Pilha->Topo = q->Prox;
  *Item = q->Prox->Item;
  free(q);
  Pilha->Tamanho--;
} /* Desempilha */

int Tamanho(TipoPilha Pilha)
{
  return (Pilha.Tamanho);
} /* Tamanho */
```

Programa B.13: Operações sobre pilhas usando apontadores

```
#define MaxTam          70
#define CancelaCarater  '#'
#define CancelaLinha    '\'
#define SaltaLinha      '@'
```

```
#define MarcaEof
typedef char TipoChave;

/* — Entram aqui os tipos do programa B.10 — */
/* — Entram aqui os operadores do programa B.11 — */
/* — Entra aqui o procedimento Imprime do programa B.15 — */
void main()
{
  TipoPilha Pilha;
  TipoItem x;

  FPVazia(&Pilha);
  x.Chave = getchar();
  if (x.Chave == '\n') x.Chave =
  while (x.Chave != MarcaEof) {
    if (x.Chave == CancelaCarater) {
      if (!Vazia(Pilha))
            Desempilha(&Pilha, &x);
    } else if (x.Chave == CancelaLinha)
      FPVazia(&Pilha);
    else if (x.Chave == SaltaLinha)
      Imprime(&Pilha);
    else {
      if (Tamanho(Pilha) == MaxTam)
            Imprime(&Pilha);
      Empilha(x, &Pilha);
    }
    x.Chave = getchar();
    if (x.Chave == '\n')
      x.Chave = ' ';
  }
  if (!Vazia(Pilha))
    Imprime(&Pilha);
} /* ET */
```

Programa B.14: Implementação do ET

```
void Imprime(TipoPilha *Pilha)
{
  TipoPilha Pilhaux;
  TipoItem x;

  FPVazia(&Pilhaux);
  while (!Vazia(*Pilha)) {
    Desempilha(Pilha, &x);
    Empilha(x, &Pilhaux);
  }
```

```
  while (!Vazia(Pilhaux)) {
    Desempilha(&Pilhaux, &x);
    putchar(x.Chave);
  }
  putchar('\n');
} /* Imprime */
```

Programa B.15: Procedimento Imprime utilizado no programa ET

```
#define MaxTam        1000

typedef int Apontador;

typedef struct {
  TipoChave Chave;
/* — outros componentes — */
} TipoItem;

typedef struct {
  TipoItem Item[MaxTam];
  Apontador Frente, Trás;
} TipoFila;
```

Programa B.16: Estrutura da fila usando arranjo

```
void FFVazia(TipoFila *Fila)
{
  Fila->Frente = 1;
  Fila->Trás = Fila->Frente;

int Vazia(TipoFila Fila)
{
  return (Fila.Frente == Fila.Trás);
} /* Vazia */

void Enfileira(TipoItem x, TipoFila *Fila)
{
  if (Fila->Trás % MaxTam + 1 == Fila->Frente)
    printf(" Erro:  fila está cheia\n");
  else {
    Fila->Item[Fila->Trás - 1] = x;
    Fila->Trás = Fila->Trás % MaxTam + 1;
  }
} /* Enfileira */

void Desenfileira(TipoFila *Fila, TipoItem *Item)
```

```
{
  if (Vazia(*Fila))
    printf(" Erro:  fila está vazia\n");
  else {
    *Item = Fila->Item[Fila->Frente - 1];
    Fila->Frente = Fila->Frente % MaxTam + 1;
  }
} /* Desenfileira */
```

Programa B.17: Operações sobre filas usando posições contíguas de memória

```
typedef struct Célula_str *Apontador;

typedef struct TipoItem {
  TipoChave Chave;
/* — outros componentes — */
} TipoItem;

typedef struct Célula_str {
  TipoItem Item;
  Apontador Prox;
} Célula;

typedef struct TipoFila {
  Apontador Frente, Trás;
} TipoFila;
```

Programa B.18: Estrutura da fila usando apontadores

```
void FFVazia(TipoFila *Fila)
{
  Fila->Frente = (Apontador) malloc(sizeof(Célula));
  Fila->Trás = Fila->Frente;
  Fila->Frente->Prox = NULL;
} /* FFVazia */

int Vazia(TipoFila Fila)
{
  return (Fila.Frente == Fila.Trás);
} /* Vazia */

void Enfileira(TipoItem x, TipoFila *Fila)
{
  Fila->Trás->Prox = (Apontador) malloc(sizeof(Célula));
  Fila->Trás = Fila->Trás->Prox;
  Fila->Trás->Item = x;
```

```
  Fila->Trás->Prox = NULL;
} /* Enfileira */

void Desenfileira(TipoFila *Fila, TipoItem *Item)
{
  Apontador q;

  if (Vazia(*Fila)) {
    printf(" Erro:  fila está vazia\n");
    return;
  }
  q = Fila->Frente;
  Fila->Frente = Fila->Frente->Prox;
  *Item = Fila->Frente->Item;
  free(q);
} /* Desenfileira */
```

Programa B.19: Operações sobre filas usando apontadores

Apêndice C

Programas C do Capítulo 3

```
typedef struct {
  ChaveTipo Chave;
  /* — outros componentes — */
} Item;
```

Programa C.1: Estrutura de um item do arquivo

```
typedef Item Vetor[n + 1];
Vetor A;
```

Programa C.2: Tipos utilizados na implementação dos algoritmos

```
void Seleção(Vetor A)
{
  Índice i, j, Min;
  Item x;

  for (i = 1; i < n; i++) {
    Min = i;
    for (j = i + 1; j <= n; j++) {
      if (A[j].Chave < A[Min].Chave)
          Min = j;
    }
    x = A[Min]; A[Min] = A[i]; A[i] = x;
  }
} /* Seleção */
```

Programa C.3: Ordenação por seleção

```
void Inserção(Vetor A)
{
  Índice i, j;
  Item x;
  for (i = 2; i <= n; i++) {
    x = A[i];
    j = i - 1;
    A[0] = x;   /* sentinela */
    while (x.Chave < A[j].Chave) {
      A[j + 1] = A[j];
      j--;
    }
    A[j + 1] = x;
  }
} /* Inserção */
```

Programa C.4: Ordenação por inserção

```
void Shellsort(Vetor A)
{
  int i, j, h;
  Item x;
  h = 1;
  do {
    h = h * 3 + 1;
  } while (h <= n);
  do {
    h /= 3;
    for (i = h + 1; i <= n; i++) {
      x = A[i];
      j = i;
      while (A[j - h].Chave > x.Chave) {
            A[j] = A[j - h];
            j -= h;
            if (j <= h)
              goto LABEL;
      }
LABEL:
      A[j] = x;
    }
  } while (h != 1);   /* Shellsort */
}
```

Programa C.5: Algoritmo Shellsort

```
void Partição(Vetor A, Índice Esq, Índice Dir, Índice *i, Índice *j)
{
  Item x, w;

  *i = Esq;
  *j = Dir;
  x = A[(*i + *j) / 2];  /* obtém o pivô x */
  do {
    while (A[*i].Chave < x.Chave)
      (*i)++;
    while (A[*j].Chave > x.Chave)
      (*j)--;
    if (*i <= *j) {
      w = A[*i];
      A[*i] = A[*j];
      A[*j] = w;
      (*i)++;
      (*j)--;
    }
  } while (*i <= *j);  /* Partição */
}
```

Programa C.6: Função Partição

```
  /* — Entra aqui a função Partição (Programa C.6) — */
void Ordena(Vetor A, Índice Esq, Índice Dir)
{
  Índice i, j;

  Partição(A, Esq, Dir, &i, &j);
  if (Esq < j)
    Ordena(A, Esq, j);
  if (i < Dir)
    Ordena(A, i, Dir);
} /* Ordena */

void Quicksort(Vetor A)
{
  Ordena(A, 1, n);
} /* Quicksort */
```

Programa C.7: Função Quicksort

```
void Refaz(Índice Esq, Índice Dir, Vetor A)
{
```

```
  Índice i;
  int j;
  Item x;

  i = Esq;
  j = i * 2;
  x = A[i];
  while (j <= Dir) {
    if (j < Dir) {
      if (A[j].Chave < A[j + 1].Chave)
          j++;
    }
    if (x.Chave >= A[j].Chave)
      goto LABEL;
    A[i] = A[j];
    i = j;
    j = i * 2;
  }
LABEL:
  A[i] = x;
} /* Refaz */

void Constrói(Vetor A)
{
  Índice Esq;

  Esq = n / 2 + 1;
  while (Esq > 1) {
    Esq--;
    Refaz(Esq, n, A);
  }
} /* Constrói */
```

Programa C.8: Função para construir o heap

```
  /* Entra aqui a função Refaz do programa C.8 */
void Heapsort(Vetor A)
{
  Índice Esq, Dir;
  Item x;

  /* constrói o heap */
  Esq = n / 2 + 1;
  Dir = n;
  while (Esq > 1) {
    Esq--;
    Refaz(Esq, Dir, A);
```

```
  }
  /* ordena o vetor */
  while (Dir > 1) {
    x = A[1];
    A[1] = A[Dir];
    A[Dir] = x;
    Dir--;
    Refaz(Esq, Dir, A);
  }
} /* Heapsort */
```

Programa C.9: Função Heapsort

Apêndice D

Programas C do Capítulo 4

```
#define Maxn          1000

typedef struct {
  TipoChave Chave;
  /*outros componentes*/
} Registro;

typedef int Índice;

typedef struct {
  Registro Item[Maxn + 1];
  Índice n;
} Tabela;
```

Programa D.1: Estrutura do tipo dicionário implementado como arranjo

```
void Inicializa(Tabela *T)
{
  T->n = 0;
} /* Inicializa */


Índice Pesquisa(TipoChave x, Tabela *T)
{
  int i;

  T->Item[0].Chave = x;
  i = T->n + 1;
  do {
    i--;
```

```
  } while (T->Item[i].Chave != x);
  return i;
} /* Pesquisa */


void Insere(Registro reg, Tabela *T)
{
  if (T->n == Maxn)
    printf("Erro:  tabela cheia\n");
  else {
    T->n++;
    T->Item[T->n] = reg;
  }
} /* Insere */
```

Programa D.2: Implementação das operações usando arranjo

```
Índice Binária(TipoChave x, Tabela *T)
{
  Índice i, Esq, Dir;

  Esq = 1;
  Dir = T->n;
  do {
    i = (Esq + Dir) / 2;
    if (x > T->Item[i].Chave)
      Esq = i + 1;
    else
      Dir = i - 1;
  } while (x != T->Item[i].Chave && Esq <= Dir);
  if (x == T->Item[i].Chave)
    return i;
  else
    return 0;
} /* Binária */
```

Programa D.3: Pesquisa binária

```
typedef struct {
  TipoChave Chave;
  /* outros componentes */
} Registro;

typedef struct Nodo_str *Apontador;

typedef struct Nodo_str {
```

```
  Registro Reg;
  Apontador Esq, Dir;
} Nodo;

typedef Apontador TipoDicionário;
```

Programa D.4: Estrutura do dicionário

```
void Pesquisa(Registro *x, Apontador *p)
{
  if (*p == NULL) {
    printf("Erro :  Registro não está presente na árvore\n");
    return;
  }
  if (x->Chave < (*p)->Reg.Chave) {
    Pesquisa(x, &(*p)->Esq);
    return;
  }
  if (x->Chave > (*p)->Reg.Chave)
    Pesquisa(x, &(*p)->Dir);
  else
    *x = (*p)->Reg;
} /* Pesquisa */
```

Programa D.5: Função para pesquisar na árvore

```
void Insere(Registro x, Apontador *p)
{
  if (*p == NULL) {
    *p = (Apontador) malloc(sizeof(Nodo));
    (*p)->Reg = x;
    (*p)->Esq = NULL;
    (*p)->Dir = NULL;
    return;
  }
  if (x.Chave < (*p)->Reg.Chave) {
    Insere(x, &(*p)->Esq);
    return;
  }
  if (x.Chave > (*p)->Reg.Chave)
    Insere(x, &(*p)->Dir);
  else
    printf(" Erro :  Registro já  existe na árvore\n");
```

```
} /* Insere */
```

Programa D.6: Função para inserir na árvore

```
void Inicializa(TipoDicionário *Dicionário)
{
  *Dicionário = NULL;
} /* Inicializa */
```

Programa D.7: Função para inicializar

```
typedef int TipoChave;

/* — Entra aqui a definição dos tipos do Programa D.4 — */
/* — Entram aqui os Programas D.7 e D.6 — */

void main()
{
TipoDicionário Dicionário;
Registro x;

  Inicializa(&Dicionário);
  scanf("%d", &x.Chave);
  while (x.Chave > 0) {
    Insere(x, &Dicionário);
    scanf("%d", &x.Chave);
  }
}
```

Programa D.8: Programa para criar a árvore

```
void Antecessor(Apontador q, Apontador *r)
{
  if ((*r)->Dir != NULL) {
    Antecessor(q, &(*r)->Dir);
    return;
  }
  q->Reg = (*r)->Reg;
  q = *r;
  *r = (*r)->Esq;
  free(q);
} /* Antecessor */

void Retira(Registro x, Apontador *p)
{
```

```
  Apontador Aux;

  if (*p == NULL) {
    printf("Erro :  Registro não está na árvore\n");
    return;
  }
  if (x.Chave < (*p)->Reg.Chave) {
    Retira(x, &(*p)->Esq);
    return;
  }
  if (x.Chave > (*p)->Reg.Chave) {
    Retira(x, &(*p)->Dir);
    return;
  }
  if ((*p)->Dir == NULL) {
    Aux = *p;
    *p = (*p)->Esq;
    free(Aux);
    return;
  }
  if ((*p)->Esq != NULL) {
    Antecessor(*p, &(*p)->Esq);
    return;
  }
  Aux = *p;
  *p = (*p)->Dir;
  free(Aux);
} /* Retira */
```

Programa D.9: Funções para retirar x da árvore

```
void Central(Apontador p)
{
  if (p != NULL) {
    Central(p->Esq);
    printf("%12d\n", p->Reg.Chave);
    Central(p->Dir);
  }
} /* Central */
```

Programa D.10: Caminhamento central

```
typedef struct {
  TipoChave Chave;
  /* outros componentes */
} Registro;
```

```
typedef enum {
  Vertical, Horizontal
} Inclinação;

typedef struct Nodo_est {
  Registro Reg;
  struct Nodo_est *Esq, *Dir;
  Inclinação BitE, BitD;
} Nodo;
```

Programa D.11: Estrutura do dicionário para árvores SBB

```
void EE(Nodo **Ap)
{
  Nodo *Ap1;

  Ap1 = (*Ap)->Esq;
  (*Ap)->Esq = Ap1->Dir;
  Ap1->Dir = *Ap;
  Ap1->BitE = Vertical;
  (*Ap)->BitE = Vertical;
  *Ap = Ap1;
} /* EE */


void ED(Nodo **Ap)
{
  Nodo *Ap1, *Ap2;

  Ap1 = (*Ap)->Esq;
  Ap2 = Ap1->Dir;
  Ap1->BitD = Vertical;
  (*Ap)->BitE = Vertical;
  Ap1->Dir = Ap2->Esq;
  Ap2->Esq = Ap1;
  (*Ap)->Esq = Ap2->Dir;
  Ap2->Dir = *Ap;
  *Ap = Ap2;
} /* ED */


void DD(Nodo **Ap)
{
  Nodo *Ap1;

  Ap1 = (*Ap)->Dir;
```

```
  (*Ap)->Dir = Ap1->Esq;
  Ap1->Esq = *Ap;
  Ap1->BitD = Vertical;
  (*Ap)->BitD = Vertical;
  *Ap = Ap1;
} /* DD */


void DE(Nodo **Ap)
{
  Nodo *Ap1, *Ap2;

  Ap1 = (*Ap)->Dir;
  Ap2 = Ap1->Esq;
  Ap1->BitE = Vertical;
  (*Ap)->BitD = Vertical;
  Ap1->Esq = Ap2->Dir;
  Ap2->Dir = Ap1;
  (*Ap)->Dir = Ap2->Esq;
  Ap2->Esq = *Ap;
  *Ap = Ap2;
} /* DE */
```

Programa D.12: Procedimentos auxiliares para árvores SBB

```
void IInsere(Registro x, Nodo **Ap, Inclinação *IAp, int *Fim)
{
  if (*Ap == NULL) {
    *Ap = (Nodo *)malloc(sizeof(Nodo));
    *IAp = Horizontal;
    (*Ap)->Reg = x;
    (*Ap)->BitE = Vertical;
    (*Ap)->BitD = Vertical;
    (*Ap)->Esq = NULL;
    (*Ap)->Dir = NULL;
    *Fim = 0;
    return;
  }
  if (x.Chave < (*Ap)->Reg.Chave) {
    IInsere(x, &(*Ap)->Esq, &(*Ap)->BitE, Fim);
    if (*Fim)
      return;
    if ((*Ap)->BitE != Horizontal) {
      *Fim = 1;
      return;
    }
    if ((*Ap)->Esq->BitE == Horizontal) {
```

```
      EE(Ap);
      *IAp = Horizontal;
      return;
    }
    if ((*Ap)->Esq->BitD == Horizontal) {
      ED(Ap);
      *IAp = Horizontal;
    }
    return;
  }
  if (x.Chave <= (*Ap)->Reg.Chave) {
    printf("Erro:  Chave já está na árvore\n");
    *Fim = 1;
    return;
  }
  IInsere(x, &(*Ap)->Dir, &(*Ap)->BitD, Fim);
  if (*Fim)
    return;
  if ((*Ap)->BitD != Horizontal) {
    *Fim = 1;
    return;
  }
  if ((*Ap)->Dir->BitD == Horizontal) {
    DD(Ap);
    *IAp = Horizontal;
    return;
  }
  if ((*Ap)->Dir->BitE == Horizontal) {
    DE(Ap);
    *IAp = Horizontal;
  }
} /* IInsere */

void Insere(Registro x, Nodo **Ap)
{
  int Fim;
  Inclinação IAp;

  IInsere(x, Ap, &IAp, &Fim);
} /* Insere */
```

Programa D.13: Procedimento para inserir na árvore SBB

```
void Inicializa(Nodo **Dicionário)
{
  *Dicionário = NULL;
```

```
} /* Inicializa */
```

Programa D.14: Procedimento para inicializa a árvore SBB

```
void EsqCurto(Nodo **Ap, int *Fim)
{ /*Folha esquerda retirada => árvore curta na altura esquerda*/
  Nodo *Ap1;

  if ((*Ap)->BitE == Horizontal) {
    (*Ap)->BitE = Vertical;
    *Fim = 1;
    return;
  }
  if ((*Ap)->BitD == Horizontal) {
    Ap1 = (*Ap)->Dir;
    (*Ap)->Dir = Ap1->Esq;
    Ap1->Esq = *Ap;
    *Ap = Ap1;
    if ((*Ap)->Esq->Dir->BitE == Horizontal) {
      DE(&(*Ap)->Esq);
      (*Ap)->BitE = Vertical;
    } else if ((*Ap)->Esq->Dir->BitD == Horizontal) {
      DD(&(*Ap)->Esq);
      (*Ap)->BitE = Vertical;
    }
    *Fim = 1;
    return;
  }
  (*Ap)->BitD = Horizontal;
  if ((*Ap)->Dir->BitE == Horizontal) {
    DE(Ap);
    *Fim = 1;
    return;
  }
  if ((*Ap)->Dir->BitD == Horizontal) {
    DD(Ap);
    *Fim = 1;
  }
} /* EsqCurto */

void DirCurto(Nodo **Ap, int *Fim)
{ /*Folha direita retirada => árvore curta na altura direita*/
  Nodo *Ap1;

  if ((*Ap)->BitD == Horizontal) {
    (*Ap)->BitD = Vertical;
    *Fim = 1;
```

```
    return;
  }
  if ((*Ap)->BitE == Horizontal) {
    Ap1 = (*Ap)->Esq;
    (*Ap)->Esq = Ap1->Dir;
    Ap1->Dir = *Ap;
    *Ap = Ap1;
    if ((*Ap)->Dir->Esq->BitD == Horizontal) {
      ED(&(*Ap)->Dir);
      (*Ap)->BitD = Vertical;
    } else if ((*Ap)->Dir->Esq->BitE == Horizontal) {
      EE(&(*Ap)->Dir);
      (*Ap)->BitD = Vertical;
    }
    *Fim = 1;
    return;
  }
  (*Ap)->BitE = Horizontal;
  if ((*Ap)->Esq->BitD == Horizontal) {
    ED(Ap);
    *Fim = 1;
    return;
  }
  if ((*Ap)->Esq->BitE == Horizontal) {
    EE(Ap);
    *Fim = 1;
  }
} /* DirCurto */

void Antecessor(Nodo *q, Nodo **r, int *Fim)
{
  if ((*r)->Dir != NULL) {
    Antecessor(q, &(*r)->Dir, Fim);
    if (!*Fim)
      DirCurto(r, Fim);
    return;
  }
  q->Reg = (*r)->Reg;
  q = *r;
  *r = (*r)->Esq;
  free(q);
  if (*r != NULL)
    *Fim = 1;
} /* Antecessor */

void IRetira(Registro x, Nodo **Ap, int *Fim)
{ /* Retira */
  Nodo *Aux;
```

```
  if (*Ap == NULL) {
    printf("Chave não está na árvore\n");
    *Fim = 1;
    return;
  }
  if (x.Chave < (*Ap)->Reg.Chave) {
    IRetira(x, &(*Ap)->Esq, Fim);
    if (!*Fim)
      EsqCurto(Ap, Fim);
    return;
  }
  if (x.Chave > (*Ap)->Reg.Chave) {
    IRetira(x, &(*Ap)->Dir, Fim);
    if (!*Fim)
      DirCurto(Ap, Fim);
    return;
  }
  *Fim = 0;
  Aux = *Ap;
  if (Aux->Dir == NULL) {
    *Ap = Aux->Esq;
    if (*Ap != NULL)
      *Fim = 1;
    return;
  }
  if (Aux->Esq == NULL) {
    *Ap = Aux->Dir;
    if (*Ap != NULL)
      *Fim = 1;
    return;
  }
  Antecessor(Aux, &Aux->Esq, Fim);
  if (!*Fim)
    EsqCurto(Ap, Fim);

  /* Encontrou chave */
} /* IRetira */


void Retira(Registro x, Nodo **Ap)
{
  int Fim;

  IRetira(x, Ap, &Fim);
```

```
} /* Retira */
```

Programa D.15: Procedimento para retirar da árvore SBB

```
#define D  "no. de bits de cada chave"  /* depende de ChaveTipo */

typedef  "a definir, dependendo da aplicação"  ChaveTipo;

typedef int IndexAmp;

typedef enum {
  Interno, Externo
} NodoTipo;

typedef struct PatNodo_str *Árvore;

typedef struct PatNodo_str {
  NodoTipo nt;
  union {
    struct {
      IndexAmp Index;
      Árvore Esq, Dir;
    } U0;
    ChaveTipo Chave;
  } UU;
} PatNodo;

typedef int Dib;
```

Programa D.16: Estrutura de dados

```
Dib Bit(IndexAmp i, ChaveTipo k)
{
  /* Retorna o i-ésimo bit da chave k a partir da esquerda */

  if (i == 0)
    return 0;
  else
    return ((k >> (i - 1)) & 1);
} /* Bit */

int EData(Árvore p)
{
  /* Verifica se p^e nodo externo */
  return (p->nt == Externo);
```

```
} /* EData */
```

Programa D.17: Funções auxiliares

```
void CrieNodos(ChaveTipo k, ChaveTipo ka, int i, Árvore *v, in t *h)
{
 /* O nodo interno contém o valor do indice do bit em que as chaves
    k e ka diferem. O nodo externo conterá a chave k a ser inserida.
    O endereço do nodo interno retorna em v; h indica se a inserção
    já foi feita */
 Árvore p, q;
 int b;

 b = 1;
 while (b && i <= D) {
  if (Bit(i, k) == Bit(i, ka))
   i++;
  else
   b = 0;
 }
 if (i > D) {
  *h = 1;  /* k já se encontra na árvore */
  return;
 }
 *h = 0;
 p = (Árvore) malloc(sizeof(PatNodo));
 q = (Árvore) malloc(sizeof(PatNodo));
 p->nt = Interno;
 p->UU.U0.Index = i;
 if (Bit(i, k) == 0)
  p->UU.U0.Esq = q;
 else
  p->UU.U0.Dir = q;
 q->nt = Externo;
 q->UU.Chave = k;
 *v = p;
} /* CrieNodos */
```

Programa D.18: Função CrieNodos

```
void Pesquise(ChaveTipo k, Árvore t)
{
 if (EData(t)) {
  if (k == t->UU.Chave)
   printf("Elemento encontrado\n");
```

```
    else
      printf("Elemento não encontrado\n");
    return;
  }
  if (Bit(t->UU.U0.Index, k) == 0)
    Pesquise(k, t->UU.U0.Esq);
  else
    Pesquise(k, t->UU.U0.Dir);
} /* Pesquise */
```

Programa D.19: Algoritmo de pesquisa

```
void Inicialize(Árvore *r)
{
  Árvore WITH;

  *r = (Árvore) malloc(sizeof(PatNodo));
  WITH = *r;
  WITH->UU.U0.Index = 0;
  WITH->UU.U0.Esq = NULL;
} /* Inicialize */
```

Programa D.20: Inicialização da árvore

```
void Insira(ChaveTipo k, Árvore *t, Árvore *u, int i, int *h)
{
  Árvore WITH;

  if (*t == NULL) { /* insere a primeira chave */
    *h = 1;
    *t = (Árvore) malloc(sizeof(PatNodo));
    WITH = *t;
    WITH->nt = Externo;
    WITH->UU.Chave = k;
    return;
  }
  if (EData(*t)) {
    CrieNodos(k, (*t)->UU.Chave, i + 1, u, h);
    return;
  }
  if ((*t)->UU.U0.Index == i + 1)
    i++;
  if (Bit((*t)->UU.U0.Index, k) == 0) {
    Insira(k, &(*t)->UU.U0.Esq, u, i, h);
    if (*h)
```

```
    return;
  if ((*t)->UU.U0.Index >= (*u)->UU.U0.Index)  /* insere nodos */
    return;
  *h = 1;
  if (Bit((*u)->UU.U0.Index, k) == 0)
    (*u)->UU.U0.Dir = (*t)->UU.U0.Esq;
  else
    (*u)->UU.U0.Esq = (*t)->UU.U0.Esq;
  (*t)->UU.U0.Esq = *u;
  return;
 }
 Insira(k, &(*t)->UU.U0.Dir, u, i, h);
 if (*h)
  return;
 if ((*t)->UU.U0.Index >= (*u)->UU.U0.Index)
  return;
 }
 Insira(k, &(*t)->UU.U0.Dir, u, i, h);
 if (*h)
  return;
 if ((*t)->UU.U0.Index >= (*u)->UU.U0.Index)
  return;
 *h = 1;
 if (Bit((*u)->UU.U0.Index, k) == 0)
  (*u)->UU.U0.Dir = (*t)->UU.U0.Dir;
 else
  (*u)->UU.U0.Esq = (*t)->UU.U0.Dir;
 (*t)->UU.U0.Dir = *u;
} /* Insira */
```

Programa D.21: Algoritmo de inserção

```
typedef char TipoChave[n+1];

typedef int Índice;

Índice h(TipoChave Chave)
{
 int i, Soma;

 Soma = 0;
 for (i = 0; i < n; i++)
  Soma += Chave[i];
 return (Soma % M);
```

```
} /* h */
```

Programa D.22: Implementação de função de transformação

```
typedef char TipoChave[n+1];

typedef struct {
  TipoChave Chave;
  /* — outros componentes — */
} TipoItem;

typedef struct Célula_str *Apontador;

typedef struct Célula_str {
  TipoItem Item;
  Apontador Prox;
} Célula;

typedef struct {
  Apontador Primeiro, Último;
} TipoLista;

typedef TipoLista TipoDicionário[M];
```

Programa D.23: Estrutura do dicionário para listas encadeadas

```
void Inicializa(TipoDicionário T)
{
  int i;

  for (i = 0; i < M; i++)
    FLVazia(&T[i]);
} /* Inicializa */

Apontador Pesquisa(TipoChave Ch, TipoDicionário T)
{
  /* Obs.: Apontador de retorno aponta para o item anterior da lista */
  Índice i;
  Apontador p;

  i = h(Ch);
  if (Vazia(T[i]))
    return NULL;  /* Pesquisa sem sucesso */
  else {
    p = T[i].Primeiro;
    while (p->Prox->Prox != NULL &&
```

```
            strncmp(Ch, p->Prox->Item.Chave, sizeof(TipoChave)))
     p = p->Prox;
    if (!strncmp(Ch, p->Prox->Item.Chave, sizeof(TipoChave)))
     return p;
    else
     return NULL;
  } /* Pesquisa sem sucesso */
} /* Pesquisa */

void Insere(TipoItem x, TipoDicionário T)
{
 if (Pesquisa(x.Chave, T) == NULL)
  Ins(x, &T[h(x.Chave)]);
 else
  printf(" Registro já  está  presente\n");
} /* Insere */


void Retira(TipoItem x, TipoDicionário T)
{
 Apontador p;

 p = Pesquisa(x.Chave, T);
 if (p == NULL)
  printf(" Registro não está  presente\n");
 else
  Ret(p, &T[h(x.Chave)], &x);
} /* Retira */
```

Programa D.24: Operações do dicionário usando listas encadeadas

```
#define Vazio          "                "
#define Retirado       "************"

typedef int Apontador;

typedef char TipoChave[n + 1];

typedef struct {
 TipoChave Chave;
 /* — outros componentes — */
} TipoItem;
```

```
typedef TipoItem TipoDicionário[M];
```

Programa D.25: Estrutura do dicionário usando *open addressing*

```
void Inicializa(TipoDicionário T)
{
  int i;

  for (i = 0; i < M; i++)
    strcpy(T[i].Chave, Vazio);
} /* Inicializa */

Apontador Pesquisa(TipoChave Ch, TipoDicionário T)
{
  int i, Inicial;

  Inicial = h(Ch);
  i = 0;
  while ( (strncmp(T[(Inicial+i) % M].Chave, Vazio, sizeof(TipoChave)))
       && (strncmp(T[(Inicial + i) % M].Chave, Ch, sizeof(TipoChave)))
       && (i < M) ) i++;
  if (!strncmp(T[(Inicial + i) % M].Chave, Ch, sizeof(TipoChave) )) {
    return ((Inicial + i) % M);
  } else
    return M; /* Pesquisa sem sucesso */
} /* Pesquisa */

void Insere(TipoItem x, TipoDicionário T)
{
  int i, Inicial;

  Inicial = h(x.Chave);
  i = 0;
  while ( (strncmp(T[(Inicial+i) % M].Chave, Vazio, sizeof(TipoChave)))
   && (strncmp(T[(Inicial+i) % M].Chave, Retirado, sizeof(TipoChave)))
   &&  (i < M) ) i++;
  if (i < M) {
    T[(Inicial + i) % M] = x;
  } else
    printf(" Tabela cheia\n");
} /* Insere */

void Retira(TipoChave Ch, TipoDicionário T)
{
  Índice i;

  i = Pesquisa(Ch, T);
```

```
  if (i < M)
    memcpy(T[i].Chave, Retirado, sizeof(TipoChave));
  else
    printf("Registro não está presente\n");
} /* Retira */
```

Programa D.26: Operações do dicionário usando *open addressing*

Apêndice E

Programas C do Capítulo 5

```
#define TamanhodaPágina  512
#define ItensPorPágina    64   /* TamanhodaPágina/TamanhoItem */

typedef struct {
  TipoChave Chave;
  /* — outros componentes — */
} Registro;

typedef struct {
  int p;
  int b; /* variando de 1 até ItensPorPágina */
} EndereçoTipo;

typedef struct {
  Registro Reg;
  EndereçoTipo Esq, Dir;
} ItemTipo;

typedef ItemTipo PáginaTipo[ItensPorPágina];
```

Programa E.1: Estrutura de dados para o sistema de paginação

```
typedef union {
  PáginaTipoA Pa;
  PáginaTipoB Pb;
  PáginaTipoC Pc;
} PáginaTipo;
```

Programa E.2: Diferentes tipos de páginas para o sistema de paginação

```
typedef struct {
  TipoChave Chave;
  /* — outros componentes — */
} Registro;

typedef struct Página_str *Apontador;

typedef struct Página_str {
  int n;
  Registro r[mm];
  Apontador p[mm + 1];
} Página;

typedef Apontador TipoDicionário;
```

Programa E.3: Estrutura do dicionário para árvore B

```
void Inicializa(TipoDicionário *Dicionário)
{
  *Dicionário = NULL;
} /* Inicializa */
```

Programa E.4: Função para inicializar uma ávore B

```
void Pesquisa(Registro *x, Apontador Ap)
{
  int i;
  if (Ap == NULL) {
    printf("Erro:  Registro não está presente na árvore\n");
    return;
  }
  i = 1;
  while (i < Ap->n && x->Chave > Ap->r[i - 1].Chave)
    i++;
  if (x->Chave == Ap->r[i - 1].Chave) {
    *x = Ap->r[i - 1];
    return;
  }
  if (x->Chave < Ap->r[i - 1].Chave)
    Pesquisa(x, Ap->p[i - 1]);
  else
    Pesquisa(x, Ap->p[i]);
```

```
} /* Pesquisa */
```

Programa E.5: Função para pesquisar na ávore B

```
void Ins(Registro Reg, Apontador Ap, int *Cresceu,
         Registro *RegRetorno, Apontador *ApRetorno)
{
  int i;

  if (Ap == NULL) {
    *Cresceu = 1;
    Atribui Reg a RegRetorno; Atribui null a ApRetorno;
    return;
  }
  i = 1;
  while (i < Ap->n && x.Chave > Ap->r[i - 1].Chave) {
    i++;
  }
  if (x.Chave == Ap->r[i - 1].Chave)
    printf("Erro: Registro já está presente na árvore\n");
  else if (x.Chave < Ap->r[i - 1].Chave) {
    Ins(x, Ap->p[i - 1], Cresceu, RegRetorno, ApRetorno);
  } else {
    Ins(x, Ap->p[i], Cresceu, RegRetorno, ApRetorno);
  }
  if (*Cresceu) {
    if (Número de registros em Ap) < mm
      Insere na página Ap e *Cresceu = 0;
  }

  /* Overflow: página tem que ser dividida */
  Cria nova página ApTemp;
  Transfere metade dos registros de Ap para ApTemp;
  Atribui registro do meio a RegRetorno;
  Atribui ApTemp a ApRetorno;
} /* Ins */

void Insere(Registro Reg, Apontador *Ap)
{
  Ins(Reg, *Ap, &Cresceu, &RegRetorno, &ApRetorno);
  if (Cresceu) {
    Cria nova página raiz para RegRetorno e ApRetorno;
  }
```

```
} /* Insere */
```

Programa E.6: Primeiro refinamento do algoritmo Insere na árvore B

```
void InsereNaPágina(Apontador Ap, Registro Reg, Apontador ApDir)
{
 int k;
 int NãoAchouPosição;

 k = Ap->n;
 NãoAchouPosição = k > 0;
 while (NãoAchouPosição) {
  if (Reg.Chave >= Ap->r[k - 1].Chave) {
   NãoAchouPosição = 0;
   break;
  }
  Ap->r[k] = Ap->r[k - 1];
  Ap->p[k + 1] = Ap->p[k];
  k--;
  if (k < 1)
   NãoAchouPosição = 0;
 }
 Ap->r[k] = Reg;
 Ap->p[k + 1] = ApDir;
 Ap->n++;
} /* InsereNaPágina */
```

Programa E.7: Função InsereNaPágina

```
void Ins(Registro Reg, Apontador Ap, int *Cresceu,
         Registro *RegRetorno, Apontador *ApRetorno)
{
 Apontador ApTemp;
 int i, j;

 if (Ap == NULL) {
  *Cresceu = 1;
  *RegRetorno = Reg;
  *ApRetorno = NULL;
  return;
 }
 i = 1;
 while (i < Ap->n && Reg.Chave > Ap->r[i - 1].Chave)
  i++;
 if (Reg.Chave == Ap->r[i - 1].Chave) {
  printf(" Erro:  Registro já está presente\n");
```

```
    *Cresceu = 0;
    return;
  }
  if (Reg.Chave < Ap->r[i - 1].Chave)
    Ins(Reg, Ap->p[i - 1], Cresceu, RegRetorno, ApRetorno);
  else
    Ins(Reg, Ap->p[i], Cresceu, RegRetorno, ApRetorno);
  if (!*Cresceu)
    return;
  if (Ap->n < mm)
  { /* Página tem espaço */
    InsereNaPágina(Ap, *RegRetorno, *ApRetorno);
    *Cresceu = 0;
    return;
  }
  /* Overflow: Página tem que ser dividida */
  ApTemp = (Apontador) malloc(sizeof(Página));
  ApTemp->n = 0;
  ApTemp->p[0] = NULL;
  if (i <= m + 1) {
    InsereNaPágina(ApTemp, Ap->r[mm - 1], Ap->p[mm]);
    Ap->n--;
    InsereNaPágina(Ap, *RegRetorno, *ApRetorno);
  } else {
    InsereNaPágina(ApTemp, *RegRetorno, *ApRetorno);
  }
  for (j = m + 2; j <= mm; j++)
    InsereNaPágina(ApTemp, Ap->r[j - 1], Ap->p[j]);
  Ap->n = m;
  ApTemp->p[0] = Ap->p[m + 1];
  *RegRetorno = Ap->r[m];
  *ApRetorno = ApTemp;
} /* Ins */

void Insere(Registro Reg, Apontador *Ap)
{
  int Cresceu;
  Registro RegRetorno;
  Apontador ApRetorno;
  Apontador ApTemp;

  Ins(Reg, *Ap, &Cresceu, &RegRetorno, &ApRetorno);
  if (Cresceu) { /* Árvore cresce na altura pela raiz */
    ApTemp = (Apontador) malloc(sizeof(Página));
    ApTemp->n = 1;
    ApTemp->r[0] = RegRetorno;
    ApTemp->p[1] = ApRetorno;
    ApTemp->p[0] = *Ap;
```

```
    *Ap = ApTemp;
  }
} /* Insere */
```

Programa E.8: Refinamento final do algoritmo Insere

```
void Reconstitui(Apontador ApPag, Apontador ApPai, int PosPai,
                 int *Diminuiu)
{
 Apontador Aux;
 int DispAux, j;

 if (PosPai < ApPai->n) {  /* Aux = Página a direita de ApPag */
  Aux = ApPai->p[PosPai + 1];
  DispAux = (Aux->n - m + 1) / 2;
  ApPag->r[ApPag->n] = ApPai->r[PosPai];
  ApPag->p[ApPag->n + 1] = Aux->p[0];
  ApPag->n++;
  if (DispAux > 0) {  /* Existe folga: transfere de Aux para ApPag */
   for (j = 1; j < DispAux; j++)
        InsereNaPágina(ApPag, Aux->r[j - 1], Aux->p[j]);
   ApPai->r[PosPai] = Aux->r[DispAux - 1];
   Aux->n -= DispAux;
   for (j = 0; j < Aux->n; j++)
        Aux->r[j] = Aux->r[j + DispAux];
   for (j = 0; j <= Aux->n; j++)
        Aux->p[j] = Aux->p[j + DispAux];
   *Diminuiu = 0;
  } else { /* Fusão: intercala Aux em ApPag e libera Aux */
   for (j = 1; j <= m; j++)
    InsereNaPágina(ApPag, Aux->r[j - 1], Aux->p[j]);
   free(Aux);
   for (j = PosPai + 1; j < ApPai->n; j++) {
    ApPai->r[j - 1] = ApPai->r[j];
    ApPai->p[j] = ApPai->p[j + 1];
   }
   ApPai->n--;
   if (ApPai->n >= m)
    *Diminuiu = 0;
  }
 } else { /* Aux = Página a esquerda de ApPag */
  Aux = ApPai->p[PosPai - 1];
  DispAux = (Aux->n - m + 1) / 2;
  for (j = ApPag->n; j >= 1; j--) {
   ApPag->r[j] = ApPag->r[j - 1];
  }
  ApPag->r[0] = ApPai->r[PosPai - 1];
```

```
    for (j = ApPag->n; j >= 0; j--) {
      ApPag->p[j + 1] = ApPag->p[j];
    }
    ApPag->n++;
    if (DispAux > 0) { /* Existe folga: transfere de Aux para ApPag */
      for (j = 1; j < DispAux; j++)
        InsereNaPágina(ApPag, Aux->r[Aux->n - j],
            Aux->p[Aux->n - j + 1]);
      ApPag->p[0] = Aux->p[Aux->n - DispAux + 1];
      ApPai->r[PosPai - 1] = Aux->r[Aux->n - DispAux];
      Aux->n -= DispAux;
      *Diminuiu = 0;
    } else { /* Fusão: intercala ApPag em Aux e libera ApPag */
      for (j = 1; j <= m; j++)
        InsereNaPágina(Aux, ApPag->r[j - 1], ApPag->p[j]);
      free(ApPag);
      ApPai->n--;
      if (ApPai->n >= m)
        *Diminuiu = 0;
    }
  }
} /* Reconstitui */

void Antecessor(Apontador Ap, int Ind, Apontador ApPai, int *Diminuiu)
{
  if (ApPai->p[ApPai->n] != NULL) {
    Antecessor(Ap, Ind, ApPai->p[ApPai->n], Diminuiu);
    if (*Diminuiu)
      Reconstitui(ApPai->p[ApPai->n], ApPai, ApPai->n, Diminuiu);
    return;
  }
  Ap->r[Ind - 1] = ApPai->r[ApPai->n - 1];
  ApPai->n--;
  *Diminuiu = ApPai->n < m;
} /* Antecessor */

void Ret(TipoChave Ch, Apontador *Ap, int *Diminuiu)
{
  int Ind, j;
  Apontador WITH;

  if (*Ap == NULL) {
    printf("Erro: registro não está na árvore\n");
    *Diminuiu = 0;
    return;
  }
  WITH = *Ap;
  Ind = 1;
```

```
  while (Ind < WITH->n && Ch > WITH->r[Ind - 1].Chave)
    Ind++;
  if (Ch == WITH->r[Ind - 1].Chave) {
    if (WITH->p[Ind - 1] == NULL) { /* Página folha */
      WITH->n--;
      *Diminuiu = WITH->n < m;
      for (j = Ind; j <= WITH->n; j++) {
            WITH->r[j - 1] = WITH->r[j];
            WITH->p[j] = WITH->p[j + 1];
      }
      return;
    }
    Antecessor(*Ap, Ind, WITH->p[Ind - 1], Diminuiu);
    if (*Diminuiu)
      Reconstitui(WITH->p[Ind - 1], *Ap, Ind - 1, Diminuiu);
    return;
  }
  if (Ch > WITH->r[Ind - 1].Chave)
    Ind++;
  Ret(Ch, &WITH->p[Ind - 1], Diminuiu);
  if (*Diminuiu)
    Reconstitui(WITH->p[Ind - 1], *Ap, Ind - 1, Diminuiu);
} /* Ret */

void Retira(TipoChave Ch, Apontador *Ap)
{
  int Diminuiu;
  Apontador Aux;

  Ret(Ch, Ap, &Diminuiu);
  if (Diminuiu && (*Ap)->n == 0) { /* Árvore diminui na altura */
    Aux = *Ap;
    *Ap = Aux->p[0];
    free(Aux);
  }
} /* Retira */
```

Programa E.9: Função Retira

```
typedef struct {
  TipoChave Chave;
  /* — outros componentes — */
} Registro;

typedef enum {
  Interna, Externa
} PáginaTipo;
```

```
typedef struct Página_str *Apontador;

typedef struct Página_str {
  PáginaTipo Pt;
  union {
    struct {
      int n;
      TipoChave r[mm];
      Apontador p[mm + 1];
    } U0;
    struct {
      int n;
      Registro r[mm2];
    } U1;
  } UU;
} Página;

typedef Apontador TipoDicionário;
```

Programa E.10: Estrutura do dicionário para árvore B*

```
void Pesquisa(Registro *x, Apontador *Ap)
{
  int i;
  Apontador WITH;

  if ((*Ap)->Pt == Interna) {
    WITH = *Ap;
    i = 1;
    while (i < WITH->UU.U1.n &&
      x->Chave > WITH->UU.U1.r[i - 1]) i++;
    if (x->Chave < WITH->UU.U1.r[i - 1])
      Pesquisa(x, &WITH->UU.U0.p[i - 1]);
    else
      Pesquisa(x, &WITH->UU.U0.p[i]);
    return;
  }
  WITH = *Ap;
  i = 1;
  while ((i < WITH->UU.U1.n &&
    x->Chave) > WITH->UU.U1.r[i - 1].Chave) i++;
  if (x->Chave == WITH->UU.U1.r[i - 1].Chave)
    *x = WITH->UU.U1.r[i - 1];
  else
    printf("Registro não está presente na árvore\n");
}
```

Programa E.11: Função para pesquisar na árvore B*

Apêndice F

Caracteres ASCII

Dec	Car	Dec	Car	Dec	Car	Dec	Car	Dec	Car
32		51	3	70	F	89	Y	108	l
33	!	52	4	71	G	90	Z	109	m
34	"	53	5	72	H	91	[	110	n
35	#	54	6	73	I	92	\	111	o
36	$	55	7	74	J	93	]	112	p
37	%	56	8	75	K	94	^	113	q
38	&	57	9	76	L	95	_	114	r
39	'	58	:	77	M	96	`	115	s
40	(	59	;	78	N	97	a	116	t
41	)	60	<	79	O	98	b	117	u
42	*	61	=	80	P	99	c	118	v
43	+	62	>	81	Q	100	d	119	w
44	,	63	?	82	R	101	e	120	x
45	-	64	@	83	S	102	f	121	y
46	.	65	A	84	T	103	g	122	z
47	/	66	B	85	U	104	h	123	{
48	0	67	C	86	V	105	i	124	|
49	1	68	D	87	W	106	j	125	}
50	2	69	E	88	X	107	k	126	~

Apêndice G

Referências Bibliográficas

Adel'son-Vel'skii, G.M. e Landis, E.M. (1962) "An Algorithm for the Organization of Information", *Doklady Akademia Nauk USSR 146 (2),* 263-266, Tradução para o Inglês em *Soviet Math. Doklay 3,* 1962, 1259-1263.

Aho, A.V., Hoperoft J.E. e Ullman J.D. (1974) *The Design and Analysis of Computer Algorithms.* Addison-Wesley, Reading, Mass.

Aho, A.V., Hoperoft, J.E. e Ullman, J.D. (1983) *Data Structures and Algorithms.* Addison-Wesley, Reading, Mass.

Albuquerque, L.C.A. e Ziviani, N. (1985) "Estudo Empírico de uma Nova Implementação para o Algoritmo de Construção da Arvore Patricia". *V Congresso da Sociedade Brasileira de Computação,* Porto Alegre, RGS, 254-267.

Árabe, J.N.C. (1992) *Comunicação Pessoal,* Belo Horizonte, MG.

Barbosa, E.F. e Ziviani, N. (1992) `"Data Structures and Access Methods for Read-Only Optical Disks". Baeza-Yates, R. e Manber, U. (Eds.) in *Computer Science: Research and Applications,* Plenum Publishing Corp. , New York, NY, 189-207.

Bayer, R. (1971) "Binary B Trees for Virtual Memory", *ACM SIGFIDET Workshop,* San Diego, 219-235.

Bayer, R. (1972) "Symmetric Binary B-Trees: Data Structure and Maintenance Algoritms", *Acta Informatica 1 (4),* 290-306.

Bayer, R. e McCreight, E.M. (1972) "Organization and Maintenance of Large Ordered Indices". *Acta Informatica 1(3),* 173-189.

Bayer, R. e Schkolnick, M. (1977) "Concurrency of Operations on B-trees: *Acta Informatica 9(1),* 1-21.

Carvalho, M.L.B. (1992) *Comunicação Pessoal,* Belo Horizonte, MG.

Clancy, M. e Cooper, D. (1982) *Oh! Pascal. W. W.* Norton, New York, NY.

Comer, D. (1979) "The Ubiquitous B-tree," *ACM Computing Surveys* 11(2), 121-137.

Cooper, D. (1983) *Standard Pascal User Reference Manual. W.W.* Norton, New York, NY.

Cormem, T.H., Leiserson, C.E. e Rivest, R.L. (1990) *Introduction to Algorithms.* McGraw-Hill e The Mit Press, Cambridge, Mass.

Dahl, O.J., Dijkstra, E.W. e Hoare, C.A.R. (1972) *Structured Programming.* Academic Press, New York, NY.

Dijkstra, E.W. (1965) "Co-operating Sequential Processes". In *Programming Languages* F. Genuys (ed.), Academic Press, New York, NY.

Dijkstra, E.W. (1971) A *Short Introduction to the Art of Programming.* Technological University Endhoven.

Dijkstra, E.W. (1976) *A Discipline of Programming.* Prentice-Hall, Englewood Cliffs, NJ.

Eisenbarth, B., Ziviani, N., Gonnet, G.H., Mehlhorn, K. e Wood, D. (1982) " The Theory of Fringe Analysis and Its Application to 2-3 Trees and B-Trees", *Information and Control 55 (1-3),* 125-174.

Feller, W. (1968) *An Introduction to Probability Theory and Its Applications. Vol.* 1, Wiley, New York, NY.

Flajolet, P. e Vitter, J.S. (1987) "Average-case Analysis of Algorithms and Data Structures". Technical Report 718, INRIA, França.

Floyd, R.W. (1964) "Treesort". *Algorithm 243, Communications ACM* 7(12), 701.

Furtado, A.L. (1984) *Comunicação Pessoal,* Rio de Janeiro, RJ.

Garey, M.R. e Johnson, D.S. (1979) *Computers and Intractability A Guide to the Theory of NP-Completeness.* Freeman, San Francisco, CA.

Gonnet, G.H. e Baeza-Yates, R. (1991) *Handbook of Algorithms and Data Structures.* Addison-Wesley, Reading, Mass., segunda edição. Graham,

R.L., Knuth, D.E. e Patashnik, O. *(1989) Concrete Mathematics.*

Addison-Wesley, Reading, Mass.

Greene, D.H. e Knuth, D.E. (1982) *Mathematics for the Analysis of Algorithms.* Birkhanser, Boston, Mass.

Guibas, L. e Sedgewick, R. (1978) "A Dichromatic Framework for Balanced Trees", *19th Annual Symposium on Foundations of Computer Science,* IEEE.

Hibbard, T.N. "Some Combinatorial Properties of Certain Trees with Applications to Searching and Sorting". *Journal of the ACM 9,* 13-28.

Hoare, C.A.R. (1962) "Quicksort". *The Computer Journal 5(1),* 10-15.

Hoare, C.A.R *(1969)* "Axiomatic Bases of Computer Programming". *Communications ACM 12 (10), 576-583.*

Horowitz, E. e Sahni, S. *(1978) Fundamentals of Computer Algorithms.* Computer Science Press, Potomac, Maryland.

Jensen, K. e Wirth, N. *(1974) Pascal User Manual and Report.* Springer-Verlag, Berlin, segunda edição.

Keehn, D. e Lacy, J. *(1974)* "VSAM Data Set Design Parameters". *IBM Systems Journal 3,* 186-212.

Knott, G. (1975) "Hashing Functions". *The Computer Journal* 18 (3), *265-378.*

Knuth, D.E. (1968) *The Art of Computer Programming, Vol.1: Fundamental Algorithms.* Addison-Wesley, Reading, Mass.

Knuth, D.E. (1971) "Mathematical Analysis of Algorithms". *Procedings IFIP Congress 71,* vol. 1, North Holland, Amsterdam, Holanda, *135-143.*

Knuth, D.E. (1973) *The Art of Computer Programming; Volume 3: Sorting and Searching.* Addison-Wesley, Reading, Mass.

Knuth, D.E. (1976) "Big Omicron and Big Omega and Big Theta". *ACM SIGACT News 8 (2),* 18-24.

Knuth, D.E. (1981) *The Art of Computer Programming, Vol. 2: Seminumerical Algoritms.* Segunda edição, Addison-Wesley, Reading, Mass.

Lister, A.M. *(1975) Fundamentals of Operating Systems.* Macmillan, London.

Lueker, *G.S.* (1980) "Some Techniques *for* Solving Recurrences". *ACM Computing Surveys* 12(4), 419-436.

Manber, U. *(1988)* "Using Induction to Design Algorithms". *Communications ACM 31 (11),* 1300-1313.

Manber, U. *(1989) Introduction to Algorithms A Creative Approach. Addi*son-Wesley, Reading, Mass.

Manber, U. e Myers, *G.* (1990) "Suffix Arrays: A New Method for On-Line String Searches". *1st ACM-SIAM Symposium on Discrete Algorithms, San* Francisco, CA, 319-327.

Morrison, D.R. (1968) "PATRICIA - Practical *Algorithm* To Retrieve Information Coded *In Alphanumeric". Journal of the ACM15(4),* 514-534.

Murta, C.D. (1992) *Comunicação Pessoal.* Belo Horizonte, *MG.*

Mehlhorm, K. (1984) *Data Structures and Algorithms, Vol.1: Sorting and Searching.* Springer-Verlaq, Berlin.

Olivié, *H.* (1980) *"Symmetric Binary* B-Trees Revisited" , Technical Report *80-01,* Interstedelijke Industriële Hogerschool Antwerpen-Mechelen, Bélgica.

Peterson, *J. e* Silberschatz, *A.* (1983) *Operating System Concepts.* Addison-Wesley, Reading, Mass.

Sedgewick, R. *(1975)* "The Analysis of Quicksort Programs". *Acta Informatica 7, 327-355.*

Sedgewick, R. (1978) *Quicksort.* Garland. *(também* publicado *como tese* de doutorado do *autor, Stanford* University, *C.S.* Department Techical *Report 75-492,* 1975).

Sedgewick, R. (1978a) "Implementing Quicksort *Programs". Communications ACM 21 (10), 847-857.*

Sedgewick, It *(1988) Algorithms.* Addison-Wesley, Reading, Mass., segunda *edição.*

Shell, D.L. (1959) "A Highspeed *Sorting* Procedure". *Communications ACM 2(7), 30-32.*

Sleator, D.D. e Tarjan, R.E. *(1985)* "Self-Adjusting Binary Search Trees". *Journal of the ACM 32, 652-686.*

Stanat, D.F. e McAllister, D.F. *(1977) Discrete Matematics in Computer Science.* Prentice-Hall, Englewood Cliffs, NJ, Capitulo 5, 218-274.

Standish, T.A. (1980) *Data Structures Techniques.* Addison-Wesley, Reading, Mass.

Tanenbaum, A.S. (1987) *Operating Systems: Design and Implementation.* Prentice-Hall, Englewood Cliffs, NJ.

Tarjan, R.E. (1983) *Data Structures and Network Algorithms.* SIAM, Philadelphia, Pennsylvania.

Tarjan, R.E. *(1985)* "Amortized Computational Complexity". *SIAM Journal on Applied and Discrete Mathematics 6,* 306-318.

Terada, R. (1991) *Desenvolvimento de Algoritmos e Estruturas de Dados.* McGraw-Hill e Makron Books do Brasil, São Paulo, SP.

Vuillemin, J. (1978) "A Data Structure for Manipulating Priority Queues". *Communications ACM 21 (4),* 309-314.

Wagner, R. *(1973)* "Indexing Design Considerations," *IBM Systems Journal4, 351-367.*

Weide, B. *(1977)* "A Survey of Analysis Techniques for Discrete Algorithms". *ACM Computing Surveys 9 (4),* 291-313.

Williams, J.W.J. (1964) "Algorithm *232". Communications ACM 7(6), 347-348.*

Wirth, N. (1971) "Program Development by Stepwise Refinement ". Com*munications ACM 14 (4),* 221-227.

Wirth, N. *(1974)* "On The Composition of well-Structured *Programs".ACM Emputing Serveys 6 (4), 247-259.*

Wirth, N. (1976) *Algorithms + Data Structures = Programs.* Prentice-Hall, Englewood Cliffs, NJ.

Wirth, N. (1986) *Algorithms and Data Structures.* Prentice-Hall, Englewood Cliffs, NJ.

Ziviani, N., Olivié, H. e Gonnet, G.H. (1985) "The Analysis of the Improved Symmetric Binary B-Tree Algorithm". *The Computer Journal 28 (4),* 417-425.

Ziviani, N. *e* Tompa, F.W. *(1982) "A* Look at Symmetric Binary B-Trees" *INFOR Canadian Journal of Operational Research and Information Processing 20 (2),* 65-81.

Índice

IMPRESSÃO E ACABAMENTO

YANGRAF

GRÁFICA E EDITORA LTDA.
TEL/FAX: (011) 218-1788
RUA: CONS. GAL. PINHEIRO 137