









**Estruturas de Dados  
e Algoritmos em**

# **JAVA**



G655e Goodrich, Michael  
Estrutura de dados e algoritmos em Java / Michael Goodrich,  
Roberto Tamassia ; tradução Bernardo Copstein, Leandro Bento  
Pompermeier. – 4. ed. – Porto Alegre : Bookman, 2007.  
600 p. ; il. p&b; 25 cm.

ISBN 978-85-60031-50-4

I. Computação – Linguagem – Java. I. Tamassia, Roberto.  
II. Título.

CDU 004.43JAVA

Catalogação na publicação: Júlia Angst Coelho – CRB 10/1712

Copyrighted material



Obra originalmente publicada sob o título *Data Structures and Algorithms in Java*, 4th Edition

Copyright© 2006, John Wiley & Sons, Inc. All rights reserved. This translation is published under license.

ISBN 0-471-73884-0

Capa: *Mário Röhnelt, arte sobre capa original*

Preparação do original: *Fábio Grespan Godinho*

Supervisão editorial: *Arysinha Jacques Affonso e Denise Weber Nowaczyk*

Editoração eletrônica: *Laser House*

Java é marca registrada da Sun Microsystems, Inc. UNIX® é uma marca registrada nos Estados Unidos e em outros países, licenciada pela X/Open Company, Ltd. Powerpoint® é marca registrada da Microsoft Corporation. Todos os outros nomes de produtos mencionados no livro são marcas registradas de seus respectivos proprietários.

Reservados todos os direitos de publicação, em língua portuguesa, à  
ARTMED® EDITORA S.A.

(BOOKMAN® COMPANHIA EDITORA é uma divisão da ARTMED® EDITORA S. A.)

Av. Jerônimo de Ornelas, 670 – Santana

90040-340 – Porto Alegre – RS

Fone: (51) 3027-7000 Fax: (51) 3027-7070

É proibida a duplicação ou reprodução deste volume, no todo ou em parte, sob quaisquer formas ou por quaisquer meios (eletrônico, mecânico, gravação, fotocópia, distribuição na Web e outros), sem permissão expressa da Editora.

#### SÃO PAULO

Av. Angélica, 1.091 – Higienópolis

01227-100 – São Paulo – SP

Fone: (11) 3665-1100 Fax: (11) 3667-1333

SAC 0800 703-3444

IMPRESSO NO BRASIL

PRINTED IN BRAZIL

Para Karen, Paul e Anna  
– *Michael T. Goodrich*

Para Isabel  
– *Roberto Tamassia*



---

## Prefácio da quarta edição

A quarta edição foi planejada para oferecer uma introdução a estruturas de dados e algoritmos, incluindo projeto, análise e implementação. Considerando os currículos baseados no *Curriculum de Ciência da Computação IEEE/ACM 2001*, este livro é apropriado para uso nos cursos CS102 (versões I/O/B), CS103 (versões I/O/B), CS111 (versão A) e CS112 (versões A/I/O/F/H). Sua utilização nestas disciplinas é discutida detalhadamente neste prefácio.

As maiores alterações, em relação à terceira edição, são as que seguem:

- Um capítulo novo sobre arranjos, listas encadeadas e recursão
- Um capítulo novo sobre gerência de memória
- Integração total com o Java 5.0
- Melhor integração com o Framework de Coleções de Java
- Melhor cobertura de iteradores
- Aumento da cobertura de listas baseadas em vetores, incluindo a substituição do uso da classe Java.util.Vector por Java.util.ArrayList
- Atualização de todas as APIs de Java para o uso de tipos genéricos
- Simplificação dos tipos abstratos de dados lista, árvore binária e lista com prioridades
- Redução dos conceitos matemáticos para as sete funções mais usadas
- Exercícios expandidos e revisados, elevando o total de exercícios de reforço, criatividade e projetos para 670. Os exercícios acrescentados incluem novos projetos sobre a manutenção da lista de maiores escores de um jogo, avaliação pós-fixada e interfixada de expressões, avaliação de árvores minimax de jogos, processamento de ordens de compra e venda de ações, escalonamento de tarefas em uma CPU, simulação *n-body*, computação do tamanho de cadeias de DNA e criação e solução de labirintos.

Este livro está relacionado com as seguintes obras:

- M.T. Goodrich, R. Tamassia, and D.M. Mount, *Data Structures and Algorithms in C++*, John Wiley & Sons, Inc., 2004. Este livro tem uma estrutura geral similar à presente obra, mas usa C++ como linguagem base (com algumas diferenças pedagógicas modestas, mas necessárias, requeridas por esta abordagem). Assim, podem ser usados em conjunto em um currículo que admite tanto Java como C++ em suas disciplinas introdutórias.
- M.T. Goodrich and R. Tamassia, *Algorithm Design: Foundations, Analysis, and Internet Examples*\*, John Wiley & Sons, Inc., 2002. É um livro-texto para uma disciplina mais avançada em estruturas de dados e algoritmos, tal como a CS210 (versões T/W/C/S) do currículo IEEE/ACM 2001.

---

## Uso como livro-texto

O projeto e a análise de estruturas de dados eficientes foi há muito reconhecido como um dos temas-chave dentro da computação, pois o estudo de estruturas de dados faz parte do núcleo essencial de disciplinas em qualquer curso de Ciência ou Engenharia da Computação. As matérias introdutórias são apresentadas em uma seqüência de duas ou três disciplinas. Estruturas de dados elementares são introduzidas em uma primeira disciplina de programação ou introdução à computação, e a estes seguem-se introduções mais profundas sobre estruturas de dados. Além disso, essas disciplinas normalmente são seguidas de estudos mais detalhados sobre estruturas de dados e algoritmos. Acredita-se que o papel central do projeto de estruturas de dados e algoritmos nos currículos é totalmente justificado, dada a importância de estruturas de dados eficientes em sistemas de software, incluindo a Internet, sistemas operacionais, bancos de dados, compiladores e sistemas de simulação científica.

---

\* Publicado pela Bookman Editora com o título Projeto de Algoritmos: Fundamentos, Análise e Exemplos da Internet.

Com o surgimento do paradigma orientado a objetos e seu uso preferencial para a implementação de software reutilizável e robusto, tentou-se manter uma visão orientada a objetos no texto. Uma das idéias principais na abordagem orientada a objetos é a de que se deve apresentar dados como sendo encapsulados com os métodos que os acessam e modificam. Ou seja, mais do que ver dados simplesmente como uma coleção de bytes e endereços, pensa-se em dados como instâncias de um *tipo abstrato de dados (TAD)* que inclui um repertório de métodos para realizar operações sobre os dados. De forma similar, soluções orientadas a objetos são freqüentemente organizadas usando-se *padrões de projeto* comuns, que facilitam a reutilização de software e aumentam sua robustez. Assim, apresenta-se cada estrutura de dados usando TADs e suas respectivas implementações, introduzindo importantes padrões de projeto como forma de organizar as implementações em classes, métodos e objetos.

Para cada TAD analisado neste livro, apresenta-se uma interface Java correspondente. Além disso, estruturas de dados concretas que implementam os TADs são fornecidas na forma de classes Java que operacionalizam as interfaces apresentadas. Também são fornecidas implementações em Java de algoritmos básicos (como ordenação e busca em grafos) e de aplicações exemplo de estruturas de dados (como busca de "tags" HTML e álbuns de fotografias). Devido às limitações de espaço, algumas vezes estão apenas fragmentos de código no texto, deixando o restante do código fonte disponível no site Web correspondente, <http://java.datastructures.net>.

O código Java que implementa as estruturas de dados básicas neste livro está organizado em um único pacote Java, *net.datastructures*. Este pacote compõe uma biblioteca coerente de estruturas de dados e algoritmos em Java especialmente projetada para fins educacionais, de forma complementar ao framework de coleções de Java.

---

## Educação com valor agregado pela Web\*

Este livro é acompanhado por um vasto site Web:

**<http://java.datastructures.net>**

Os alunos são encorajados a usar este site juntamente com o livro para auxiliar nos exercícios e melhorar o entendimento dos temas. Os professores também são bem-vindos, devendo usar o site para auxiliá-los no planejamento, na organização e na apresentação do material de seus cursos.

### Para o estudante

Para todos os leitores e em especial para os estudantes, incluiu-se:

- todos os códigos fonte Java apresentados neste livro;
- a versão para estudantes do pacote *net.datastructures*;
- as transparências em formato PDF (quatro por página);
- um banco de dados de dicas sobre *todos* os exercícios, indexados pelo número do problema;
- animações Java e applets interativas sobre estruturas de dados e algoritmos;
- links para outras fontes sobre estruturas de dados e algoritmos.

Acredita-se que as animações Java e as applets interativas possam ser especialmente interessantes, uma vez que permitem que o leitor interaja com as diferentes estruturas de dados, o que conduz a um melhor entendimento dos diferentes TADs. Além disso, as dicas podem ser muito úteis para qualquer um que necessite de uma ajuda para começar certos exercícios.

---

\* Esta seção refere-se ao site da editora original e todos os suplementos aqui listados estão em inglês.

## Para o professor\*

Para os professores que usarem este livro, incluem-se os seguintes recursos adicionais:

- solução para mais de 200 dos exercícios do livro;
- exercícios adicionais indexados por palavra-chave;
- o pacote `net.datastructures` completo;
- transparências em formato PowerPoint;

As transparências são editáveis, de maneira que o professor que adote este livro tem liberdade para personalizar suas apresentações.

## Um recurso para ministrar estruturas de dados e algoritmos

Este livro contém vários fragmentos de códigos Java e fragmentos de pseudo-código, e mais de 670 exercícios divididos, grosso modo, em 40% de exercícios de reforço, 40% de exercícios de criatividade e 20% de projetos de programação.

Pode ser usado para as disciplinas CS102 (versões I/O/B), CS103 (versões I/O/B), CS111 (versão A) e/ou CS112 (versões A/I/O/F/H) do Currículo de Ciência da Computação da IEEE/ACM 2001, usando-se as unidades de instrução como definido na Tabela 0.1.

Unidade de instrução	Material relevante
PL1. Visão geral sobre linguagens de programação	Capítulos 1 e 2
PL2. Máquinas virtuais	Seções 14.1.1, 14.1.2 e 14.1.3
PL3. Introdução à tradução de linguagens	Seção 1.9
PL4. Declarações e tipos	Seções 1.1, 2.4 e 2.5
PL5. Mecanismos de abstração	Seções 2.4, 5.1, 5.2, 5.3, 6.1.1, 6.2, 6.4, 6.3, 7.1, 7.3.1, 8.1, 9.1, 9.3, 11.6, e 13.1
PL6. Programação Orientada a Objetos	Capítulos 1 e 2 e seções 6.2.2, 6.3, 7.3.7, 8.1.2 & 13.3.1
PF1. Construções fundamentais em programação	Capítulos 1 e 2
PF2. Algoritmos e solução de problemas	Seções 1.9 e 4.2
PF3. Estrutura de dados fundamentais	Seções 3.1, 5.1-3.2, 5.3, 6.1-6.4, 7.1-7.3, 8.1-8.3, 9.1-9.4, 10.1 e 13.1
PF4. Recursão	Seção 3.5
SE1. Projeto de software	Capítulo 2 e seções 6.2.2, 6.3, 7.3.7, 8.1.2 e 13.3.1
SE2. Usando APIs	Seções 2.4, 5.1, 5.2, 5.3, 6.1.1, 6.2, 6.4, 6.3, 7.1, 7.3.1, 8.1, 9.1, 9.3, 11.6 e 13.1
AL1. Análise de algoritmos básicos	Capítulo 4
AL2. Estratégias algorítmicas	Seções 11.1.1, 11.7.1, 12.2.1, 12.4.2 e 12.5.2
AL3. Computação de algoritmos fundamentais	Seções 8.1.4, 8.2.3, 8.3.5, 9.2 & 9.3.3 e Capítulos 11, 12 e 13
DS1. Funções, relações e conjuntos	Seções 4.1, 8.1 e 11.6
DS3. Técnicas de prova	Seções 4.3, 6.1.4, 7.3.3, 8.3, 10.2, 10.3, 10.4, 10.5, 11.2.1, 11.3, 11.6.2, 13.1, 13.3.1, 13.4 e 13.5
DS4. Contagem	Seções 2.2.3 e 11.1.5
DS5. Grafos e árvores	Capítulos 7, 8, 10 e 13
DS6. Probabilidade discreta	Apêndice A e seções 9.2.2, 9.4.2, 11.2.1 e 11.7

**Tabela 0.1** Material para as unidades do currículo de computação da IEEE/ACM 2001

\* Professores interessados em receber material de apoio (em inglês) devem entrar em contato com a Bookman Editora pelo endereço [secretariaeditorial@artmed.com.br](mailto:secretariaeditorial@artmed.com.br) e anexar comprovante de docência.

Copyrighted material

---

# Sumário

---

<b>1 Conceitos Básicos de Programação Java .....</b>	<b>23</b>
<b>1.1 Iniciando: classes, tipos e objetos .....</b>	<b>24</b>
1.1.1 Tipos básicos .....	26
1.1.2 Objetos .....	28
1.1.3 Tipos enumerados .....	34
<b>1.2 Métodos .....</b>	<b>34</b>
<b>1.3 Expressões.....</b>	<b>39</b>
1.3.1 Literais .....	39
1.3.3 Conversores e autoboxing/unboxing em expressões .....	42
<b>1.4 Controle de fluxo .....</b>	<b>44</b>
1.4.1 Os comandos if e switch .....	44
1.4.2 Laços .....	46
1.4.3 Expressões explícitas de controle de fluxo .....	48
<b>1.5 Arranjos .....</b>	<b>49</b>
1.5.1 Declarando arranjos .....	51
1.5.2 Arranjos são objetos .....	52
<b>1.6 Entrada e saída simples .....</b>	<b>53</b>
<b>1.7 Um programa de exemplo .....</b>	<b>55</b>
<b>1.8 Classes aninhadas e pacotes .....</b>	<b>58</b>
<b>1.9 Escrevendo um programa em Java .....</b>	<b>59</b>
1.9.1 Projeto .....	60
1.9.2 Pseudocódigo .....	60
1.9.3 Codificação .....	61
1.9.4 Teste e depuração .....	64
<b>1.10 Exercícios .....</b>	<b>66</b>
<b>2 Projeto Orientado a Objetos .....</b>	<b>69</b>
<b>2.1 Objetivos, princípios e padrões .....</b>	<b>70</b>
2.1.1 Objetivos do projeto orientado a objetos .....	70
2.1.2 Princípios de projeto orientado a objetos .....	71
2.1.3 Padrões de projeto .....	73
<b>2.2 Herança e polimorfismo .....</b>	<b>74</b>
2.2.1 Herança .....	74
2.2.2 Polimorfismo .....	75
2.2.3 Usando herança em Java .....	76

<b>2.3 Exceções .....</b>	<b>84</b>
2.3.1 Lançando exceções .....	84
2.3.2 Capturando exceções .....	85
<b>2.4 Interfaces e classes abstratas .....</b>	<b>87</b>
2.4.1 Implementando interfaces .....	87
2.4.2 Herança múltipla e interfaces .....	89
2.4.3 Classes abstratas e tipagem forte .....	90
<b>2.5 Conversão e genéricos .....</b>	<b>91</b>
2.5.1 Conversão .....	91
2.5.2 Genéricos .....	94
<b>2.6 Exercícios .....</b>	<b>96</b>
<b>3 Arranjos, Listas Encadeadas e Recursão.....</b>	<b>101</b>
<b>3.1 Usando arranjos.....</b>	<b>102</b>
3.1.1 Armazenando os registros de um jogo em um arranjo .....	102
3.1.2 Ordenando um arranjo .....	107
3.1.3 Métodos de java.util para arranjos e números aleatórios .....	109
3.1.4 Criptografia simples com strings e arranjos de caracteres .....	111
3.1.5 Arranjos bidimensionais e jogos de posição .....	114
<b>3.2 Listas simplesmente encadeadas .....</b>	<b>117</b>
3.2.1 Inserção em uma lista simplesmente encadeada .....	119
3.2.2 Removendo um elemento em uma lista simplesmente encadeada .....	120
<b>3.3 Listas duplamente encadeadas .....</b>	<b>121</b>
3.3.1 Inserção no meio de uma lista duplamente encadeada .....	123
3.3.2 Remoção do meio de uma lista duplamente encadeada .....	125
3.3.3 Implementação de uma lista duplamente encadeada .....	125
<b>3.4 Listas encadeadas circulares e ordenação de listas encadeadas .....</b>	<b>128</b>
3.4.1 Listas encadeadas circulares e a brincadeira do "Pato, Pato, Ganso" .....	128
3.4.2 Ordenando uma lista encadeada .....	131
<b>3.5 Recursão .....</b>	<b>133</b>
3.5.1 Recursão linear .....	137
3.5.2 Recursão binária .....	141
3.5.3 Recursão múltipla .....	143
<b>3.6 Exercícios .....</b>	<b>145</b>
<b>4 Ferramentas de Análise .....</b>	<b>149</b>
<b>4.1 As sete funções usadas neste livro .....</b>	<b>150</b>
4.1.1 A função constante .....	150
4.1.2 A função logaritmo .....	150
4.1.3 A função linear .....	151
4.1.4 A função n-log-n .....	151

4.1.5 A função quadrática . . . . .	152
4.1.6 A função cúbica e outras polinomiais . . . . .	152
4.1.7 A função exponencial . . . . .	154
4.1.8 Comparando taxas de crescimento . . . . .	155
<b>4.2 Análise de algoritmos . . . . .</b>	<b>156</b>
4.2.1 Estudos experimentais . . . . .	157
4.2.2 Operações primitivas . . . . .	158
4.2.3 Notação assintótica . . . . .	159
4.2.4 Análise assintótica . . . . .	163
4.2.5 Usando a notação $O$ . . . . .	164
4.2.6 Um algoritmo recursivo para calcular potência . . . . .	167
<b>4.3 Técnicas simples de justificativa . . . . .</b>	<b>168</b>
4.3.1 Através de exemplos . . . . .	168
4.3.2 O ataque “contra” . . . . .	168
4.3.3 Indução e invariantes em laços . . . . .	169
<b>4.4 Exercícios . . . . .</b>	<b>171</b>
 <b>5 Pilhas e Filas . . . . .</b>	<b>177</b>
<b>5.1 Pilhas . . . . .</b>	<b>178</b>
5.1.1 O tipo abstrato de dados pilha . . . . .	178
5.1.2 Uma implementação baseada em arranjos . . . . .	181
5.1.3 Implementando uma pilha usando uma lista encadeada genérica . . . . .	185
5.1.4 Invertendo um arranjo usando uma pilha . . . . .	187
5.1.5 Verificando parênteses e tags HTML . . . . .	188
<b>5.2 Filas . . . . .</b>	<b>191</b>
5.2.1 O tipo abstrato de dados fila . . . . .	191
5.2.2 Uma implementação simples baseada em arranjos . . . . .	193
5.2.3 Implementando uma fila usando uma lista encadeada genérica . . . . .	195
5.2.4 Escalonadores round-robin . . . . .	196
<b>5.3 Filas com dois finais . . . . .</b>	<b>198</b>
5.3.1 O tipo abstrato de dados deque . . . . .	198
5.3.2 Implementando um deque . . . . .	199
<b>5.4 Exercícios . . . . .</b>	<b>201</b>
 <b>6 Listas e Iteradores . . . . .</b>	<b>207</b>
<b>6.1 Listas arranjo . . . . .</b>	<b>208</b>
6.1.1 O tipo abstrato de dados lista arranjo . . . . .	208
6.1.2 O Padrão adaptador . . . . .	209
6.1.3 Uma implementação simples usando arranjo . . . . .	209
6.1.4 A interface simples e a classe Java.util.ArrayList . . . . .	211
6.1.5 Implementando uma lista arranjo usando arranjos extensíveis . . . . .	212

<b>6.2 Listas de nodos .....</b>	<b>215</b>
6.2.1 Operações baseadas em nodos .....	215
6.2.2 Posições .....	216
6.2.3 O tipo abstrato de dados lista de nodos .....	216
6.2.4 Implementação usando lista duplamente encadeada.....	219
<b>6.3 Iteradores .....</b>	<b>224</b>
6.3.1 Os tipos abstratos de dados iterador e iterável.....	224
6.3.2 O laço de Java para-cada .....	226
6.3.3 Implementando iteradores.....	226
6.3.4 Iteradores de lista em Java .....	228
<b>6.4 Os TADs de lista e o framework de coleções .....</b>	<b>229</b>
6.4.1 O framework de coleções do Java .....	229
6.4.2 A classe <code>java.util.LinkedList</code> .....	231
6.4.3 Seqüências .....	231
<b>6.5 Estudo de caso: a heurística mover-para-frente .....</b>	<b>233</b>
6.5.1 Usando uma lista ordenada e uma classe aninhada.....	233
6.5.2 Usando uma lista com a heurística mover-para-frente .....	235
6.5.3 Possíveis usos de uma lista de favoritos.....	236
<b>6.6 Exercícios .....</b>	<b>238</b>
<b>7 Árvores.....</b>	<b>245</b>
<b>7.1 Árvores genéricas .....</b>	<b>246</b>
7.1.1 Definição de árvore e propriedades.....	247
7.1.2 O tipo abstrato de dados árvore .....	249
7.1.3 Implementando uma árvore.....	250
<b>7.2 Algoritmos de caminhamento em árvores .....</b>	<b>251</b>
7.2.1 Altura e profundidade .....	252
7.2.2 Caminhamento prefixado .....	254
7.2.3 Caminhamento pós-fixado .....	256
<b>7.3 Árvores binárias.....</b>	<b>259</b>
7.3.1 O TAD árvore binária .....	260
7.3.2 Uma interface de árvore binária em Java .....	261
7.3.3 Propriedades de árvores binárias .....	261
7.3.4 Estruturas encadeadas para árvores binárias .....	263
7.3.5 Uma estrutura baseada em lista arranjo para árvore binária.....	270
7.3.6 Caminhamentos sobre árvores binárias .....	272
7.3.7 O padrão do método modelo .....	278
<b>7.4 Exercícios .....</b>	<b>281</b>
<b>8 Fila de Prioridade .....</b>	<b>291</b>
<b>8.1 O tipo abstrato de dados fila de prioridade.....</b>	<b>292</b>
8.1.1 Chaves, prioridades e relações de ordem total .....	292
8.1.2 Entradas e comparadores .....	293

8.1.3	O TAD fila de prioridade . . . . .	295
8.1.4	Ordenando com uma fila de prioridade . . . . .	297
<b>8.2</b>	<b>Implementando uma fila de prioridade com seqüências . . . . .</b>	<b>298</b>
8.2.1	Implementação com uma seqüência não-ordenada . . . . .	298
8.2.2	Implementação com uma seqüência ordenada . . . . .	299
8.2.3	Selection sort e insertion sort . . . . .	301
<b>8.3</b>	<b>Heaps . . . . .</b>	<b>303</b>
8.3.1	A estrutura de dados heap . . . . .	303
8.3.2	Árvores binárias completas e suas representações . . . . .	305
8.3.3	Implementando uma fila de prioridade com um heap . . . . .	309
8.3.4	Implementação em Java . . . . .	313
8.3.5	Heap-sort . . . . .	316
8.3.6	Construção bottom-up do heap ★ . . . . .	317
<b>8.4</b>	<b>Filas de prioridade adaptáveis . . . . .</b>	<b>320</b>
8.4.1	Métodos do TAD fila de prioridade adaptável . . . . .	320
8.4.2	Localizadores . . . . .	321
8.4.3	Implementando uma fila de prioridade adaptável . . . . .	322
<b>8.5</b>	<b>Exercícios . . . . .</b>	<b>324</b>
<b>9</b>	<b>Mapas e Dicionários . . . . .</b>	<b>331</b>
<b>9.1</b>	<b>O tipo abstrato de dados mapa . . . . .</b>	<b>332</b>
9.1.1	Uma implementação simples de mapa . . . . .	334
<b>9.2</b>	<b>Tabelas de hash . . . . .</b>	<b>335</b>
9.2.1	Arranjo de buckets . . . . .	335
9.2.2	Funções de hash . . . . .	336
9.2.3	Códigos de hash . . . . .	336
9.2.4	Funções de compressão . . . . .	339
9.2.5	Esquema para tratamento de colisões . . . . .	340
9.2.6	Uma implementação Java para tabelas de hash . . . . .	344
9.2.7	Fatores de carga e rehashing . . . . .	347
9.2.8	Aplicação: contador de freqüência de palavras . . . . .	348
<b>9.3</b>	<b>O TAD dicionário . . . . .</b>	<b>348</b>
9.3.1	Dicionários baseados em seqüências e auditorias . . . . .	350
9.3.2	Implementação de um dicionário com tabela de hash . . . . .	352
9.3.3	Tabelas de pesquisa ordenada e pesquisa binária . . . . .	352
<b>9.4</b>	<b>Skip list . . . . .</b>	<b>356</b>
9.4.1	Pesquisa e alteração em uma skip list . . . . .	357
9.4.2	Uma análise probabilística das skip lists ★ . . . . .	361
<b>9.5</b>	<b>Extensões e aplicações de dicionários . . . . .</b>	<b>363</b>
9.5.1	Suporando localizadores em um dicionário . . . . .	363
9.5.2	O TAD dicionário ordenado . . . . .	364
9.5.3	Banco de dados de vôos e conjuntos máximos . . . . .	364
<b>9.6</b>	<b>Exercícios . . . . .</b>	<b>367</b>

<b>10 Árvores de Pesquisa.....</b>	<b>373</b>
<b>    10.1 Árvores binárias de pesquisa.....</b>	<b>374</b>
10.1.1 Pesquisa .....	374
10.1.2 Operações de atualização.....	376
10.1.3 Implementação Java .....	379
<b>    10.2 Árvores AVL .....</b>	<b>383</b>
10.2.1 Operações de atualização.....	385
10.2.2 Implementação Java .....	388
<b>    10.3 Árvores splay .....</b>	<b>392</b>
10.3.1 Espalhamento .....	392
10.3.2 Quando espalhar.....	393
10.3.3 Análise amortizada do espalhamento ★ .....	396
<b>    10.4 Árvores (2,4) .....</b>	<b>401</b>
10.4.1 Árvore genérica de pesquisa.....	401
10.4.2 Operações de atualização em árvores (2,4) .....	405
<b>    10.5 Árvores vermelho-pretas.....</b>	<b>410</b>
10.5.1 Operações de atualização.....	412
10.5.2 Implementação Java .....	420
<b>    10.6 Exercícios.....</b>	<b>425</b>
<b>11 Ordenação, Conjuntos e Seleção.....</b>	<b>431</b>
<b>    11.1 Merge-sort .....</b>	<b>432</b>
11.1.1 Divisão e conquista .....	432
11.1.2 Junção de arranjos e listas .....	433
11.1.3 O tempo de execução do merge-sort .....	438
11.1.4 Implementações Java do merge-sort .....	439
11.1.5 O merge-sort e suas relações de recomendação ★ .....	441
<b>    11.2 Quick-sort.....</b>	<b>442</b>
11.2.1 Quick-sort randômico .....	448
11.2.2 Quick-sort in-place .....	450
<b>    11.3 Um limite inferior para ordenação .....</b>	<b>452</b>
<b>    11.4 Bucket-sort e radix-sort .....</b>	<b>453</b>
11.4.1 Bucket-sort .....	454
11.4.2 Radix-sort .....	455
<b>    11.5 Comparando algoritmos de ordenação.....</b>	<b>456</b>
<b>    11.6 O TAD conjunto e estruturas union/find .....</b>	<b>458</b>
11.6.1 Uma simples implementação de conjunto .....	458
11.6.2 Partições com operações de union-find .....	461
11.6.3 Uma implementação de partição baseada em árvore ★ .....	462
<b>    11.7 Seleção .....</b>	<b>465</b>
11.7.1 Poda e busca .....	465
11.7.2 Quick-select randômico .....	466
11.7.3 Analisando o quick-select randômico .....	467
<b>    11.8 Exercícios.....</b>	<b>468</b>

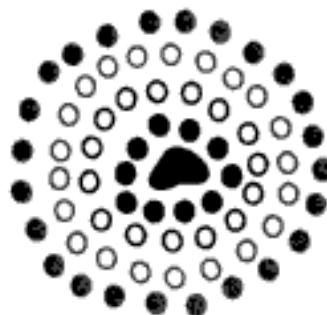
<b>12 Processamento de Texto .....</b>	<b>475</b>
<b>12.1 Operações com strings.....</b>	<b>476</b>
12.1.1 A classe Java String .....	477
12.1.2 A classe Java StringBuffer.....	477
<b>12.2 Algoritmos para procura de padrões.....</b>	<b>478</b>
12.2.1 Força bruta .....	478
12.2.2. O algoritmo Boyer-Moore .....	480
12.2.3 O algoritmo de Knuth-Morris-Pratt .....	483
<b>12.3 Tries.....</b>	<b>487</b>
12.3.1 Tries-padrão.....	487
12.3.2 Tries comprimidos .....	489
12.3.3 Tries de sufixos .....	492
12.3.4 Mecanismos de busca	495
<b>12.4 Compressão de textos .....</b>	<b>495</b>
12.4.1 O algoritmo de codificação de Huffman .....	497
12.4.2 O método guloso.....	497
<b>12.5 Testando a similaridade de textos .....</b>	<b>498</b>
12.5.1 O problema da maior subseqüência comum .....	498
12.5.2 Programação dinâmica .....	499
12.5.3 Aplicando programação dinâmica ao problema da LCS.....	499
<b>12.6 Exercícios.....</b>	<b>502</b>
<b>13 Grafos.....</b>	<b>507</b>
<b>13.1 O tipo abstrato de dados grafo.....</b>	<b>508</b>
13.1.1 O TAD grafo .....	512
<b>13.2 Estruturas de dados para grafos .....</b>	<b>512</b>
13.2.1 A lista de arestas .....	512
13.2.2 A lista de adjacências .....	515
13.2.3 A matriz de adjacência .....	516
<b>13.3 Caminhamento em grafos.....</b>	<b>518</b>
13.3.1 Pesquisa em profundidade .....	518
13.3.2 Implementando a pesquisa em profundidade.....	522
13.3.3 Caminhamento em largura .....	528
<b>13.4 Grafos dirigidos .....</b>	<b>531</b>
13.4.1 Caminhamento em um dígrafo .....	532
13.4.2 Fechamento transitivo .....	535
13.4.3 Grafos acíclicos dirigidos .....	536
<b>13.5 Grafos ponderados .....</b>	<b>539</b>
<b>13.6 Caminhos mínimos .....</b>	<b>541</b>
13.6.1 O algoritmo de Dijkstra .....	542
<b>13.7 Árvores de cobertura mínima.....</b>	<b>549</b>
13.7.1 Algoritmo de Kruskal .....	551
13.7.2 O algoritmo Prim-Jarnik.....	554
<b>13.8 Exercícios.....</b>	<b>556</b>

<b>14 Memória .....</b>	<b>567</b>
<b>14.1 Gerenciamento de Memória .....</b>	<b>568</b>
14.1.1 Pilhas na máquina virtual de Java .....	568
14.1.2 Alocando espaço na memória heap .....	571
14.1.3 Coleta de lixo .....	572
<b>14.2 Memória externa e caching .....</b>	<b>574</b>
14.2.1 A hierarquia de memória .....	574
14.2.2 Estratégias de cache .....	575
<b>14.3 Pesquisa externa e árvores-B.....</b>	<b>579</b>
14.3.1 Árvores (a,b).....	580
14.3.2 Árvores-B.....	581
<b>14.4 Ordenando memória externa.....</b>	<b>582</b>
14.4.1 Merge genérico .....	583
<b>14.5 Exercícios .....</b>	<b>584</b>
<b>A Fatos Matemáticos Úteis .....</b>	<b>587</b>
<b>Bibliografia.....</b>	<b>593</b>
<b>Índice .....</b>	<b>597</b>

# 1

# Conceitos Básicos de Programação Java

---



## Conteúdo

---

<b>1.1 Iniciando: classes, tipos e objetos . . . . .</b>	<b>24</b>
1.1.1 Tipos básicos . . . . .	26
1.1.2 Objetos . . . . .	28
1.1.3 Tipos enumerados. . . . .	34
<b>1.2 Métodos . . . . .</b>	<b>34</b>
<b>1.3 Expressões . . . . .</b>	<b>39</b>
1.3.1 Literais . . . . .	39
1.3.3 Conversores e autoboxing/unboxing em expressões . . . . .	42
<b>1.4 Controle de fluxo . . . . .</b>	<b>44</b>
1.4.1 Os comandos if e switch . . . . .	44
1.4.2 Laços . . . . .	46
1.4.3 Expressões explícitas de controle de fluxo . . . . .	48
<b>1.5 Arranjos . . . . .</b>	<b>49</b>
1.5.1 Declarando arranjos . . . . .	51
1.5.2 Arranjos são objetos . . . . .	52
<b>1.6 Entrada e saída simples. . . . .</b>	<b>53</b>
<b>1.7 Um programa de exemplo . . . . .</b>	<b>55</b>
<b>1.8 Classes aninhadas e pacotes . . . . .</b>	<b>58</b>
<b>1.9 Escrevendo um programa em Java . . . . .</b>	<b>59</b>
1.9.1 Projeto . . . . .	60
1.9.2 Pseudocódigo . . . . .	60
1.9.3 Codificação . . . . .	61
1.9.4 Teste e depuração . . . . .	64
<b>1.10 Exercícios . . . . .</b>	<b>66</b>

## 1.1 Iniciando: classes, tipos e objetos

Construir estruturas de dados e algoritmos requer a comunicação de instruções detalhadas para um computador. Uma excelente maneira de fazer isso é usar uma linguagem de programação de alto nível tal como Java. Este capítulo apresenta uma visão geral da linguagem Java assumindo que o leitor esteja familiarizado com alguma linguagem de programação de alto nível. Este livro, entretanto, não provê uma descrição completa da linguagem Java. Existem aspectos importantes da linguagem que não são relevantes para o projeto de estruturas de dados e que não são incluídos aqui, tais como threads e sockets. O leitor que desejar aprender mais sobre Java deve observar as notas do final deste capítulo. Iniciamos com um programa que imprime "Hello Universe!" na tela, que é mostrado e dissecado na Figura 1.1.

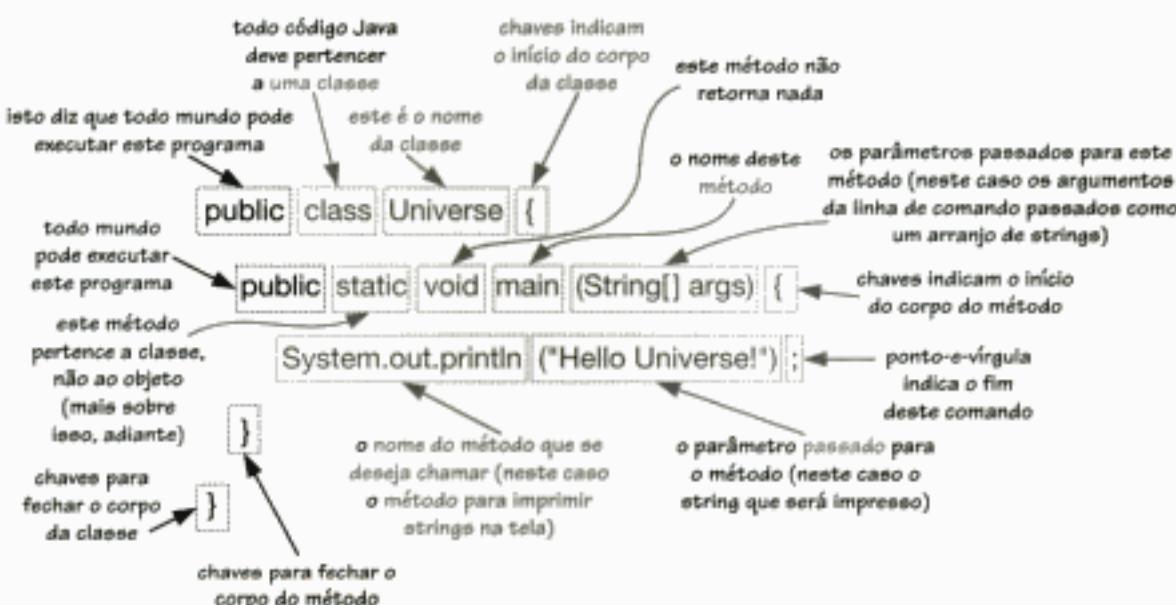


Figura 1.1 O programa "Hello Universe!"

Os principais “atores” em um programa Java são os *objetos*. Os objetos armazenam dados e fornecem os métodos para acessar e modificar esses dados. Todo objeto é instância de uma **classe** que define o *tipo* do objeto, bem como os tipos de operações que executa. Os **membros** críticos de uma classe Java são os seguintes (classes também podem conter definições de classes aninhadas, mas essa é uma discussão para mais tarde):

- Dados de objetos Java são armazenados em *variáveis de instância* (também chamadas de *campos*). Por essa razão, se um objeto de uma classe deve armazenar dados, então sua classe deve especificar variáveis de instância para esse fim. As variáveis de instância podem ser de tipos básicos (tais como inteiros, números de ponto flutuante ou booleanos) ou podem se referir a objetos de outras classes.
- As operações que podem atuar sobre os dados e que expressam as “mensagens” às quais os objetos respondem são chamadas de *métodos*, e estes consistem de construtores, subprogramas e funções. Eles definem o comportamento dos objetos daquela classe.

### Como as classes são declaradas

Resumindo, um objeto é uma combinação específica de dados e dos métodos capazes de processar e comunicar esses dados. As classes definem os *tipos* dos objetos; por essa razão, objetos são também chamados de instâncias da classe que os define, e usam o nome da classe como seu tipo.

Um exemplo de definição de uma classe Java é apresentado no Trecho de código 1.1.

```

public class Counter {
    protected int count; // uma simples variável de instância inteira
    /** O construtor default para um objeto Counter */
    Counter() { count = 0; }
    /** Um método de acesso para recuperar o valor corrente do contador */
    public int getCount() { return count; }
    /** Um método modificador para incrementar o contador */
    public void incrementCount() { count++; }
    /** Um método modificador para decrementar o contador */
    public void decrementCount() { count--; }
}

```

**Trecho de código 1.1** A classe Counter para um contador simples que pode ser acessado, incrementado e decrementado.

Neste exemplo, observa-se que a definição da classe está delimitada por chaves, isto é, começa por um “{” e termina com um “}”. Em Java, qualquer conjunto de comandos entre chaves “{” e “}” define um **bloco** de programa.

Assim como a classe Universe, a classe Counter é pública, o que significa que qualquer outra classe pode criar e usar um objeto Counter. O Counter tem uma variável de instância – um inteiro chamado count. Esta variável é inicializada com zero no método construtor, Counter, que é chamado quando se deseja criar um novo objeto Counter (este método sempre tem o mesmo nome que a classe a qual pertence). Esta classe também tem um método de acesso, getCount, que retorna o valor corrente do contador. Finalmente, esta classe tem dois métodos de atualização – o método incrementCount, que incrementa o contador, e o método decrementCount, que decrementa o contador. Na verdade, esta é uma classe extremamente aborrecida, mas pelo menos mostra a sintaxe e a estrutura de uma classe Java. Mostra também que uma classe Java não precisa ter um método chamado main (mas tal classe não consegue fazer nada sozinha).

O nome da classe, método ou variável em Java é chamado de **identificador**, e pode ser qualquer string de caracteres desde que inicie por uma letra e seja composto por letras, números e caracteres sublinhados (onde “letra” e “número” podem ser de qualquer língua escrita definida no conjunto de caracteres Unicode). Listam-se as exceções a esta regra geral para identificadores Java na Tabela 1.1.

Palavras reservadas			
abstract	else	interface	switch
boolean	extends	long	synchronized
break	false	native	this
byte	final	new	throw
case	finally	null	throws
catch	float	package	transient
char	for	private	true
class	goto	protected	try
const	if	public	void
continue	implements	return	volatile
default	import	short	while
do	instanceof	static	
double	int	super	

**Tabela 1.1** Lista de palavras reservadas Java. Estas palavras não podem ser usadas como nomes de variáveis ou de métodos em Java.

## Modificadores de classes

Os modificadores de classes são palavras reservadas opcionais que precedem a palavra reservada **class**. Até agora, foram vistos exemplos que usavam a palavra reservada **public**. Em geral, os diferentes modificadores de classes e seu significado são os que seguem:

- O modificador de classe **abstract** descreve uma classe que possui métodos abstratos. Muitos abstratos são declarados com a palavra reservada **abstract** e são vazios (isto é, não possuem um bloco de comandos definindo o código do método). Se uma classe tem apenas métodos abstratos e nenhuma variável de instância, é mais adequado considerá-la uma interface (ver Seção 2.4), de forma que uma classe **abstract** é normalmente uma mistura de métodos abstratos e métodos verdadeiros. (Discutem-se classes abstratas e seus usos na Seção 2.4).
- O modificador de classe **final** descreve uma classe que não pode ter subclasses. (Discute-se esse conceito no próximo capítulo).
- O modificador de classe **public** descreve uma classe que pode ser instanciada ou estendida por qualquer coisa definida no mesmo pacote ou por qualquer coisa que *importe* a classe. (Isso é melhor detalhado na Seção 1.8.) Todas as classes públicas são declaradas em arquivo próprio exclusivo nomeado *classname.java*, onde “*classname*” é o nome da classe.
- Se o modificador de classe **public** não é usado, então a classe é considerada **amigável**. Isso significa que pode ser usada e instanciada por qualquer classe do mesmo *pacote*. Esse é o modificador de classe default.

---

### 1.1.1 Tipos básicos

Os tipos dos objetos são determinados pela classe de origem. Em nome da eficiência e da simplicidade, Java ainda oferece os seguintes *tipos básicos* (também chamados de *tipos primitivos*) que não são objetos:

<b>boolean</b>	valor booleano: true ou false
<b>char</b>	caráter Unicode de 16 bits
<b>byte</b>	inteiro com sinal em complemento de dois de 8 bits
<b>short</b>	inteiro com sinal em complemento de dois de 16 bits
<b>int</b>	inteiro com sinal em complemento de dois de 32 bits
<b>long</b>	inteiro com sinal em complemento de dois de 64 bits
<b>float</b>	número de ponto flutuante de 32 bits (IEEE 754-1985)
<b>double</b>	número de ponto flutuante de 64 bits (IEEE 754-1985)

Uma variável declarada como tendo um desses tipos simplesmente armazena um valor deste tipo, em vez de uma referência para um objeto. Constantes inteiras, tais como 14 ou 195, são do tipo **int**, a menos que seguidas de imediato por um “L” ou “l”, sendo, neste caso, do tipo **long**. Constantes de ponto flutuante, como 3.1415 ou 2.158e5, são do tipo **double**, a menos que seguidas de imediato por um ‘F’ ou um ‘f’, sendo, neste caso, do tipo **float**. O Trecho de código 1.2 apresenta uma classe simples que define algumas variáveis locais de tipos básicos no método main.

```
public class Base {
    public static void main (String[ ] args) {
        boolean flag = true;
        char ch = 'A';
        byte b = 12;
        short s = 24;
        int i = 257;
```

```

long l = 890L;           // Observar o uso do "L" aqui
float f = 3.1415F;      // Observar o uso do "F" aqui
double d = 2.1828;
System.out.println ("flag = " + flag); // o "+" indica concatenação de strings
System.out.println ("ch = " + ch);
System.out.println ("b = " + b);
System.out.println ("s = " + s);
System.out.println ("i = " + i);
System.out.println ("l = " + l);
System.out.println ("f = " + f);
System.out.println ("d = " + d);
}
}

```

**Trecho de código 1.2** A classe Base mostrando o uso dos tipos básicos.

### Comentários

Observar o uso de comentários neste e nos outros exemplos. Os comentários são anotações para uso de humanos, e não são processadas pelo compilador Java. Java permite dois tipos de comentários – comentários de bloco e comentários de linha – usados para definir o texto a ser ignorado pelo compilador. Em Java, usa-se um `/*` para começar um bloco de comentário e um `*/` para fechá-lo. Deve-se destacar os comentários iniciados por `/**`, pois tais comentários tem um formato especial que permite que um programa chamado Javadoc os leia e automaticamente gere documentação para programas Java. A sintaxe e interpretação dos comentários Javadoc será discutida na Seção 1.9.3

Além de comentários de bloco, Java usa o `//` para começar comentários de linha e ignorar tudo mais naquela linha. Por exemplo:

```

/*
 * Este é um bloco de comentário
 */
// Este é um comentário de linha

```

### Saída da classe Base

A saída resultante da execução da classe Base (método main) é mostrada na Figura 1.2.

```

flag = true
ch = A
b = 12
s = 24
i = 257
l = 890
f = 3.1415
d = 2.1828

```

**Figura 1.2** Saída da classe Base.

Mesmo não se referindo a objetos, variáveis dos tipos básicos são úteis no contexto de objetos, na medida em que são usadas para definir variáveis de instâncias (ou campos) dentro de um objeto. Por exemplo, a classe Counter (Trecho de código 1.1) possui uma única variável de instância do tipo `int`. Uma outra característica adicional de Java é o fato de que variáveis de instância

sempre recebem um valor inicial quando o objeto que as contém é criado (seja zero, falso ou um caractere nulo, dependendo do tipo).

### 1.1.2 Objetos

Em Java, um objeto novo é criado a partir de uma classe usando-se o operador **new**. O operador **new** cria um novo objeto a partir de uma classe especificada e retorna uma **referência** para este objeto. Para criar um objeto de um tipo específico, deve-se seguir o uso do operador **new** por uma chamada a um construtor daquele tipo de objeto. Pode-se usar qualquer construtor que faça parte da definição da classe, incluindo o construtor default (que não recebe argumentos entre os parênteses). Na Figura 1.3, apresentam-se vários exemplos de uso do operador **new** que criam novos objetos e atribuem uma referência para os mesmos a uma variável.

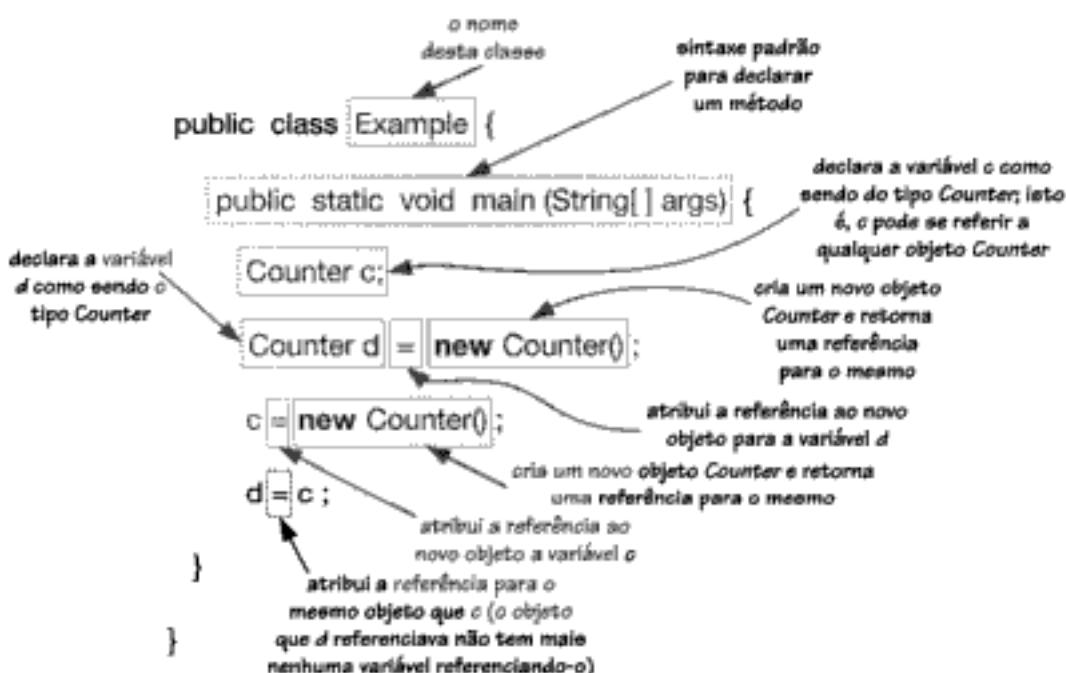


Figura 1.3 Exemplos de uso do operador **new**.

A chamada do operador **new** sobre um tipo de classe faz com que ocorram três eventos:

- Um novo objeto é dinamicamente alocado na memória, e todas as variáveis de instância são inicializadas com seus valores padrão. Os valores padrão são **null** para variáveis objeto e 0 para todos os tipos base, exceto as variáveis **boolean** (que são **false** por default).
- O construtor para o novo objeto é chamado com os parâmetros especificados. O construtor atribui valores significativos para as variáveis de instância e executa as computações adicionais que devam ser feitas para criar este objeto.
- Depois do construtor retornar, o operador **new** retorna uma referência (isto é, um endereço de memória) para o novo objeto recém criado. Se a expressão está na forma de uma atribuição, então este endereço é armazenado na variável objeto, e então a variável objeto passa a **referir** o objeto recém criado.

### Objetos numéricos

Às vezes, quer-se armazenar números como objetos, mas os tipos básicos não são objetos, como já se observou. Para contornar esse problema, Java define uma classe especial para cada tipo bá-

sico numérico. Essas classes são chamadas de *classes numéricas*. Na Tabela 1.2, estão os tipos básicos numéricos e as classes numéricas correspondentes, juntamente com exemplos de como se criam e se acessam os objetos numéricos. Desde o Java 5.0, a operação de criação é executada automaticamente sempre que se passa um número básico para um método que esteja esperando o objeto correspondente. Da mesma forma, o método de acesso correspondente é executado automaticamente sempre que se deseja atribuir o valor do objeto Número correspondente a um tipo numérico básico.

<i>Tipo base</i>	<i>Nome da classe</i>	<i>Exemplo de criação</i>	<i>Exemplo de acesso</i>
<b>byte</b>	Byte	n = new Byte((byte)34)	n.byteValue()
<b>short</b>	Short	n = new Short((short)100)	n.shortValue()
<b>int</b>	Integer	n = new Integer(1045)	n.intValue()
<b>long</b>	Long	n = new Long(10849L)	n.longValue()
<b>float</b>	Float	n = new Float(3.934F)	n.floatValue()
<b>double</b>	Double	n = new Double(3.934)	n.doubleValue()

**Tabela 1.2** Classes numéricas de Java. Para cada classe é fornecido o tipo básico correspondente e expressões exemplificadoras de criação e acesso a esses objetos. Em cada linha, se admite que a variável n é declarada com o nome de classe correspondente.

## Objetos string

Uma string é uma seqüência de caracteres que provêm de algum *alfabeto* (conjunto de todos os *caracteres* possíveis). Cada caractere c que compõe uma string s pode ser referenciado por seu índice na string, a qual é igual ao número de caracteres que vem antes de c em s (desta forma, o primeiro caractere tem índice 0). Em Java, o alfabeto usado para definir strings é o conjunto internacional de caracteres Unicode, um padrão de codificação de caracteres de 16 bits que cobre as línguas escritas mais usadas. Outras linguagens de programação tendem a usar o conjunto de caracteres ASCII, que é menor (corresponde a um subconjunto do alfabeto Unicode baseado em um padrão de codificação de 7 bits). Além disso, Java define uma classe especial embutida de objetos chamados objetos String.

Por exemplo, P pode ser

"hogs and dogs"

que tem comprimento 13 e pode ter vindo da página Web de alguém. Neste caso, o caractere de índice 2 é 'g' e o caractere de índice 5 é 'a'. Por outro lado, P poderia ser a string "CGTAATAG-TTAATCCG", que tem comprimento 16 e pode ser proveniente de uma aplicação científica de seqüenciamento de DNA, onde o alfabeto é {G, C, A, T}.

## Concatenação

O processamento de strings implica em lidar com strings. A operação básica para combinar strings chama-se *concatenação*, a qual toma uma string P e uma string Q e as combina em uma nova string denotada P+Q, que consiste de todos os caracteres de P seguidos por todos os caracteres de Q. Em Java, o operador "+" age exatamente desta maneira quando aplicado sobre duas strings. Sendo assim, em Java é válido (e muito útil) escrever uma declaração de atribuição do tipo:

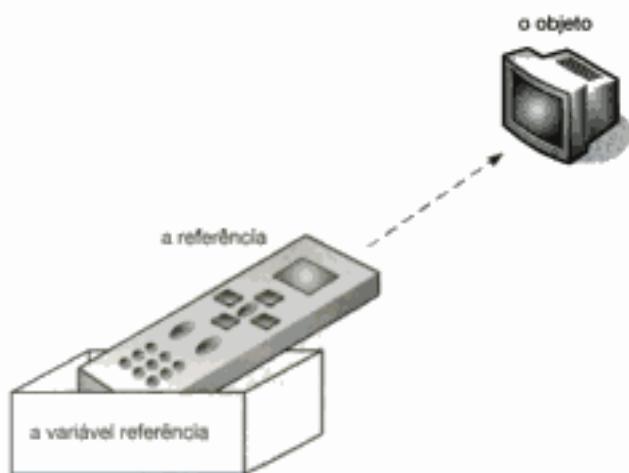
```
String s = "kilo" + "meters";
```

Essa declaração define uma variável s que referencia objetos da classe String e lhe atribui a string "kilometers". (Mais adiante, neste capítulo, serão discutidos mais detalhadamente comandos de atribuição e expressões como a apresentada). Pressupõe-se ainda que todo objeto

Java tem um método predefinido chamado `toString()` que retorna a string associada ao objeto. Esta descrição da classe `String` deve ser suficiente para a maioria dos usos. Analisaremos a classe `String` e sua “parente”, a classe `StringBuffer`, na Seção 12.1.

## Referências para objetos

Como mencionado acima, a criação de um objeto novo envolve o uso do operador `new` para alojar espaço em memória para o objeto e usar o construtor do objeto para inicializar esse espaço. A localização ou *endereço* deste espaço normalmente é atribuída para uma variável **referência**. Consequentemente, uma variável referência pode ser entendida como sendo um “ponteiro” para um objeto. Isso é como se a variável fosse o suporte de um controle remoto que pudesse ser usado para controlar o objeto recém-criado (o dispositivo). Ou seja, a variável tem uma maneira de apontar para o objeto e solicitar que o mesmo faça coisas ou acessar seus dados. Este conceito pode ser visto na Figura 1.4.



**Figura 1.4** Demonstrando o relacionamento entre objetos e variáveis referência. Quando se atribui uma referência para um objeto (isto é, um endereço de memória) para uma variável referência, é como se fosse armazenado um controle remoto do objeto naquela variável.

## O operador ponto

Toda a variável referência para objeto deve referir algum objeto, a menos que seja `null`, caso em que não aponta para nada. Seguindo com a analogia do controle remoto, uma referência `null` é um suporte de controle remoto vazio. Inicialmente, a menos que se faça a variável referência apontar para alguma coisa através de uma atribuição, ela é `null`.

Pode haver, na verdade, várias referências para um mesmo objeto, e cada referência para um objeto específico pode ser usada para chamar métodos daquele objeto. Esta situação corresponde a existirem vários controles remotos capazes de atuar sobre o mesmo dispositivo. Qualquer um dos controles pode ser usado para fazer alterações no dispositivo (como alterar o canal da televisão). Observe que se um controle remoto é usado para alterar o dispositivo, então o (único) objeto apontado por todos os controles se altera. Da mesma forma, se uma variável referência for usada para alterar o estado do objeto, então seu estado muda para todas as suas referências. Este comportamento vem do fato de que são muitas referências, mas todas apontando para o mesmo objeto.

Um dos principais usos de uma variável referência é acessar os membros da classe a qual pertence o objeto, a instância da classe. Ou seja, uma variável referência é útil para acessar os métodos e as variáveis de instância associadas com um objeto. Este acesso é feito através do operador ponto (“.”). Chama-se um método associado com um objeto usando o nome da variável referência seguido do operador ponto, e então o nome do método e seus parâmetros.

Isso ativa o método com o nome especificado associado ao objeto referenciado pela variável referência. Opcionalmente, podem ser passados vários parâmetros. Se existirem vários métodos com o mesmo nome definido para este objeto, então a máquina de execução do Java irá usar aquele cujo número de parâmetros e tipos melhor combinem. O nome de um método combinado com a quantidade e o tipo de seus parâmetros chama-se de **assinatura** do método, uma vez que todas essas partes são usadas para determinar o método correto para executar uma certa chamada de método. Considerem-se os seguintes exemplos:

```
oven.cookDinner();
oven.cookDinner(food);
oven.cookDinner(food, seasoning);
```

Cada uma dessas chamadas se refere, na verdade, a métodos diferentes, definidos com o mesmo nome na classe a qual pertencem. Observa-se, entretanto, que a assinatura de um método em Java não inclui o tipo de retorno do método, de maneira que Java não permite que dois métodos com a mesma assinatura retornem tipos diferentes.

### Variáveis de instância

Classes Java podem definir **variáveis de instância**, também chamadas de **campos**. Essas variáveis representam os dados associados com os objetos de uma classe. As variáveis de instância devem ter um **tipo**, que pode tanto ser um **tipo básico** (como **int**, **float**, **double**) ou um **tipo referência** (como na analogia do controle remoto), isto é, uma classe, como **String**, uma interface (ver Seção 2.4) ou um arranjo (ver Seção 1.5). Uma instância de variável de um tipo básico armazena um valor do tipo básico, enquanto que variáveis de instância, declaradas usando-se um nome de classe, armazenam uma **referência** para um objeto daquela classe.

Continuando com a analogia entre variáveis referência e controles remotos, variáveis de instância são como parâmetros do dispositivo que podem tanto ser lidos, como alterados usando-se o controle remoto (tais como os controles de volume e canal do controle remoto de uma televisão). Dada uma variável referência *v*, que aponta para um objeto *o*, pode-se acessar qualquer uma das variáveis de instância de *o* que as regras de acesso permitirem. Por exemplo, variáveis de instância **públicas** podem ser acessadas por qualquer pessoa. Usando o operador ponto, pode-se **obter** o valor de qualquer variável de instância, *i*, usando-se *v.i* em uma expressão aritmética. Da mesma forma pode-se **alterar** o valor de qualquer variável de instância *i*, escrevendo *v.i* no lado esquerdo do operador de atribuição ("="). (Ver Figura 1.5.) Por exemplo, se **gnome** se refere a um objeto **Gnome** que tem as variáveis de instância públicas **name** e **age**, então os seguintes comandos são possíveis:

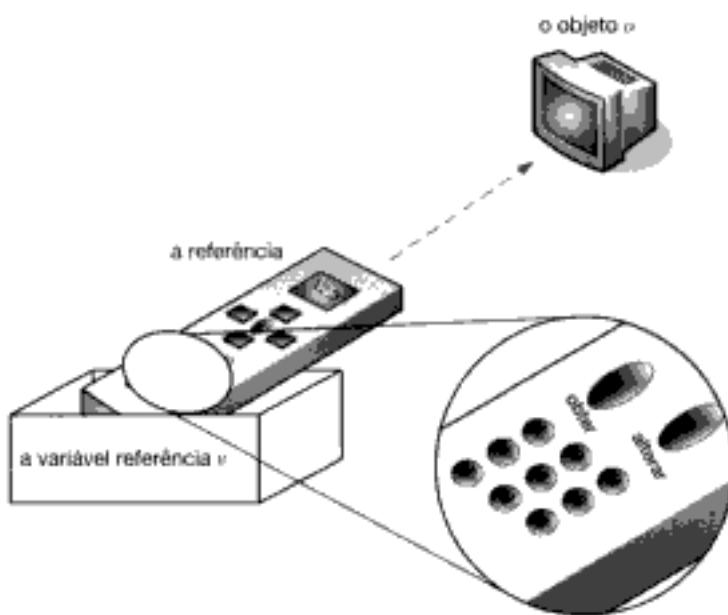
```
gnome.name = "ProfessorSmythe";
gnome.age = 132;
```

Entretanto, uma referência para objeto não tem de ser apenas uma variável referência. Pode ser qualquer expressão que retorna uma referência para objeto.

### Modificadores de variáveis

Em alguns casos, o acesso direto a uma variável de instância de um objeto pode não estar habilitado. Por exemplo, uma variável de instância declarada como **privada** em alguma classe só pode ser acessada pelos métodos definidos dentro da classe. Tais variáveis de instância são parecidas com parâmetros de dispositivo que não podem ser acessados diretamente pelo controle remoto. Por exemplo, alguns dispositivos tem parâmetros internos que só podem ser lidos ou alterados por técnicos da fábrica (e o usuário não está autorizado a alterá-los sem violar a garantia do dispositivo).

Quando se declara uma variável de instância, pode-se, opcionalmente, definir um modificador de variável, seguido pelo tipo e identificador daquela variável. Além disso, também é opcio-



**Figura 1.5** Demonstrando a maneira pela qual uma referência para objeto pode ser usada para obter ou alterar variáveis de instância em um objeto (assumindo que se tem acesso a estas variáveis).

nal atribuir um valor inicial para a variável (usando o operador de atribuição “=”). As regras para o nome da variável são as mesmas de qualquer outro identificador Java. O tipo da variável pode ser tanto um tipo básico, indicando que a variável armazena valores daquele tipo, ou um nome de classe, indicando que a variável é uma **referência** para um objeto desta classe. Por fim, o valor inicial opcional que se pode atribuir a uma variável de instância deve combinar com o tipo da variável. Como exemplo, definiu-se a classe *Gnome*\*<sup>1</sup>, que contém várias definições de variáveis de instância, apresentada no Trecho de código 1.3.

O **escopo** (ou visibilidade) de uma variável de instância pode ser controlado através do uso dos seguintes **modificadores de variáveis**:

- **public**: qualquer um pode acessar variáveis de instância públicas.
- **protected**: apenas métodos do mesmo pacote ou subclasse podem acessar variáveis de instância protegidas.
- **private**: apenas métodos da mesma classe (excluindo métodos de uma subclasse) podem acessar variáveis de instâncias privadas.
- Se nenhum dos modificadores acima for usado, então a variável de instância é considerada amigável. Variáveis de instância amigáveis podem ser acessadas por qualquer classe no mesmo pacote. Os pacotes são discutidos detalhadamente na Seção 1.8.

Além dos modificadores de escopo de variável, existem também os seguintes modificadores de uso:

- **static**: a palavra reservada **static** é usada para declarar uma variável que é associada com a classe, não com instâncias individuais daquela classe. Variáveis static são usadas para armazenar informações globais sobre uma classe (por exemplo, uma variável static pode ser usada para armazenar a quantidade total de objetos *Gnome* criados). Variáveis static existem mesmo se nenhuma instância de sua classe for criada.
- **final**: uma variável de instância final é um tipo de variável para o qual se **deve** atribuir um valor inicial, e para a qual, a partir de então, não é possível atribuir um novo valor. Se for

\* N. de T. *Gnome*.

de um tipo básico, então é uma constante (como a constante MAX\_HEIGHT na classe Gnome). Se uma variável objeto é **final**, então irá sempre se referir ao mesmo objeto (mesmo se o objeto alterar seu estado interno).

```
public class Gnome {
    // Variáveis de instância:
    public String name;
    public int age;
    public Gnome gnomeBuddy;
    private boolean magical = false;
    protected double height = 2.6;
    public static final int MAX_HEIGHT = 3; // altura máxima
    // Construtores:
    Gnome(String nm, int ag, Gnome bud, double hgt) { // totalmente parametrizado
        name = nm;
        age = ag;
        gnomeBuddy = bud;
        height = hgt;
    }
    Gnome() { // Constructor default
        name = "Rumple";
        age = 204;
        gnomeBuddy = null;
        height = 2.1;
    }
    // Métodos:
    public static void makeKing (Gnome h) {
        h.name = "King " + h.getRealName();
        h.magical = true; // Apenas a classe Gnome pode referenciar este campo.
    }
    public void makeMeKing () {
        name = "King " + getRealName();
        magical = true;
    }
    public boolean isMagical() { return magical; }
    public void setHeight(int newHeight) { height = newHeight; }
    public String getName() { return "I won't tell!"; }
    public String getRealName() { return name; }
    public void renameGnome(String s) { name = s; }
```

#### Trecho de código 1.3 A classe Gnome.

Observa-se o uso das variáveis de instância no exemplo da classe Gnome. As variáveis age, magical e height\* são de tipos básicos, a variável name é uma referência para uma instância da classe predefinida String, e a variável gnomeBuddy\*\* é uma referência para um objeto da classe sendo definida. A declaração da variável de instância MAX\_HEIGHT\*\*\* está tirando proveito desses dois modificadores para definir uma “variável” que tem um valor constante fixo. Na verdade, valores constantes associados a uma classe sempre devem ser declarados **static** e **final**.

\* N. de T. “Idade”, “mágica” e “altura”, respectivamente.

\*\* N. de T. Companheiro do duende.

\*\*\* N. de T. Largura máxima.

### 1.1.3 Tipos enumerados

Desde a versão 5.0, Java suporta tipos enumerados chamados **enums**. Esses tipos são permitidos apenas para que se possa obter valores provenientes de conjuntos específicos de valores. Eles são declarados dentro de uma classe como segue:

```
modificador enum nome { nome_valor0, nome_valor1, ..., nome_valorn-1 }
```

onde *modificador* pode ser vazio, **public**, **protected** ou **private**. O nome desta enumeração, *nome*, pode ser qualquer identificador Java. Cada um dos identificadores de valor, *nome\_value<sub>i</sub>*, é o nome de um possível valor que variáveis desse tipo podem assumir. Cada um desses nomes de valor pode ser qualquer identificador Java legal, mas, por convenção, normalmente começam por letra maiúscula. Por exemplo, a seguinte definição de tipo enumerado pode ser útil em um programa que deve lidar com datas:

```
public enum Day { MON, TUE, WED, THU, FRI, SAT, SUN };
```

Uma vez definido, um tipo enumerado pode ser usado na definição de outras variáveis da mesma forma que um nome de classe. Entretanto, como o Java conhece todos os nomes dos valores possíveis para um tipo enumerado, se um tipo enumerado for usado em uma expressão string, o Java irá usar o nome do valor automaticamente. Tipos enumerados também possuem alguns métodos predefinidos, incluindo o método `valueOf`, que retorna o valor enumerado que é o mesmo que uma determinada string. Um exemplo de uso de tipo enumerado pode ser visto no Trecho de código 1.4.

```
public class DayTripper {
    public enum Day { MON, TUE, WED, THU, FRI, SAT, SUN };
    public static void main(String[] args) {
        Day d = Day.MON;
        System.out.println("Initially d is " + d);
        d = Day.WED;
        System.out.println("Then it is " + d);
        Day t = Day.valueOf("WED");
        System.out.println("I say d and t are the same: " + (d == t));
    }
}
```

A saída deste programa é:

```
Initially d is MON
Then it is WED
I say d and t are the same: true
```

**Trecho de código 1.4** Um exemplo de uso de tipo enumerado.

---

## 1.2 Métodos

Os métodos em Java são conceitualmente similares a procedimentos e funções em outras linguagens de alto nível. Normalmente correspondem a “trechos” de código que podem ser chamados em um objeto específico (de alguma classe). Os métodos podem admitir parâmetros como argumentos, e seu comportamento depende do objeto ao qual pertencem e dos valores passados por qualquer parâmetro. Todo método em Java é especificado no corpo de uma classe. A definição de um método comprehende duas partes: a **assinatura**, que define o nome e os parâmetros do método, e o **corpo**, que define o que o método realmente faz.

Um método permite ao programador enviar uma mensagem para um objeto. A assinatura do método especifica como uma mensagem deve parecer e o corpo do método especifica o que o objeto irá fazer quando receber tal mensagem.

## Declarando métodos

A sintaxe da definição de um método é como segue:

```
modificadores tipo nome(tipo0, parâmetro0, ..., tipon-1, parâmetron-1) {
    // corpo do método ...
}
```

Cada uma das partes desta declaração é importante e será descrita em detalhes nesta seção. A seção de *modificadores* usa os mesmos tipos de modificadores de escopo que podem ser usados para variáveis, tais como **public**, **protected** e **static**, com significados parecidos. A seção *tipo* define o tipo de retorno do método. O *nome* é o nome do método, e pode ser qualquer identificador Java válido. A lista de parâmetros e seus tipos declararam as variáveis locais que correspondem aos valores que são passados como argumentos para o método. Cada declaração de tipo, *tipo<sub>i</sub>*, pode ser qualquer nome tipo Java e cada *parâmetro<sub>i</sub>* pode ser qualquer identificador Java. Esta lista de identificadores e seus tipos pode ser vazia, o que significa que não existem valores para serem passados para este método quando for acionado. As variáveis parâmetro, assim como as variáveis de instância da classe, podem ser usadas dentro do corpo do método. Da mesma forma, os outros métodos desta classe podem ser chamados de dentro do corpo de um método.

Quando um método de uma classe é acionado, é chamado para uma instância específica da classe, e pode alterar o estado daquele objeto (exceto o método **static**, que é associado com a classe propriamente dita). Por exemplo, invocando-se o método que segue em um *gnome* particular, altera-se seu nome.

```
public void renameGnome (String s) {
    name = s; // Alterando a variável de instância nome deste gnome.
}
```

## Modificadores de métodos

À semelhança das variáveis de instância, modificadores de métodos podem restringir o escopo de um método:

- **public**: qualquer um pode chamar métodos públicos.
- **protected**: apenas métodos do mesmo pacote ou subclasse podem chamar um método protegido.
- **private**: apenas métodos da mesma classe (excluindo os métodos de subclasses) podem chamar um método privado.
- Se nenhum dos modificadores acima for usado, então o método é considerado amigável. Métodos amigáveis só podem ser chamados por objetos de classes do mesmo pacote.

Os modificadores de método acima podem ser precedidos por modificadores adicionais:

- **abstract**: um método declarado como **abstract** não terá código. A lista de parâmetros de um método abstrato é seguida por um ponto-e-vírgula, sem o corpo do método. Por exemplo:

```
public abstract void setHeight (double newHeight);
```

Métodos abstratos só podem ocorrer em classes abstratas. A utilidade desta construção será analisada na Seção 2.4.

- **final**: este é um método que não pode ser sobreescrito por uma subclasse.

- **static**: este é um método que é associado com a classe propriamente dita e não com uma instância em particular. Métodos static também podem ser usados para alterar o estado de variáveis static associadas com a classe (desde que estas variáveis não tenham sido declaradas como sendo **final**).

### Tipos de retorno

Uma definição de método deve especificar o tipo do valor que o método irá retornar. Se o método não retorna um valor, então a palavra reservada **void** deve ser usada. Se o tipo de retorno é **void**, o método é chamado de **procedimento**, caso contrário, é chamado de **função**. Para retornar um valor em Java, um método deve usar a palavra reservada **return** (e o tipo retornado deve combinar com o tipo de retorno do método). Na seqüência, um exemplo de método (interno à classe **Gnome**) que tem a forma de uma função:

```
public boolean isMagical () {
    return magical;
}
```

Assim que um **return** é executado em uma função Java, a execução do método termina.

Funções Java podem retornar apenas um valor. Para retornar múltiplos valores em Java, deve-se combiná-los em um **objeto composto** cujas variáveis de instância incluem todos os valores desejados e, então, retornar uma referência para este objeto composto. Além disso, pode-se alterar o estado interno de um objeto que é passado para um método como outra forma de “retornar” vários resultados.

### Parâmetros

Os parâmetros de um método são definidos entre parênteses, após o nome do mesmo, separados por vírgulas. Um parâmetro consiste em duas partes: seu tipo e o seu nome. Se um método não tem parâmetros, então apenas um par de parênteses vazio é usado.

Todos os parâmetros em Java são passados **por valor**, ou seja, sempre que se passa um parâmetro para um método, uma cópia do parâmetro é feita para uso no contexto do corpo do método. Ao se passar uma variável **int** para um método, o valor daquela variável é copiado. O método pode alterar a cópia, mas não o original. Quando se passa uma referência do objeto como parâmetro para um método, então essa referência é copiada da mesma forma. É preciso lembrar que se podem ter muitas variáveis diferentes referenciando o mesmo objeto. A alteração da referência recebida dentro de um método não irá alterar a referência que foi passada para o mesmo. Por exemplo, ao passar uma referência **g** da classe **Gnome** para um método que chama este parâmetro de **h**, então o método pode alterar a referência **h** de maneira que ela aponte para outro objeto, porém **g** continuará a referenciar o mesmo objeto anterior. O método, contudo, pode usar a referência **h** para mudar o estado interno do objeto, alterando assim o estado do objeto apontado por **g** (desde que **g** e **h** referenciem o mesmo objeto).

### Métodos construtores

Um **construtor** é um tipo especial de método que é usado para inicializar objetos novos quando de sua criação. Java tem uma maneira especial de declarar um construtor e uma forma especial de invocá-lo. Primeiro será analisada a sintaxe de declaração de um construtor:

```
modificadores tipo nome(tipo0 parâmetro0, ..., tipon-1 parâmetron-1) {
    // corpo do construtor ...
}
```

Vê-se que a sintaxe é igual à de qualquer outro método, mas existem algumas diferenças essenciais. O nome do construtor *name*, deve ser o mesmo nome da classe que constrói. Se a classe se chama Fish ("peixe"), então o construtor deve se chamar Fish da mesma forma. Além disso, um construtor não possui parâmetro de retorno – seu tipo de retorno é o mesmo que seu nome implicitamente (que é também o nome da classe). Os modificadores de construtor, indicados acima como *modifiers*, seguem as mesmas regras que os métodos normais, exceto pelo fato de que construtores **abstract**, **static** ou **final** não são permitidos.

Por exemplo:

```
public Fish (int w, String n) {
    weight = w;
    name = n;
}
```

### Definição e invocação de um construtor

O corpo de um construtor é igual ao corpo de um método normal, com um par de pequenas exceções. A primeira diferença diz respeito ao conceito conhecido como cadeia de construtores, tópico discutido na Seção 2.2.3 e que não é importante a esta altura.

A segunda diferença entre o corpo de um construtor e o corpo de um método comum é que o comando **return** não é permitido no corpo de um construtor. A finalidade deste corpo é ser usado para a inicialização dos dados associados com os objetos da classe correspondente, de forma que os mesmos fiquem em um estado inicial estável quando criados.

Métodos construtores são ativados de uma única forma: **devem** ser chamados através do operador **new**. Assim, a partir da ativação, uma nova instância da classe é automaticamente criada, e seu construtor é então chamado para inicializar as variáveis de instância e executar outros procedimentos de configuração. Por exemplo, considere-se a seguinte ativação de construtor (que corresponde também a uma declaração da variável *myFish*\*):

```
Fish myFish = new Fish (7, "Wally");
```

Uma classe pode ter vários construtores, mas cada um deve ter uma *assinatura* diferente, ou seja, devem ser distinguíveis pelo tipo e número de parâmetros que recebem.

### O método main

Certas classes Java destinam-se a ser utilizadas por outras classes, e outras têm como finalidade definir programas executáveis\*\*. Classes que definem programas executáveis devem conter um outro tipo especial de método para uma classe – o método **main**. Quando se deseja executar um programa executável Java, referencia-se o nome da classe que define este programa, por exemplo, disparando o seguinte comando (em um Shell Windows, Linux ou UNIX):

```
java Aquarium
```

Neste caso, o sistema de execução de Java procura por uma versão compilada da classe *Aquarium* ("aquário"), e então ativa o método especial **main** dessa classe. Esse método deve ser declarado como segue:

```
public static void main(String[ ] args) {
    // corpo do método main ...
}
```

\* N. de T. Meu peixe.

\*\* N. de T. Utiliza-se a expressão "programa executável" (*stand-alone program*) neste contexto para indicar um programa que é executado sem a necessidade de um navegador, não se referindo a um arquivo binário executável.

Os argumentos passados para o método main pelo parâmetro args são os argumentos de linha de comando fornecidos quando o programa é chamado. A variável args é um arranjo de objetos String; ou seja, uma coleção de strings indexadas, com a primeira string sendo args[0], a segunda sendo args[1] e assim por diante. (Falaremos mais sobre arranjos na Seção 1.5.)

### Chamando um programa Java a partir da linha de comando

Programas Java podem ser chamados a partir da linha de comando usando o comando `Java` seguido do nome da classe Java que contém o método main que se deseja executar, mais qualquer argumento opcional. Por exemplo, o programa `Aquarium` poderia ter sido definido para receber um parâmetro opcional que especificasse o número de peixes no aquário. O programa poderia ser ativado digitando-se o seguinte em uma janela Shell:

```
java Aquarium 45
```

para especificar que se quer um aquário com 45 peixes dentro dele. Neste caso, args[0] se refere à string "45". Uma característica interessante do método main é que permite a cada classe definir um programa executável, e um dos usos deste método é testar os outros métodos da classe. Desta forma, o uso completo do método main é uma ferramenta eficaz para a depuração de coleções de classes Java.

### Blocos de comandos e variáveis locais

O corpo de um método é um ***bloco de comandos***, ou seja, uma seqüência de declarações e comandos executáveis definidos entre chaves "{}". O corpo de um método e outros blocos de comandos podem conter também blocos de comandos aninhados. Além de comandos que executam uma ação, tal como ativar um método de algum objeto, os blocos de comandos podem conter declarações de ***variáveis locais***. Essas variáveis são declaradas no corpo do comando, em geral no início (mas entre as chaves "{}"). As variáveis locais são similares a variáveis de instância, mas existem apenas enquanto o bloco de comandos está sendo executado. Tão logo o fluxo de controle saia do bloco, todas as variáveis locais internas do mesmo não podem mais ser referenciadas. Uma variável local pode ser tanto um ***tipo base*** (tal como `int`, `float`, `double`), como uma ***referência*** para uma instância de alguma classe. Comandos e declarações simples em Java sempre se encerram com ponto-e-vírgula, ou seja um ";".

Existem duas formas de declarar variáveis locais:

```
tipo nome;  
tipo nome = valor_inicial;
```

A primeira declaração simplesmente define que o identificador, *nome*, é de um tipo específico. A segunda declaração define o identificador, seu tipo e também inicializa a variável com um valor específico. Seguem alguns exemplos de inicialização de variáveis locais:

```
{
    double r;
    Point p1 = new Point(3, 4);
    Point p2 = new Point(8, 2);
    int i = 512;
    double e = 2.71828;
}
```

## 1.3 Expressões

Variáveis e constantes são usadas em *expressões* para definir novos valores e para modificar variáveis. Nesta seção, discute-se com mais detalhes como as expressões Java funcionam. Elas envolvem o uso de *literais*, *variáveis* e *operadores*. Como as variáveis já foram examinadas, serão focados rapidamente os literais e analisados os operadores com mais detalhe.

### 1.3.1 Literais

Um *literal* é qualquer valor “constante” que pode ser usado em uma atribuição ou outro tipo de expressão. Java admite os seguintes tipos de literais:

- A referência para objeto **null** (este é o único literal que é um objeto e pertence à classe genérica **Object** por definição).
- Booleano: **true** e **false**.
- Inteiro: o default para um inteiro como 176 ou -52 é ser do tipo **int**, que corresponde a um inteiro de 32 bits. Um literal representando um inteiro longo deve terminar por um “L” ou “L”, por exemplo, 176L ou -52L, e corresponde a um inteiro de 64 bits.
- Ponto flutuante: o default para números de ponto flutuante, tais como 3.1415 e 10035.23, é ser do tipo **double**. Para especificar um literal **float**, ele deve terminar por um “F” ou um “f”. Literais de ponto flutuante em notação exponencial também são aceitos, como por exemplo, 3.14E2 ou 0.19e10; a base assumida é 10.
- Caracteres: assume-se que constantes de caracteres em Java pertencem ao alfabeto Unicode. Normalmente, um caractere é definido como um símbolo individual entre aspas simples. Por exemplo, ‘a’ e ‘?’ são constantes caractere. Além desses, Java define as seguintes constantes especiais de caracteres:

'\n'	(nova linha)	'\t'	(tabulação)
'\b'	(retorna um espaço)	'\r'	(retorno do carro)
'\f'	(alimenta formulário)	'\\'	(barra invertida)
'\"'	(aspas simples)	'\"'	(aspas duplas)

- Strings: uma string é uma seqüência de caracteres entre aspas duplas, por exemplo, o que segue é um string literal

"cachorros não sobem em árvores"

### 1.3.2 Operadores

As expressões em Java implicam em concatenar literais e variáveis usando operadores. Os operadores de Java serão analisados nesta seção.

#### O operador de atribuição

O operador-padrão de atribuição em Java é “=”. É usado na atribuição de valores para variáveis de instância ou variáveis locais. Sua sintaxe é:

*variável* = *expressão*

onde *variável* se refere a uma variável que pode ser referenciada no bloco de comandos que contém esta expressão. O valor de uma operação de atribuição é o valor da expressão que é atribuída. Sendo assim, se *i* e *j* são declaradas do tipo *int*, é correto ter um comando de atribuição como o seguinte:

```
i = j = 25; // funciona porque o operador '=' é avaliado da direita para a esquerda
```

## Operadores aritméticos

Os operadores que seguem são os operadores binários aritméticos de Java:

+	adição
-	subtração
*	multiplicação
/	divisão
%	operador módulo

O operador módulo também é conhecido como o operador de “resto”, na medida em que fornece o resto de uma divisão de números inteiros. Com freqüência, usamos “mod” para indicar o operador de módulo, e o definimos formalmente como:

$$n \bmod m = r$$

de maneira que

$$n = mq + r,$$

para um inteiro *q* e  $0 \leq r < n$ .

Java também fornece o operador unário menos (-), que pode ser colocado na frente de qualquer expressão aritmética para inverter seu sinal. É possível utilizar parênteses em qualquer expressão para definir a ordem de avaliação. Java utiliza ainda uma regra de precedência de operadores bastante intuitiva para determinar a ordem de avaliação quando não são usados parênteses. Ao contrário de C++, Java não permite a sobrecarga de operadores.

## Operadores de incremento e decremento

Da mesma forma que C e C++, Java oferece operadores de incremento e decremento. De forma mais específica, oferece os operadores incremento de um (++) e decremento de um (--). Se tais operadores são usados na frente de um nome de variável, então 1 é somado ou subtraído à variável, e seu valor é empregado na expressão. Se for utilizado depois do nome da variável, então primeiro o valor é usado, e depois a variável é incrementada ou decrementada de 1. Assim, por exemplo, o trecho de código

```
int i = 8;
int j = i++;
int k = ++i;
int m = i--;
int n = 9 + i++;
```

atribui 8 para *j*, 10 para *k*, 10 para *m*, 18 para *n* e deixa *i* com o valor 10.

## Operadores lógicos

Java oferece operadores padrão para comparações entre números:

<	menor que
<=	menor que ou igual a
==	igual a

**!=** diferente de  
**>=** maior que ou igual a  
**>** maior que

Os operadores **==** e **!=** também podem ser usados com referências para objetos. O tipo resultante de uma comparação é **boolean**.

Os operadores que trabalham com valores **boolean** são os seguintes:

**!** negação (prefixado)  
**&&** e condicional  
**||** ou condicional

Os operadores booleanos **&&** e **||** não avaliarão o segundo operando em suas expressões (para a direita) se isso não for necessário para determinar o valor da expressão. Este recurso é útil, por exemplo, para construir expressões booleanas onde primeiro se testa se uma determinada condição se aplica (tal como uma referência não ser **null**) e, então, se testa uma condição que geraria uma condição de erro, se o primeiro teste falhasse.

### Operadores sobre bits

Java fornece, também, os seguintes operadores sobre bits para inteiros e booleanos:

**~** complemento sobre bits (operador prefixado unário)  
**&** e sobre bits  
**|** ou sobre bits  
**^** ou exclusivo sobre bits  
**<<** deslocamento de bits para esquerda, preenchendo com zeros  
**>>** deslocamento de bits para a direita, preenchendo com bits de sinal  
**>>>** deslocamento de bits para a direita, preenchendo com zeros

### Operadores operacionais de atribuição

Além do operador de atribuição padrão (**=**), Java também oferece um conjunto de outros operadores de atribuição que têm efeitos colaterais operacionais. Esses outros tipos de operadores são da forma:

*variável op = expressão*

onde *op* é um operador binário. Esta expressão é equivalente a

*variável = variável op expressão*

excetuando-se que, se *variável* contém uma expressão (por exemplo, um índice de arranjo), a expressão é avaliada apenas uma vez. Assim, o fragmento de código

```
a[5] = 10;
i = 5;
a[i++] += 2;
```

deixa *a[5]* com o valor 12 e *i* com o valor 6.

### Concatenação de strings

As strings podem ser compostas usando o operador de **concatenação** (**+**), de forma que o código

```
String rug = "carpet";
String dog = "spot";
String mess = rug + dog;
String answer = mess + "will cost me" + 5 + "dollars!";
```

terá o efeito de fazer `answer` apontar para a string

```
"carpetspot will cost me 5 dollars!"
```

Esse exemplo também mostra como Java converte constantes que não são string em strings, quando estas estão envolvidas em uma operação de concatenação de strings.

### Precedência de operadores

Os operadores em Java têm uma dada preferência, ou precedência, que determina a ordem na qual as operações são executadas quando a ausência de parênteses ocasiona ambigüidades na avaliação. Por exemplo, é necessário que exista uma forma de decidir se a expressão “`5+2*3`” tem valor 21 ou 11 (em Java o valor é 11).

A Tabela 1.3 apresenta a precedência dos operadores em Java (que, coincidentemente, é a mesma de C).

Precedência de operadores		
	Tipo	Símbolos
1	operadores pós-fixados operadores pré-fixados <i>cast</i> (coerção)	<i>exp</i> <code>++</code> <i>exp</i> <code>--</code> <code>++exp</code> <code>--exp</code> <code>+exp</code> <code>-exp</code> <code>-exp</code> <code>!exp</code> ( <i>type</i> ) <i>exp</i>
2	mult./div.	<code>*</code> <code>/</code> <code>%</code>
3	soma/subt.	<code>+</code> <code>-</code>
4	deslocamento	<code>&lt;&lt;</code> <code>&gt;&gt;</code> <code>&gt;&gt;&gt;</code>
5	comparação	<code>&lt;</code> <code>&lt;=</code> <code>&gt;</code> <code>&gt;=</code> <code>instanceof</code>
6	igualdade	<code>==</code> <code>!=</code>
7	“e” bit a bit	<code>&amp;</code>
8	“xor” bit a bit	<code>^</code>
9	“ou” bit a bit	<code> </code>
10	“e”	<code>&amp;&amp;</code>
11	“ou”	<code>  </code>
12	condicional	<i>expressão booleana?</i> <code>valor se true : valor se false</code>
13	atribuição	<code>=</code> <code>+=</code> <code>-=</code> <code>*=</code> <code>/=</code> <code>%=</code> <code>&gt;&gt;=</code> <code>&lt;&lt;=</code> <code>&gt;&gt;&gt;=</code> <code>&amp;=</code> <code>^=</code> <code> =</code>

**Tabela 1.3** As regras de precedência de Java. Os operadores em Java são avaliados de acordo com a ordem acima se não forem utilizados parênteses para determinar a ordem de avaliação. Os operadores na mesma linha são avaliados da esquerda para a direita (exceto atribuições e operações prefixadas, que são avaliadas da direita para esquerda), sujeitos à regra de avaliação condicional para as operações booleanas **e** e **ou**. As operações são listadas da precedência mais alta para a mais baixa (usamos *exp* para indicar uma expressão atômica ou entre parênteses). Sem parênteses, os operadores de maior precedência são executados depois de operadores de menor precedência.

Discutiu-se até agora quase todos os operadores listados na Tabela 1.3. Uma exceção notável é o operador condicional, o que implica avaliar uma expressão booleana e então tomar o valor apropriado, dependendo de a expressão booleana ser verdadeira ou falsa. (O uso do operador **instanceof** será analisado no próximo capítulo.)

### 1.3.3 Conversores e autoboxing/unboxing em expressões

A conversão é uma operação que nos permite alterar o tipo de uma variável. Em essência, pode-se **converter** uma variável de um tipo em uma variável equivalente de outro tipo. Os conversores

podem ser úteis para fazer certas operações numéricas e de entrada e saída. A sintaxe para converter uma variável para um tipo desejado é a seguinte:

*(tipo) exp*

onde *tipo* é o tipo que se deseja que a expressão *exp* assuma. Existem dois tipos fundamentais de conversores que podem ser aplicados em Java. Pode-se tanto converter tipos de base numérica como tipos relacionados com objetos. Agora, será discutida a conversão de tipos numéricos e strings e a conversão de objetos será analisada na Seção 2.5.1. Por exemplo, pode ser útil converter um **int** em um **double** de maneira a executar operações como uma divisão.

### Conversores usuais

Quando se converte um **double** em um **int**, pode-se perder a precisão. Isso significa que o valor double resultante será arredondado para baixo. Mas pode-se converter um **int** em um **double** sem esta preocupação. Por exemplo, considere o seguinte:

```
double d1 = 3.2;
double d2 = 3.9999;
int i1 = (int)d1;           // i1 tem valor 3
int i2 = (int)d2;           // i2 tem valor 3
double d3 = (double)i2;    // d3 tem valor 3.0
```

### Convertendo operadores

Alguns operadores binários, como o de divisão, terão resultados diferentes dependendo dos tipos de variáveis envolvidas. Devemos ter cuidado para garantir que tais operações executem seus cálculos em valores do tipo desejado. Quando usada com inteiros, por exemplo, a divisão não mantém a parte fracionária. No caso de uso com double, a divisão conserva esta parte, como ilustra o exemplo a seguir:

```
int i1 = 3;
int i2 = 6;
dresult = (double)i1 / (double)i2; // dresult tem valor 0.5
dresult = i1 / i2;               // dresult tem valor 0.0
```

Observe que a divisão normal para números reais foi executada quando i1 e i2 foram convertidos em double. Quando i1 e i2 não foram convertidos, o operador “/” executou uma divisão inteira e o resultado de i1 / i2 foi o **int** 0. Java executou uma **conversão implícita** para atribuir um valor **int** ao resultado **double**. Vamos estudar a conversão implícita a seguir.

### Conversores implícitos e autoboxing/unboxing

Existem casos onde o Java irá executar uma **conversão implícita**, de acordo com o tipo da variável sendo atribuída, desde que não haja perda de precisão. Por exemplo:

```
int irest, i = 3;
double dresult, d = 3.2;
dresult = i / d;           // dresult tem valor 0.9375. i foi convertido para double
iresult = i / d;           // perda de precisão -> Isto é em um erro de compilação;
iresult = (int) i / d;     // irest é 0, uma vez que a parte fracionária será perdida.
```

Considerando que Java não executará conversões implícitas onde houver perda de precisão, a conversão explícita da última linha do exemplo é necessária.

A partir do Java 5.0, existe um novo tipo de conversão implícita entre objetos numéricos, tais como **Integer** e **Float**, e seus tipos básicos relacionados, tais como **int** e **float**. Sempre que um ob-

jeto numérico for esperado como parâmetro para um método, o tipo básico correspondente pode ser informado. Neste caso, o Java irá proceder uma conversão implícita chamada *autoboxing*, que irá converter o tipo base para o objeto numérico correspondente. Da mesma forma, sempre que um tipo base for esperado em uma expressão envolvendo um objeto numérico, o objeto numérico será convertido no tipo base correspondente em uma operação chamada de *unboxing*.

Existem, entretanto, alguns cuidados a serem tomados no uso de boxing e unboxing. O primeiro é que se uma referência numérica for `null`, então qualquer tentativa de unboxing irá gerar um erro de `NullPointerException`. Em segundo, o operador “`==`” é usado tanto para testar a igualdade de dois valores numéricos como se duas referências para objetos apontam para o mesmo objeto. Sendo assim, quando se testa a igualdade, deve-se evitar a conversão implícita provida por autoboxing/unboxing. Por fim, a conversão implícita de qualquer tipo toma tempo, logo devemos minimizar nossa confiança nela se performance for um requisito.

Circunstancialmente, existe uma situação em Java em que apenas a conversão implícita é permitida, que é na concatenação de strings. Sempre que uma string é concatenada com qualquer objeto ou tipo base, o objeto ou tipo base é automaticamente convertido em uma string. Entretanto, a conversão explícita de um objeto ou tipo base para uma string não é permitida. Portanto, as seguintes atribuições são incorretas:

```
String s = (String) 4.5;           // Isso está errado!
String t = "Value = " + (String) 13; // Isso está errado!
String u = 22;                   // Isso está errado!
```

Para executar conversões para string, deve-se, ao invés disso, usar o método `toString` apropriado ou executar uma conversão implícita via operação de concatenação. Assim, os seguintes comandos estão corretos:

```
String s = "" + 4.5;           // correto, porém, mau estilo de programação
String t = "Value = " + 13;    // correto
String u = Integer.toString(22); // correto
```

## 1.4 Controle de fluxo

O controle de fluxo em Java é similar ao oferecido em outras linguagens de alto nível. Nesta seção, revisa-se a estrutura básica e a sintaxe do controle de fluxo em Java, incluindo retorno de métodos, comando **condicional**, comandos de **seleção múltipla**, laços e formas restritas de “desvios” (os comandos **break** e **continue**).

### 1.4.1 Os comandos if e switch

Em Java, comandos condicionais funcionam da mesma forma que em outras linguagens. Eles fornecem a maneira de tomar uma decisão e então executar um ou mais blocos de comandos diferentes baseados no resultado da decisão.

O comando if

A sintaxe básica do comando **if** é a que segue:

```
if (expr_boleana)
    comando_se_verdade
else
    comando_se_falso
```

onde *expr\_booleana* é uma expressão booleana e *comando\_se\_verdade* e *comando\_se\_falso* podem ser um comando simples ou um bloco de comandos entre chaves ("{" e "}"). Observa-se que, diferentemente de outras linguagens de programação, os valores testados por um comando **if** devem ser uma expressão booleana. Particularmente, não são uma expressão inteira. Por outro lado, como em outras linguagens similares, a cláusula **else** (e seus comandos associados) são opcionais. Existe também uma forma de agrupar um conjunto de testes booleanos como segue:

```
if (primeira_expressão_booleana)
    comando_se_verdade
else if (segunda_expressão_booleana)
    segundo_comando_se_verdade
else
    comando_se_falso
```

Se a primeira expressão booleana for falsa, então a segunda expressão booleana será testada, e assim por diante. Um comando **if** pode ter qualquer quantidade de cláusulas **else if**.

Por exemplo, a estrutura a seguir está correta:

```
if (snowLevel < 2) {
    goToClass();
    comeHome();
}
else if (snowLevel < 5) {
    goSledding();
    haveSnowballFight();
}
else
    stayAtHome();
```

### Comando switch

Java oferece o comando **switch** para controle de fluxo multivalorado, o que é especialmente útil com tipos enumerados. O exemplo a seguir é indicativo (baseado na variável *d* do tipo Day da Seção 1.1.3).

```
switch (d) {
    case MON:
        System.out.println("This is tough.");
        break;
    case TUE:
        System.out.println("This is getting better.");
        break;
    case WED:
        System.out.println("Half way there.");
        break;
    case THU:
        System.out.println("I can see the light.");
        break;
    case FRI:
        System.out.println("Now we are talking.");
        break;
    default:
        System.out.println("Day off!");
        break;
}
```

O comando **switch** avalia uma expressão inteira ou enumeração e faz com que o fluxo de controle desvie para o ponto marcado com o valor dessa expressão. Se não existir um ponto com tal marca, então o fluxo é desviado para o ponto marcado com “**default**”. Entretanto, este é o único desvio explícito que o comando **switch** executa, e, a seguir, o controle “cai” através das cláusulas **case** se o código dessas cláusulas não for terminado por uma instrução **break** (que faz o fluxo de controle desviar para a próxima linha depois do comando **switch**).

### 1.4.2 Laços

Outro mecanismo de controle de fluxo importante em uma linguagem de programação é o laço. Java possui três tipos de laços.

#### Laços while

O tipo mais simples de laço em Java é o laço **while**. Este tipo de laço testa se uma certa condição é satisfeita e executa o corpo do laço enquanto esta condição for **true**. A sintaxe para testar uma condição antes de o corpo do laço ser executado é a seguinte:

```
while (expressão booleana)
    corpo do laço
```

No início de cada iteração, o laço testa a expressão booleana, *boolean exp*, e então, se esta resultar **true**, executa o corpo do laço, *loop statement*. Da mesma forma que o laço **for**, o corpo do laço também pode ser um bloco de comandos.

Considere-se, por exemplo, um gnomo tentando regar todas as cenouras de seu canteiro de cenouras, o que faz até seu regador ficar vazio. Se o regador estiver vazio logo no início, escreve-se o código para executar esta tarefa como segue:

```
public void waterCarrots() {
    Carrot current = garden.findNextCarrot();
    while (!waterCan.isEmpty()) {
        water(current, waterCan);
        current = garden.findNextCarrot();
    }
}
```

Lembre-se que “!” em Java é o operador “not”.

#### Laços for

Outro tipo de laço é o laço **for**. Na sua forma mais simples, os laços **for** oferecem uma repetição codificada baseada em um índice inteiro. Em Java, entretanto, pode-se fazer muito mais. A funcionalidade de um laço **for** é significativamente mais flexível. Sua estrutura se divide em quatro seções: inicialização, condição, incremento e corpo.

#### Definindo um laço for

Esta é a sintaxe de um laço **for** em Java

```
for (inicialização; condição; incremento)
    corpo do laço
```

onde cada uma das seções de *inicialização*, *condição* e *incremento* podem estar vazias.

Na seção *inicialização*, pode-se declarar uma variável índice que será válida apenas no escopo do laço **for**. Por exemplo, quando se deseja um laço indexado por um contador, e não há necessidade desse contador fora do contexto do laço **for**, então declara-se algo como o que segue:

```
for (int counter = 0; condição; incremento)
    corpo_do_laço
```

que declara uma variável *counter* cujo escopo é limitado apenas ao corpo do laço.

Na seção *condição*, especifica-se a condição de repetição ("enquanto") do laço. Esta deve ser uma expressão booleana. O corpo do laço **for** será executado toda a vez que a *condição* resultar **true**, quando avaliada no início de uma iteração potencial. Assim que a *condição* resultar **false**, então o corpo do laço não será executado e, em seu lugar, o programa executa o próximo comando depois do laço **for**.

Na seção de *incremento*, declara-se o comando de incremento do laço. O comando de incremento pode ser qualquer comando válido, o que permite uma flexibilidade significativa para a programação. Assim, a sintaxe do laço **for** é equivalente ao que segue:

```
inicialização;
while (condição) {
    comandos_do_laço
    incremento;
}
```

exceto pelo fato de que um laço **while** não pode ter uma condição booleana vazia, enquanto que um laço **for** pode. O exemplo a seguir apresenta um exemplo simples de laço **for** em Java:

```
public void eatApples (Apples apples) {
    numApples = apples.getNumApples ();
    for (int x = 0; x < numApples; x++) {
        eatApple (apples.getApple (x));
        spitOutCore ();
    }
}
```

Neste exemplo, a variável de laço *x* foi declarada como **int** *x* = 0. Antes de cada iteração, o laço testa a condição "*x* < numApples" e executa o corpo do laço apenas se isso for verdadeiro. Por último, ao final de cada iteração, o laço usa a expressão *x*++ para incrementar a variável *x* do laço antes de testar a condição novamente.

Desde a versão 5.0, Java inclui o laço *for-each*, que será discutido na Seção 6.3.2.

## Laços do-while

Java tem ainda outro tipo de laço além do laço **for** e do laço **while** padrão – o laço **do-while**. Enquanto que os primeiros testam a condição antes de executar a primeira iteração com o corpo do laço, o laço **do-while** testa a condição após o corpo do laço. A sintaxe de um laço **do-while** é mostrada a seguir:

```
do
    corpo_do_laço
while (condição)
```

Mais uma vez, o *corpo do laço* pode ser um comando ou um bloco de comandos, e a *condição* será uma expressão booleana. Em um laço **do-while**, repete-se o corpo do laço enquanto a expressão resultar verdadeira a cada avaliação.

Suponha-se, por exemplo, que se deseja solicitar uma entrada ao usuário e posteriormente fazer algo útil com essa entrada. (Entradas e saídas em Java serão examinadas com mais detalhes

na Seção 1.6.) Uma condição possível para sair do laço, neste caso, é quando o usuário entra uma string vazia. Entretanto, mesmo neste caso, pode-se querer manter a entrada e informar ao usuário que ela saiu. O exemplo a seguir ilustra o caso:

```
public void getUserInput( ) {
    String input;
    do {
        input = getInputStream();
        handleInput(input);
    } while (input.length( ) > 0);
}
```

Observe-se a condição de saída do exemplo. Mais especificamente, está escrita para ser consistente com a regra de Java que diz que laços **do-while** se encerram quando a condição *não* é verdadeira (ao contrário da construção repeat-until usada em outras linguagens).

### 1.4.3 Expressões explícitas de controle de fluxo

Java também oferece comandos que permitem alterações explícitas no fluxo de controle de um programa.

#### Retornando de um método

Se um método Java é declarado com o tipo de retorno **void**, então o fluxo de controle retorna quando encontra a última linha de código do método ou quando encontra um comando **return** sem argumentos. Entretanto, se um método é declarado com um tipo de retorno, ele é uma função e deverá terminar retornando o valor da função como um argumento do comando **return**. O exemplo seguinte (correto) ilustra o retorno de uma função:

```
// Verifica um aniversário específico
public boolean checkBDay (int date) {
    if (date == Birthday.MIKES_BDAY){
        return true;
    }
    return false;
}
```

Conclui-se que o comando **return** deve ser o último comando executado em uma função, já que o resto do código nunca será alcançado.

Existe uma diferença significativa entre um comando ser a última linha de código a ser *executada* em um método ou ser a última linha de código do método propriamente dita. No exemplo anterior, a linha **return true;** claramente não é a última linha do código escrito para a função, mas pode ser a última linha executada (se a condição envolvendo date for **true**). Esse comando interrompe de forma explícita o fluxo de controle do método. Existem dois outros comandos explícitos de controle de fluxo que são usados em conjunto com laços e com o comando **switch**.

#### O comando **break**

O uso típico do comando **break** tem a seguinte sintaxe simples:

```
break;
```

É usado para “sair” do bloco internamente mais aninhado dos comandos **switch**, **for**, **while** ou **do-while**. Quando executado, um comando **break** faz com que o fluxo de controle seja desviado para a próxima linha depois do laço ou **switch** que contém o **break**.

O comando **break** também pode ser usado de forma rotulada para desviar para o laço ou comando **switch** de aninhamento mais externo. Neste caso, ele tem a sintaxe:

```
break label;
```

onde *label* é um identificador em Java usado para rotular um laço ou um comando **switch**. Este tipo de rótulo só pode aparecer no início da declaração de um laço; não existem outras formas de comandos “go to” em Java.

O uso de um rótulo com comando **break** é ilustrado no seguinte exemplo:

```
public static boolean hasZeroEntry (int[ ][ ] a) {
    boolean foundFlag = false;

    zeroSearch:
    for (int i=0; i<a.length; i++) {
        for (int j=0; j<a[i].length; j++) {
            if (a[i][j] == 0) {
                foundFlag = true;
                break zeroSearch;
            }
        }
    }
    return foundFlag;
}
```

O exemplo acima usa arranjos que serão abordados na Seção 3.1

### O comando **continue**

O outro comando que altera explicitamente o fluxo de controle em um programa Java é o comando **continue**, que tem a seguinte sintaxe:

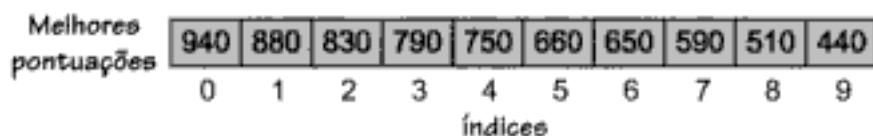
```
continue label;
```

onde *label* é um identificador em Java usado para rotular o laço. Como já foi mencionado anteriormente, não existem comandos “go to” explícitos em Java. Da mesma forma, o comando **continue** só pode ser usado dentro de laços (**for**, **while** e **do-while**). O comando **continue** faz com que a execução pule os passos restantes do laço na iteração atual (mas continue o laço se a condição for satisfeita).

## 1.5 Arranjos

Uma tarefa comum em programação é a manutenção de um conjunto numerado de objetos relacionados. Por exemplo, deseja-se que um jogo de videogame mantenha a relação das dez melhores pontuações. Em vez de se usar dez variáveis diferentes para esta tarefa, prefere-se usar um único nome para o conjunto e usar índices numéricos para referenciar as pontuações mais altas dentro do conjunto. Da mesma forma, deseja-se que um sistema de informações médicas mantenha a relação de pacientes associados aos leitos de um certo hospital. Novamente, não é necessário inserir 200 variáveis no programa apenas porque o hospital tem 200 leitos.

Nestes casos, minimiza-se o esforço de programação pelo uso de **arranjos**, que são coleções numeradas de variáveis do mesmo tipo. Cada variável ou **célula** em um arranjo tem um **índice**, que referencia o valor armazenado na célula de forma única. As células de um arranjo *a* são numeradas 0, 1, 2 e assim por diante. A Figura 1.6 apresenta o desenho de um arranjo contendo as melhores pontuações do videogame.



**Figura 1.6** Desenho de um arranjo com as dez (*int*) melhores pontuações de um videogame.

Essa forma de organização é extremamente útil, na medida em que permite computações interessantes. Por exemplo, o método a seguir soma todos os valores armazenados em um arranjo de inteiros:

```
/** Soma todos os valores de um arranjo de inteiros */
public static int sum(int[] a) {
    int total = 0;
    for (int i=0; i<a.length; i++) // observe-se o uso da variável length
        total += a[i];
    return total;
}
```

Este exemplo tira vantagem de um recurso interessante de Java que permite determinar a quantidade de células mantidas por um arranjo, ou seja, seu *tamanho*\*. Em Java um arranjo *a* é um tipo especial de objeto, e o tamanho de *a* está armazenado na variável de instância *length*. Isto é, jamais será necessário ter de adivinhar o tamanho de um arranjo em Java, visto que o tamanho de um arranjo pode ser acessado como segue:

*nome\_do\_arranjo.length*

onde *nome\_do\_arranjo* é o nome do arranjo. Assim, as células de um arranjo *a* são numeradas 0, 1, 2, e assim por diante até *a.length* – 1.

### Elementos e capacidade de um arranjo

Cada objeto armazenado em um arranjo é chamado de *elemento* do arranjo. O elemento número 0 é *a[0]*, o elemento número 1 é *a[1]*, o elemento número 2 é *a[2]*, e assim por diante. Uma vez que o comprimento de um arranjo determina o número máximo de coisas que podem ser armazenadas no arranjo, também pode-se referir ao comprimento de um arranjo como sendo sua *capacidade*. O trecho de código que segue apresenta outro exemplo simples de uso de arranjos, que conta o número de vezes que um certo número aparece em um arranjo.

```
/** Conta o número de vezes que um inteiro aparece em um arranjo */
public static int findCount(int[] a, int k) {
    int count = 0;
    for (int e: a) { // observe-se o uso do laço "foreach"
        if (e == k) // deve-se verificar se o elemento corrente é igual a k
            count++;
    }
    return count;
}
```

### Erros de limites

É um erro perigoso tentar indexar um arranjo *a* usando um número fora do intervalo de 0 a *a.length* – 1. Tal referência é dita estar *fora de faixa*. Referências fora de faixa tem sido fre-

\* N. de T. Apesar da expressão em inglês *length* indicar comprimento, é mais comum entre programadores Java o uso da expressão *tamanho* do arranjo.

quietamente exploradas por hackers usando um método chamado *ataque do estouro do buffer*\* comprometendo a segurança de sistemas de computação escritos em outras linguagens em vez de Java. Por questões de segurança, os índices de arranjo são sempre verificados em Java para constatar se não estão fora de faixa. Se um índice de arranjo está fora de faixa, o ambiente de execução de Java sinaliza uma condição de erro. O nome desta condição é `ArrayIndexOutOfBoundsException`. Esta verificação auxilia para que Java evite uma série de problemas (incluindo problemas de ataque de estouro de buffer) com os quais outras linguagens tem de lutar.

Pode-se evitar erros de índice fora de faixa tendo certeza de que as indexações sempre serão feitas dentro de um arranjo `a`, usando valores inteiros entre 0 e `a.length`. Uma forma simples de fazer isso é usando com cuidado o recurso das operações booleanas de Java já apresentado. Por exemplo, um comando como o que segue nunca irá gerar um erro de índice fora de faixa:

```
if ((i >= 0) && (i < a.length) && (a[i] > 2))
    x = a[i];
```

pois a comparação "`a[i] > 5`" só será executada se as duas primeiras comparações forem bem-sucedidas.

### 1.5.1 Declarando arranjos

Uma forma de declarar e inicializar um arranjo é a seguinte:

```
tipo_do_elemento[] nome_do_arranjo = { val_inic_0, val_inic_1, ..., val_inic_N-1};
```

O `tipo_do_elemento` pode ser qualquer tipo base de Java ou um nome de classe e `nome_do_arranjo` pode ser qualquer identificador Java válido. Os valores de inicialização devem ser do mesmo tipo que o arranjo. Por exemplo, considere-se a seguinte declaração de um arranjo que é inicializado para conter os primeiros dez números primos:

```
int[] primes = {2, 3, 5, 7, 11, 13, 17, 19, 23, 29};
```

Além de declarar um arranjo e inicializar todos os seus valores na declaração, pode-se declarar um arranjo sem inicializá-lo. A forma desta declaração é a que segue:

```
tipo_do_elemento[] nome_do_arranjo;
```

Um arranjo criado desta forma é inicializado com zeros se o tipo do arranjo for um tipo numérico. Arranjos de objetos são inicializados com referências `null`. Uma vez criado um arranjo desta forma, pode-se criar o conjunto de células mais tarde usando a sintaxe a seguir:

```
new tipo_do_elemento[comprimento]
```

onde `comprimento` é um inteiro positivo que denota o comprimento do arranjo criado. Normalmente, esta expressão aparece em comandos de atribuição com o nome do arranjo do lado esquerdo do operador de atribuição. Então, por exemplo, o seguinte comando define uma variável arranjo chamada `a` e, mais tarde, atribuem-se à mesma um arranjo de dez células, cada uma do tipo `double`, que então é inicializado:

```
double[] a;
// ... vários passos ...
a = new double[10];
for (int k=0; k < a.length; k++)
    a[k] = 1.0;
}
```

\* N. de T. Em inglês, *buffer overflow attack*.

As células do novo arranjo “a” são indexadas usando o conjunto inteiro {0,1,2, ..., 9} (lembre-se que os arranjos em Java sempre iniciam a indexação em 0), e, da mesma forma que qualquer arranjo Java, todas as células deste arranjo são do mesmo tipo – **double**.

### 1.5.2 Arranjos são objetos

Arranjos em Java são tipos especiais de objetos. Na verdade, esta é a razão pela qual pode-se usar o operador **new** para criar uma nova instância de arranjo. Um arranjo pode ser usado da mesma forma que qualquer outro objeto de Java, mas existe uma sintaxe especial (usando colchetes, “[” e “]”) para se referenciar a seus membros. Um arranjo Java pode fazer tudo que um objeto genérico pode fazer. Como se trata de um objeto, o nome de um arranjo em Java é, na verdade, uma referência para o lugar na memória onde o arranjo está armazenado. Assim, não existe nada de tão especial em se usar o operador ponto e a variável de instância `length`, para se referir ao comprimento de um arranjo como no exemplo “`a.length`”. O nome `a`, neste caso, é apenas uma referência ou ponteiro para o arranjo subjacente.

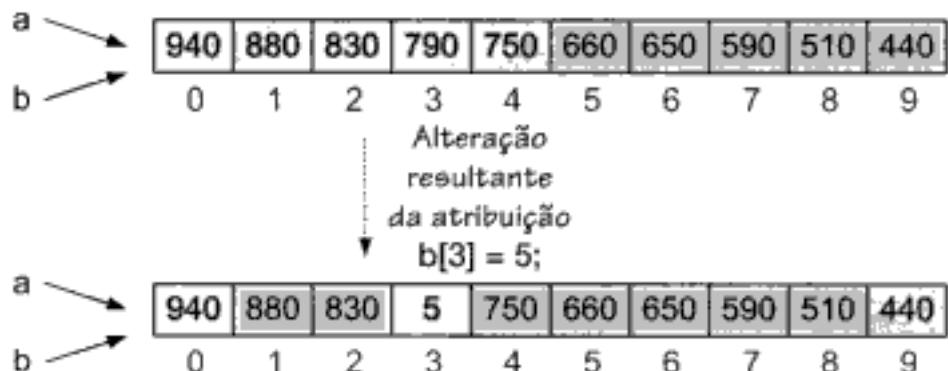
O fato de que arranjos em Java são objetos tem uma implicação importante quando se usa nomes de arranjos em expressões de atribuição. Quando se escreve algo tal como

`b = a;`

em um programa Java, na verdade significa que agora tanto `b` como `a` se referem ao mesmo arranjo. Então, ao se escrever algo como

`b[3] = 5;`

se está alterando `a[3]` para 5. Este ponto crucial é demonstrado na Figura 1.7.



**Figura 1.7** Desenho da atribuição de um arranjo de objetos. Apresenta-se o resultado da atribuição de “`b[3] = 5;`” depois de previamente ter executado “`b = a`”.

### Clonando um arranjo

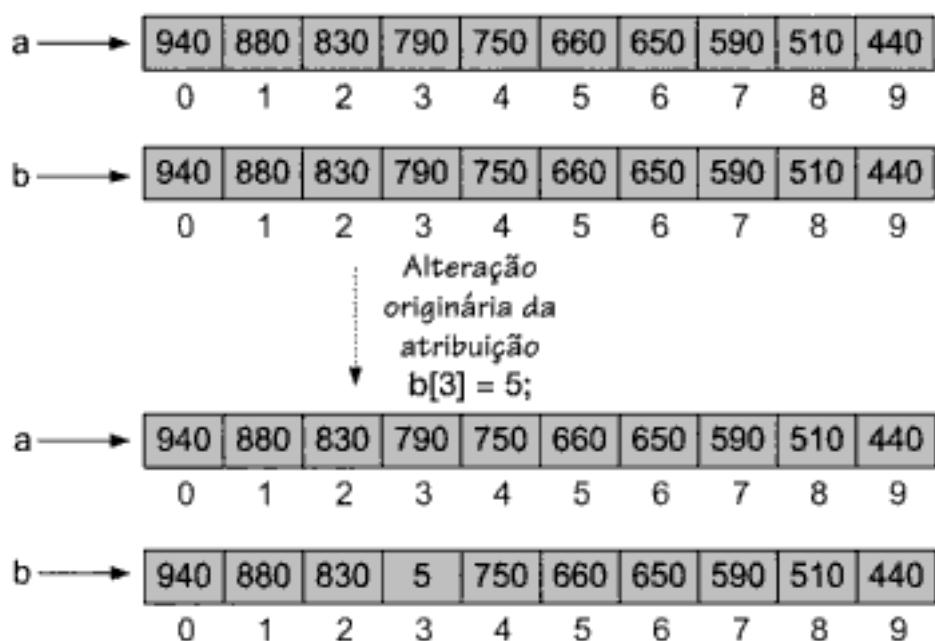
Se, por outro lado, for necessário criar uma cópia exata do arranjo `a` e atribuir esse arranjo para a variável arranjo `b`, pode-se escrever:

`b = a.clone();`

que copia todas as células em um novo arranjo e o atribui a `b`, de maneira que este aponta para o novo arranjo. Na verdade, o método `clone` é um método predefinido de todo objeto Java, e cria uma cópia exata de um objeto. Neste caso, se então escrever-se

`b[3] = 5;`

o novo arranjo (copiado) terá o valor 5 atribuído para a célula de índice 3, mas `a[3]` irá permanecer inalterado. Demonstra-se esta situação na Figura 1.8.



**Figura 1.8** Demonstração da clonagem de um arranjo de objetos. Demonstra-se o resultado da atribuição  $b[3] = 5$  após a atribuição “ $b = a.clone()$ ”.

Detalhando, pode-se afirmar que as células de um arranjo são copiadas quando o mesmo é clonado. Se as células são de um tipo base, como `int`, os valores são copiados. Mas se as células são referências para objetos, então essas referências são copiadas. Isso significa que existem duas maneiras de referenciar tais objetos. As consequências deste fato são exploradas no Exercício R-1.1.

## 1.6 Entrada e saída simples

Java oferece um conjunto rico de classes e métodos para executar entrada e saída. Existem classes para executar projetos de interfaces gráficas com o usuário, incluindo diálogos e menus suspensos, assim como métodos para a exibição e a entrada de texto e números. Java também oferece métodos para lidar com objetos gráficos, imagens, sons, páginas Web e eventos de mouse (tais como cliques, deslocamentos do mouse e arrasto). Além do mais, muitos desses métodos de entrada e saída podem ser usados tanto em programas executáveis quanto em applets. Infelizmente, detalhar como cada um desses métodos funciona para construir interfaces gráficas sofisticadas com o usuário está além do escopo deste livro. Entretanto, em nome de uma maior abrangência, descreve-se nesta seção como se pode fazer entrada e saída simples em Java.

Em Java, a entrada e saída simples é feita através da janela console de Java. Dependendo do ambiente Java que se está empregando, esta janela pode ser a janela especial usada para exibição e entrada de texto, ou é a janela que se utiliza para passar comandos para nosso sistema operacional (tais janelas costumam ser chamadas de janelas de console, janelas DOS ou janelas de terminal).

### Métodos de saída

Java oferece um objeto static embutido chamado `System.out`, que envia a saída para o dispositivo de saída-padrão. Alguns sistemas operacionais permitem aos usuários redirecionar a saída-padrão para arquivos ou até mesmo como entrada para outros programas, embora a saída-padrão seja a janela de console de Java. O objeto `System.out` é uma instância da classe `java.io.PrintStream`. Essa classe define métodos para um fluxo buferizado de saída, o que significa que os caracteres são colocados em uma localização temporária, chamada **buffer**, que é esvaziada quando a janela de console estiver pronta para imprimir os caracteres.

Mais especificamente, a classe `java.io.PrintStream` fornece os seguintes métodos para executar saídas simples (usamos `base_type` para indicar qualquer um dos possíveis tipos básicos):

- `print(Object o)`: imprime o objeto `o` usando seu método `toString`;
- `print(String s)`: imprime a string `s`;
- `print(base_type b)`: imprime o valor de `b` conforme seu tipo básico;
- `println(String s)`: imprime a string `s`, seguida pelo caractere de nova linha.

### Um exemplo de saída

Considere, por exemplo, o seguinte trecho de código:

```
System.out.print("Java values: ");
System.out.print(3.1415);
System.out.print(',');
System.out.print(15);
System.out.println(" (double,char,int) .");
```

Quando executado, este trecho de código produz a seguinte saída na janela console de Java:  
`Java values: 3.1415,15 (double,char,int) .`

### Entrada simples usando a classe `java.util.Scanner`

Assim como existe um objeto especial para enviar a saída para a janela de console de Java, existe também um objeto especial, chamado `System.in`, para executar a entrada de dados a partir da janela de console de Java. Tecnicamente, a entrada vem, na verdade, do “dispositivo de entrada-padrão”, o qual, por default, é o teclado do computador ecoando os caracteres na janela de console de Java. O objeto `System.in` é um objeto associado com o dispositivo de entrada padrão. Uma maneira simples de ler a entrada usando este objeto é utilizá-lo para criar um objeto `Scanner`, através da expressão:

```
new Scanner(System.in)
```

A classe `Scanner` inclui uma série de métodos convenientes para ler do fluxo de entrada. Por exemplo, o programa que segue usa um objeto `Scanner` para processar a entrada:

```
import java.io.*;
import java.util.Scanner;
public class InputExample {
    public static void main(String args[ ]) throws IOException {
        Scanner s = new Scanner(System.in);
        System.out.print("Enter your height in centimeters: ");
        float height = s.nextFloat();
        System.out.print("Enter your weight in kilograms: ");
        float weight = s.nextFloat();
        float bmi = weight/(height*height)*10000;
        System.out.println("Your body mass index is " + bmi + ". ");
    }
}
```

Quando executado, este programa gera o seguinte no console de Java:

```
Enter your height in centimeters: 180
Enter your weight in kilograms: 80.5
Your body mass index is 24.84568.
```

## Métodos de java.util.Scanner

A classe Scanner lê o fluxo de entrada e divide o mesmo em *tokens*, que são strings de caracteres contíguos separados por *delimitadores*, que correspondem a caracteres separadores. O delimitador padrão é o espaço em branco, ou seja, tokens são separados por strings de espaços, tabulações ou nova-linha, por default. Tokens tanto podem ser lidos imediatamente como strings ou um objeto Scanner pode converter um token para um tipo base, se o token estiver sintaticamente correto. Para tanto, a classe Scanner inclui os seguintes métodos para lidar com tokens:

- `hasNext()`: retorna **true** se e somente se existe mais um token no strings de entrada.
- `next()`: retorna o próximo token do fluxo de entrada; gera um erro se não existem mais tokens.
- `hasNextType(Tipo)`: retorna **true** se e somente se existe mais um token no fluxo de entrada e se pode ser interpretado como sendo do tipo base correspondente, *Tipo*, onde *Tipo* pode ser Boolean, Byte, Double, Float, Int, Long ou Short.
- `nextType(Tipo)`: retorna o próximo token do fluxo de entrada, retornando-o com o tipo base correspondente a *Tipo*; gera uma erro se não existem mais tokens ou se o próximo token não pode ser interpretado como sendo do tipo base correspondente a *Tipo*.

Além disso, objetos Scanner podem processar a entrada linha por linha, ignorando os delimitadores e mesmo podem procurar por padrões em linhas. Os métodos para processar a entrada desta forma incluem os seguintes:

- `hasNextLine()`: retorna **true** se e somente se o fluxo de entrada tem outra linha de texto.
- `nextLine()`: avança a entrada até o final da linha corrente e retorna toda a entrada que foi deixada para trás.
- `findInLine(String s)`: procura encontrar um string que combine com o padrão (expressão regular) *s* na linha corrente. Se o padrão for encontrado, ele é retornado e o scanner avança para o primeiro caractere depois do padrão. Se o padrão não for encontrado, o scanner retorna **null** e não avança.

Esses métodos podem ser usados com os anteriores, como no exemplo que segue:

```
Scanner input = new Scanner(System.in);
System.out.print("Please enter an integer: ");
while (!input.hasNextInt()) {
    input.nextLine();
    System.out.print("That's not an integer; please enter an integer: ");
}
int i = input.nextInt();
```

## 1.7 Um programa de exemplo

Nesta seção, será descrito um exemplo simples de programa Java que ilustra muitas das construções definidas anteriormente. O exemplo consiste em duas classes: CreditCard, que define objetos que representam cartões de crédito; e Test, que testa as funcionalidades da classe CreditCard. Os objetos que representam cartões de crédito, definidos pela classe CreditCard, são versões simplificadas dos cartões de crédito tradicionais. Eles têm um número de identificação, informações de identificação do proprietário e do banco que os emitiram e informações sobre o saldo corrente e o limite de crédito. Não debitam juros ou pagamentos atrasados, mas restringem pagamentos que possam fazer com que o saldo vá além do limite de gastos.

## A classe CreditCard

A classe CreditCard é apresentada no Trecho de código 1.5. Ela define cinco variáveis de instância, todas exclusivas, e possui um construtor simples que inicializa essas variáveis.

Ela também define cinco **métodos de acesso** que permitem acessar o valor corrente dessas variáveis de instância. Evidentemente, as variáveis de instância poderiam ter sido definidas como públicas, o que faria com que os métodos de acesso fossem duvidosos. A desvantagem dessa abordagem direta, porém, é que permite ao usuário modificar as variáveis de instância do objeto diretamente, enquanto que, em muitos casos como este, é preferível restringir a alteração de variáveis de instância a métodos especiais chamados de **métodos de atualização**. No Trecho de código 1.5, inclui-se dois métodos de atualização, chargeIt e makePayment.

Além disso, é conveniente incluir **métodos de ação**, que com freqüência definem as ações específicas do comportamento do objeto. Para demonstrar isso, define-se um método de ação, o printCard, como um método estático, que também está incluído no Trecho de código 1.5.

## A classe test

A classe CreditCard é testada na classe Test. Observa-se aqui o uso de um arranjo de objetos CreditCard, wallet, e como se usam iterações para fazer débitos e pagamentos. Apresenta-se o código completo da classe Test no Trecho de código 1.6. Para simplificar, a classe Test não produz nenhum gráfico elaborado, simplesmente envia a saída para o console de Java. Apresenta-se esta saída no Trecho de código 1.7. Observa-se a diferença na maneira em que se utilizam os métodos não-estáticos chargeIt e makePayment e o método estático printCard.

```
public class CreditCard {
    // Variáveis de instância:
    private String number;
    private String name;
    private String bank;
    private double balance;
    private int limit;
    // Construtor:
    CreditCard(String no, String nm, String bk, double bal, int lim) {
        number = no;
        name = nm;
        bank = bk;
        balance = bal;
        limit = lim;
    }
    // Métodos de acesso:
    public String getNumber() { return number; }
    public String getName() { return name; }
    public String getBank() { return bank; }
    public double getBalance() { return balance; }
    public int getLimit() { return limit; }
    // Métodos de ação:
    public boolean chargeIt(double price) { // Debita
        if (price + balance > (double) limit)
            return false; // Não há dinheiro suficiente para debitar
        balance += price;
        return true; // Neste caso o débito foi efetivado
    }
}
```

```

public void makePayment(double payment) { // Faz um pagamento
    balance -= payment;
}
public static void printCard(CreditCard c) { // Imprime informações sobre o cartão
    System.out.println("Number = " + c.getNumber());
    System.out.println("Name = " + c.getName());
    System.out.println("Bank = " + c.getBank());
    System.out.println("Balance = " + c.getBalance()); // conversão implícita
    System.out.println("Limit = " + c.getLimit()); // conversão implícita
}
}

```

Trecho de código 1.5 A classe CreditCard.

```

public class Test {
    public static void main(String[] args) {
        CreditCard wallet[] = new CreditCard[10];
        wallet[0] = new CreditCard("5391 0375 9387 5309",
                                  "John Bowman", "California Savings", 0.0, 2500);
        wallet[1] = new CreditCard("3485 0399 3395 1954",
                                  "John Bowman", "California Federal", 0.0, 3500);
        wallet[2] = new CreditCard("6011 4902 3294 2994",
                                  "John Bowman", "California Finance", 0.0, 5000);
        for (int i=1; i<=16; i++) {
            wallet[0].chargeIt((double)i);
            wallet[1].chargeIt(2.0*i); // conversão implícita
            wallet[2].chargeIt((double)3*i); // conversão explícita
        }
        for (int i=0; i<3; i++) {
            CreditCard.printCard(wallet[i]);
            while (wallet[i].getBalance() > 100.0) {
                wallet[i].makePayment(100.0);
                System.out.println("New balance = " + wallet[i].getBalance());
            }
        }
    }
}

```

Trecho de código 1.6 A classe Test.

```

Number = 5391 0375 9387 5309
Name = John Bowman
Bank = California Savings
Balance = 136.0
Limit = 2500
New balance = 36.0
Number = 3485 0399 3395 1954
Name = John Bowman
Bank = California Federal
Balance = 272.0
Limit = 3500
New balance = 172.0
New balance = 72.0

```

```

Number = 6011 4902 3294 2994
Name = John Bowman
Bank = California Finance
Balance = 408.0
Limit = 5000
New balance = 308.0
New balance = 208.0
New balance = 108.0
New balance = 8.0

```

**Trecho de código 1.7** Saída da classe Test.

## 1.8 Classes aninhadas e pacotes

A linguagem Java usa uma abordagem prática e genérica para organizar as classes de um programa. Toda a classe pública definida em Java deve ser fornecida em um arquivo separado. O nome do arquivo é o nome da classe com uma terminação *.java*. Desta forma, a classe **public class** SmartBoard, é definida em um arquivo chamado *SmartBoard.java*. Nesta seção, são apresentadas duas maneiras interessantes pelas quais Java permite que várias classes sejam organizadas.

### Classes aninhadas

Java permite que definições de classes sejam feitas dentro, isto é, **aninhadas** dentro das definições de outras classes. Este é um tipo de construção útil que será explorada diversas vezes neste livro na implementação de estruturas de dados. O uso principal de classes aninhadas é para definir uma classe fortemente conectada com outra. Por exemplo, a classe de um editor de textos pode definir uma classe cursor relacionada. Definindo a classe cursor como classe aninhada dentro da definição da classe editor, mantém-se a definição destas duas classes altamente relacionadas juntas no mesmo arquivo. Além disso, permite que ambas acessem os métodos públicos uma da outra. Um aspecto técnico relacionado a classes aninhadas é que classes aninhadas podem ser declaradas como **static**. Esta declaração implica que a classe aninhada está associada com a classe mais externa, mas não com uma instância da classe mais externa, isto é, um objeto específico.

### Pacotes

Um conjunto de classes relacionadas, todas pertencentes ao mesmo subdiretório, pode ser um **package** (pacote) Java. Cada arquivo em um pacote se inicia com a linha:

```
package nome_do_pacote;
```

O subdiretório que contém o pacote deve ter o mesmo nome que o pacote. É possível, também, definir um pacote em um único arquivo que contenha diversas definições de classe, mas quando for compilado, todas as classes o serão em arquivos separados no mesmo subdiretório.

Em Java, pode-se usar classes que estão definidas em outros pacotes prefixando os nomes das classes com pontos (isto é, usando o caractere ".") que corresponde à estrutura de diretório dos outros pacotes.

```

public boolean Temperature(TA.Measures.Thermometer thermometer,
                           int temperature) {
    // ...
}

```

A função Temperature recebe a classe Thermometer como parâmetro. Thermometer é definida no pacote TA, em um subpacote chamado Measures. Os pontos em TA.Measures.Thermometer têm correspondência direta com a estrutura de diretório do pacote TA.

Toda a digitação necessária para fazer referência a uma classe fora do pacote corrente pode-se tornar cansativa. Em Java, é possível usar a palavra reservada **import** para incluir classes externas ou pacotes inteiros no arquivo corrente. Para importar uma classe individual de um pacote específico, digita-se no início do arquivo o seguinte:

```
import nome_do_pacote.nome_da_classe;
```

Por exemplo, pode-se digitar

```
package Project;
import TA.Measures.Thermometer;
import TA.Measures.Scale;
```

no início do pacote Project para indicar que se está importando as classes TA.Measures.Thermometer e TA.Measures.Scale. O ambiente de execução de Java irá procurar essas classes para verificar os identificadores com as classes, métodos e variáveis de instância que se usa no programa.

Também se pode importar um pacote inteiro utilizando a seguinte sintaxe:

```
import (packageName).*;
```

Por exemplo:

```
package student;
import TA.Measures.*;
public boolean Temperature(Thermometer thermometer, int temperature) {
    // ...
}
```

Nos casos em que dois pacotes têm classes com o mesmo nome, **deve-se** referenciar especificamente o pacote que contém a classe. Por exemplo, supondo que ambos os pacotes Gnomes e Cooking tenham uma classe chamada Mushroom ("cogumelo"). Se for determinado um comando **import** para cada pacote, deve-se especificar que classe se quer designar:

```
Gnomes.Mushroom shroom = new Gnomes.Mushroom ("purple");
Cooking.Mushroom topping = new Cooking.Mushroom ();
```

Se não for especificado o pacote (ou seja, no exemplo anterior apenas foi empregada uma variável do tipo **Mushroom**), o compilador irá sinalizar um erro de "classe ambígua".

Resumindo a estrutura de um programa Java, pode-se ter variáveis de instância e métodos dentro de uma classe, além de classes dentro de um pacote.

## 1.9 Escrevendo um programa em Java

O processo de escrever um programa em Java envolve três etapas fundamentais:

1. projeto,
2. codificação,
3. teste e depuração.

Cada uma será resumida nesta seção.

### 1.9.1 Projeto

A etapa de projeto é talvez o passo mais importante no processo de escrever um programa. É na fase de projeto que se decide como dividir as tarefas do programa em classes, como essas classes irão interagir com os dados que irão armazenar e que ações cada uma irá executar. Um dos maiores desafios com o qual os programadores iniciantes se deparam em Java é determinar que classes definir para executar as tarefas do seu programa. Ainda que seja difícil obter prescrições genéricas, existem algumas regras práticas que podem ser aplicadas quando se está procurando definir as classes:

- **Responsabilidades:** dividir o trabalho entre diferentes *atores*, cada um com responsabilidades diferentes. Procurar descrever as responsabilidades usando verbos de ação. Os atores irão formar as classes do programa.
- **Independência:** definir, se possível, o trabalho de cada classe de forma independente das outras classes. Subdividir as responsabilidades entre as classes de maneira que cada uma tenha autonomia sobre algum aspecto do programa. Fornecer os dados (como variáveis de instância) para as classes que têm competência sobre as ações que requerem acesso a esses dados.
- **Comportamento:** as consequências de cada ação executada por uma classe terão de ser bem compreendidas pelas classes que interagem com ela. Portanto, é preciso definir o comportamento de cada uma com cuidado e precisão. Esses comportamentos irão definir os métodos que a classe executa. O conjunto de comportamentos de uma classe é algumas vezes chamado de *protocolo*, porque espera-se que os comportamentos de uma classe sejam agrupados como uma unidade coesa.

A definição das classes juntamente com seus métodos e variáveis de instância determina o projeto de um programa Java. Um bom programador, com o tempo, vai desenvolver naturalmente grande habilidade em executar essas tarefas à medida que a experiência lhe ensinar a observar padrões nos requisitos de um programa que se parecem com padrões já vistos.

---

### 1.9.2 Pseudocódigo

Freqüentemente, programadores são solicitados a descrever algoritmos de uma maneira que seja compreensível para olhos humanos, em vez de escrever um código real. Tais descrições são chamadas de *pseudocódigo*. Pseudo-código não é um programa de computador, mas é mais estruturado que a prosa normal. Pseudo-código é uma mistura de língua-natural com estruturas de programação de alto-nível que descrevem as idéias principais que estão por trás da implementação de uma estrutura de dados ou algoritmo. Não existe, portanto, uma definição precisa de uma linguagem de *pseudocódigo*, em razão de sua dependência da língua natural. Ao mesmo tempo, para auxiliar na clareza, o pseudo-código mistura língua natural com construções de padrão de linguagens de programação. As construções que foram escolhidas são consistentes com as modernas linguagens de alto nível, tais como C, C++ e Java.

Estas construções incluem:

- **Expressões:** usam-se símbolos matemáticos padrão para expressões numéricas e booleanas. Usa-se a seta para esquerda ( $\leftarrow$ ) como operador de atribuição em comandos de atribuição (equivalente ao operador = de Java) e o sinal de igual (=) para o relacional de igualdade em expressões booleanas (equivalente ao relacional “==” em Java).
- **Declarações de métodos:** **Algoritmo** nome(*param1*, *param2*, ... ) declara um nome de método novo e seus parâmetros.

- **Estruturas de decisão:** se condição, então ações caso verdade [senão, ações caso falso]. Usa-se identação para indicar quais ações devem ser incluídas nas ações caso verdade e nas ações caso falso.
- **Laços enquanto:** enquanto condição, faça ações. Usa-se indentação para indicar quais ações devem ser incluídas no laço.
- **Laços repete:** repete ações até condição. Usa-se indentação para indicar que ações devem ser incluídas no laço.
- **Laços for:** for definição de variável de incremento, faça ações. Usa-se identação para indicar que ações devem ser incluídas no laço.
- **Indexação de arranjos:** A[i] representa a  $i$ -ésima célula do arranjo A. As células de um arranjo A com  $n$  células são indexadas de A[0] até A[n - 1] (de forma consistente com Java).
- **Chamadas de métodos:** objeto.método(argumentos) (objeto é opcional se for subentendido).
- **Retorno de métodos:** retorna valor. Esta operação retorna o valor especificado para o método que chamou o método corrente.
- **Comentários:** { os comentários vão aqui }. Os comentários são colocados entre chaves.

Quando se escreve pseudo-código, deve-se ter em mente que se está escrevendo para um leitor humano, não para um computador. Assim, o esforço deve ser no sentido de comunicar idéias de alto-nível, e não detalhes de implementação. Ao mesmo tempo, não se deve omitir passos importantes. Como em muitas outras formas de comunicação humana, encontrar o balanceamento correto é uma habilidade importante, que é refinada pela prática.

### 1.9.3 Codificação

Como mencionado anteriormente, um dos passos-chave na codificação de um programa orientado a objetos é codificar a partir de descrições de classes e seus respectivos métodos. Para acelerar o desenvolvimento dessa habilidade, serão estudados, em diferentes momentos ao longo deste texto, vários **padrões de projeto** para os projetos de programas orientados a objeto (ver Seção 2.1.3). Esses padrões fornecem moldes para a definição de classes e interações entre essas classes.

Muitos programadores não fazem seus projetos iniciais em um computador, e sim usando **cartões CRC**. *Componente-responsabilidade-colaborador*, ou CRC, são simples cartões indexados que subdividem as tarefas especificadas para um programa. A idéia principal por trás desta ferramenta é que cada cartão represente um componente, o qual, no final, se transformará em uma classe de nosso programa. Escreve-se o nome do componente no topo do cartão. No lado esquerdo do mesmo, anotam-se as responsabilidades desse componente. No lado direito, listam-se os colaboradores desse componente, isto é, os outros componentes com os quais o primeiro terá de interagir de maneira a cumprir suas finalidades. O processo de projeto é iterativo através de um ciclo ação/ator, onde primeiro se identifica uma ação (ou seja, uma responsabilidade) e, então, se determina o ator (ou seja, um componente) mais adequado para executar tal ação. O processo de projeto está completo quando se tiver associado atores para todas as ações.

A propósito, ao utilizar cartões indexados para executar nosso projeto, assume-se que cada componente terá um pequeno conjunto de responsabilidades e colaboradores. Esta premissa não é acidental, uma vez que auxilia a manter os programas gerenciáveis.

Uma alternativa ao uso dos cartões CRC é o uso de diagramas UML (Linguagem de Modelagem Unificada\*) para expressar a organização de um programa e pseudo-código para expressar algoritmos. Diagramas UML são uma notação visual padrão para expressar projetos orientados a

\* N. de T. Unified Modeling Language.

objetos. Existem muitas ferramentas auxiliadas por computador capazes de construir diagramas UML. A descrição de algoritmos em pseudo-código, por outro lado, é uma técnica que será utilizada ao longo deste livro.

Tendo decidido sobre as classes de nosso programa, juntamente com suas responsabilidades, pode-se começar a codificação. Cria-se o código propriamente dito das classes do programa usando tanto um editor de textos independente (por exemplo emacs, WordPad ou vi) como um editor embutido em um *ambiente integrado de desenvolvimento* (IDE<sup>\*</sup>), tal como o Eclipse ou o Borland JBuilder.

Após completar a codificação de uma classe (ou pacote), se compila o arquivo para código executável usando um compilador. Quando não se está usando um IDE, então se compila o programa chamando um programa tal como javac sobre o arquivo. Estando em uso um IDE, então se compila o programa clicando o botão apropriado. Felizmente se o programa não tiver erros de sintaxe, então o processo de compilação irá criar arquivos com a extensão “.class”.

Se o programa contiver erros de sintaxe, estes serão identificados, e se terá de voltar ao editor de textos para consertar as linhas de código com problema. Eliminados todos os erros de sintaxe e criado o código compilado correspondente, pode-se executar o programa tanto chamando um comando, tal como “java” (fora de um IDE), ou clicando o botão de execução apropriado (dentro de um IDE). Quando um programa Java estiver executando dessa forma, o ambiente de execução localiza os diretórios contendo as classes criadas e quaisquer outras referenciadas a partir destas, usando uma variável especial de ambiente do sistema operacional. Essa variável é chamada de “CLASSPATH”, e a ordem dos diretórios a serem pesquisados é fornecida como uma lista de diretórios, separados por vírgulas se em Unix/Linux ou por ponto-e-vírgulas se em DOS/Windows. Um exemplo de atribuição para a variável CLASSPATH no sistema operacional DOS/Windows pode ser o seguinte:

```
SET CLASSPATH=.;C:\java;C:\Program Files\Java\
```

Um exemplo de atribuição para CLASSPATH no sistema operacional Unix/Linux pode ser:

```
setenv CLASSPATH ".:/usr/local/java/lib:/usr/netscape/classes"
```

Em ambos os casos, o ponto (“.”) se refere ao diretório atual a partir do qual o ambiente de execução foi chamado.

## Javadoc

Para incentivar o bom uso de comentários em bloco e a produção automática de documentação, o ambiente de programação Java vem com um programa para a geração de documentação chamado *javadoc*. Esse programa examina uma coleção de arquivos fontes Java que tenham sido comentados usando-se certas palavras reservadas, chamadas de *tags*, e produz uma série de documentos HTML que descrevem as classes, métodos, variáveis e constantes contidas nestes arquivos. Por razões de espaço, não se usa o estilo de comentários do javadocs em todos os programas exemplificadores contidos neste livro, mas se incluiu um exemplo de javadoc no Trecho de código 1.8, bem como em outros disponíveis no site da Web que acompanha este livro.

Cada comentário javadoc é um comentário em bloco que se inicia com “/\*”, termina com “\*/” e tem cada linha entre estas duas iniciada por um único asterisco, “\*” que é ignorado. Presupõe-se que o bloco de comentário deve começar com uma frase descritiva seguida por uma linha em branco, e posteriormente por linhas especiais que começam por tags javadoc. Um comentário em bloco que venha imediatamente antes de uma definição de classe, uma declaração de variável ou uma definição de método é processado pelo javadoc em um comentário que se refere à classe, à variável ou ao método.

\* N. de T. A abreviatura, já consagrada, corresponde à expressão em inglês *integrated development environment*.

```


    /**
     * Esta classe define um ponto (x,y) não alterável no plano
     *
     * @author Michael Goodrich
     */
    public class XYPoint {
        private double x,y; // variáveis de instância privada para as coordenadas

        /**
         * Constrói um ponto (x,y) em uma localização específica
         *
         * @param xCoor A abscissa do ponto
         * @param yCoor A ordenada do ponto
         */

        public XYPoint(double xCoor, double yCoor) {
            x = xCoor;
            y = yCoor;
        }

        /**
         * Retorna o valor da abscissa
         *
         * @return abscissa
         */
        public double getX() { return x; }

        /**
         * Retorna o valor da ordenada
         *
         * @return ordenada
         */
        public double getY() { return x; }
    }


```

**Trecho de código 1.8** Um exemplo de definição de classe usando o estilo javadoc de comentário. Observa-se que esta classe inclui apenas duas variáveis de instância, um construtor e dois métodos de acesso.

As tags javadoc mais importantes são as seguintes:

- `@author texto`: identifica os autores (um por linha) de uma classe;
- `@exception descrição do nome de uma exceção`: identifica uma condição de erro sinalizada por este método (ver Seção 2.3);
- `@param descrição de um nome de parâmetro`: identifica um parâmetro aceito por este método;
- `@return descrição`: descreve o tipo de retorno de um método e seu intervalo de valores.

Existem outras tags como estas; o leitor interessado deve consultar a documentação online do javadoc para um estudo mais aprofundado.

### Clareza e estilo

É possível fazer programas fáceis de ler e entender. Bons programadores devem, portanto, ser cuidadosos com seu estilo de programação, desenvolvendo-o de forma a comunicar os aspectos importantes do projeto de um programa tanto para os usuários como para os computadores.

Alguns dos princípios mais importantes sobre bons estilos de programação são os seguintes:

- *Usar nomes significativos para identificadores.* Devem-se escolher nomes que possam ser lidos em voz alta, que refletem a ação, a responsabilidade ou os dados que o identificador está nomeando. A tradição na maioria dos círculos de Java é usar maiúsculas na primeira letra de cada palavra que compõem um identificador, excetuando-se a primeira palavra de identificadores de variáveis ou métodos. Então, segundo esta tradição, “Date”, “Vector”, “DeviceManage” identificam classes e “isFull()”, “insertItem()”, “studentName” e “studentHeigth” referem-se, respectivamente, a métodos e a variáveis.
- *Usar constantes ou tipos enumerados em vez de valores.* A clareza, a robustez e a manutenção serão melhoradas se forem incluídos uma série de valores constantes em uma definição de classe. Estes poderão então ser usados nesta e em outras classes para fazer referência a valores especiais desta classe. A tradição Java é usar apenas maiúsculas em tais constantes, como mostrado abaixo:

```
public class Student {  
    public static final int MIN_CREDITS = 12; // créditos mínimos por período  
    public static final int MAX_CREDITS = 24; // créditos máximos por período  
    public static final int FRESHMAN = 1; // código de calouro  
    public static final int SOPHOMORE = 2; // código de aluno do primeiro ano  
    public static final int JUNIOR = 3; // código para júnior  
    public static final int SENIOR = 4; // código para sênior  
  
    // definições de variáveis de instância, construtores e métodos seguem aqui...  
}
```

- *Indentar os blocos de comandos.* Os programadores normalmente indentam cada bloco de comandos com quatro espaços; neste livro, entretanto, usam normalmente dois espaços, para evitar que o código extravase as margens do livro.
- *Organizar as classes conforme a seguinte ordem:*
  1. constantes,
  2. variáveis de instância,
  3. construtores,
  4. métodos.

Alguns programadores Java preferem colocar as declarações de variáveis de instância por último. Aqui, opta-se por colocá-las antes, de forma que se possa ler cada classe seqüencialmente, compreendendo os dados com que cada método está lidando.

- *Usar comentários para acrescentar significado ao programa e explicar construções ambíguas ou confusas.* Comentários de linha são úteis para explicações rápidas e não precisam ser frases completas. Comentários em bloco são úteis para explicar os propósitos de um método ou as seções de código complicadas.

---

#### 1.9.4 Teste e depuração

Teste é o processo de verificar a correção de um programa; depuração é o processo de seguir a execução de um programa para descobrir seus erros. Teste e depuração são, em geral, as atividades que mais consomem tempo durante o desenvolvimento de um programa.

## Teste

Um plano de testes cuidadoso é parte essencial da escrita de um programa. Apesar de a verificação da correção de um programa para todas as entradas possíveis ser normalmente impraticável, pode-se privilegiar a execução do programa a partir de subconjuntos representativos das entradas. Na pior das hipóteses, deve-se ter certeza de que cada método do programa tenha sido testado pelo menos uma vez (cobertura de método). Melhor ainda, cada linha de código do programa deve ser executada pelo menos uma vez (cobertura de comandos).

Em geral, as entradas dos programas falham em *casos especiais*. Tais casos precisam ser cuidadosamente identificados e testados. Por exemplo, quando se testa um método que ordena (isto é, coloca em ordem) um arranjo de inteiros, deve-se considerar as seguintes entradas:

- se o arranjo tiver tamanho zero (nenhum elemento);
- se o arranjo tiver um elemento;
- se todos os elementos do arranjo forem iguais;
- se o arranjo já estiver ordenado;
- se o arranjo estiver ordenado na ordem inversa.

Além das entradas especiais para o programa, deve-se também analisar condições especiais para as estruturas usadas pelo programa. Por exemplo, usando-se um arranjo para armazenar dados, é preciso ter certeza de que os casos-limite, tais como a inserção/remoção no início ou no fim do arranjo que armazena os dados, estão sendo convenientemente tratados.

Se é essencial usar conjuntos de testes definidos manualmente, também é fundamental executar o programa a partir de grandes conjuntos de dados gerados randomicamente. A classe Random do pacote `java.util` oferece vários métodos para a geração de números randômicos.

Existe uma hierarquia entre as classes e métodos de um programa, induzida pelas relações de “ativador-ativado”. Isto é, um método *A* está acima de um método *B* na hierarquia, se *A* chamar *B*. Existem duas estratégias de teste principais, *top-down* e *bottom-up*, que diferem na ordem em que os métodos são testados.

O teste bottom-up é executado desde métodos de mais baixo nível até os de mais alto nível. Ou seja, métodos de mais baixo nível que não ativam outros métodos são testados primeiro, seguidos pelos métodos que chamam apenas um método de baixo nível, e assim por diante. Esta estratégia garante que os erros encontrados em um método nunca são causados por um método de nível mais baixo aninhado no mesmo.

O teste bottom-up é executado do topo para a base da hierarquia de métodos. Normalmente é usado em conjunto com *terminadores*, uma técnica de rotina de inicialização que substitui métodos de mais baixo nível por um *tampão*<sup>\*</sup>, um substituto para o método que simula a saída do método original. Por exemplo, se o método *A* chama o método *B* para pegar a primeira linha de um arquivo, quando se testa *A* pode-se substituir *B* por um tampão que retorna uma string fixa.

## Depuração

A técnica mais simples de depuração consiste em usar *comandos de impressão* (usando o método `System.out.println(string)`) para rastrear os valores das variáveis durante a execução do programa. Um problema desta abordagem é que os comandos de impressão, por vezes, necessitam ser removidos ou comentados antes de o programa poder ser executado.

Uma melhor abordagem é executar o programa com um *depurador*, que é um ambiente especializado para controlar e monitorar a execução de um programa. A funcionalidade básica

\* N. de T. Em inglês, *stub*.

oferecida por um depurador é a inserção de *pontos de parada*\* no código. Quando um programa é executado com um depurador, ele interrompe a cada ponto de parada. Enquanto o programa está parado, o valor corrente das variáveis pode ser verificado. Além de pontos de parada fixos, depuradores mais avançados permitem a especificação de *pontos de parada condicionais*, que são disparados apenas se uma determinada condição for satisfeita.

As ferramentas-padrão de Java incluem um depurador básico chamado jdb, controlado por linhas de comando. Os IDEs para programação em Java oferecem ambientes de depuração avançados com interface gráfica com o usuário.

## 1.10 Exercícios

Para obter ajuda e o código fonte dos exercícios, visite [java.datastructures.net](http://java.datastructures.net).

### Reforço

- R-1.1 Suponha que seja criado um arranjo *A* de objetos GameEntry, que possui um campo inteiro scores, e que *A* seja clonado e o resultado seja armazenado em um arranjo *B*. Se o valor de *A[4].score* for imediatamente alterado para 550, qual o valor do campo score do objeto GameEntry referenciado por *B[4]*?
- R-1.2 Modifique a classe CreditCard do Trecho de código 1.5 de maneira a debitar juros em cada pagamento.
- R-1.3 Modifique a classe CreditCard do Trecho de código 1.5 de maneira a debitar uma taxa por atraso para qualquer pagamento feito após a data de vencimento.
- R-1.4 Modifique a classe CreditCard do Trecho de código 1.5 para incluir *métodos modificadores* que permitam ao usuário modificar variáveis internas da classe CreditCard de forma controlada.
- R-1.5 Modifique a declaração do primeiro laço **for** da classe Test do Trecho de código 1.6 de maneira que os débitos possam, mais cedo ou mais tarde, fazer com que um dos três cartões ultrapasse seu limite de crédito. Qual é esse cartão?
- R-1.6 Escreva uma pequena função em Java, *inputAllBaseTypes* que recebe diferentes valores de cada um dos tipos base na entrada padrão e o imprime de volta no dispositivo de saída padrão.
- R-1.7 Escreva uma classe Java, Flower, que tenha três variáveis de instância dos tipos **String**, **int** e **float** representando, respectivamente, o nome da flor, seu número de pétalas e o preço. A classe pode incluir um construtor que initialize cada variável adequadamente, além de métodos para alterar o valor de cada tipo e recuperar o valor de cada tipo.
- R-1.8 Escreva uma pequena função em Java, *isMultiple*, que recebe dois valores **long**, *n* e *m*, e retorna **true** se e somente se *n* é múltiplo de *m*, isto é, *n = mi* para algum inteiro *i*.
- R-1.9 Escreva uma pequena função Java *isOdd*, que recebe um **int** *i* e retorna **true** se e somente se *i* é par. Entretanto, esta função não pode usar operadores de multiplicação, módulo ou divisão.

\* N. de T. Em inglês, *breakpoints*.

- R-1.10 Escreva uma pequena função em Java que receba um inteiro  $n$  e retorne a soma de todos os inteiros menores que  $n$ .
- R-1.11 Escreva uma pequena função em Java que receba um inteiro  $n$  e retorne a soma de todos os inteiros pares menores que  $n$ .

## Criatividade

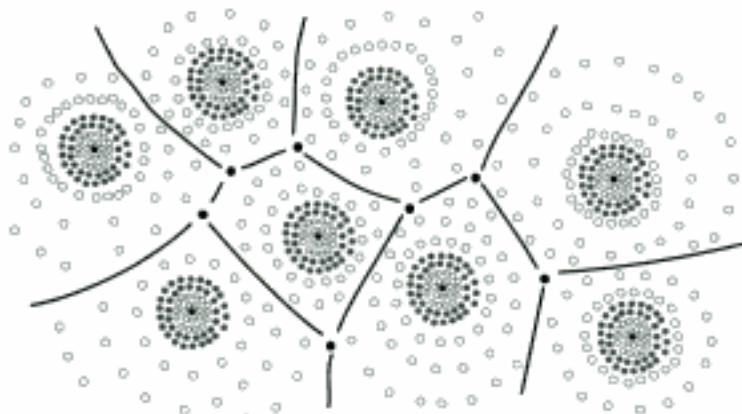
- C-1.1 Escreva uma pequena função Java que recebe um arranjo de valores `int` e determina se existe um par de números no arranjo cujo produto seja par.
- C-1.2 Escreva um método Java que recebe um arranjo de valores `int` e determina se todos os números são diferentes entre si (isto é, se são valores distintos).
- C-1.3 Escreva um método em Java que receba um arranjo contendo o conjunto de todos os inteiros no intervalo de 1 a 52 e embaralhe os mesmos de forma aleatória. O método deve exibir as possíveis sequências com igual probabilidade.
- C-1.4 Escreva um pequeno programa em Java que exiba todas as strings possíveis de serem formadas usando os caracteres 'c', 'a', 'x', 'b', 'o' e 'n' apenas uma vez.
- C-1.5 Escreva um pequeno programa em Java que receba linhas de entrada pelo dispositivo de entrada padrão, e escreva as mesmas no dispositivo de saída padrão na ordem contrária. Isto é, cada linha é exibida na ordem correta, mas a ordem das linhas é invertida.
- C-1.6 Escreva um pequeno programa em Java que receba dois arranjos  $a$  e  $b$  de tamanho  $n$  que armazenam valores `int` e retorne o produto escalar de  $a$  por  $b$ . Isto é, retorna um arranjo  $c$  de tamanho  $n$  onde  $c[i] = a[i] \cdot b[i]$ , para  $i = 0, \dots, n - 1$ .

## Projetos

- P-1.1 Uma punição comum para alunos de escola é escrever a mesma frase várias vezes. Escreva um programa executável em Java que escreva a mesma frase uma centena de vezes: "Eu não mandarei mais spam para meus amigos". Seu programa deve numerar as frases e "acidentalmente" fazer oito erros aleatórios diferentes de digitação.
- P-1.2 (Para aqueles que conhecem os métodos de interface gráfica com o usuário em Java.) Defina uma classe `GraphicalTest` que teste a funcionalidade da classe `CreditCard` do Trecho de código 1.5, usando campos de entrada de texto e botões.
- P-1.3 O *paradoxo do aniversário* diz que a probabilidade de duas pessoas em uma sala terem a mesma data de aniversário é maior que 50% desde que  $n$ , o número de pessoas na sala, seja maior que 23. Esta propriedade não é realmente um paradoxo, mas muitas pessoas se surpreendem. Projete um programa em Java que possa testar esse paradoxo por uma série de experimentos sobre aniversários gerados aleatoriamente, testando o paradoxo para  $n = 5, 10, 15, 20, \dots, 100$ .

## Observações sobre o capítulo

Para mais informações sobre a linguagem de programação Java, indicamos ao leitor alguns dos melhores livros de Java: Arnold e Gosling [7], Campione e Walrath [19], Cornell e Horstmann [26], Flanagan [34] e Horstmann [51], assim como a página de Java da Sun (<http://www.java.sun.com>).



## Conteúdo

---

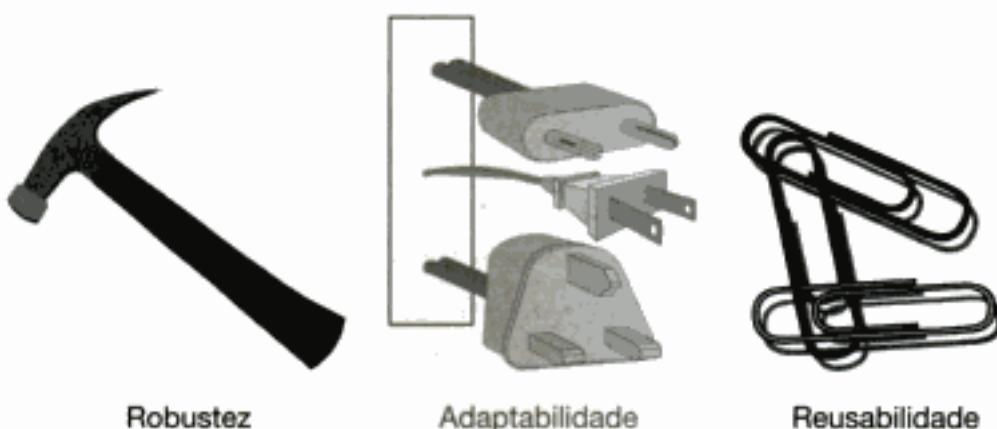
<b>2.1</b>	<b>Objetivos, princípios e padrões</b>	<b>70</b>
2.1.1	Objetivos do projeto orientado a objetos	70
2.1.2	Princípios de projeto orientado a objetos	71
2.1.3	Padrões de projeto	73
<b>2.2</b>	<b>Herança e polimorfismo</b>	<b>74</b>
2.2.1	Herança	74
2.2.2	Polimorfismo	75
2.2.3	Usando herança em Java	76
<b>2.3</b>	<b>Exceções</b>	<b>84</b>
2.3.1	Lançando exceções	84
2.3.2	Capturando exceções	85
<b>2.4</b>	<b>Interfaces e classes abstratas</b>	<b>87</b>
2.4.1	Implementando interfaces	87
2.4.2	Herança múltipla e interfaces	89
2.4.3	Classes abstratas e tipagem forte	90
<b>2.5</b>	<b>Conversão e genéricos</b>	<b>91</b>
2.5.1	Conversão	91
2.5.2	Genéricos	94
<b>2.6</b>	<b>Exercícios</b>	<b>96</b>

## 2.1 Objetivos, princípios e padrões

Como o próprio nome indica, os “atores” principais do paradigma de projetos orientados a objetos são chamados de *objetos*. Um objeto se origina de uma *classe*, que é uma especificação tanto dos *campos* de dados, também chamados de *variáveis de instância* que um objeto contém, como dos *métodos* (operações) que pode executar. Cada classe apresenta para o mundo exterior uma visão concisa e consistente dos objetos que são instâncias dessa classe, sem detalhes desnecessários ou acesso às estruturas internas dos objetos. Essa abordagem de computação visa a atingir diversos objetivos e a incorporar vários princípios de projeto que serão discutidos neste capítulo.

### 2.1.1 Objetivos do projeto orientado a objetos

Implementações de software devem buscar *robustez*, *adaptabilidade* e *reusabilidade* (ver Figura 2.1).



**Figura 2.1** Objetivos de um projeto orientado a objetos.

#### Robustez

Todo bom programador quer produzir software que seja correto, o que significa um programa que produz as saídas certas para todas as entradas previstas pela aplicação do programa. Além disso, é desejável que um software seja *robusto*, ou seja, capaz de lidar com entradas inesperadas que não estão explicitamente definidas em sua aplicação. Por exemplo, se um programa está esperando por um inteiro positivo (isto é, representando o preço de um item), mas recebe um inteiro negativo, deve ser capaz de se recuperar com elegância desse erro. No caso de *aplicações de missão crítica*, nas quais um erro de software pode causar ferimentos ou a perda da vida, a utilização de um software que não é robusto pode ser mortal. A importância disso foi enfatizada na década de 80, em acidentes envolvendo o Therac-25, uma máquina de terapia com radiação que aplicou superdoses em seis pacientes entre 1985 e 1987, alguns dos quais morreram de complicações resultantes das doses excessivas de radiação. Em todos os seis acidentes, detectou-se que a causa era proveniente de um erro de software.

#### Adaptabilidade

Os projetos modernos de software, tais como editores de texto, navegadores para Web e aplicativos de pesquisa na Internet, são normalmente programas grandes que devem durar muitos anos. O software, portanto, deve ser capaz de evoluir ao longo do tempo em resposta a alterações nas condições de seu ambiente. Sendo assim, a *adaptabilidade* (também chamada de *capacidade*



Um TAD é materializado por uma estrutura de dados concreta que, em Java, é modelada por uma *classe*. Uma classe define os dados que serão armazenados e as operações suportadas pelos objetos que são instâncias dessa classe. Além disso, ao contrário das interfaces, as classes especificam *como* as operações são executadas. Diz-se que uma classe em Java *implementa uma interface* quando seus métodos incluem todos os métodos declarados na interface, provendo um corpo para os mesmos. Entretanto, uma classe pode ter mais métodos que aqueles definidos pela interface.

### Encapsulamento

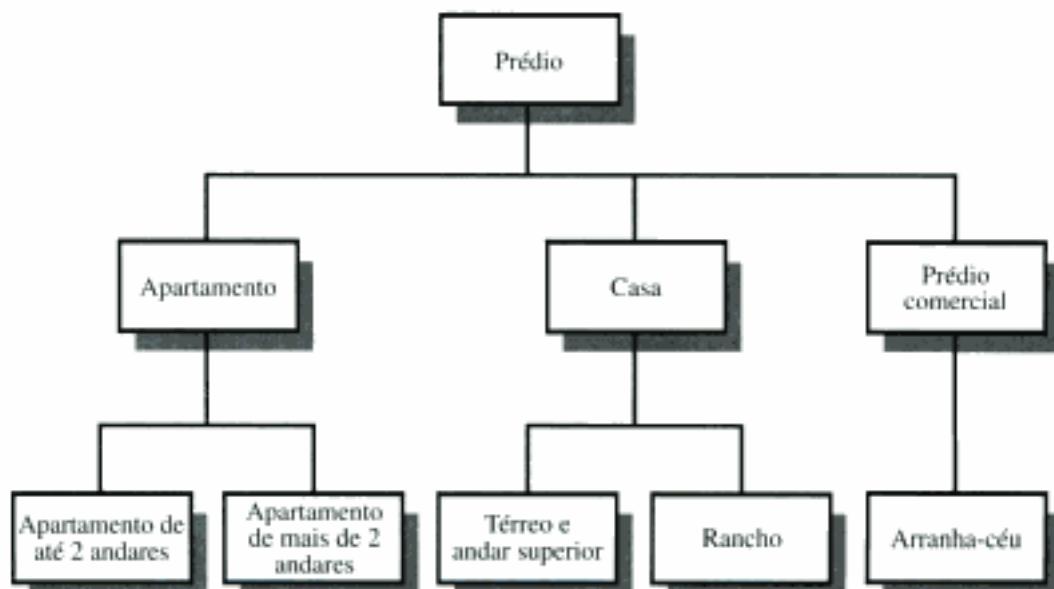
Outro princípio importante em projeto orientado a objetos é o conceito de *encapsulamento*, que estabelece que os diferentes componentes de um sistema de software não devem revelar detalhes de suas respectivas implementações. Uma das maiores vantagens do encapsulamento é que ele oferece ao programador liberdade na implementação dos detalhes do sistema. A única restrição ao programador é manter a interface abstrata que é percebida pelos de fora.

### Modularidade

Além de abstração e do encapsulamento, outro princípio fundamental de projeto orientado a objetos é a *modularidade*. Sistemas modernos de software normalmente são compostos por vários componentes diferentes que devem interagir corretamente, fazendo com que o sistema como um todo funcione de forma adequada. Para manter essas interações corretas, é necessário que os diversos componentes estejam bem organizados. Na abordagem orientada a objetos, essa organização se centra no conceito de *modularidade*. A modularidade se refere a uma estrutura de organização na qual os diferentes componentes de um sistema de software são divididos em unidades funcionais separadas.

### Organização hierárquica

A estrutura imposta pela modularidade auxilia a tornar o software reutilizável. Se os módulos do software forem escritos de uma forma abstrata para resolver problemas genéricos, então os módulos podem ser reutilizados quando instâncias do mesmo problema geral surgirem em outros contextos.



**Figura 2.3** Exemplo de uma hierarquia “é um” compreendendo prédios arquitetônicos.

Por exemplo, a estrutura de definição de uma parede é a mesma de casa para casa, sendo normalmente definida em termos de barrotes de duas por quatro polegadas, espaçados por uma distância específica, etc. O arquiteto organizado pode, assim, reutilizar suas definições de parede de uma casa para outra. Ao reutilizar tais definições, algumas partes podem exigir adaptações, por exemplo, uma parede em um edifício comercial pode ser similar à de uma casa, mas o sistema elétrico pode ser diferente.

Uma forma natural de organizar vários componentes estruturais de um pacote de software é de uma forma **hierárquica**, que agrupa definições abstratas similares juntas nível a nível, partindo do mais específico para o mais genérico, à medida que se percorre a hierarquia. Um uso normal de tais hierarquias ocorre em um gráfico organizacional, no qual cada arco que sobe pode ser lido como “é um”, como em “um rancho é uma casa é um prédio”. Esse tipo de hierarquia também é útil no projeto de software quando agrupa funcionalidades comuns no nível mais geral e vê comportamentos especializados como uma extensão do comportamento geral.

### 2.1.3 Padrões de projeto

Uma das maiores vantagens do paradigma de projeto orientado a objetos é que ele facilita o desenvolvimento de software reusável, robusto e adaptável. Projetar código orientado a objetos de qualidade exige mais do que simplesmente entender as metodologias de projeto orientado a objetos. Requer o uso efetivo das técnicas de projeto orientado a objetos.

Cientistas da computação e profissionais da área desenvolveram uma variedade de conceitos organizacionais e metodologias para projetar softwares orientados a objetos de qualidade que sejam concisos, corretos e reutilizáveis. Um conceito especialmente relevante no contexto deste livro é o conceito de **padrão de projeto**, que descreve uma solução para um determinado problema de projeto de software “típico”. Um padrão provê um esquema genérico de uma solução que pode ser aplicada em muitas situações diferentes. Descreve os elementos principais da solução de uma forma abstrata que pode ser especializada para o problema específico que se apresenta. Consiste em um nome que identifica o padrão, um contexto que descreve os cenários para os quais se aplica, um esquema que descreve como é aplicado e um resultado que descreve e analisa o que o padrão produz.

Diversos padrões de projeto serão mostrados neste livro, e será indicado como podem ser aplicados com consistência no projeto de implementações de qualidade de estruturas de dados e algoritmos. Esses padrões se organizam naturalmente em dois grupos: padrões para resolver problemas de projeto de algoritmos e padrões para resolver problemas de engenharia de software. Alguns dos padrões para projeto de algoritmos que serão apresentados incluem:

- recursão (Seção 3.5);
- amortização (Seção 6.1.4);
- divisão e conquista (Seção 11.1.1);
- poda e busca também conhecido como diminuição e conquista (Seção 11.7.1);
- força bruta (Seção 12.2.1);
- o método guloso (Seção 12.4.2);
- programação dinâmica (Seção 12.5.2).

Da mesma forma, alguns dos padrões para engenharia de software apresentados são:

- posicionamento (Seção 6.2.2);
- adaptador (Seção 6.1.2)
- iteradores (Seção 6.3);
- método do esquema (Seção 7.3.7, 11.6 e 13.3.2);
- composição (Seção 8.1.2);
- comparador (Seção 8.1.2);
- decorador (Seção 13.3.1).

Em vez de explicar, inicialmente, cada um desses conceitos, os mesmos serão introduzidos ao longo do texto, como se verá na seqüência. Para cada padrão, seja para engenharia de algoritmo, seja para engenharia de software, será explicado seu uso genérico e ilustrado pelo menos um exemplo concreto.

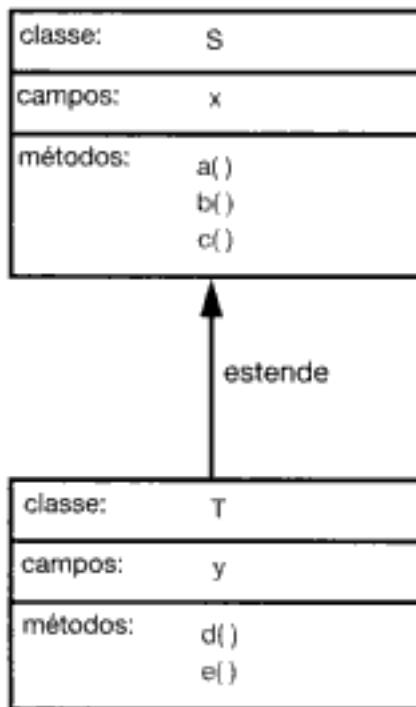
## 2.2 Herança e polimorfismo

Para tirar proveito de relacionamentos hierárquicos comuns em projetos de software, a abordagem de projeto orientado a objetos oferece maneiras de reutilizar código.

### 2.2.1 Herança

O paradigma de orientação a objetos oferece uma estrutura hierárquica e modular para reutilização de código através de uma técnica conhecida como **herança**. Essa técnica permite projetar classes genéricas que podem ser especializadas em classes mais particulares, em que as classes especializadas reutilizam o código das mais genéricas. A classe genérica, também conhecida por **classe base** ou **superclasse**, define variáveis de instância “genéricas” e métodos que se aplicam em uma variada gama de situações. A classe que **especializa**, **estende** ou **herda de** uma superclasse não necessita fornecer uma nova implementação para os métodos genéricos, uma vez que os herda. Deve apenas definir aqueles métodos que são especializados para esta **subclasse** em particular.

**Exemplo 2.1** Considere a classe S que define objetos com o campo x, e três métodos a(), b() e c(). Supõe-se que a classe T que estende S seja definida incluindo um campo adicional, y, e dois métodos, d() e e(). A classe T herdará a variável de instância e os métodos a(), b() e c() de S. Os relacionamentos entre as classes S e T são apresentados no **diagrama de classes com herança** da Figura 2.4. Cada caixa nesse diagrama indica uma classe, com seu nome, campos (ou variáveis de instância) e métodos incluídos como retângulos aninhados.



**Figura 2.4** Um diagrama de classe com herança. Cada caixa indica uma classe, com seu nome, campos e métodos, e uma seta entre as caixas denota um relacionamento de herança.

## Criação de objetos e referências

Quando um objeto *o* é criado, aloca-se memória para seus campos, e esses mesmos campos são inicializados para valores iniciais específicos. Normalmente, associa-se o novo objeto *o* com uma variável que serve de “ligação” com o objeto *o*, e diz-se que é a mesma **referência** *o*. Quando se deseja acessar o objeto *o* (para acessar seus campos ou ativar seus métodos), pode-se tanto solicitar a execução de um dos métodos de *o* (definidos na classe a qual *o* pertence) ou procurar um dos campos de *o*. Na verdade, a principal maneira pela qual um objeto *p* interage com outro objeto *o* é o envio de uma mensagem de *p* para *o* que invoque um dos métodos de *o*, por exemplo, para fazer *o* imprimir uma descrição de si mesmo, para que *o* converta a si mesmo em uma string ou para que retorne o valor de um de seus campos de dados. Uma forma secundária pela qual *p* pode interagir com *o* é através do acesso direto de *p* a um dos campos de *o*, mas isso só é possível se *o* tiver dado permissão para objetos do tipo de *p* fazê-lo. Por exemplo, uma instância da classe `Integer` de Java armazena um inteiro em uma variável de instância e fornece várias operações para acessar esse dado, incluindo métodos para convertê-lo em outros tipos numéricos ou em uma string de dígitos e também para converter uma string de dígitos em um número. Ela não permite, entretanto, o acesso direto à variável de instância, uma vez que tais detalhes estão escondidos.

## Ativação dinâmica

Quando um programa deseja ativar um método `a()` de algum objeto *o*, ele envia uma mensagem para *o*, o que normalmente é feito usando-se a sintaxe do operador ponto (Seção 1.3.2) da seguinte forma: “*o.a()*”. Na versão compilada desse programa, o código correspondente a essa ativação ordena ao ambiente de execução que examine a classe *T* de *o* para verificar se a classe *T* suporta o método `a()`, e, em caso positivo, executa o mesmo. Mais especificamente, o ambiente de execução examina a classe *T* para verificar se ela define o método `a()`. Se assim ocorre, então esse método é executado. Se *T* não define o método `a()`, então o ambiente de execução examina *S*, a superclasse de *T*. Se *S* define `a()`, então esse método é executado. Por outro lado, se *S* não define `a()`, então o ambiente de execução repete a busca na superclasse de *S*. Essa busca continua subindo a hierarquia de classes até que se encontre o método `a()`, que é então executado, ou que se encontre a classe de nível mais alto (por exemplo, a classe `Object` em Java) sem o método `a()`, o que gera um erro de execução. O algoritmo que processa a mensagem *o.a()* para encontrar o método a ser disparado é chamado de algoritmo de **ativação dinâmica** (ou de **ligação dinâmica**), o qual oferece um mecanismo efetivo para localizar software reutilizado. Ele também permite o uso de outra técnica poderosa de programação orientada a objetos: o **polimorfismo**.

---

### 2.2.2 Polimorfismo

Literalmente, “polimorfismo” significa “muitas formas”. No contexto de projeto orientado a objetos, entretanto, refere-se à habilidade de uma variável de objeto de assumir formas diferentes. Linguagens orientadas a objetos, tais como Java, referenciam objetos usando variáveis referência. Uma variável referência *o* deve especificar que tipo de objeto é capaz de referenciar em termos de uma classe *S*. Isso implica, entretanto, que *o* também pode se referir a qualquer objeto pertencente à classe *T* derivada de *S*. Agora, será analisado o que acontece se *S* define um método `a()` e *T* também define um método `a()`. O algoritmo de ativação dinâmica de métodos sempre inicia sua busca pela classe mais restritiva à qual se aplica. Quando *o* se refere a um objeto da classe *T* e *o.a()* é invocado, então é ativada a versão de *T* do método `a()`, em lugar da versão de *S*. Neste caso, diz-se que *T* **sobrecreve** o método `a()` de *S*. Por outro lado, se *o* se refere a um objeto da classe *S* (que, ao contrário, não é um objeto da classe *T*), quando *o.a()* for ativado, será executada a versão de *S* de `a()`. Um polimorfismo como esse é útil porque aquele que chama *o.a()* não precisa saber quando

*o* se refere a uma instância de T ou S para poder executar a versão correta de *a()*. Dessa forma, a variável de objeto *o* pode ser **polimórfica**, ou assumir muitas formas, dependendo da classe específica dos objetos aos quais está se referindo. Esse tipo de funcionalidade permite a uma classe especializada T estender uma classe S, herdar os métodos padrão de S e redefinir outros métodos de S, de maneira que sejam incluídos como propriedades específicas dos objetos T.

Algumas linguagens orientadas a objetos, como Java, também oferecem uma técnica relacionada a polimorfismo que é chamada de **sobrecarga** de métodos. A sobrecarga ocorre quando uma única classe T tem vários métodos com o mesmo nome, desde que cada um tenha uma **assinatura** diferente. A assinatura de um método é uma combinação entre seu nome, o tipo e a quantidade de argumentos que são passados para o mesmo. Dessa forma, mesmo que vários métodos de uma classe tenham o mesmo nome, eles são distinguíveis pelo compilador pelo fato de terem diferentes assinaturas, ou seja, na verdade são desiguais. Em linguagens que possibilitam a sobrecarga de métodos, o ambiente de execução determina qual método ativar para uma determinada chamada de método que percorre a hierarquia de classes em busca do primeiro método cuja assinatura combine com a do método que está sendo invocado. Por exemplo, imagine uma classe T que define o método *a()*, derivada da classe U que define o método *a(x,y)*. Se um objeto *o* da classe T recebe a mensagem “*o.a(x,y)*”, então a versão de U do método *a* é ativada (com os dois parâmetros, *x* e *y*). Assim, o verdadeiro polimorfismo aplica-se apenas a métodos que têm a mesma assinatura mas estão definidos em classes diferentes.

A herança, o polimorfismo e a sobrecarga de métodos suportam o desenvolvimento de software reutilizável. Podem-se estabelecer classes que herdam as variáveis e os métodos de instância genéricos e que, a seguir, definem novas variáveis e métodos de instância mais específicos que lidam com os aspectos particulares dos objetos da nova classe.

### 2.2.3 Usando herança em Java

Existem duas formas básicas de se usar herança de classes em Java: **extensão** e **especialização**.

#### Especialização

Quando se usa a especialização, estamos refinando uma classe genérica em subclasses específicas. Tais subclasses normalmente possuem uma relação do tipo “é um” com sua superclasse. Estas subclasses herdam todos os métodos da superclasse. Para cada método herdado, caso funcione corretamente independente do fato de estar sendo usado pela especialização, nenhum esforço adicional é necessário. Se, por outro lado, um método genérico da superclasse não funcionar corretamente na subclasse, então é necessário sobrepor o método para obter a funcionalidade correta da subclasse. Por exemplo, pode-se ter uma classe genérica Dog (“cachorro”) que possui o método *drink* (“beber”) e o método *sniff* (“farejar”). Ao especializar a classe Bloodhound (“cão de caça”), provavelmente não será necessário especializar o método *drink*, na medida em que todos os cachorros bebem da mesma forma. Contudo, provavelmente será necessário sobrepor o método *sniff*, na medida em que um cão de caça tem um faro muito mais sensível que um cão “genérico”. Dessa forma, a classe Bloodhound especializa os métodos da superclasse Dog.

#### Extensão

Ao usar a extensão, por outro lado, utiliza-se herança para reutilizar o código escrito para os métodos da superclasse, mas será necessário adicionar novos métodos que não estão presentes na superclasse, de maneira a estender sua funcionalidade. Por exemplo, reconsiderando a classe

Dog, pode-se querer criar a subclasse BorderCollie ("cão pastor") que herda todos os métodos genéricos da classe Dog mas adiciona um método novo, herd ("arrebanhar"), uma vez que cães pastores têm um instinto para o pastoreio que não está presente nos cães genéricos. Adicionando o novo método, estende-se estendendo a funcionalidade do cão genérico.

Em Java, cada classe pode estender apenas uma única classe. Mesmo que uma classe não faça uso explícito da cláusula **extends**, ainda assim é derivada de exatamente uma classe, neste caso a classe `java.lang.Object`. Em virtude desta propriedade, diz-se que Java possibilita apenas *herança simples* entre classes.

### Tipos de sobrecarga de método

Dentro da declaração de uma classe nova, Java usa dois tipos de sobrecarga de método, o *refinamento* e a *substituição*. Na sobrecarga por substituição, o novo método substitui completamente o método da superclasse que está sendo sobreescrito (como no caso do método `sniff` da classe `Bloodhound` mencionada anteriormente). Em Java, todos os métodos normais de uma classe utilizam este tipo de comportamento na sobrecarga.

Na sobrecarga por refinamento, entretanto, um método não substitui o método de sua superclasse, mas ao contrário, adiciona código ao de sua superclasse. Em Java, todos os construtores utilizam sobrecarga por refinamento, um esquema chamado de *encadeamento de construtores*. Isto é, um construtor inicia sua execução chamando o construtor de sua superclasse. Essa chamada pode ser feita de forma explícita ou implícita. Para chamar o construtor de uma superclasse, explicitamente, usa-se a palavra reservada **super**, para fazer referência à superclasse. (Por exemplo, `super()` chama o construtor da superclasse que não tem parâmetros.) Se nenhuma chamada explícita é feita no corpo do construtor, entretanto, o compilador insere automaticamente, na primeira linha do construtor, uma chamada para o método `super()`. (Existe uma exceção a esta regra geral que será discutida na próxima seção.) Em resumo, em Java os construtores usam sobrecarga por refinamento, enquanto que os métodos normais usam a substituição.

### A palavra reservada **this**

Algumas vezes, em uma classe Java, é conveniente referenciar a instância corrente da classe. Java oferece uma palavra reservada, chamada **this**, para tal referência. A referência **this** é útil, por exemplo, quando se deseja passar o objeto corrente como parâmetro de algum método. Outra aplicação é referenciar um campo do objeto corrente cujo nome está em conflito com o nome de uma variável definida no bloco corrente, como no programa apresentado no Trecho de código 2.1

```
public class ThisTester {
    public int dog = 2; // variável de instância
    public void clobber() {
        int dog = 5; // um cachorro diferente!
        System.out.println("The dog local variable = " + dog);
        System.out.println("The dog field = " + this.dog);
    }
    public static void main(String args[ ]) {
        ThisTester t = new ThisTester();
        t.clobber();
    }
}
```

**Trecho de código 2.1** Programa exemplo que ilustra o uso da referência **this** para resolver a ambigüidade entre um campo do objeto corrente e a variável local com mesmo nome.

Quando este programa é executado, imprime o seguinte:

```
The dog local variable = 5.0
The dog field = 2
```

### Um exemplo de herança em Java

Para tornar as noções de herança e polimorfismo mais concretas, serão analisados alguns exemplos simples em Java.

Particularmente, serão utilizados alguns exemplos de classes que percorrem e imprimem progressões numéricas. Uma progressão numérica é uma seqüência de números em que o valor de cada um depende de um ou mais valores anteriores. Por exemplo, uma **progressão aritmética** determina o próximo número por meio de adições, e uma **progressão geométrica**, por multiplicações. Em qualquer caso, uma progressão exige uma forma determinativa do valor inicial, bem como uma maneira de identificar o valor corrente.

Inicia-se definindo a classe, Progression, apresentada no Trecho de código 2.2, que estabelece os campos e métodos “genéricos” de uma progressão numérica. Em particular, dois campos inteiros longos são definidos:

- first: o primeiro valor da progressão;
- cur: o valor atual da progressão;

bem como os três métodos a seguir:

`firstValue()`: Retorna a progressão ao primeiro valor e retorna esse valor.

`nextValue()`: Avança a progressão para o próximo valor e retorna esse valor.

`printProgression(n)`: Retorna a progressão para o início e imprime os primeiros *n* valores da progressão.

Diz-se que o método `printProgression` não tem saída no sentido de que não retorna nenhum valor, enquanto que os métodos `firstValue` e `nextValue` retornam ambos inteiros longos. Ou seja, `firstValue` e `nextValue` são funções, enquanto `printProgression` é um procedimento.

A classe `Progression` também inclui o método `Progression()`, que é um método **construtor**. Lembre-se que os métodos construtores inicializam as variáveis de instância no momento da criação do objeto. A classe `Progression` visa a ser uma superclasse genérica, a partir da qual classes especializadas são derivadas, de maneira que o código do construtor será incluído nos construtores de cada uma que estende a classe `Progression`.

```
/**
 * Uma classe de progressão numérica.
 */
public class Progression {
    /** Primeiro valor da progressão. */
    protected long first;
    /** Valor atual da progressão. */
    protected long cur;
    /** Construtor default. */
    Progression() {
        cur = first = 0;
    }
    /** Reinicializa a progressão com o valor inicial.
     *
     * @return valor inicial
    }
```

```

/*
protected long firstValue() {
    cur = first;
    return cur;
}

/** Avança a progressão para o próximo valor.
 *
 * @return próximo valor da progressão
 */
protected long nextValue() {
    return ++cur; // próximo valor default
}

/** Imprime os primeiros valores n da progressão
 *
 * @param número n de valores a serem impressos
 */
public void printProgression(int n) {
    System.out.print(firstValue());
    for (int i = 2; i <= n; i++)
        System.out.print(" " + nextValue());
    System.out.println(); // termina a linha
}
}

```

**Trecho de código 2.2** Classe genérica de progressão numérica.

### Uma classe de progressão aritmética

Será analisada, na seqüência, a classe ArithProgression, apresentada no Trecho de código 2.3. Essa classe define uma progressão em que cada valor é determinado pela adição de um incremento fixo, inc, ao valor anterior. Ou seja, ArithProgression define uma progressão aritmética. ArithProgression herda os campos first e cur, bem como os métodos firstValue() e printProgression() da classe Progression. Adiciona um novo campo, inc, para armazenar o incremento, e dois construtores para inicializá-lo. Finalmente, sobrescreve o método nextValue() para adequá-lo à forma pela qual será obtido o próximo termo de uma progressão aritmética.

O polimorfismo está presente neste caso. Quando uma referência para Progression aponta para um objeto da classe ArithProgression, então os métodos firstValue() e nextValue() de ArithProgression serão usados. Este polimorfismo também é real na versão herdada de printProgression(*n*), pois as chamadas dos métodos firstValue() e nextValue() são implícitas para o objeto corrente (chamado de *this* em Java), que neste caso corresponderá à classe ArithProgression.

### Exemplo de construtores e da palavra reservada **this**

Na definição da classe ArithProgression, foram adicionados dois métodos construtores: um método default, sem parâmetros, e uma versão parametrizada que recebe um inteiro para ser usado como incremento da progressão. O construtor default, na verdade, chama o método parametrizado usando a palavra reservada **this** e passando 1 como valor do parâmetro a ser usado como incremento. Esses dois construtores ilustram a sobrecarga de métodos na qual um nome de método pode ter várias versões dentro de uma mesma classe, uma vez que, na verdade, um método é especificado pelo seu nome, pela classe do objeto que o ativa e pelos tipos dos parâmetros que são passados para o mesmo – sua assinatura. Neste caso, a sobrecarga é dos métodos construtores (um construtor default e um construtor parametrizado).

A chamada `this(1)` ao construtor parametrizado, como primeiro comando do construtor default, configura uma exceção à regra geral de ativação em cascata de construtores, discutida na Seção 2.2.3. Na verdade, quando o primeiro comando de um construtor  $C'$  chama outro construtor  $C''$  da mesma classe usando a referência `this`, o construtor da superclasse não é automaticamente acionado por  $C'$ . Observa-se que o construtor da superclasse pode ser, eventualmente, acionado ao longo da cadeia, tanto de forma implícita como explícita. No caso particular da classe `ArithProgression`, o construtor default da superclasse (`Progression`) é ativado implicitamente como primeiro comando do construtor paramétrico de `ArithProgression`.

Construtores serão discutidos em mais detalhes na Seção 1.2.

```
/*
 * Progressão aritmética.
 */
class ArithProgression extends Progression
{
    /** Incremento. */
    protected long inc;

    // Herda as variáveis first e cur.

    /** Construtor default inicializa com incremento de 1. */
    ArithProgression() {
        this(1);
    }

    /** Construtor paramétrico fornece o incremento. */
    ArithProgression(long increment) {
        inc = increment;
    }

    /** Avança a progressão acrescentando o incremento ao valor atual.
     *
     * @return próximo valor da progressão
     */
    protected long nextValue() {
        cur += inc;
        return cur;
    }

    // Herda os métodos firstValue() e printProgression(int).
}
```

**Trecho de código 2.3** Classe de progressão aritmética que herda da progressão genérica apresentada no Trecho de código 2.2.

### Uma classe de progressão geométrica

Será definida em seguida a classe `GeomProgression`, apresentada no Trecho de código 2.4, que permite tanto navegar através de uma progressão geométrica como imprimi-la, sendo esta determinada pela multiplicação do valor prévio por uma base  $b$ . Uma progressão geométrica é como uma progressão genérica, exceto pela forma como se determina o próximo valor. Desta forma, `GeomProgression` é declarada como subclasse da classe `Progression`. Da mesma forma que `ArithProgression`, a classe `GeomProgression` herda os campos `first` e `cur`, bem como os métodos `firstValue` e `printProgression` da classe `Progression`.

```


/*
 * Progressão geométrica
 */
class GeomProgression extends Progression {

    /** Base */
    protected long base;

    // Herda as variáveis first e cur.

    /** Construtor default inicializa o valor base com 2. */
    GeomProgression() {
        this(2);
    }

    /** Construtor paramétrico fornece o valor base.
     *
     * @param base é o valor base da progressão.
     */
    GeomProgression(long b) {
        base = b;
        first = 1;
        cur = first;
    }

    /** Avança a progressão multiplicando a base pelo valor corrente.
     *
     * @return próximo valor da progressão
     */
    protected long nextValue() {
        cur *= base;
        return cur;
    }

    // Herda os métodos firstValue() e printProgression(int).
}


```

**Trecho de código 2.4** Classe de progressão geométrica.

### Uma classe de progressão Fibonacci

Como último exemplo, será definida a classe `FibonacciProgression`, que representa um outro tipo de progressão, a **progressão Fibonacci**, na qual o próximo valor é definido pela soma dos valores atuais com os anteriores. A classe `FibonacciProgression` será apresentada no Trecho de código 2.5. Observe-se o uso do construtor parametrizado na classe `FibonacciProgression`, oferecendo uma forma diferente de iniciar a progressão.

```


/*
 * Progressão Fibonacci
 */
class FibonacciProgression extends Progression {

    /** Valor anterior. */
    long prev;

    // Herda as variáveis first e cur.
}


```

```

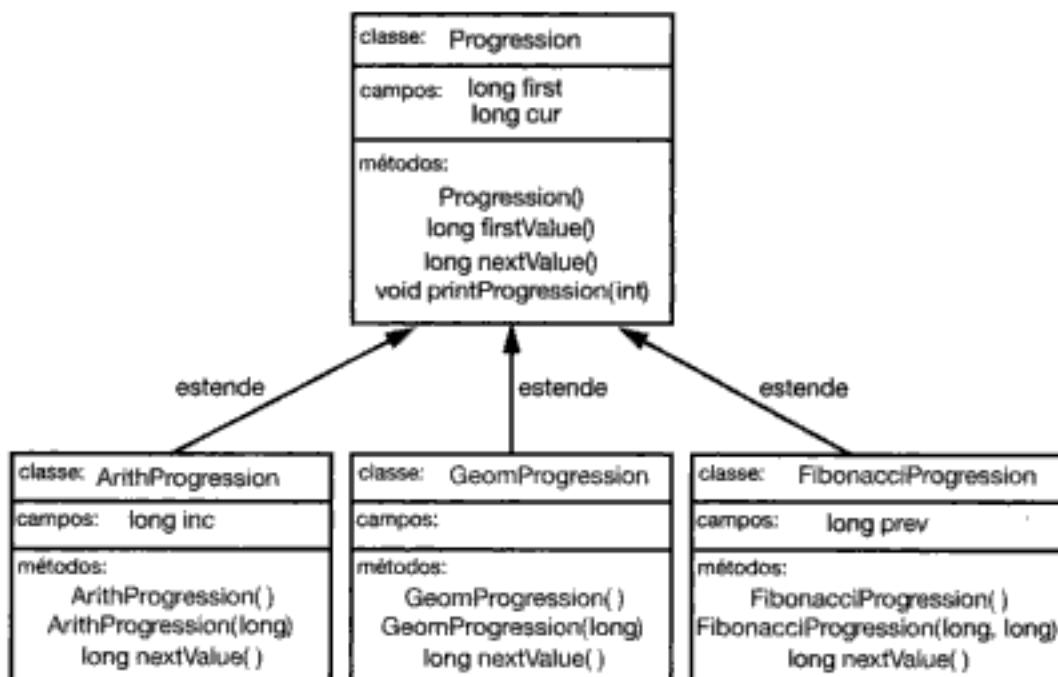
/** Construtor default inicializa os dois primeiros valores como sendo 0 e 1. */
FibonacciProgression( ) {
    this(0, 1);
}
/** Construtor paramétrico fornece o primeiro e o segundo valores.
 *
 * @param value1 é o primeiro valor.
 * @param value2 é o segundo valor.
 */
FibonacciProgression(long value1, long value2) {
    first = value1;
    prev = value2 - value1; // valor fictício que antecede o primeiro
}
/** Avança a progressão somando o valor anterior no valor atual.
 *
 * @return próximo valor da progressão
 */
protected long nextValue( ) {
    long temp = prev;
    prev = cur;
    cur += temp;
    return cur;
}
// Herda os métodos firstValue( ) e printProgression(int).
}

```

**Trecho de código 2.5** Classe para a progressão Fibonacci.

Para visualizar a maneira pela qual as três diferentes progressões são derivadas da classe genérica Progression, apresenta-se o diagrama de herança correspondente na Figura 2.5.

Para completar o exemplo, define-se a classe Tester, indicada no Trecho de código 2.6, que executa um teste simples com cada uma das três classes. Nesta classe, a variável prog é polimórf-



**Figura 2.5** Diagrama de herança da classe Progression e suas subclasses.

fica durante a execução do método main, uma vez que referencia, alternadamente, objetos das classes ArithProgression, GeomProgression e FibonacciProgression. Quando o método main da classe Tester é ativado pelo sistema de execução de Java, a saída resultante é a apresentada no Trecho de código 2.7.

O exemplo desta seção é propositadamente pequeno, mas faz uma demonstração simples do uso de herança em Java. A classe Progression, suas subclasses e o programa de teste, entretanto, têm uma série de defeitos que não são percebidos à primeira vista. Um dos problemas é que as progressões geométrica e Fibonacci crescem rapidamente, e não existe previsão de tratamento para o estouro inevitável dos inteiros longos envolvidos. Por exemplo, uma vez que  $3^{40} > 2^{63}$ , uma progressão geométrica de base  $b = 3$  irá estourar a capacidade de um inteiro longo após 40 iterações. Da mesma forma, o 94º número da série de Fibonacci é maior que  $2^{63}$ , logo, a progressão Fibonacci irá explodir a capacidade de um inteiro longo após 94 iterações. Outro problema é que não se admitem valores iniciais arbitrários para uma progressão Fibonacci. Por exemplo, pode-se considerar uma progressão de Fibonacci iniciada em 0 e -1? Lidar com os erros de entrada ou com as condições de erro que ocorrem durante a execução de um programa Java requer algum mecanismo para tratá-los. Este tópico será discutido a seguir.

```
/** Programa teste para as classes de progressão */
class TestProgression {
    public static void main(String[] args) {
        Progression prog;
        // testa ArithProgression
        System.out.println("Arithmetic progression with default increment:");
        prog = new ArithProgression();
        prog.printProgression(10);
        System.out.println("Arithmetic progression with increment 5:");
        prog = new ArithProgression(5);
        prog.printProgression(10);
        // testa GeomProgression
        System.out.println("Geometric progression with default base:");
        prog = new GeomProgression();
        prog.printProgression(10);
        System.out.println("Geometric progression with base 3:");
        prog = new GeomProgression(3);
        prog.printProgression(10);
        // testa FibonacciProgression
        System.out.println("Fibonacci progression with default start values:");
        prog = new FibonacciProgression();
        prog.printProgression(10);
        System.out.println("Fibonacci progression with start values 4 and 6:");
        prog = new FibonacciProgression(4,6);
        prog.printProgression(10);
    }
}
```

**Trecho de código 2.6** Programa para testar as classes de progressão.

Arithmetic progression with default increment:

0 1 2 3 4 5 6 7 8 9

Arithmetic progression with increment 5:

0 5 10 15 20 25 30 35 40 45

Geometric progression with default base:

1 2 4 8 16 32 64 128 256 512

Geometric progression with base 3:

1 3 9 27 81 243 729 2187 6561 19683

Fibonacci progression with default start values:

0 1 1 2 3 5 8 13 21 34

Fibonacci progression with start values 4 and 6:

4 6 10 16 26 42 68 110 178 288

**Trecho de código 2.7** Saída do programa TestProgression apresentado no Trecho de código 2.6.

## 2.3 Exceções

As exceções são eventos inesperados que ocorrem durante a execução de um programa. Uma exceção pode ser o resultado de uma condição de erro ou de uma simples entrada inesperada. De qualquer forma, em linguagens orientadas a objetos como Java, as exceções são vistas como objetos.

### 2.3.1 Lançando exceções

Em Java, exceções são *lançadas* por trechos de código que detectam algum tipo de condição inesperada. Podem também ser lançadas pelo ambiente de execução de Java se este encontra uma situação imprevista como, por exemplo, execução além da memória de um objeto. Uma exceção lançada é *capturada* por trechos de código capazes de tratar a exceção de alguma forma, ou então o programa é encerrado de maneira inesperada. (Trata-se, mais adiante sobre captura de exceções.)

As exceções se originam quando um pedaço de código Java encontra algum tipo de problema durante a execução, e *lança* um objeto de exceção identificado com um nome descritivo. Por exemplo, eliminando o décimo elemento de uma seqüência de apenas cinco elementos, o código irá lançar uma *BoundaryViolationException*. Esta ação poderia ser executada, por exemplo, pelo seguinte trecho de código:

```
if (insertIndex > A.length) {
    throw new
        BoundaryViolationException("No element at index " + insertIndex);
}
```

Em geral, é conveniente instanciar o objeto de exceção no momento em que a exceção está para ser lançada. Desta forma, um comando de *lançamento* típico é escrito assim:

```
throw new exception_type(param0, param1, ...paramn-1)
```

onde *exception\_type* é o tipo da exceção e os “*param*” formam a lista dos parâmetros do construtor desta exceção.

As exceções também são lançadas pelo ambiente de execução de Java. Por exemplo, o equivalente ao caso anterior é *ArrayIndexOutOfBoundsException*. Se existe um vetor de seis elementos e solicita-se o nono elemento, esta exceção será lançada pelo ambiente de execução de Java.

A cláusula *throws*

Quando um método é declarado, é adequado especificar os tipos de exceção que ele pode lançar. Esta convenção tem tanto um propósito funcional como de cortesia, porque permite que o usuário saiba o que esperar. Além disso, possibilita que o compilador Java saiba para quais exceções deve estar preparado. O exemplo seguinte apresenta este tipo de definição de método:

```
public void goShopping( ) throws ShoppingListTooSmallException,
    OutOfMoneyException {
    // corpo do método
}
```

Especificando-se todas as exceções que podem ser disparadas por um método, preparam-se os outros para lidar com todos os casos excepcionais que podem resultar do seu uso. Outro benefício da declaração de exceções é que não é necessário capturar essas exceções neste método. Algumas vezes isso é apropriado, principalmente quando outro código é o responsável pelas circunstâncias que podem levar à exceção.

O exemplo seguinte ilustra uma exceção que é “passada adiante”:

```
public void getReadyForClass( ) throws ShoppingListTooSmallException,
    OutOfMoneyException {
    goShopping( ); // Eu não tenho que testar ou capturar exceções
                    // que a função goShopping( ) pode lançar porque
                    // getReadyForClass( ) passa as mesmas adiante.
    makeCookiesForTA( );
}
```

Uma função pode declarar que é capaz de lançar quantas exceções quiser. Tal lista pode ser simplificada, entretanto, se todas as exceções que podem ser lançadas forem subclasses da mesma. Neste caso, deve-se declarar apenas que o método lança a superclasse adequada.

### Tipos de lançamentos

Java define as classes `Exception` e `Error` como subclasses de `Throwable`, o que implica que todos esses objetos podem ser lançados e capturados. Além disso, define a classe `RunTimeException` como subclasse de `Exception`. A classe `Error` é usada para condições anormais que ocorram no ambiente de execução, tais como executar sem memória suficiente. Os erros podem ser capturados mas, provavelmente, não o serão, porque, em geral, sinalizam problemas que não podem ser tratados de forma refinada. Uma mensagem de erro ou a interrupção súbita da execução pode ser a melhor saída que se pode esperar nesses casos. A classe `Exception` é a raiz da hierarquia de exceções. As exceções especializadas (por exemplo, `BoundaryViolationException`) devem ser definidas por herança de `Exception` ou de `RunTimeException`. Observa-se que as exceções que não forem subclasses de `RunTimeException` devem ser declaradas na cláusula `throws` de qualquer método que possa lançá-las.

#### 2.3.2 Capturando exceções

Quando uma exceção é lançada, deve ser *capturada*, ou o programa se encerrará. Uma exceção que ocorra em qualquer método pode ser passada pelo método que o ativou ou pode ser por ele capturada. Quando uma exceção é capturada, ela pode ser analisada e tratada. A forma geral de se lidar com exceções é “**tentar**”\* executar um trecho de código que tenha a possibilidade de lançar uma exceção. Se a exceção for lançada, então a mesma será *capturada*\*\*, fazendo com que o fluxo de controle desvie para um bloco `catch` predefinido. Usando o bloco de *captura*\*\*\*, pode-se então lidar com a circunstância excepcional.

\* N. de T. Analogia com o comando `try` de Java.

\*\* N. de T. Analogia com o comando `catch` de Java.

\*\*\* N. de T. Bloco `catch`.

A sintaxe genérica para um **bloco try-catch** em Java é a seguinte:

```
try
    bloco_principal_de_comandos
catch(tipo_da_exceção, variável1)
    bloco_de_comandos1
catch(tipo_da_exceção2, variável2)
    bloco_de_comandos2

...
finally
    bloco_de_comandosn
```

onde deve existir pelo menos uma cláusula **catch**, mas a cláusula **finally** é opcional. Cada *tipo\_da\_exceção* é do tipo de alguma exceção, e cada *variável* é um nome de variável válido em Java.

O ambiente Java inicia executando um bloco **try-catch** como este, pela execução do conjunto de instruções *bloco\_principal\_de\_comandos*. Se essa execução não gerar exceções, então o fluxo de controle continua no primeiro comando após a última linha do bloco **try-catch**, a menos que este inclua o bloco opcional **finally**. O bloco **finally**, quando existe, é executado indiferentemente se as exceções forem lançadas ou capturadas. Então, neste caso, se nenhuma exceção é lançada, a execução segue pelo bloco **try-catch**, pula o bloco **finally** e continua com o primeiro comando depois da última linha do bloco **try-catch**.

Se, por outro lado, o bloco *principal\_de\_comandos* gera uma exceção, então a execução do bloco **try-catch** termina neste ponto e desvia para o bloco **catch** mais próximo cujo *tipo\_da\_exceção* combinar com a exceção lançada. A *variável* deste comando **catch** referencia o objeto da exceção propriamente dito, que pode ser usado no bloco do comando **catch** acionado. Uma vez encerrada a execução do bloco **catch**, o controle é passado para o bloco opcional **finally**, se ele existir, ou imediatamente para o primeiro comando após a última linha do bloco **try-catch**, se não existir bloco **finally**. Caso contrário, se não houver nenhum bloco **catch** capaz de tratar a exceção, então o controle será passado para o bloco opcional **finally**, se existir, e então a exceção é repassada para o método chamador.

A seguir, será examinado o exemplo do trecho de código:

```
int index = Integer.MAX_VALUE; // 2.14 Bilhões
try
    // Este código deve ter problemas...
{
    String toBuy = shoppingList[index];
}
catch (ArrayIndexOutOfBoundsException aioobx)
{
    System.out.println("The index "+index+" is outside the array.");
}
```

Se este código não capturar a exceção lançada, o fluxo de controle irá sair imediatamente do método e retornar para o código que o ativou. Neste ponto, o ambiente de execução de Java procurará novamente um **catch**. Se não existir bloco **catch** no código que chama este método, o fluxo de controle irá pular para o código que o ativa e assim por diante. No caso de nenhum bloco de código capturar a exceção, o ambiente de execução de Java (a origem do fluxo de execução do programa) irá capturar a exceção. Nesse momento, uma mensagem de erro e o conteúdo da pilha são impressos na tela e o programa é encerrado.

O exemplo que segue é de uma mensagem de erro real:

```
java.lang.NullPointerException: Returned a null locator
at java.awt.Component.handleEvent(Component.java:900)
```

```

at java.awt.Component.postEvent(Component.java:838)
at java.awt.Component.postEvent(Component.java:845)
at sun.awt.motif.MButtonPeer.action(MButtonPeer.java:39)
at java.lang.Thread.run(Thread.java)

```

Uma vez capturada a exceção, existem várias possibilidades de escolha para o programador. Uma delas é imprimir uma mensagem de erro e terminar o programa. Existem casos interessantes em que a melhor maneira de tratar uma exceção é ignorá-la (isto pode ser feito com um bloco **catch** vazio).

Ignorar uma exceção é normal, por exemplo, quando o programador não está preocupado com o fato de ocorrerem exceções ou não. Outra forma legítima de tratar uma exceção é criar e lançar uma outra, possivelmente alguma que expresse a situação excepcional com maior precisão. O exemplo a seguir apresenta essa situação:

```

catch (ArrayIndexOutOfBoundsException aioobx) {
    throw new ShoppingListTooSmallException(
        "Product index is not in the shopping list");
}

```

A melhor maneira de tratar uma exceção (embora nem sempre seja possível), porém, é encontrar o problema, consertá-lo e continuar a execução.

## 2.4 Interfaces e classes abstratas

Para que dois objetos possam interagir, eles precisam “conhecer” as várias mensagens que cada um pode aceitar, ou seja, os métodos que cada objeto suporta. Para garantir esse “conhecimento”, o paradigma de projeto orientado a objetos solicita que as classes especifiquem a *interface de programação da aplicação* (API\*), ou simplesmente *interface*, que seus objetos apresentam para os outros objetos. Na abordagem de TADs (ver a Seção 2.1.2) usada para estruturas de dados neste livro, uma interface que define um TAD é especificada como uma definição de tipo e uma coleção de métodos para esse tipo, com os parâmetros de cada método sendo dos tipos determinados. Esta especificação, por sua vez, é garantida pelo compilador ou pelo ambiente de execução que requer que os tipos dos parâmetros realmente passados para os métodos confirmem exatamente com o tipo indicado na interface. Este requisito é conhecido como *tipagem forte*. Ter de definir interfaces e ter que lidar com tipagem forte é uma sobrecarga para o programador, mas a tarefa é recompensada porque certifica o princípio do encapsulamento e, normalmente, detecta erros de programação que de outra forma passariam desapercebidos.

### 2.4.1 Implementando interfaces

O principal elemento estrutural de Java que garante uma API é a *interface*. Uma interface é uma coleção de declarações de métodos sem dados e sem corpo. Ou seja, os métodos de uma interface são sempre vazios (ou seja, são simples assinaturas de métodos). Quando uma classe implementa uma interface, ela deve programar todos os métodos declarados na mesma. Dessa forma, a interface impõe que a classe que a implementa tenha métodos com assinaturas específicas.

Suponha, por exemplo, que se deseja criar um inventário de antiguidades, classificando-as como objetos de vários tipos e de várias características. Pode-se, por exemplo, identificar alguns dos objetos como vendíveis e, neste caso, podemos implementar a interface *Sellable* (“vendível”) apresentada no Trecho de código 2.8.

\* N. de T. Do inglês *application programming interface*.

Pode-se então definir uma classe concreta, *Photograph* (fotografia), apresentada no Trecho de código 2.9, que implementa a interface *Sellable*, indicando que se está disposto a vender qualquer um dos objetos *Photograph*: esta classe define um objeto que implementa cada um dos métodos da interface *Sellable*, como requerido. Além disso, adiciona um método, *isColor* (é colorida), específico de objetos *Photograph*.

Outro tipo de objeto da coleção pode ser algo que se possa transportar. Para tais objetos, se define a interface apresentada no Trecho de código 2.10.

```
/** Interfaces para objetos que podem ser vendidos. */
public interface Sellable {
    /** descrição do objeto */
    public String description();
    /** lista de preços em centavos */
    public int listPrice();
    /** preço mais baixo em centavos que se pode aceitar */
    public int lowestPrice();
}
```

#### Trecho de código 2.8 Interface *Sellable*.

```
/** Classe de fotografias que podem ser vendidas. */
public class Photograph implements Sellable {
    private String descript; // descrição desta foto
    private int price; // preço estabelecido
    private boolean color; // true se a foto for a cores

    public Photograph(String desc, int p, boolean c) { // construtor
        descript = desc;
        price = p;
        color = c;
    }

    public String description() { return descript; }
    public int listPrice() { return price; }
    public int lowestPrice() { return price/2; }
    public boolean isColor() { return color; }
}
```

#### Trecho de código 2.9 Classe *Photograph* implementando a interface *Sellable*.

```
/** Interface para objetos que podem ser transportados. */
public interface Transportable {
    /** peso em gramas */
    public int weight();
    /** se o objeto é ou não é perigoso */
    public boolean isHazardous();
}
```

#### Trecho de código 2.10 Interface *Transportable*\*.

Pode-se definir então a classe *BoxedItem*, apresentada no Trecho de código 2.11, para quaisquer antigüidades que se possam vender, empacotar e transportar. Dessa forma, a classe *BoxedI-*

\* N. de T. Transportável.

tem implementa os métodos da interface **Sellable** e da interface **Transportable**, além de acrescentar métodos especializados para determinar o valor do seguro para o transporte de um pacote e para determinar as dimensões do mesmo.

```
/** Classe de objetos que podem ser vendidos, empacotados e despachados. */
public class BoxedItem implements Sellable, Transportable {
    private String descript; // descrição do item
    private int price; // preço de tabela em centavos
    private int weight; // peso em gramas
    private boolean haz; // true se o objeto for perigoso
    private int height=0; // altura da caixa em centímetros
    private int width=0; // largura da caixa em centímetros
    private int depth=0; // profundidade da caixa em centímetros
    /** Construtor */
    public BoxedItem(String desc, int p, int w, boolean h) {
        descript = desc;
        price = p;
        weight = w;
        haz = h;
    }
    public String description() { return descript; }
    public int listPrice() { return price; }
    public int lowestPrice() { return price/2; }
    public int weight() { return weight; }
    public boolean isHazardous() { return haz; }
    public int insuredValue() { return price*2; }
    public void setBox(int h, int w, int d) {
        height = h;
        width = w;
        depth = d;
    }
}
```

Trecho de código 2.11 Classe BoxedItem.

A classe **BoxedItem** também apresenta outro recurso de classes e interfaces em Java – uma classe pode implementar várias interfaces – o que nos permite grande flexibilidade para definir classes que se adaptam a múltiplas APIs. Enquanto uma classe Java só pode estender uma única classe, elas podem implementar muitas interfaces.

## 2.4.2 Herança múltipla e interfaces

A habilidade de estender de mais de uma classe é conhecida como **herança múltipla**. Em Java, a herança múltipla é permitida para interfaces, mas não para classes. A razão desta regra é que métodos de interfaces não têm corpos, enquanto que métodos de classes os possuem. Dessa forma, se Java permitisse a herança múltipla para classes, poderia haver confusão caso uma classe tentasse estender duas classes que contivessem métodos com a mesma assinatura. Essa confusão, entretanto, não existe para interfaces porque os métodos são vazios. Então, como não há confusão possível e existem situações em que a herança múltipla de interfaces é útil, Java as permite.

Um uso para a herança múltipla de interfaces é uma aproximação da técnica de herança múltipla conhecida como **mistura**. Ao contrário de Java, algumas linguagens orientadas a objetos tais como C++ e Smalltalk permitem a herança múltipla de classes concretas, e não apenas de

interfaces. Em tais linguagens, é comum a criação das chamadas classes *mistura*, que não são projetadas para serem instanciadas, mas apenas para proporcionar funcionalidades adicionais a classes existentes. Entretanto, tal tipo de herança não é permitida em Java, então os programadores podem se aproximar dela através de interfaces. Pode-se usar a herança múltipla de interfaces como um mecanismo para “misturar” os métodos de duas ou mais interfaces não-relacionadas para definir uma outra que combina suas funcionalidades, talvez agregando mais alguns métodos próprios. Analisando novamente o exemplo das antiguidades, pode-se definir uma interface para descrever itens seguráveis, como segue:

```
public interface InsurableItem extends Transportable, Sellable {  
    /** Retorna o valor segurado em centavos. */  
    public int insuredValue();  
}
```

Esta interface mistura os métodos da interface Transportable com os da interface Sellable, e adiciona um método extra, insuredValue. Tal interface permite definir a classe BoxedItem de outra forma:

```
public class BoxedItem2 implements InsurableItem {  
    // ...o resto do código fica exatamente como antes  
}
```

Neste caso, observa-se que o método insuredValue não é opcional, o que acontecia na declaração de BoxedItem fornecida anteriormente.

Entre as interfaces de Java que se aproximam ao conceito da mistura, destacam-se java.lang.Cloneable, que acrescenta a capacidade de cópia a uma classe, java.lang.Comparable, a qual agrupa a capacidade de comparação a uma classe (impondo uma ordem natural a suas instâncias), e java.util.Observer, que adiciona o recurso de atualização a classes que desejam ser notificadas quando certos objetos “observáveis” têm seu estado alterado.

---

#### 2.4.3 Classes abstratas e tipagem forte

Uma classe abstrata é aquela que contém uma declaração de método vazia (isto é, uma declaração de método sem implementação) e definições concretas de métodos e variáveis de instância. Por isso, uma classe abstrata situa-se entre uma interface e uma classe concreta. Da mesma forma que uma interface, uma classe abstrata não pode ser instanciada, ou seja, nenhum objeto pode ser criado a partir de uma classe abstrata. Uma subclasse de uma classe abstrata deve prover a implementação dos métodos abstratos de sua superclasse ou será também considerada abstrata. Da mesma forma que uma classe concreta, porém, uma classe abstrata A pode estender outra classe abstrata, e tanto classes concretas como abstratas podem mais tarde estender A. Por fim, pode-se definir outra classe que não é abstrata e estender (subclasses) uma superclasse abstrata, e esta nova classe deve fornecer o código de todos os métodos abstratos. Sendo assim, classes abstratas usam a chamada herança de especificação, mas também permitem especialização e extensão (ver Seção 2.2.3).

##### A classe java.lang.Number

De fato, já se viu um exemplo de classe abstrata. As chamadas classes numéricas de Java (apresentadas na Tabela 1.2) especializam uma classe abstrata chamada java.lang.Number. Cada classe numérica concreta, como java.lang.Integer e java.lang.Double, estendem a classe java.lang.Number e implementam os detalhes dos métodos abstratos da superclasse. Em especial, os métodos intValue, floatValue, doubleValue e longValue são todos abstratos na classe java.lang.Number. Cada classe numérica concreta deve especificar os detalhes desses métodos.

## Tipagem forte

Uma variável de objeto pode ser vista como sendo de vários tipos. O tipo primário de um objeto  $o$  é a classe  $C$  especificada no momento em que  $o$  é instanciado. Além disso,  $o$  é do tipo de cada superclasse  $S$  de  $C$ , e do tipo  $I$  de cada interface  $I$  implementada por  $C$ .

Entretanto, uma variável só pode ser declarada como sendo de apenas um tipo (uma classe ou uma interface), o que determina como a variável será usada e como certos métodos irão agir sobre a mesma. Da mesma forma, um método tem um único tipo de retorno. Em geral, uma expressão tem um tipo único.

Reforçando o fato de que todas as variáveis são tipadas e que as operações declaram os tipos esperados, Java usa a técnica da **tipagem forte** para auxiliar na prevenção de erros. Mas com exigências rígidas para tipos, há necessidade de trocar ou converter um tipo em outro. Estas conversões devem ser especificadas por um operador de **conversão**. Já se discutiu (Seção 1.3.3) como as conversões funcionam para os tipos base. Na seqüência, será discutido como elas funcionam para variáveis referência.

## 2.5 Conversão e genéricos

Nesta seção, discute-se a conversão entre variáveis referência, bem como a técnica conhecida como genéricos, que permite que se evite o uso de conversão explícita em muitos casos.

### 2.5.1 Conversão

A discussão começa pelos métodos de conversão de tipo para objetos.

#### Conversões ampliadas

Uma **conversão ampliada** ocorre quando um tipo  $T$  é convertido para um tipo “ampliado”  $U$ . Os casos a seguir são exemplos de conversões ampliadas:

- $T$  e  $U$  são classes e  $U$  é uma superclasse de  $T$ .
- $T$  e  $U$  são interfaces e  $U$  é uma superinterface de  $T$ .
- $T$  é uma classe que implementa uma interface  $U$ .

Conversões ampliadas são feitas automaticamente para armazenar o resultado de uma expressão em uma variável sem a necessidade de conversão explícita. Assim, pode-se atribuir diretamente o resultado de uma expressão do tipo  $T$  em uma variável  $v$  do tipo  $U$  quando a conversão de  $T$  para  $U$  for uma conversão ampliada. O exemplo do trecho de código que segue mostra que uma expressão do tipo `Integer` (um objeto `Integer` recém criado) pode ser atribuído a uma variável do tipo `Number`.

```
Integer i = new Integer(3);
Number n = i; // conversão ampliada de Integer para Number
```

A correção de uma conversão ampliada pode ser verificada pelo compilador e sua validade não precisa ser testada pelo ambiente de execução de Java durante a execução.

#### Conversões reduzidas

Uma **conversão reduzida** ocorre quando um tipo  $T$  é convertido em um tipo “reduzido”  $S$ . Os casos a seguir são exemplos de conversões reduzidas:

- $T$  e  $S$  são classes e  $S$  é uma subclasse de  $T$ .
- $T$  e  $S$  são interfaces e  $S$  é uma subinterface de  $T$ .
- $T$  é uma interface implementada pela classe  $S$ .

Normalmente, uma conversão reduzida de referências requer uma conversão explícita. Além disso, a correção de uma conversão reduzida pode não ser verificável pelo compilador. Assim, sua validade deve ser testada pelo ambiente de execução de Java durante a execução do programa.

O exemplo do trecho de código que segue mostra como usar um conversor para executar uma conversão reduzida do tipo `Number` para o tipo `Integer`.

```
Number n = new Integer(2); // conversão ampliada de Integer para Number
Integer i = (Integer) n; // conversão reduzida de Number para Integer
```

No primeiro comando, um objeto novo da classe `Integer` é criado e atribuído a uma variável do tipo `Number`. Logo, uma conversão ampliada ocorre nessa atribuição, e nenhum conversor é necessário. No segundo comando, atribui-se `n` a variável `i` do tipo `Integer` usando um conversor. Esta atribuição é possível porque `n` se refere a um objeto do tipo `Integer`. Entretanto, como a variável `n` é do tipo `Number`, ocorre uma conversão reduzida e um conversor é necessário.

### Exceções de conversão

Em Java, é possível converter uma referência `o` do tipo `T` em um tipo `S`, desde que o objeto referenciado por `o` seja na verdade do tipo `S`. Se, por outro lado, o objeto `o` não for do tipo `S`, então tentar converter `o` para o tipo `S` irá causar uma exceção chamada `ClassCastException`. Esta regra é apresentada no trecho de código que segue:

```
Number n;
Integer i;
n = new Integer(3);
i = (Integer) n; // Isso é legal
n = new Double(3.1415);
i = (Integer) n; // Isso é ilegal
```

Para evitar problemas como esse, e evitar ter de poluir o código com blocos `try-catch`, toda a vez que se usa conversões, Java fornece uma maneira de se ter certeza que uma conversão de objeto será correta. A saber, é fornecido o operador `instanceof`, que permite testar se uma variável refere-se a um objeto de uma certa classe (ou implementa uma determinada interface). A sintaxe para usar este operador é `referência_para_objeto instanceof tipo_referenciado`, onde `referência_para_objeto` é uma expressão que retorna uma referência para objeto e `tipo_referenciado` é o nome de uma classe, interface ou enumeração (Seção 1.1.3). Se `referência_para_objeto` é também uma instância para `tipo_referenciado`, então a expressão anterior retorna `true`. Caso contrário ela retorna `false`. Assim, pode-se evitar que uma `ClassCastException` seja lançada modificando o trecho de código como segue:

```
Number n;
Integer i;
n = new Integer(3);
if (n instanceof Integer)
    i = (Integer) n; // Isso é legal
n = new Double(3.1415);
if (n instanceof Integer)
    i = (Integer) n; // Isso não será tentado
```

### Conversões com interfaces

Interfaces permitem forçar que objetos implementem certos métodos, mas usar variáveis interface com objetos concretos por vezes implica no uso de conversores. Suponha, por exemplo, que se deseja declarar a interface `Person` ("pessoa") apresentada no Trecho de código 2.12. Observe que

o método `equalTo` da interface `Person` recebe um parâmetro do tipo `Person`. Logo, pode-se passar um objeto de qualquer classe que implemente a interface `Person` para este método.

```
public interface Person {
    public boolean equalTo (Person other); // Esta é a mesma pessoa?
    public String getName(); // Retorna o nome desta pessoa
    public int getAge(); // Retorna a idade desta pessoa
}
```

#### Trecho de código 2.12 Interface Person.

No Trecho de código 2.13 é apresentada uma classe, `Student`, que implementa `Person`. O método `equalTo` assume que o argumento (declarado do tipo `Person`) é também do tipo `Student`, e executa uma conversão reduzida do tipo `Person` (uma interface) para o tipo `Student` (uma classe). Esta conversão é permitida neste caso porque é uma conversão reduzida da classe `T` para uma interface `U`, onde o objeto obtido de `T` é tal que `T` estende `S` (ou `T = S`) e `S` implementa `U`.

```
public class Student implements Person {
    String id;
    String name;
    int age;
    public Student (String i, String n, int a) { // construtor simples
        id = i;
        name = n;
        age = a;
    }
    protected int studyHours() { return age/2; } // apenas um "chute"
    public String getID() { return id; } // ID do estudante
    public String getName() { return name; } // da interface Person
    public int getAge() { return age; } // da interface Person
    public boolean equalTo (Person other) { // da interface Person
        Student otherStudent = (Student) other; // converte Person para Student
        return (id.equals (otherStudent.getID())); // compara os IDs
    }
    public String toString() { // para impressão
        return "Student (ID: " + id +
            ", Name: " + name +
            ", Age: " + age + ")";
    }
}
```

#### Trecho de código 2.12 Classe Student que implementa a interface Person.

Em função da premissa assumida na implementação do método `equalTo`, é necessário ter certeza que uma aplicação que use objetos da classe `Student` não irá tentar comparar objetos `Student` com outros tipos de objetos, pois, neste caso, a conversão no método `equalTo` irá falhar. Por exemplo, se a aplicação gerencia um diretório de objetos `Student` e não usa outros tipos de objetos `Person`, então a premissa será satisfeita.

A capacidade de executar conversões reduzidas de tipos interface para classes permite que se escrevam estruturas de dados genéricas que façam apenas suposições mínimas sobre os elementos que elas armazenam. No Trecho de código 2.14, esboça-se como construir um diretório de pares de objetos que implementam a interface `Person`. O método `remove` pesquisa o conteúdo do diretório e remove o par de pessoas especificado, se este existir, e, da mesma forma que o método `findOther`, usa o método `equalTo` para fazê-lo.

```

public class PersonPairDirectory {
    // as variáveis de instância vão aqui
    public PersonPairDirectory() { /* o construtor default vai aqui */ }
    public void insert(Person person, Person other) { /* o código de inserção vai aqui */ }
    public Person findOther(Person person) { return null; } // substituto para find
    public void remove(Person person, Person other) { /* o código de remoção vai aqui */ }
}

```

#### Trecho de código 2.14 Classe PersonPairDirectory.

Supondo que se preencha um diretório, myDirectory, com pares de objetos Student que representam colegas de quarto, para encontrar o companheiro de um certo estudante, smart\_one (“esperto”), pode-se tentar fazer o seguinte (o que está *errado*):

```
Student cute_one = myDirectory.findOther(smart_one); // errado !
```

O comando anterior provoca um erro de compilação do tipo “*explicit cast required*”\*. O problema é que se está tentando fazer uma conversão reduzida sem um conversor explícito. A saber, o valor retornado pelo método *findOther* é do tipo Person, enquanto que a variável *cute\_one*, que está sendo atribuída, é de um tipo “menor” Student, uma classe que implementa a interface Person. Logo, usa-se um conversor para converter o tipo Person para o tipo Student, como segue:

```
Student cute_one = (Student) myDirectory.findOther(smart_one);
```

A conversão do tipo Person retornado pelo método *findOther* para o tipo Student funciona sempre que houver certeza que a camada para *myDirectory.findOther* fornece um objeto Student. Em geral, interfaces são ferramentas valiosas para o projeto de estruturas genéricas, pois podem ser especializadas por outros programadores por meio do uso de conversores.

### 2.5.2 Genéricos

A partir da versão 5.0, Java inclui um *framework genérico* que permite o uso de tipos abstratos de dados de uma forma que evita muitas conversões explícitas. Um *tipo genérico* é um tipo que não é definido em tempo de compilação, mas que é especificado em tempo de execução. O framework genérico permite que se defina uma classe em termos de um conjunto de *parâmetros de tipo* que podem ser usados, por exemplo, para abstrair os tipos de algumas variáveis internas da classe. Os sinais de maior e menor são usados para delimitar a lista de parâmetros de tipo. Ainda que qualquer identificador válido possa ser usado como parâmetro, letras maiúsculas individuais são usadas por convenção. Dada uma classe que foi definida com esses parâmetros, instancia-se um objeto dessa classe usando-se *tipos reais* para indicar os tipos concretos a serem usados.

No Trecho de código 2.15 é apresentada a classe Pair, que armazena pares valor-chave, onde os tipos da chave e dos valores são especificados pelos parâmetros K e V, respectivamente. O método main cria duas instâncias desta classe, uma para o par String-Integer (para armazenar uma dimensão e seu valor, por exemplo) e outra para o par Student-Double (para armazenar a nota de um estudante, por exemplo).

```

public class Pair<K, V> {
    K key;
    V value;
    public void set(K k, V v) {

```

\* N. de T. Requer conversor explícito.

```

        key = k;
        value = v;
    }
    public K getKey() { return key; }
    public V getValue() { return value; }
    public String toString() {
        return "[" + getKey() + ", " + getValue() + "]";
    }
    public static void main (String[ ] args) {
        Pair<String, Integer> pair1 = new Pair<String, Integer>();
        pair1.set(new String("height"), new Integer(36));
        System.out.println(pair1);
        Pair<Student, Double> pair2 = new Pair<Student, Double>();
        pair2.set(new Student("A5976", "Sue", 19), new Double(9.5));
        System.out.println(pair2);
    }
}

```

**Trecho de código 2.15** Exemplo usando a classe Student do Trecho de código 2.13.

A saída resultante da execução deste método é apresentada a seguir:

```
[height, 36]
[Student(ID: A5976, Name: Sue, Age: 19), 9.5]
```

No exemplo anterior, o tipo real do parâmetro podia ser qualquer tipo. Para restringir o tipo do parâmetro real, pode-se usar a cláusula **extends**, como mostrado na seqüência, onde a classe PersonPairDirectoryGeneric é definida em termos do parâmetro de tipo genérico P, parcialmente especificado, declarando-se que o mesmo estende a classe Person.

```

public class PersonPairDirectoryGeneric,P extends Person {
    // ... as variáveis de instância vão aqui ...
    public PersonPairDirectoryGeneric() /* o construtor default vai aqui */
    public void insert (P person, P other) /* o código de inserção vai aqui */
    public P findOther (P person) { return null; } // substituta para find
    public void remove (P person, P other) /* o código de remoção vai aqui */
}

```

Esta classe pode ser comparada com a classe PersonPairDirectory no Trecho de código 2.14. Dada a classe anterior, pode-se declarar uma variável referindo uma instância de PersonPairDirectoryGeneric, que armazene pares de objetos do tipo Student:

```
PersonPairDirectoryGeneric,Student. myStudentDirectory;
```

Para esta instância, o método findOther retorna um valor do tipo Student. Logo, o comando a seguir, que não usa conversores, está correto:

```
Student cute one 5 myStudentDirectory.findOther(smart one);
```

O framework genérico permite a definição de versões genéricas de métodos. Neste caso, pode-se incluir a definição genérica entre os modificadores dos métodos. Por exemplo, na seqüência, apresenta-se a definição de um método que compara as chaves de quaisquer dois objetos Pair, desde que suas chaves implementem a interface Comparable.

```

public static <K extends Comparable,V,L,W> int
    comparePairs(Pair<K,V> p, Pair<L,W> q) {
    return p.getKey().compareTo(q.getKey()); // a chave de p implementa compareTo
}

```

Existe um problema importante relacionado aos tipos genéricos, pois os elementos armazenados em um arranjo não podem ser um variável de tipo ou um tipo parametrizado. Java permite que um arranjo seja definido com um tipo parametrizado, mas não permite que um tipo parametrizado seja usado para criar um arranjo novo. Felizmente, permite-se que um arranjo definido com um tipo parametrizado seja inicializado com um arranjo não paramétrico recém criado. Mesmo assim, este último mecanismo faz com que o compilador de Java gere um aviso, porque não é 100% seguro em relação aos tipos. Esta questão é demonstrada a seguir:

```
public static void main(String[ ] args) {
    Pair<String, Integer>[ ] a = new Pair[10]; // correto, mas gera um aviso
    Pair<String, Integer>[ ] b = new Pair<String, Integer>[10]; // errado !!
    a[0] = new Pair<String, Integer>(); // correto
    a[0].set("Dog", 10);           // este comando e o próximo estão corretos
    System.out.println("First pair is "+a[0].getKey() + ", " + a[0].getValue());
}
```

## 2.6 Exercícios

Para obter ajuda e o código-fonte dos exercícios, visite [java.datastructures.net](http://java.datastructures.net).

### Reforço

- R-2.1 Duas interfaces podem estender uma a outra? Por que sim ou por que não?
- R-2.2 Forneça três exemplos de aplicações críticas de software.
- R-2.3 Forneça um exemplo de aplicação de software em que a capacidade de adaptação pode significar a diferença entre um período longo de vendas e a falência.
- R-2.4 Descreva um componente de um editor de textos com interface GUI (que não seja o menu “editar”) e os métodos que ele encapsula.
- R-2.5 Desenhe o diagrama de herança para os seguintes conjuntos de classes:
  - A classe Goat (bode) estende Object, acrescentando uma variável de instância tail e os métodos milk() e jump()\*.
  - A classe Pig (porco) estende Object, acrescentando uma variável de instância nose e os métodos eat() e wallow()\*\*.
  - A classe Horse (cavalo) estende a classe Object, acrescentando as variáveis de instância height e color e os métodos run() e jump()\*\*\*.
  - A classe Racer (corredor) estende a classe Horse, acrescentando o método race()\*\*\*\*.
  - A classe Equestrian (“cavaleiro”) estende Horse, acrescentando a variável de instância weight e os métodos trot() e isTrained()\*\*\*\*\*.
- R-2.6 Escreva um pequeno trecho de código Java que use as classes de progressão da Seção 2.2.3 para encontrar o oitavo valor de uma série de Fibonacci que inicia com 2 e 2 como sendo seus dois valores iniciais.

\* N. de T. “Rabo”, “leite” e “pular”, respectivamente.

\*\* N. de T. “Nariz”, “comer” e “chafurdar”, respectivamente.

\*\*\* N. de T. “Altura”, “cor”, “correr” e “pular”, respectivamente.

\*\*\*\* N. de T. “Corrida”.

\*\*\*\*\* N. de T. “Peso”, “trotar” e “está treinado”, respectivamente.

- R-2.7 Se forem escolhidos `inc = 128`, quantas chamadas ao método `nextValue` da classe `ArithProgression`, da Seção 2.2.3, podem ser feitas antes de ser provocado um overflow de inteiro longo?
- R-2.8 Considere uma variável de instância `p` declarada do tipo `Progression`, usando as classes da Seção 2.2.3. Suponha que na realidade `p` faça referência a uma instância da classe `GeometricProgression`, que foi criada usando o construtor default. Convertendo `p` para o tipo `Progression` e ativando `p.firstValue()`, qual será o valor retornado? Por quê?
- R-2.9 Analise a herança de classes do Exercício R-2.5 e faça `d` ser uma variável objeto do tipo `Horse`. Se `d` se refere a um objeto real do tipo `Equestrian`, ela pode ser convertida para a classe `Racer`? Por que sim ou por que não?
- R-2.10 Escreva um exemplo de trecho de código em Java que execute uma referência para arranjo que possivelmente esteja fora de faixa e, se isso ocorrer, que o programa capture a exceção e imprima a seguinte mensagem de erro:
- `"Don't try buffer overflow attacks in Java!"`
- R-2.11 Analise o seguinte trecho de código extraído de um pacote:

```

public class Maryland extends State {
    Maryland() { /* construtor nulo */ }
    public void printMe() { System.out.println("Read it."); }
    public static void main(String[] args) {
        Region mid = new State();
        State md = new Maryland();
        Object obj = new Place();
        Place usa = new Region();
        md.printMe();
        mid.printMe();
        ((Place) obj).printMe();
        obj = md;
        ((Maryland) obj).printMe();
        obj = usa;
        ((Place) obj).printMe();
        usa = md;
        ((Place) usa).printMe();
    }
}
class State extends Region {
    State() { /* construtor nulo */ }
    public void printMe() { System.out.println("Ship it."); }
}
class Region extends Place {
    Region() { /* construtor nulo */ }
    public void printMe() { System.out.println("Box it."); }
}
class Place extends Object {
    Place() { /* construtor nulo */ }
    public void printMe() { System.out.println("Buy it."); }
}

```

O que é exibido após a ativação do método `main()` da classe `Maryland`?

- R-2.12 Escreva um pequeno método Java que conte o número de vogais em uma string.
- R-2.13 Escreva um pequeno método Java que remove toda a pontuação de um string *s* armazenando uma frase. Por exemplo, esta operação deve transformar a string "Let's try, Mike" em "Lets try Mike".
- R-2.14 Escreva um pequeno programa que recebe como entrada três inteiros, *a*, *b* e *c*, a partir do console Java e determina se eles podem ser usados em uma fórmula aritmética correta (na ordem em que foram fornecidos), como em "*a* + *b* = *c*", "*a* = *b* - *c*" ou "*a*\**b* = *c*".
- R-2.15 Escreva um pequeno programa que cria uma classe Pair, que armazena dois objetos declarados como tipos genéricos. Faça um exemplo de uso deste programa criando e imprimindo pares de objetos Pair que contenham cinco tipos diferentes de pares, tais como <Integer, String> e <Float, Long>.
- R-2.16 Parâmetros genéricos não são incluídos na assinatura da declaração de um método, de maneira que não se pode ter métodos diferentes na mesma classe que tenha diferentes parâmetros genéricos mas os mesmos nomes e tipos e quantidade de parâmetros. Como se pode alterar a assinatura dos métodos em conflito de maneira a contornar esse problema?

### Criatividade

- C-2.1 Explique por que o algoritmo de ativação dinâmica de Java que define o método que será ativado para uma determinada mensagem *o.a()* nunca irá entrar em laço infinito.
- C-2.2 Escreva uma classe em Java que estende a classe Progression, criando uma progressão em que cada valor é o módulo da diferença entre os dois valores anteriores. Deve-se incluir um construtor que inicie com 2 e 200 como sendo os dois valores iniciais e um construtor parametrizado, que inicie com um par de valores informados como iniciais.
- C-2.3 Escreva uma classe em Java que estende a classe Progression, criando uma progressão em que cada valor é a raiz quadrada do valor anterior. (Observe que não é mais possível representar os valores como inteiros.) Você deve incluir um construtor default que inicie com 65.536 como primeiro valor, e um construtor parametrizado que inicie com um valor informado (**double**) como primeiro valor.
- C-2.4 Reescreva todas as classes da hierarquia da classe Progression de forma que todos os valores sejam da classe BigInteger, de maneira a evitar overflows.
- C-2.5 Escreva um programa que consiste de três classes, *A*, *B* e *C*, tais que *B* estende *A* e *C* estende *B*. Cada classe deve definir uma variável de instância chamada "x" (isto é, cada uma tem sua própria variável chamada *x*). Descreva uma forma que permita a um método de *C* acessar e alterar a versão da variável *x* de *A* sem alterar as versões de *B* ou *C*.
- C-2.6 Escreva um conjunto de classes Java que possam simular uma aplicação Internet onde um usuário, Alice, periodicamente crie pacotes que deseja enviar para Bob. Um processo da Internet está sempre verificando se Alice tem algum pacote para enviar e, se tiver, despacha-os para o computador de Bob, e Bob está periodicamente verificando se o seu computador tem um pacote de Alice, e, se tiver, lê o mesmo e o deleta.

---

## Projetos

- P-2.1 Escreva um programa Java que recebe um documento e exibe um gráfico de barras com as freqüências de cada letra do alfabeto que aparece no documento.
  - P-2.2 Escreva um programa Java que simule uma calculadora portátil. O programa deve ser capaz de processar entradas tanto de uma GUI como do console Java, para acionar os botões e então exibir o conteúdo da tela após a execução de cada operação. No mínimo, a calculadora deve ser capaz de efetuar as operações aritméticas básicas e as operações zerar/limpar.
  - P-2.3 Complete o código da classe PersonPairDirectory do Trecho de código 2.14, assumindo que pares de pessoas são armazenadas em um arranjo com capacidade 1000. O diretório deve manter o registro de quantas pessoas estão registradas no momento.
  - P-2.4 Escreva um programa Java que recebe um inteiro positivo maior que 2 como entrada e exibe o número de vezes que alguém pode, repetidamente, dividir este número por 2 antes de obter um resultado menor que 2.
  - P-2.5 Escreva um programa Java que “faça troco”. O programa deve receber dois números como entrada, um o valor pago e o outro o valor devido. Ele deve então retornar a quantidade de cada tipo de nota e moeda que deve ser devolvido como troco, devido a diferença entre o que foi pago e o que foi cobrado. Os valores das notas e moedas podem ser os do sistema monetário de qualquer governo. Tente projetar o programa de maneira que ele retorne a menor quantidade de notas e moedas possível.
- 

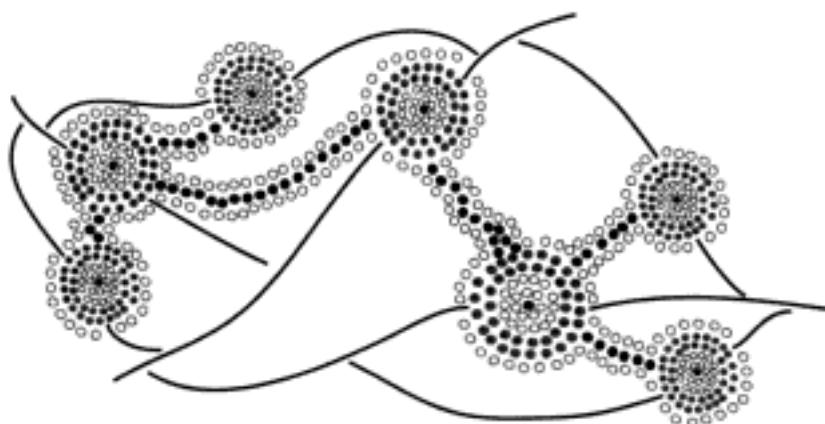
## Observações sobre o capítulo

Para uma revisão abrangente dos desenvolvimentos recentes da ciência e da engenharia da computação, sugere-se *The Computer Science and Engineering Handbook* [92]. Para mais informações sobre o incidente com o Terac-25, ver o artigo de Leveson e Turner [66].

Para o leitor preocupado com estudos avançados em programação orientada a objetos, indicam-se os livros de Booch [14], Budd[17] e Liskov e Guttag[69]. Liskov e Guttag [69] também oferecem uma análise interessante sobre tipos abstratos de dados, da mesma forma que o artigo científico de Cardelli e Wegner [20], assim como o capítulo do livro de Demurjian[28] em *The Computer Science and Engineering Handbook* [92]. Padrões de projeto são descritos no livro de Gamma *et al* [38]. A notação dos diagramas de herança de classe que foi utilizada é derivada do livro de Gamma *et al*.



# 3 Arranjos, Listas Encadeadas e Recursão



## Conteúdo

<b>3.1 Usando arranjos.....</b>	<b>102</b>
3.1.1 Armazenando os registros de um jogo em um arranjo .....	102
3.1.2 Ordenando um arranjo .....	107
3.1.3 Métodos de <code>java.util</code> para arranjos e números aleatórios .....	109
3.1.4 Criptografia simples com strings e arranjos de caracteres.....	111
3.1.5 Arranjos bidimensionais e jogos de posição.....	114
<b>3.2 Listas simplesmente encadeadas .....</b>	<b>117</b>
3.2.1 Inserção em uma lista simplesmente encadeada.....	119
3.2.2 Removendo um elemento em uma lista simplesmente encadeada .....	120
<b>3.3 Listas duplamente encadeadas .....</b>	<b>121</b>
3.3.1 Inserção no meio de uma lista duplamente encadeada .....	123
3.3.2 Remoção do meio de uma lista duplamente encadeada .....	125
3.3.3 Implementação de uma lista duplamente encadeada.....	125
<b>3.4 Listas encadeadas circulares e ordenação de listas encadeadas .....</b>	<b>128</b>
3.4.1 Listas encadeadas circulares e a brincadeira do "Pato, Pato, Ganso"....	128
3.4.2 Ordenando uma lista encadeada .....	131
<b>3.5 Recursão.....</b>	<b>133</b>
3.5.1 Recursão linear .....	137
3.5.2 Recursão binária .....	141
3.5.3 Recursão múltipla .....	143
<b>3.6 Exercícios .....</b>	<b>145</b>

### 3.1 Usando arranjos

Nesta seção, serão exploradas algumas aplicações de arranjos que foram introduzidas na Seção 1.5.

#### 3.1.1 Armazenando os registros de um jogo em um arranjo

A primeira aplicação a ser estudada armazena registros em um arranjo – em especial os registros dos maiores scores de um videogame. Armazenar registros em um arranjo é um uso comum para arranjos, e poderia ter-se escolhido armazenar os prontuários de pacientes de um hospital ou os nomes dos estudantes de uma aula de estruturas de dados. Em vez disso, optou-se por armazenar os registros dos maiores scores por ser uma aplicação simples, mas que apresenta alguns conceitos de estruturas de dados importantes que serão usados em outras implementações ao longo deste livro.

Inicialmente começa-se perguntando o que se deseja armazenar em um registro de score. Obviamente, um dos componentes é um inteiro representando o score propriamente dito que se pode chamar de `score`. Seria interessante incluir também o nome da pessoa que obteve o score, e que será chamado simplesmente de `name`. Pode-se continuar acrescentando campos para representar a data da obtenção do score ou estatísticas do jogo que levaram a tal score. Entretanto, este exemplo será mantido simples dispondo apenas de dois campos: `score` e `name`. No Trecho de código 3.1, apresenta-se uma classe Java que representa um registro do jogo.

```
public class GameEntry {
    protected String name; // nome da pessoa que obteve o score
    protected int score; // valor do score
    /** Construtor que cria um registro do jogo */
    public GameEntry(String n, int s) {
        name = n;
        score = s;
    }
    /** Recupera o campo nome */
    public String getName() { return name; }
    /** Recupera o campo score */
    public int getScore() { return score; }
    /** Retorna uma string com a representação deste registro */
    public String toString() {
        return "(" + name + ", " + score + ")";
    }
}
```

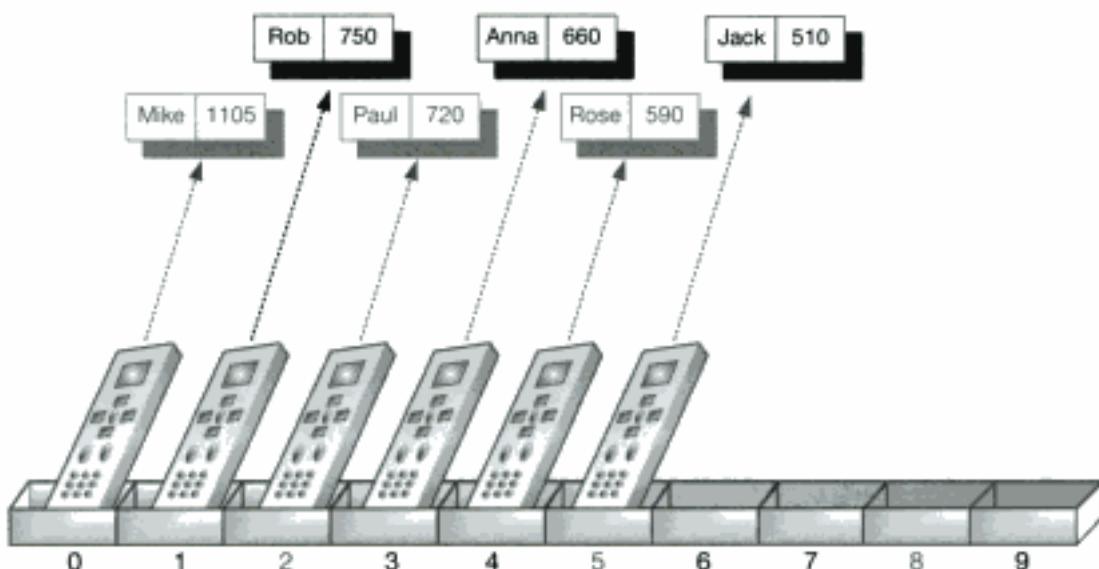
**Trecho de código 3.1** Código Java da classe `GameEntry`. Observa-se que foram incluídos métodos para retornar o nome e o score de um objeto registro, bem como um método que retorna uma representação string do mesmo.

#### Uma classe para os maiores scores

Suponha que se deseja armazenar os maiores scores em um arranjo chamado `entries`. A quantidade de scores que se deseja armazenar pode ser 10, 20 ou 50, de maneira que será usado um nome simbólico `maxEntries`, que irá representar a quantidade de scores que se deseja armazenar. Naturalmente, deve-se atribuir um valor para esta variável, mas, usando a variável ao longo do código, esta alteração é fácil de ser feita posteriormente, se for o caso. Define-se então o arranjo, `entries`, para ser um arranjo com comprimento `maxEntries`. Inicialmente, este arranjo armazena apenas entradas `null`, mas à medida que os usuários jogam o videogame, preenchem-se as entradas do ar-

ranjo com referências para novos objetos da classe GameEntry. Desta forma, é necessário definir métodos para atualizar as referências para GameEntry no arranjo entries.

A maneira de manter as entradas do arranjo organizadas é simples: armazena-se o conjunto de objetos GameEntry ordenados pelo valor dos scores, do maior para o menor. Se o número de objetos GameEntry é menor que maxEntries, então deixa-se que as últimas posições do arranjo armazenem referências **null**. Esta abordagem previne que existam células vazias ou “buracos” na série contínua de células do arranjo entries que armazena os registros do jogo indexados de 0 em diante. A Figura 3.1 ilustra uma instância desta estrutura de dados, e o código Java correspondente é fornecido no Trecho de código 3.2. No Exercício C-3.1, explora-se como o acréscimo de registros pode ser simplificado para o caso quando não é necessário preservar a ordem relativa.



**Figura 3.1** Esquema de um arranjo de comprimento 10, armazenando referências para seis objetos GameEntry nas células indexadas de 0 a 5 e com as restantes sendo referências **null**.

```
/** Classe que armazena os maiores scores em um arranjo em ordem não decrescente */
public class Scores {
    public static final int maxEntries = 10; // Quantidade de scores que serão armazenados
    protected int numEntries; // número real de registros
    protected GameEntry[] entries; // arranjo de registros (nomes & scores)
    /** Construtor default */
    public Scores() {
        entries = new GameEntry[maxEntries];
        numEntries = 0;
    }
    /** Retorna uma representação string da lista de scores */
    public String toString() {
        String s = "[";
        for (int i = 0; i < numEntries; i++) {
            if (i > 0) s += ", ";
            s += entries[i];
        }
        return s + "]";
    }
    // ... os métodos para atualizar o conjunto de scores vão aqui ...
}
```

**Trecho de código 3.2** Classe para manter um conjunto de objetos GameEntry.

Nota-se que foi incluído um método, `toString()`, que produz uma representação string dos maiores scores armazenados no arranjo `entries`. Este método é muito útil para depuração. Neste caso, a string será uma lista, separada por vírgulas, dos objetos `GameEntry` armazenados no arranjo `entries`. Esta lista é produzida por um laço `for` simples, que acrescenta uma vírgula antes de cada registro que antecede ao primeiro. Com tal representação string, pode-se imprimir o estado do arranjo `entries` durante a depuração, de maneira a testar como estão as coisas antes e depois das atualizações.

## Inserção

Uma das atualizações mais comuns que se deseja fazer com o arranjo `entries` dos scores mais altos é o acréscimo de um novo registro. Supondo que se deseja inserir um novo objeto `GameEntry`, `e`. Neste caso, levaremos em consideração como será executada a seguinte operação de atualização sobre uma instância da classe `Scores`:

`add(e)`: insere o registro `e` na coleção de maiores scores. Se a coleção está cheia, então `e` é acrescentado apenas se o seu score é maior que o menor score armazenado no conjunto `e`, neste caso, `e` substitui a entrada com menor score.

O maior desafio para implementar esta operação é descobrir onde `e` deve entrar no arranjo `entries`, e abrir espaço para `e`.

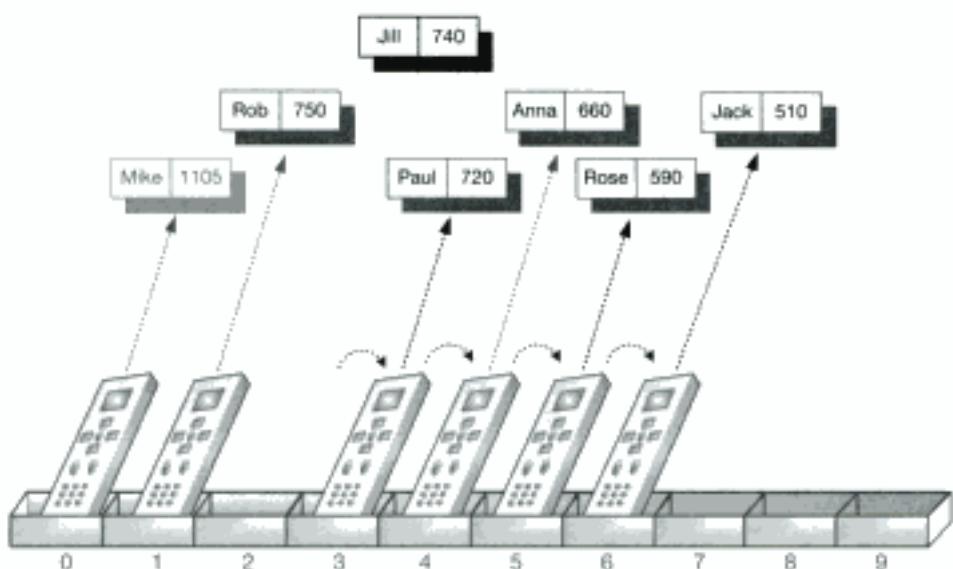
## Visualizando a inserção de um registro

Para visualizar o processo de inserção, imagina-se que o arranjo `entries` armazena controles remotos que representam referências para objetos `GameEntry` que não são nulos, listados da esquerda para direita, do maior para o menor score.

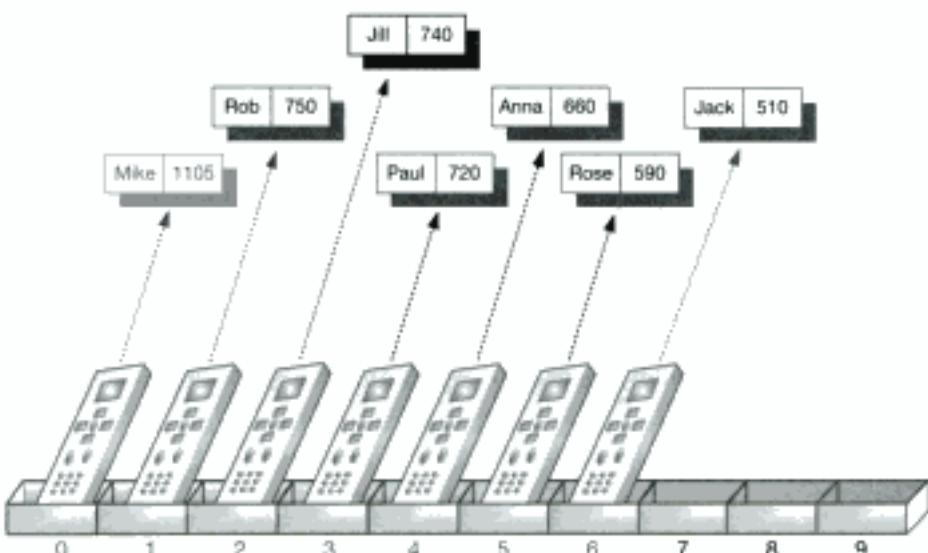
Dado o novo registro, `e`, é necessário determinar a que posição ele pertence. Inicia-se esta pesquisa pelo final do arranjo `entries`. Se a última referência do arranjo não é `null`, e seu score é maior que o score de `e`, então pode-se parar por aí. Neste caso, `e` não é um dos maiores scores – ele não deve pertencer ao arranjo de maiores scores. Caso contrário, sabe-se que `e` deve pertencer ao arranjo, e também sabe-se que o último registro armazenado no arranjo não deve mais pertencer ao mesmo. Na seqüência, move-se para a penúltima referência do arranjo. Se esta referência é `null`, ou aponta para um objeto `GameEntry` cujo score é menor que o referenciado por `e`, esta referência deve ser movida uma célula para a direita. Além disso, se esta referência foi movida, então é necessário repetir esta comparação com a próxima célula, desde que ainda não tenha sido encontrado o ínicio do arranjo. Continua-se comparando e deslocando as referências para os registros até atingir o ínicio do arranjo ou até comparar o score de `e` com um score maior. Neste caso, identificou-se a posição a qual `e` pertence (ver Figura 3.2).

Uma vez que foi identificado o lugar do arranjo `entries` ao qual o objeto `e` pertence, armazena-se a referência `e` nesta posição. Sendo assim, entendendo as referências para objetos como controles remotos, acrescentou-se um controle remoto especialmente projetado para `e` nesta posição do arranjo `entries` (ver Figura 3.3).

Os detalhes do algoritmo para acrescentar um registro novo `e` no arranjo `entries` são similares a esta descrição informal, e são fornecidos em Java no Trecho de código 3.3. Observa-se o uso de um laço para mover as referências. O número de vezes que se executa este laço depende do número de referências que é necessário mover para abrir espaço para o registro novo. Se existem 0, 1 ou mesmo poucas referências para mover, este método `add` será muito rápido. Mas se existirem várias para mover, então este método pode se tornar um tanto lento. Observa-se também que se o arranjo está cheio, e se executa um `add` sobre o mesmo, ou será removida a referência para o último registro do arranjo ou a inserção do novo registro, `e`, irá falhar.



**Figura 3.2** Preparando para acrescentar um novo objeto GameEntry no arranjo entries. De maneira a abrir espaço para nova referência, deve-se deslocar as referências dos registros com scores menores que o do novo uma célula para a direita.



**Figura 3.3** Acrescentando uma referência para um novo objeto GameEntry ao arranjo entries. A referência foi inserida no índice 2, uma vez que todas as referências para objetos GameEntry com scores menores que o do registro novo foram deslocadas para a direita.

```
/** Tenta inserir um novo score na coleção (se ele for grande o suficiente) */
public void add(GameEntry e) {
    int newScore = e.getScore();
    // o novo registro é correspondente mesmo a um dos maiores scores?
    if (numEntries == maxEntries) { // o arranjo está cheio
        if (newScore <= entries[numEntries - 1].getScore())
            return; // neste caso, a nova entrada, e, não é um dos maiores scores
    }
    else // o arranjo não está cheio
        numEntries++;
    // localiza o lugar onde o novo registro é (com score grande) deve ficar
    int i = numEntries - 1;
    for (; (i >= 1) && (newScore > entries[i - 1].getScore()); i--)
        entries[i] = entries[i - 1];
    entries[i] = e;
}
```

```

        entries[i] = entries[i - 1];      // move a entrada i uma posição para direita
        entries[i] = e;                  // acrescenta o novo score as entradas
    }
}

```

**Trecho de código 3.3** Código Java para inserção de um objeto GameEntry.

### Remoção de objetos

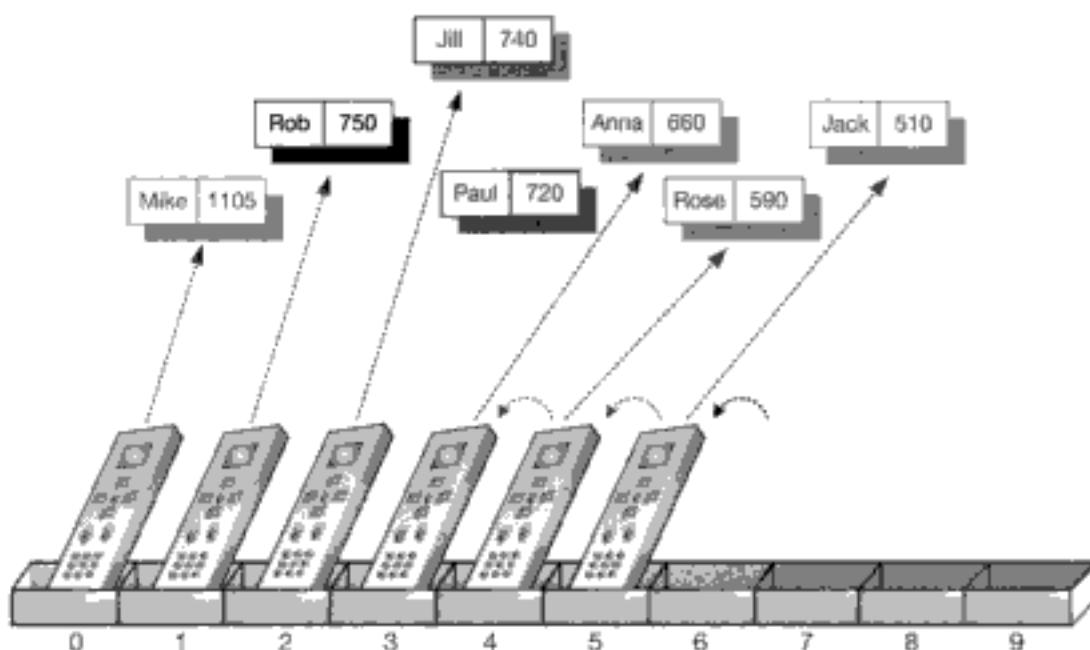
Suponha que um *expert* use o videogame e coloque seu nome na lista de melhores scores. Neste caso, será necessário dispor de um método que permita remover um registro desta lista. Por essa razão, será analisado como remover uma referência para um objeto GameEntry do arranjo entries. Isto é, será analisado como se pode implementar a operação que segue:

**remove(*i*):** remove e retorna o registro *e* de índice *i* do arranjo **entries**. Se o índice *i* estiver fora dos limites do arranjo, então este método lança uma exceção; caso contrário, as entradas do arranjo são atualizadas de maneira a remover o objeto sob o índice *i*, e todos os objetos anteriormente armazenados em índices mais altos que *i* são “movidos” de maneira a preencher a posição liberada pelo objeto removido.

Esta implementação de **remove** usa um algoritmo semelhante ao de inserção de objetos, porém ao contrário. Novamente, pode-se entender o arranjo **entries** como um arranjo de controles-remotos que apontam para objetos GameEntry. Para remover a referência para o objeto no índice *i*, começa-se pelo índice *i* e move-se todas as referências armazenadas em índices mais altos que *i* uma célula para a esquerda. (ver a Figura 3.4).

### Alguns detalhes sutis sobre remoção de registros

Os detalhes da execução de uma operação de remoção possuem alguns aspectos sutis. O primeiro é que para remover e retornar o registro (identificado como *e*), localizado sob o índice *i* no arranjo, deve-se primeiramente salvar *e* em uma variável temporária. Usa-se esta variável para retornar *e* quando a remoção estiver completa. O segundo aspecto sutil é que, movendo as referências



**Figura 3.4** Esquema da remoção do índice 3 em um arranjo que armazena referências para objetos GameEntry.

maiores que  $i$  uma célula para a esquerda, não se vai até o fim do arranjo — pára-se na penúltima referência. Para-se antes do fim porque a última referência não tem nenhuma referência a sua direita (conseqüentemente, não existe referência a ser movida para a última posição do arranjo `entries`). No lugar da última referência, é suficiente simplesmente se atribuir nulo para a mesma. Conclui-se retornando a referência para o registro removido (que não possui mais nenhuma referência apontando para o mesmo no arranjo `entries`). Ver o Trecho de código 3.4.

```
/** Remove e retorna o score armazenado no índice  $i$  */
public GameEntry remove(int i) throws IndexOutOfBoundsException {
    if ((i < 0) || (i >= numEntries))
        throw new IndexOutOfBoundsException("Invalid index: " + i);
    GameEntry temp = entries[i]; // armazena temporariamente o objeto a ser removido
    for (int j = i; j < numEntries - 1; j++) // conta a partir de  $i$ 
        entries[j] = entries[j+1]; // move uma célula para esquerda
    entries[numEntries - 1] = null; // anula o último score
    numEntries--;
    return temp; // retorna o objeto removido
}
```

**Trecho de código 3.4** Código Java para a execução da operação de remoção.

Estes métodos de adição e remoção de objetos em um arranjo de maiores scores são simples. Entretanto, eles formam a base das técnicas que são usadas de forma repetida para construir estruturas de dados mais sofisticadas. Naturalmente, essas outras estruturas são mais genéricas que a estrutura de arranjo descrita, e normalmente oferecerão muito mais operações do que o que se pode fazer com apenas `add` e `remove`. Porém, estudar a estrutura de dados concreta do arranjo, como está sendo feito agora, é um ótimo ponto de partida para se entender as demais estruturas, uma vez que todos as estruturas de dados são implementadas a partir de dados concretos.

Na verdade, mais adiante neste livro, será estudada uma das classes de coleções de Java, `ArrayList`, que é mais geral que a estrutura de arranjo analisada aqui. A classe `ArrayList` tem métodos para fazer uma série de coisas que se deseja fazer com um arranjo, além de eliminar os problemas que ocorrem quando se acrescenta um objeto em um arranjo cheio. O `ArrayList` elimina este erro copiando automaticamente os objetos em um arranjo maior. Em vez de discutir esse processo aqui, entretanto, será visto mais sobre como isso é feito quando a classe `ArrayList` for analisada em detalhes.

### 3.1.2 Ordenando um arranjo

Na seção anterior, trabalhou-se intensivamente para mostrar como acrescentar ou remover objetos em um determinado índice  $i$  de um arranjo, mantendo a ordenação dos objetos. Nesta seção, será estudada uma maneira de iniciar com um arranjo contendo objetos que estão fora de ordem, e então colocá-los em ordem. Isso é conhecido como o problema da *ordenação*.

#### Um algoritmo de inserção ordenada simples

Serão estudados diversos algoritmos de ordenação neste livro, a maioria no Capítulo 11. Para introduzir o assunto, entretanto, nesta seção será descrito um algoritmo de ordenação simples chamado de *inserção ordenada*. Neste caso, descreve-se uma versão específica do algoritmo onde a entrada é um arranjo de elementos comparáveis. Categorias mais gerais de algoritmos de ordenação serão consideradas posteriormente neste livro.

O algoritmo de ordenação simples funciona como segue. Inicia-se com o primeiro caractere do arranjo. Um caractere por si só já está ordenado. Então, considera-se o próximo caractere. Se for menor que o primeiro, então invertem-se as posições de ambos. Na seqüência, considera-se o terceiro caractere do arranjo. Desloca-se o mesmo para a esquerda até que esteja na posição correta em relação aos dois primeiros. Considera-se então o quarto caractere, e desloca-se o mesmo para a esquerda até que esteja na posição correta em relação aos outros três. Continua-se procedendo desta forma com o quinto, o sexto e assim por diante, até que todo o arranjo esteja ordenado. Combinando esta descrição informal com construções de programação, pode-se expressar o algoritmo de inserção ordenada como apresentado no Trecho de código 3.5.

#### **Algoritmo InsertionSort( $A$ )**

**Entrada:** Um arranjo  $A$  com  $n$  elementos comparáveis

**Saída:** O arranjo  $A$  com elemento reorganizados em ordem não decrescente

**Para**  $i \leftarrow 1$  até  $n - 1$  **faça**

Inserir  $A[i]$  na localização correta dentre  $A[0], A[1], \dots, A[i - 1]$ .

**Trecho de código 3.5** Descrição de alto nível do algoritmo de inserção ordenada.

Esta é uma boa descrição de alto nível do algoritmo de inserção ordenada. Ela demonstra também por que o algoritmo é chamado de “inserção ordenada”: porque cada interação do laço principal insere o próximo elemento na parte ordenada do arranjo que vem antes dele. Antes que se possa codificar esta descrição, entretanto, é necessário trabalhar melhor os detalhes da operação de inserção.

Aprofundando um pouco mais esses detalhes, esta descrição será reescrita usando dois laços aninhados. O laço mais externo considera um elemento do arranjo de cada vez, e o laço mais interno desloca o elemento para a localização adequada no subarranjo (ordenado) de caracteres, que está à sua esquerda.

Refinando os detalhes da inserção ordenada

Refinando os detalhes, então, descreve-se o algoritmo como apresentado no Trecho de código 3.6.

#### **Algoritmo InsertionSort( $A$ )**

**Entrada:** Um arranjo  $A$  com  $n$  elementos comparáveis

**Saída:** O arranjo  $A$  com elemento reorganizados em ordem não decrescente

**Para**  $i \leftarrow 1$  até  $n - 1$  **faça**

{ Inserir  $A[i]$  na localização correta dentre  $A[0], A[1], \dots, A[i - 1]$ . }

$cur \leftarrow A[i]$

$j \leftarrow i - 1$

**Enquanto**  $j \geq 0$  e  $a[j] > cur$  **faça**

$A[j+1] \leftarrow A[j]$

$j \leftarrow j - 1$

$A[j + 1] \leftarrow cur$  {  $cur$  agora está na posição correta }

**Trecho de código 3.6** Descrição de nível intermediário do algoritmo de inserção ordenada.

Esta descrição está muito mais próxima do código real, uma vez que explica melhor como inserir o elemento  $A[i]$  no subarranjo que o antecede. Ele ainda usa uma descrição informal da movimentação dos elementos se eles estiverem fora de ordem, mas isso não é uma coisa muito difícil de resolver.

### Descrição Java da inserção ordenada

Agora, pode-se apresentar o código Java para esta versão simples do algoritmo de inserção ordenada. Apresenta-se esta descrição no Trecho de código 3.7, para o caso especial em que *A* é o arranjo de caracteres *a*.

```
/** Inserção ordenada de um arranjo de caracteres em ordem não decrescente */
public static void insertionSort(char[ ] a) {
    int n = a.length;
    for (int i = 1; i < n; i++) {           // índice do segundo caractere em a
        char cur = a[i];                  // o caractere corrente a ser inserido
        int j = i - 1;                    // inicia comparando a célula a esquerda de i
        while ((j >= 0) && (a[j] > cur)) // enquanto a[j] está fora de ordem em relação a cur
            a[j + 1] = a[j] --;          // move a[j] para a direita e decrementa j
        a[j + 1] = cur;                // este é o local correto de cur
    }
}
```

**Trecho de código 3.7** Código Java para executar a inserção ordenada sobre um arranjo de caracteres.

A Figura 3.5 ilustra um exemplo de execução do algoritmo de inserção ordenada.

Algo interessante ocorre com este algoritmo se o arranjo já está previamente ordenado. Neste caso, o laço interno faz apenas uma comparação, verifica que não é necessária nenhuma troca e retorna para o laço mais externo. Isto é, executa-se apenas uma iteração do laço mais interno para cada iteração do laço mais externo. Assim, neste caso, executa-se o número mínimo de comparações. Naturalmente, tem-se muito mais trabalho quando o arranjo está completamente desordenado. Na verdade, a maior carga de trabalho irá ocorrer se o arranjo estiver em ordem decrescente.

### 3.1.3 Métodos de java.util para arranjos e números aleatórios

Como arranjos são extremamente importantes, Java fornece uma grande quantidade de métodos predefinidos que executam tarefas comuns sobre arranjos. Estes métodos apresentam-se como métodos estáticos da classe `java.util.Arrays`. Isto é, eles estão associados a classe `java.util.Arrays` propriamente dita, e não com uma instância particular da classe. A descrição de alguns destes métodos, entretanto, terá de esperar até que os conceitos em que são baseados sejam estudados.

#### Alguns dos métodos mais simples de `java.util.Arrays`

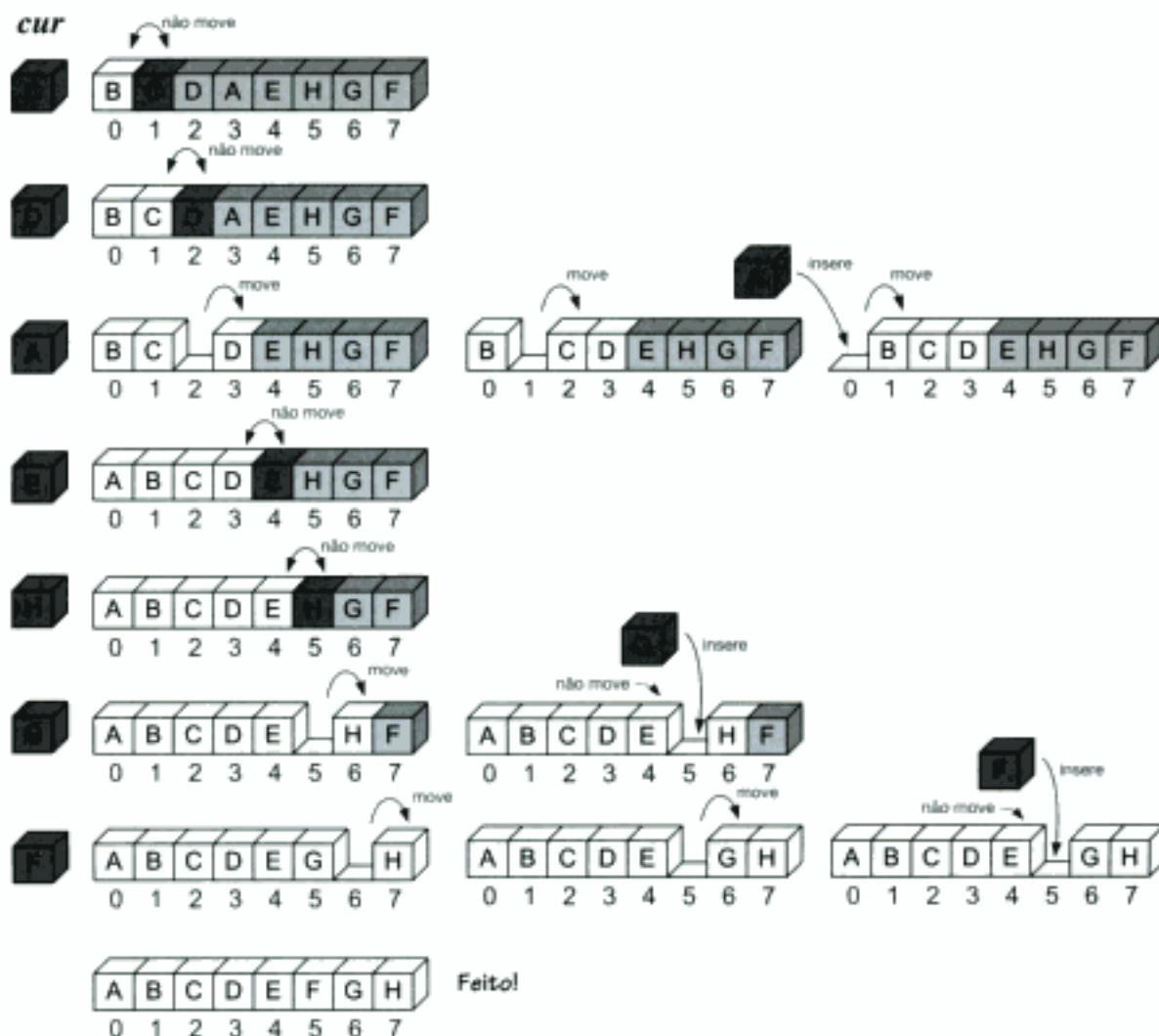
São listados a seguir alguns dos métodos mais simples da classe `Java.util.Arrays` que não necessitam maiores explicações:

`equals(A, B)`: retorna true se e somente se os arranjos *A* e *B* são iguais. Dois arranjos são considerados iguais se eles têm o mesmo número de elementos, e todo par correspondente de elementos nos dois arranjos é igual. Isto é, *A* e *B* tem os mesmos elementos na mesma ordem.

`fill(A, x)`: armazena o elemento *x* em todas as células de *A*.

`sort(A)`: ordena o arranjo *A* usando a ordenação natural de seus elementos.

`toString(A)`: retorna a representação de *A* sob a forma de uma string.



**Figura 3.5** Execução do algoritmo de inserção ordenada sobre um arranjo de oito caracteres. Apresenta-se a parte completa (ordenada) do arranjo em branco e colore-se o próximo elemento a ser inserido na parte ordenada com cinza claro. Destaca-se, também, o caractere na esquerda, uma vez que ele é armazenado na variável *cur*. Cada linha corresponde a uma iteração do laço mais externo e cada cópia do arranjo em uma mesma linha corresponde a uma iteração do laço mais interno. Cada comparação é indicada com um arco. Além disso, indica-se quando o resultado de uma comparação resulta em movimentação ou não.

Por exemplo, a string a seguir será retornada pelo método `toString` ativado sobre o arranjo de inteiros  $A = [4,5,2,3,5,7,10]$ :

`[4,5,2,3,5,7,10]`

Observa que, pela lista anterior, Java possui um algoritmo de ordenação predefinido. Este, entretanto, não é o algoritmo de inserção ordenada apresentado anteriormente. É um algoritmo chamado de quick-sort, que normalmente executa muito mais rápido que o de inserção ordenada. O algoritmo quick-sort será estudado na Seção 11.2.

### Um exemplo usando números pseudo-aleatórios

No Trecho de código 3.8 é apresentado um pequeno (mas completo) programa Java que usa os métodos listados.

```

import java.util.Arrays;
import java.util.Random;
/** Programa que apresenta alguns usos para arranjos */
public class ArrayTest {
    public static void main(String[ ] args) {
        int num[ ] = new int[10];
        Random rand = new Random( ); // Um gerador de números pseudo-aleatórios
        rand.setSeed(System.currentTimeMillis( )); // usa o tempo corrente como semente
        // preenche o arranjo com números pseudo-aleatórios entre 0 e 99, inclusive
        for (int i = 0; i < num.length; i++)
            num[i] = rand.nextInt(100); // o próximo número pseudo aleatório
        int[ ] old = (int[ ]) num.clone( ); // clona o arranjo num
        System.out.println("arrays equal before sort: " + Arrays.equals(old,num));
        Arrays.sort(num); // ordena o arranjo num (old não é modificado)
        System.out.println("arrays equal after sort: " + Arrays.equals(old,num));
        System.out.println("old = " + Arrays.toString(old));
        System.out.println("num = " + Arrays.toString(num));
    }
}

```

**Trecho de código 3.8** Programa teste ArrayTest que usa vários dos métodos predefinidos da classe Array.

O programa ArrayTest usa outro recurso de Java – a habilidade de gerar números pseudo-aleatórios, isto é, números que são estatisticamente aleatórios (mas não verdadeiramente aleatórios). Neste caso, usa o objeto Java.util.Random, que é um *gerador de números pseudo-aleatórios*, isto é, um objeto que calcula ou “gera” uma seqüência de números que são estatisticamente aleatórios. Tal gerador necessita, entretanto, de um ponto para começar, chamado de *semente*. A seqüência de números aleatórios para uma dada semente será sempre a mesma. Neste programa, a semente escolhida é o tempo decorrido em milissegundos a partir de 1º de janeiro de 1970 (usando o método System.currentTimeMillis), que será diferente cada vez que o programa for executado. Uma vez selecionada a semente, pode-se repetidamente obter um número randômico entre 0 e 99 chamando-se o método nextInt com o argumento 100. Na sequência, apresenta-se um exemplo de saída deste programa:

```

arrays equal before sort: true
arrays equal after sort: false
old = [41,38,48,12,28,46,33,19,10,58]
num = [10,12,19,28,33,38,41,46,48,58]

```

A propósito, existe uma pequena chance dos arranjos old e num permanecerem iguais após a ordenação de num, que é o caso em que num já estiver ordenado antes de ser clonado. A chance disso ocorrer, entretanto, é menor que uma em quatro milhões.

### 3.1.4 Criptografia simples com strings e arranjos de caracteres

Uma das aplicações primárias de arranjos é a representação de strings de caracteres. Isto é, objetos strings são normalmente armazenados internamente como um arranjo de caracteres. Mesmo que strings possam ser representadas de alguma outra forma, existe uma relação natural entre strings e arranjos de caracteres – ambos usam índices para referenciar os caracteres. Em função desse relacionamento, Java torna simples a criação de strings a partir de arranjos de caracteres e

vice-versa. Mais especificamente, para criar um objeto da classe `String` a partir de um arranjo de caracteres `A`, simplesmente usa-se a expressão,

```
new String(A)
```

Isto é, um dos construtores da classe `String` recebe um arranjo de caracteres como argumento e retorna um string com os mesmos caracteres e na mesma ordem que o arranjo. Por exemplo, o string que será criado a partir do arranjo `A = [a, c, a, t]` é `acat`. Da mesma forma, dada uma string `S`, pode-se criar uma representação sob a forma de arranjo de caracteres para `S` usando a expressão,

```
S.toCharArray()
```

Isto é, a classe `String` tem um método `toCharArray` que retorna um arranjo (do tipo `char[]`) com os mesmos caracteres que `S`. Por exemplo, se `toCharArray` for ativado sobre o string `adog`, será obtido o arranjo `B = [a, d, o, g]`.

### A Cifra de César

Uma área onde é útil ter a capacidade de alternar entre uma string e um arranjo de caracteres é de volta novamente é em *criptografia*, a ciência das mensagens secretas e suas aplicações. Este campo estuda formas de executar *criptografia*, que recebe uma mensagem, chamada de *texto limpo* e converte o mesmo em uma mensagem misturada, chamada de *texto cifrado*. Da mesma forma, a criptografia também estuda maneiras de fazer a *decriptografia*, que recebe um texto cifrado e retorna o texto limpo original.

Discutivelmente, o esquema de criptografia mais antigo é a Cifra de César, que recebeu este nome em homenagem a Julio César, que usou este esquema para proteger importantes mensagens militares (todas as mensagens de César eram escritas em Latim, naturalmente, o que as tornava incompreensíveis para a maioria das pessoas). A Cifra de César é uma maneira simples de confundir uma mensagem escrita em uma linguagem que forma palavras a partir de um alfabeto.

A Cifra de César implica em substituir cada letra de uma mensagem pela letra que está a três letras de distância no alfabeto da língua. Assim, em uma mensagem em inglês, substitui-se cada A por um D, cada B por um E, cada C por um F, e assim por diante. Continua-se com esta abordagem até o W, que é substituído pelo Z. Então, faz-se o padrão de substituição *girar*, substituindo-se o X por A, o Y por B e o Z por C.

### Usando caracteres como índices de arranjo

Se as letras forem numeradas como se fossem os índices de um arranjo, então A corresponde a 0, B a 1, C a 2, e assim por diante, e então se pode escrever a Cifra de César como uma fórmula simples:

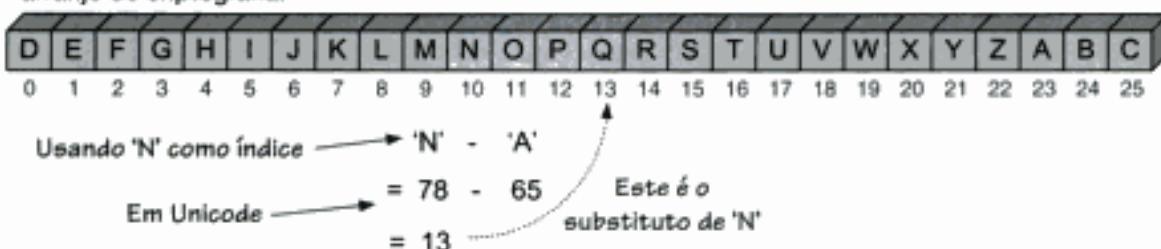
Substitua cada letra  $i$  pela letra  $(i + 3) \bmod 26$ ,

onde `mod` é o operador **módulo**, que retorna o resto de uma divisão inteira. Este operador é denotado `%` em Java, e é exatamente o operador necessário para facilitar o “giro” ao redor do fim do alfabeto. Para  $26 \bmod 26$ , a resposta é 0,  $27 \bmod 26$  resulta 1 e  $28 \bmod 26$  resulta 2. O algoritmo de decriptografia para a Cifra de César é exatamente o contrário: substitui-se cada letra por uma três posições antes com o “giro” para A, B e C.

Pode-se capturar esta regra de substituição usando arranjos para encriptar e desencriptar. Uma vez que todo caractere em Java é, na verdade, armazenado como um número – seu valor Unicode – pode-se usar letras como índices de um arranjo. Para um caractere `c` maiúsculo, por exemplo, pode-se usar `c` como índice de arranjo pegando o valor Unicode de `c` e subtraindo `A`. Naturalmente, isso funciona apenas para letras maiúsculas, de maneira que será necessário que as mensagens secretas sejam maiúsculas. Pode-se então usar um arranjo `encrypt`, que representa a regra de criptografia, de maneira que `encrypt[i]` contém a letra que substitui a letra número  $i$ .

(que corresponde a  $c - A$  para um caractere maiúsculo em Unicode). Este uso é demonstrado na Figura 3.6. Da mesma forma, um arranjo, `decrypt`, pode representar a regra de descriptografia, de maneira que `decrypt[i]` contém a letra que substitui a letra número  $i$ .

arranjo de criptografia:



**Figura 3.6** Demonstração do uso de caracteres maiúsculos como índices de arranjos. Neste caso, para executar a regra de substituição do mecanismo de criptografia da Cifra de César.

No Trecho de código 3.9, fornece-se uma classe Java simples, mas completa, para executar a Cifra de César, que usa a abordagem apresentada e também faz uso das conversões entre strings e arranjos de caracteres. Quando se executa este programa (para executar um teste simples), o resultado é a seguinte saída:

```

Encryption order = DEFGHIJKLMNOPQRSTUVWXYZABC
Decryption order = XYZABCDEFGHIJKLMNOPQRSTUVWXYZ
WKH HDJOH LV LQ SODB; PHHW DW MRH'V.
THE EAGLE IS IN PLAY; MEET AT JOE'S.

/** Classe para criptografar e descriptografar usando a Cifra de César. */
public class Caesar {
    public static final int ALPHASIZE = 26; //Alfabeto em inglês (somente letras maiúsculas)
    public static final char[] alpha = {'A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I',
        'J', 'K', 'L', 'M', 'N', 'O', 'P', 'Q', 'R', 'S', 'T', 'U', 'V', 'W', 'X', 'Y', 'Z'};
    protected char[] encrypt = new char[ALPHASIZE]; // Encryption array
    protected char[] decrypt = new char[ALPHASIZE]; // Decryption array
    /** Construtor que inicializa os arranjos de criptografar e descriptografar */
    public Caesar() {
        for (int i=0; i<ALPHASIZE; i++)
            encrypt[i] = alpha[(i + 3) % ALPHASIZE]; // gira o alfabeto 3 posições
        for (int i=0; i<ALPHASIZE; i++)
            decrypt[encrypt[i] - 'A'] = alpha[i]; // descriptografar é o contrário da criptografia
    }
    /** Método de criptografia */
    public String encrypt(String secret) {
        char[] mess = secret.toCharArray(); // o arranjo com a mensagem
        for (int i=0; i<mess.length; i++) // laço de criptografia
            if (Character.isUpperCase(mess[i])) // tem-se uma letra para trocar
                mess[i] = encrypt[mess[i] - 'A']; // usa a letra como índice
        return new String(mess);
    }
    /** Método de descriptografar */
    public String decrypt(String secret) {
        char[] mess = secret.toCharArray(); // o arranjo com a mensagem
        for (int i=0; i<mess.length; i++) // laço de descriptografar
            if (Character.isUpperCase(mess[i])) // tem-se uma letra para trocar
                mess[i] = decrypt[mess[i] - 'A']; // usa a letra como índice
        return new String(mess);
    }
}

```

```
}

/** Um método main simples para testar a Cifra de César */
public static void main(String[ ] args) {
    Caesar cipher = new Caesar(); // Cria um objeto com a Cifra de César
    System.out.println("Encryption order = " + new String(cipher.encrypt()));
    System.out.println("Decryption order = " + new String(cipher.decrypt()));
    String secret = "THE EAGLE IS IN PLAY; MEET AT JOE'S .";
    secret = cipher.encrypt(secret);
    System.out.println(secret); // o texto cifrado
    secret = cipher.decrypt(secret);
    System.out.println(secret); // deve ser texto limpo novamente
}
}
```

Trecho de código 3.9 Uma classe Java simples, mas completa, para a Cifra de César.

### 3.1.5 Arranjos bidimensionais e jogos de posição

Muitos jogos de computador, sejam eles de estratégia, de simulação ou de conflito, usam um tabuleiro bidimensional. Programas que lidam com tais *jogos de posição* necessitam uma maneira de representar objetos em um espaço bidimensional. Uma forma natural de fazer isso é usando um *arranjo de duas dimensões*, onde se usam dois índices, por exemplo, *i* e *j*, para referenciar as células do arranjo. O primeiro índice normalmente se refere a um número de linha, e o segundo a um número de coluna. Dado tal arranjo, pode-se manter tabuleiros bidimensionais, bem como executar outros tipos de cálculos envolvendo os dados armazenados nas linhas e colunas.

Arranjos em Java são unidimensionais; usa-se um único índice para acessar cada célula do arranjo. Apesar disso, existe uma maneira de definir arranjos de duas dimensões em Java – pode-se criar um arranjo de duas dimensões como um arranjo de arranjos. Isto é, pode-se definir um arranjo bidimensional como sendo um arranjo onde cada uma de suas células é outro arranjo. Tal arranjo bidimensional é por vezes chamado de *matriz*. Em Java, declara-se um arranjo bidimensional como segue:

```
int[ ][ ] Y = new int[8][10];
```

Este comando cria um “arranjo de arranjo” de duas dimensões, *Y*, que é  $8 \times 10$ , tendo 8 linhas e 10 colunas. Isto é, *Y* é um arranjo de comprimento 8 onde cada elemento de *Y* é um arranjo de comprimento 10, de inteiros. (Ver a Figura 3.7.) O que segue são usos válidos para o arranjo *Y* e as variáveis *int*, *i* e *j*:

```
Y[i][i + 1] = Y[i][i] + 3;
i = a.length;
j = Y[4].length;
```

Arranjos bidimensionais têm várias aplicações em análise numérica. Em vez de entrar em detalhes sobre tais aplicações, entretanto, explora-se uma aplicação de arranjos de duas dimensões para implementar um jogo posicional simples.

#### Jogo da Velha

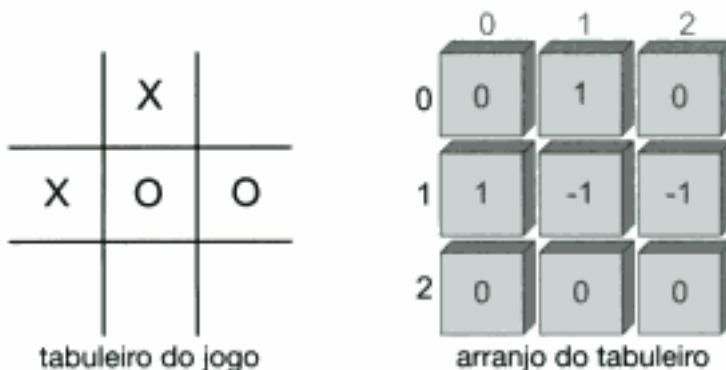
Como todas as crianças em idade escolar sabem, o *jogo da velha* é um jogo que se joga em um tabuleiro de 3 por 3 posições. Dois jogadores – X e O – se alternam colocando suas respectivas marcas nas células deste tabuleiro, iniciando pelo jogador X. Se um dos jogadores for bem-sucedido em obter três de suas marcas em uma linha, coluna ou diagonal, então será o vencedor.

	0	1	2	3	4	5	6	7	8	9
0	22	18	709	5	33	10	4	56	82	440
1	45	32	830	120	750	660	13	77	20	105
2	4	880	45	66	61	28	650	7	510	67
3	940	12	36	3	20	100	306	590	0	500
4	50	65	42	49	88	25	70	126	83	288
5	398	233	5	83	59	232	49	8	365	90
6	33	58	632	87	94	5	59	204	120	829
7	62	394	3	4	102	140	183	390	16	26

**Figura 3.7** Demonstração de um arranjo bidimensional Y que tem 8 linhas e 10 colunas. O valor Y[3][5] é 100 e o valor Y[6][2] é 632.

Este realmente não é um jogo posicional muito sofisticado, e não é muito divertido de jogar, pois um bom jogador sempre pode forçar o empate. A graça do jogo da velha está no fato de que é um exemplo simples para demonstrar como arranjos de duas dimensões podem ser usados em jogos de posição. Programas para jogos posicionais mais complicados tais como damas, xadrez ou os populares jogos de simulação são todos baseados na mesma abordagem apresentada aqui para o jogo da velha. (ver o Exercício P-7.8)

A idéia básica é usar um arranjo bidimensional, board, para manter o tabuleiro do jogo. As células deste arranjo armazenam valores que indicam se a célula está vazia ou armazena um X ou O. Isto é, board é uma matriz  $3 \times 3$ , cujas células da linha do meio consistem nas células board[1][0], board[1][1], board[1][2]. Neste caso, definiu-se que as células do arranjo board seriam inteiros, com um 0 indicando uma célula vazia, um 1 indicando um X e um -1 indicando O. Esta codificação permite uma maneira simples de testar se uma dada configuração de tabuleiro é vencedora para X ou O apenas testando se os valores de uma linha, coluna ou diagonal somam 3 ou -3. Demonstra-se esta abordagem na Figura 3.8.



**Figura 3.8** Demonstração do tabuleiro de jogo da velha e do arranjo de inteiros bidimensional que representa o mesmo.

Apresenta-se uma classe Java completa para manter um tabuleiro de jogo da velha para dois jogadores nos Trechos de código 3.10 e 3.11. Apresenta-se um exemplo de saída na Figura 3.9. Observa-se que este código serve apenas para manter o tabuleiro do jogo e registrar os movimentos; ele não executa nenhuma estratégia nem permite que se jogue contra o computador. Tal programa seria um bom projeto em uma aula de inteligência artificial.

```
/** Simulação do jogo da velha (não tem estratégia) */
public class TicTacToe {
    protected static final int X = 1, O = -1; // jogadores
    protected static final int EMPTY = 0; // célula vazia
```

```

protected int board[ ][ ] = new int[3][3];      // tabuleiro
protected int player;                         // jogador corrente
/** Construtor */
public TicTacToe( ) { clearBoard(); }
/** Limpa o tabuleiro */
public void clearBoard() {
    for (int i = 0; i < 3; i++)
        for (int j = 0; j < 3; j++)
            board[i][j] = EMPTY;           // toda a célula deve estar vazia
    player = X;                          // o primeiro jogador é 'X'
}
/** Coloca um X ou O na posição i,j */
public void putMark(int i, int j) throws IllegalArgumentException {
    if ((i < 0) || (i > 2) || (j < 0) || (j > 2))
        throw new IllegalArgumentException("Invalid board position");
    if (board[i][j] != EMPTY)
        throw new IllegalArgumentException("Board position occupied");
    board[i][j] = player;             // insere a marca do jogador corrente
    player = -player;               // troca os jogadores (usa o fato de que O = -X)
}
/** Verifica se a configuração do tabuleiro é vencedora para algum jogador */
public boolean isWin(int mark) {
    return ((board[0][0] + board[0][1] + board[0][2] == mark*3) // linha 0
        || (board[1][0] + board[1][1] + board[1][2] == mark*3) // linha 1
        || (board[2][0] + board[2][1] + board[2][2] == mark*3) // linha 2
        || (board[0][0] + board[1][0] + board[2][0] == mark*3) // coluna 0
        || (board[0][1] + board[1][1] + board[2][1] == mark*3) // coluna 1
        || (board[0][2] + board[1][2] + board[2][2] == mark*3) // coluna 2
        || (board[0][0] + board[1][1] + board[2][2] == mark*3) // diagonal
        || (board[2][0] + board[1][1] + board[0][2] == mark*3)); // diagonal
}
/** Retorna o jogador vencedor ou indica um empate */
public int winner() {
    if (isWin(X))
        return(X);
    else if (isWin(O))
        return(O);
    else
        return(0);
}

```

**Trecho de código 3.10** Uma classe Java simples e completa para jogar jogo da velha entre dois jogadores (continua no Trecho de código 3.11).

```

/** Retorna uma string de caracteres que representa o tabuleiro corrente */
public String toString() {
    String s = "";
    for (int i=0; i<3; i++) {
        for (int j=0; j<3; j++) {
            switch (board[i][j]) {
                case X: s += "X"; break;
                case O: s += "O"; break;
                case EMPTY: s += " "; break;
            }
        }
    }
}

```

```

        if (j < 2) s += " | ";
        // limite da coluna
    }
    if (i < 2) s += "\n-----\n";
    // limite da linha
}
return s;
}

/** Testa a execução de um jogo simples
public static void main(String[] args) {
    TicTacToe game = new TicTacToe();
    /* Jogada de X */          /* Jogada de O */
    game.putMark(1,1);         game.putMark(0,2);
    game.putMark(2,2);         game.putMark(0,0);
    game.putMark(0,1);         game.putMark(2,1);
    game.putMark(1,2);         game.putMark(1,0);
    game.putMark(2,0);
    System.out.println(game.toString());
    int winningPlayer = game.winner();
    if (winningPlayer != 0)
        System.out.println(winningPlayer + " wins");
    else
        System.out.println("Tie");
}
}

```

**Trecho de código 3.11** Uma classe Java simples e completa para jogar o jogo da velha entre dois jogadores (continuação do Trecho de código 3.10).

```

O | X | O
-----
O | X | X
-----
X | O | X
Tie

```

**Figura 3.9** Exemplo de saída do jogo da velha.

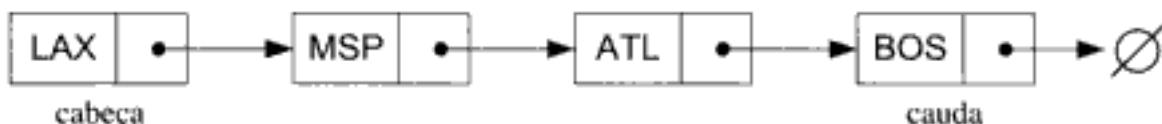
### 3.2 Listas simplesmente encadeadas

Na seção anterior, foi apresentada a estrutura de dados arranjo, e discutiram-se algumas de suas aplicações. Arranjos são interessantes e simples para armazenar coisas em uma certa ordem, mas têm o problema de não serem muito adaptáveis, uma vez que deve-se prever o tamanho  $N$  do arranjo.

Existem, entretanto, outras maneiras de armazenar uma seqüência de elementos que não têm este problema. Nesta seção, será explorada uma importante alternativa de implementação conhecida como lista simplesmente encadeada.

Uma *lista encadeada*, em sua forma mais simples, é uma coleção de *nodos* que juntos formam uma ordem linear. A ordem é determinada como no jogo de criança “siga o chefe”, no qual cada nodo é um objeto que armazena uma referência para um elemento e uma referência, chamada *next*, para outro nodo (ver a Figura 3.10).

Pode parecer estranho que um nodo tenha uma referência para outro nodo, mas este esquema funciona facilmente. A referência *next* dentro de um nodo pode ser vista como uma *ligação* ou um *ponteiro* para outro nodo. Da mesma forma, a movimentação de um nodo para outro seguindo a referência *next* é conhecida como *salto pela ligação* ou *salto pelo ponteiro*. O primeiro e o último nodos de uma lista encadeada são normalmente chamados de *cabeça* (*head*) e *cauda*



**Figura 3.10** Exemplo de uma lista simplesmente encadeada cujos elementos são strings indicando códigos de aeroportos. Os ponteiros next de cada nodo são representados como setas. O objeto `null` é denotado como  $\emptyset$ .

(*tail*) da lista, respectivamente. Assim, pode-se saltar pelas ligações da lista iniciando na cabeça e terminando na cauda. Identifica-se cauda por ser o nodo que possui uma referência next nula, o que indica o fim da lista. Uma lista encadeada definida desta forma é conhecida como uma *lista simplesmente encadeada*.

Da mesma forma que um arranjo, uma lista simplesmente encadeada mantém seus elementos em uma certa ordem. Esta ordem é determinada pela cadeia de ligações next que parte de um nodo para seu sucessor na lista. Ao contrário de um arranjo, uma lista encadeada não tem um tamanho fixo predeterminado e usa um espaço proporcional à quantidade de seus elementos. Além disso, os nodos de uma lista encadeada não são indexados. Assim, apenas examinando um nodo individual, não é possível dizer se ele é o segundo, quinto ou o vigésimo da lista.

### Implementando uma lista simplesmente encadeada

Para implementar uma lista simplesmente encadeada, define-se uma classe `Node`, como mostrado no Trecho de código 3.12, a qual especifica o tipo dos objetos que serão armazenados nos nodos da lista. Aqui, se assume que os elementos são strings. No Capítulo 5, se descreve como definir nodos que podem armazenar tipos arbitrários de elementos. Definida a classe `Node`, pode-se definir a classe `SLinkedList`, apresentada no Trecho de código 3.13, definindo a lista encadeada real. Esta classe mantém a referência para o nodo cabeça e uma variável que conta o número total de nodos.

```
/** Nodo de uma lista simplesmente encadeada de strings */
public class Node {
    private String element; // assumimos que os elementos são strings
    private Node next;
    /** Cria um nodo com um dado elemento e o próximo nodo */
    public Node(String s, Node n) {
        element = s;
        next = n;
    }
    /** Retorna o elemento deste nodo */
    public String getElement() { return element; }
    /** Retorna o próximo elemento deste nodo */
    public Node getNext() { return next; }
    // Métodos modificadores:
    /** Define o elemento deste nodo */
    public void setElement(String newElem) { element = newElem; }
    /** Define o próximo elemento deste nodo */
    public void setNext(Node newNext) { next = newNext; }
}
```

**Trecho de código 3.12** Implementação de um nodo de uma lista simplesmente encadeada.

```
/** Lista simplesmente encadeada */
public class SLinkedList {
    protected Node head; // nodo cabeça da lista
```

```

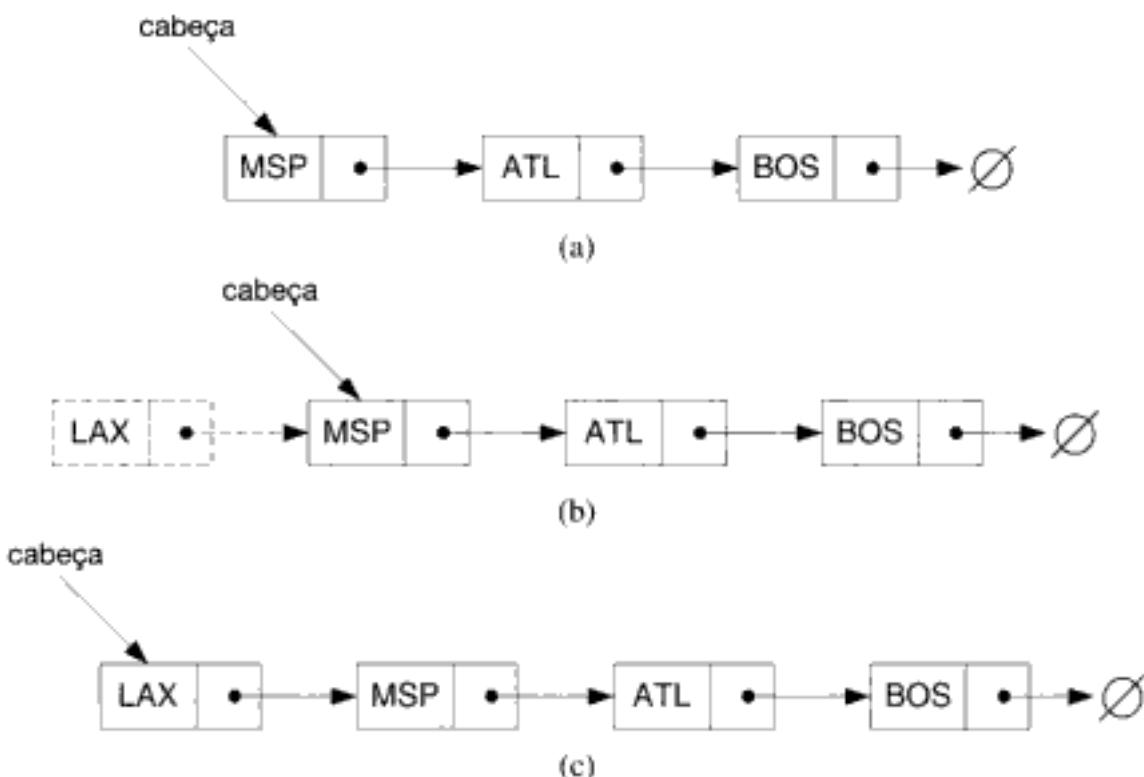
protected long size;      // número de nodos da lista
/** Construtor default que cria uma lista vazia */
public SLinkedList() {
    head = null;
    size = 0;
}
// ... os métodos de pesquisa e atualização vão aqui ...
}

```

**Trecho de código 3.13** Implementação parcial da classe de uma lista simplesmente encadeada.

### 3.2.1 Inserção em uma lista simplesmente encadeada

Quando se usa uma lista simplesmente encadeada, pode-se facilmente inserir um elemento na cabeça da lista, como pode ser visto na Figura 3.11 e no Trecho de código 3.14. A idéia principal é que se cria um nodo novo, define-se sua ligação next para referir o mesmo objeto que head e, então, define-se head para apontar para o novo nodo.



**Figura 3.11** Inserção de um elemento na cabeça de uma lista simplesmente encadeada: (a) antes da inserção; (b) criação do novo nodo; (c) depois da inserção.

#### Algoritmo addFirst(v)

```

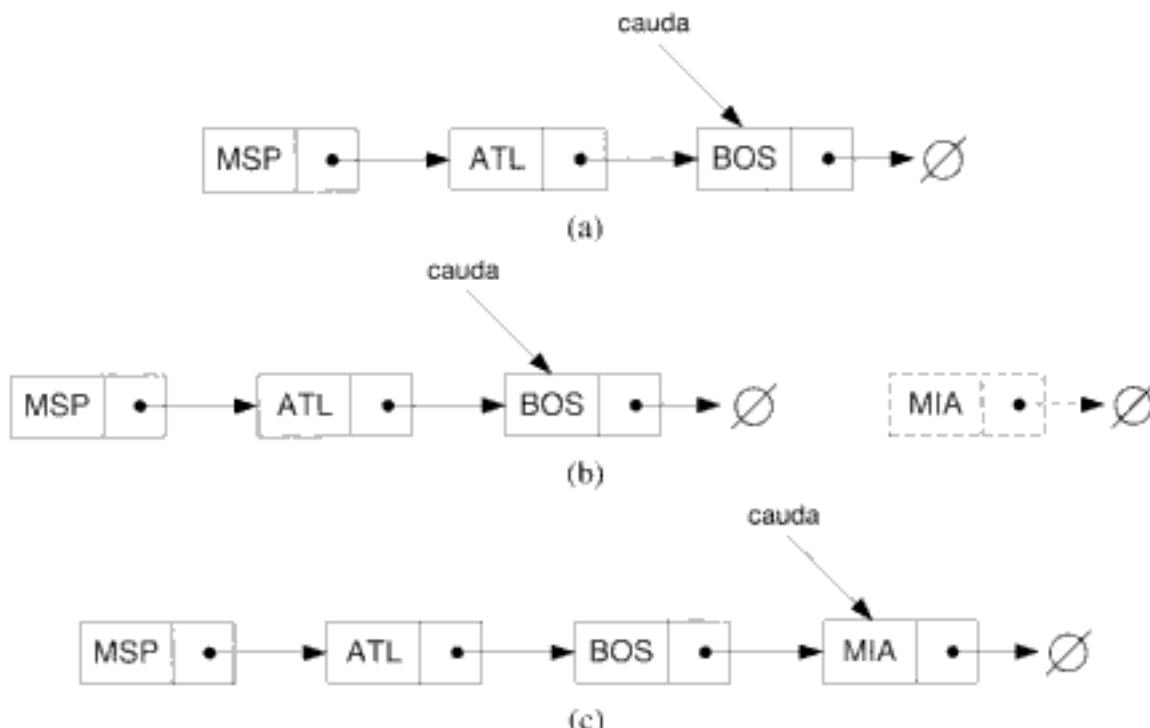
v.setNext(head)          { faz v apontar para o nodo cabeça antigo }
head ← v                { faz a variável head apontar para o nodo novo }
size ← size + 1          { incrementa o nodo contador }

```

**Trecho de código 3.14** Inserção de um nodo novo *v* no início de uma lista simplesmente encadeada. Observa-se que este método funciona mesmo que a lista esteja vazia. Observa-se também que se definiu o ponteiro *next* do novo nodo *v* *antes* de fazer a variável *head* apontar para *v*.

Inserindo um elemento na cauda de uma lista simplesmente encadeada

Pode-se inserir um elemento na cauda de uma lista simplesmente encadeada com facilidade desde que se mantenha uma referência para o nodo cauda, como mostrado na Figura 3.12. Neste caso, cria-se um nodo novo, atribui-se `null` para sua referência `next`, faz-se com que a referência `next` do nodo cauda aponte para este novo objeto, e que a referência para a cauda propriamente dita, `tail`, aponte para o nodo novo. Os detalhes aparecem no Trecho de código 3.15.



**Figura 3.12** Inserção na cauda de uma lista simplesmente encadeada: (a) antes da inserção; (b) criação do nodo novo; (c) depois da inserção. Observe que se define o valor da referência `next` de `tail` em (b), antes de fazer a variável `tail` apontar para o nodo novo em (c).

#### Algoritmo addLast(v)

```

v.setNext(null)           { faz com que o nodo novo, v, aponte para null}
tail.setNext(v)           { faz com que o nodo cauda antigo aponte para o nodo novo}
tail ← v                 { faz a variável tail apontar para o nodo novo}
size ← size + 1          { incrementa o contador de nodos}

```

**Trecho de código 3.15** Inserção de um nodo novo no final de uma lista simplesmente encadeada. Este método também funciona se a lista está vazia. Observe que o valor do ponteiro `next` do nodo cauda antigo é alterado antes que de se fazer a variável `tail` apontar para o nodo novo.

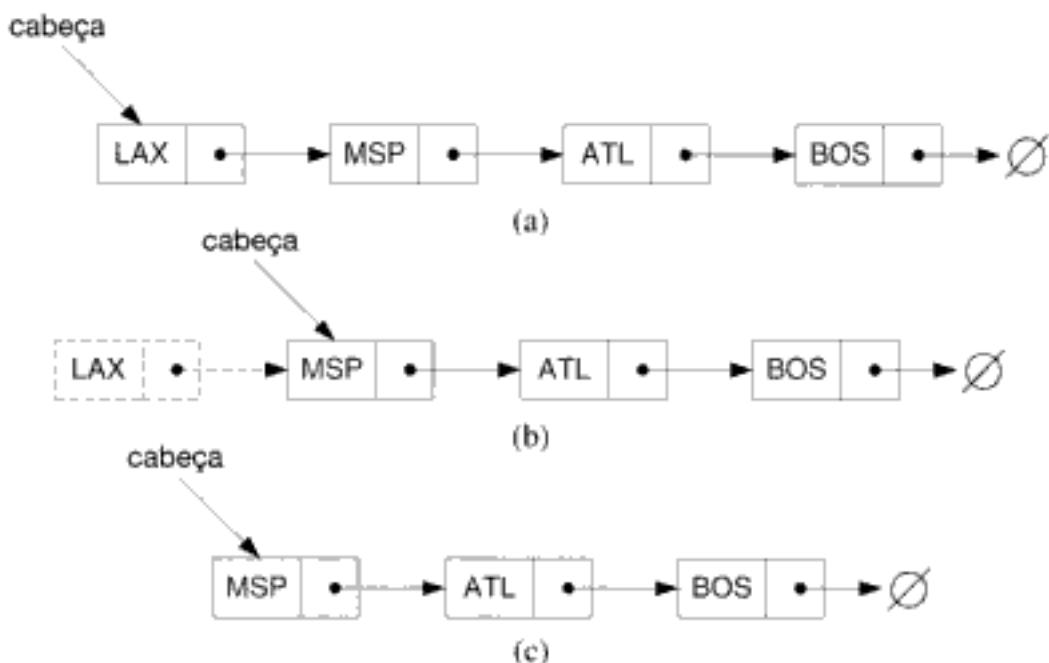
### 3.2.2 Removendo um elemento em uma lista simplesmente encadeada

A operação inversa da inserção de um novo elemento na cabeça de uma lista encadeada é a remoção de um elemento da cabeça da lista. Esta operação é demonstrada na Figura 3.13 e detalhada no Trecho de código 3.16.

#### Algoritmo removeFirst():

```
se head = null então
```

Indica um erro: a lista está vazia.



**Figura 3.13** Remoção de um elemento da cabeça de uma lista simplesmente encadeada: (a) antes da remoção; (b) “desconectando” o antigo nodo novo; (c) após a remoção.

```

t ← head
head ← head.getNext()           { faz head apontar para o próximo nodo (ou null)}
t.setNext(null)                  { atribui null para o ponteiro next do nodo removido}
size ← size - 1                 { decrementa o contador de nodos}

```

**Trecho de código 3.16** Removendo um nodo no inicio de uma lista simplesmente encadeada.

Infelizmente, não é possível deletar o nodo da cauda da lista com a mesma facilidade. Mesmo se houver uma referência diretamente para o último nodo da lista, é necessário acessar o nodo *antes* do último para conseguir removê-lo. Mas não se pode atingir o nodo antes da cauda seguindo as conexões a partir da cauda. A única maneira de acessar este nodo é iniciar a partir da cabeça da lista pesquisando ao longo da mesma. Mas a seqüência de saltos pelas ligações pode consumir um tempo considerável.

### 3.3 Listas duplamente encadeadas

Como visto na última seção, remover um elemento da cauda de uma lista simplesmente encadeada não é fácil. Na verdade, consome muito tempo remover qualquer nodo, exceto a cabeça em uma lista simplesmente encadeada, uma vez que não existe uma forma rápida de acessar o nodo na frente daquele que se quer remover. Na verdade, existem várias aplicações nas quais é necessário acessar rapidamente o nodo predecessor. Para tais aplicações, é interessante ter uma maneira de se mover em ambas as direções em uma lista encadeada.

Existe um tipo de lista encadeada que permite o deslocamento em ambas as direções – para frente e para trás – em uma lista encadeada. É uma lista **duplamente encadeada**. Tais listas permitem uma grande variedade de operações rápidas de atualização, incluindo inserções e remoções em ambas extremidades e no meio. Um nodo em uma lista duplamente encadeada armazena duas referências – uma ligação *next*, que aponta para o próximo nodo da lista, e uma ligação *prev*, que aponta para o nodo anterior.

O Trecho de código 3.17 apresenta uma implementação Java de um nodo de uma lista duplamente encadeada na qual se assume que os elementos são strings. No Capítulo 5, discute-se como definir nodos para tipos arbitrários de elementos.

```
/** Nodo de uma lista duplamente encadeada de strings */
public class DNode {
    protected String element; // String armazenada pelo nodo
    protected DNode next, prev; // Ponteiros para o nodo seguinte e o anterior
    /** Construtor que cria um nodo com os campos fornecidos */
    public DNode(String e, DNode p, DNode n) {
        element = e;
        prev = p;
        next = n;
    }
    /** Retorna o elemento deste nodo */
    public String getElement() { return element; }
    /** Retorna o nodo anterior a este */
    public DNode getPrev() { return prev; }
    /** Retorna o nodo seguinte a este */
    public DNode getNext() { return next; }
    /** Atribui o elemento deste nodo */
    public void setElement(String newElem) { element = newElem; }
    /** Atribui o nodo anterior deste nodo */
    public void setPrev(DNode newPrev) { prev = newPrev; }
    /** Atribui o nodo seguinte a este nodo */
    public void setNext(DNode newNext) { next = newNext; }
}
```

**Trecho de código 3.17** Classe Java DNode representando um nodo de uma lista duplamente encadeada que armazena uma string.

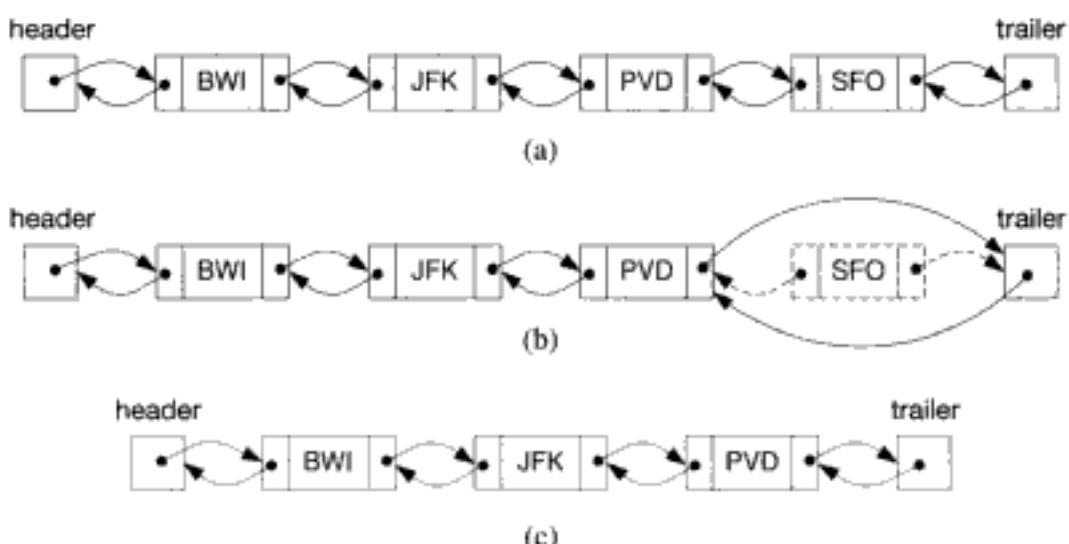
### Sentinelas da cabeça e da cauda

Para simplificar a programação, é conveniente acrescentar nodos especiais em ambas as extremidades de uma lista duplamente encadeada: um nodo *cabeçalho* (*header*) antes da cabeça da lista e um nodo *final* (*trailer*) após a cauda da lista. Estes nodos “falsos” ou *sentinelas* não armazenam nenhum elemento. O cabeçalho tem uma referência next válida e uma referência prev nula, enquanto que o final tem uma referência prev válida e uma referência next nula. Uma lista duplamente encadeada com estas sentinelas é apresentada na Figura 3.14. Observa-se que o objeto lista encadeada terá simplesmente de armazenar referências para estas duas sentinelas e um contador size para manter o número de elementos na lista (sem contar os sentinelas).



**Figura 3.14** Uma lista duplamente encadeada com sentinelas, header e trailer, marcando as extremidades da lista. Uma lista vazia terá estas sentinelas apontando uma para outra. Não se desenha o ponteiro prev nulo do header nem o ponteiro next nulo do final.

Inserir ou remover elementos em qualquer extremidade de uma lista duplamente encadeada é fácil de fazer. Na verdade, a ligação prev elimina a necessidade de percorrer a lista para obter o nodo que antecede a cauda. A Figura 3.15 mostra a remoção na cauda de uma lista duplamente encadeada e os detalhes desta operação no Trecho de código 3.18.



**Figura 3.15** Remoção de um nodo na extremidade de uma lista duplamente encadeada com sentinelas para o cabeçalho e o final: (a) antes de deletar a cauda; (b) deletando a cauda; (c) após a deleção.

**Algoritmo removeLast( ):**

```

se size = 0 então
    Indica um erro: a lista está vazia
    v ← trailer.getPrev()           {último nodo}
    u ← v.getPrev()                {nodo antes do último nodo}
    trailer.setPrev(u)
    u.setNext(trailer)
    v.setPrev(null)
    v.setNext(null)
    size = size - 1
  
```

**Trecho de código 3.18** Remoção do último nodo de uma lista duplamente encadeada. A variável `size` mantém a quantidade de elementos na lista. Observa-se que este método também funciona se a lista tiver tamanho 1.

Com a mesma facilidade pode-se inserir um novo elemento no início de uma lista simplesmente encadeada, como pode ser visto na Figura 3.16 e no Trecho de código 3.19.

**Algoritmo addFirst(v):**

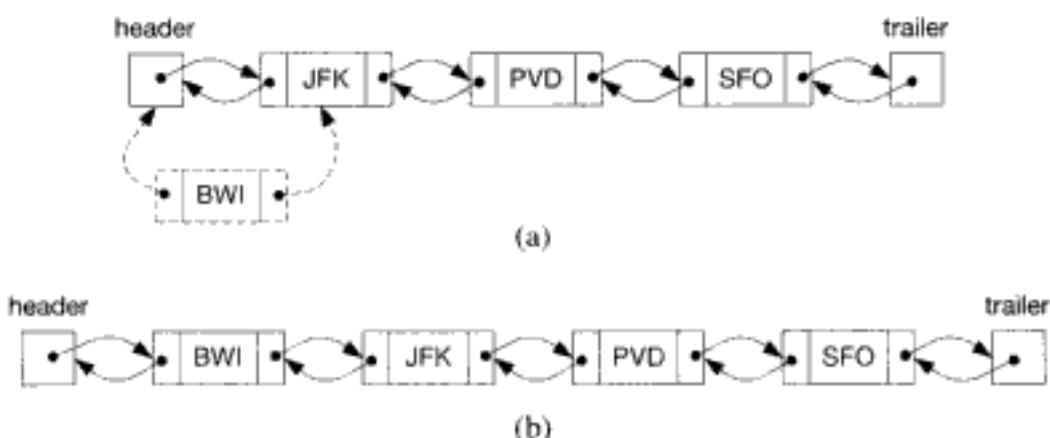
```

w ← header.getNext()           {primeiro nodo}
v.setNext(w)
v.setPrev(header)
w.setPrev(v)
header.setNext(v)
size = size + 1
  
```

**Trecho de código 3.19** Inserção de um nodo novo `v` no início de uma lista duplamente encadeada. A variável `size` mantém a quantidade de elementos na lista. Observa-se que este método também trabalha sobre uma lista vazia.

### 3.3.1 Inserção no meio de uma lista duplamente encadeada

Listas duplamente encadeadas são úteis para mais do que inserir e remover elementos no inicio e no fim da lista. Elas também são convenientes para manter uma lista de elementos e



**Figura 3.16** Acrescentando um elemento no início: (a) durante; (b) depois.

permitir inserções no meio da lista. Dado um nodo  $v$  de uma lista duplamente encadeada (que até pode ser o nodo cabeça, mas não a cauda), pode-se facilmente inserir um novo nodo  $z$  imediatamente após  $v$ . Mais especificamente, considere  $w$  como o nodo que segue  $v$ . Executam-se os seguintes passos:

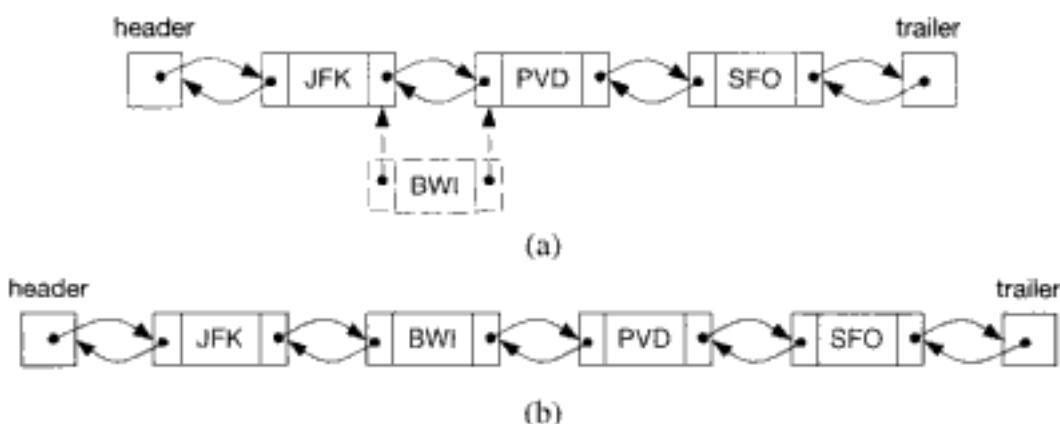
1. faça a ligação  $\text{prev}$  de  $z$  se referir a  $v$
2. faça a ligação  $\text{next}$  de  $z$  se referir a  $w$
3. faça a ligação  $\text{prev}$  de  $w$  se referir a  $z$
4. faça a ligação  $\text{next}$  de  $v$  se referir a  $z$

Este método é apresentado em detalhes no Trecho de código 3.20 e é demonstrado na Figura 3.17. Lembrando do uso das sentinelas cabeça e final, observe que este algoritmo funciona mesmo que  $v$  seja o nodo cauda (o nodo que antecede o final).

**Algoritmo addAfter( $v,z$ ):**

```
w ← v.getNext()      {nodo que segue v}
z.setPrev(v)          {conecta z a seu predecessor, v}
z.setNext(w)          {conecta z a seu sucessor, w}
w.setPrev(z)          {conecta w a seu novo predecessor, z}
v.setNext(z)          {conecta v a seu novo sucessor, z}
size ← size + 1
```

**Trecho de código 3.20** Inserção de um novo nodo  $z$  depois de um nodo  $v$  em uma lista duplamente encadeada.



**Figura 3.17** Acrescentando um nodo novo depois do nodo que armazena JFK: (a) criando um nodo novo com o elemento BWI e conectando o mesmo; (b) depois da inserção.

### 3.3.2 Remoção do meio de uma lista duplamente encadeada

Da mesma forma, é fácil remover um nodo  $v$  do meio de uma lista duplamente encadeada. Acessam-se os nodos  $u$  e  $w$  em ambos os lados de  $v$  usando os métodos `getPrev` e `getNext` de  $v$  (estes nodos devem existir uma vez que se está usando sentinelas). Para remover o nodo  $v$ , basta fazer  $u$  e  $w$  apontarem um para o outro em vez de apontarem para  $v$ . Esta operação é conhecida como **desconexão** de  $v$ . Atribui-se nulo para os ponteiros `next` e `prev` de  $v$  de maneira a não manter referências antigas para a lista. O algoritmo é apresentado no Trecho de código 3.21 e ilustrado na Figura 3.18.

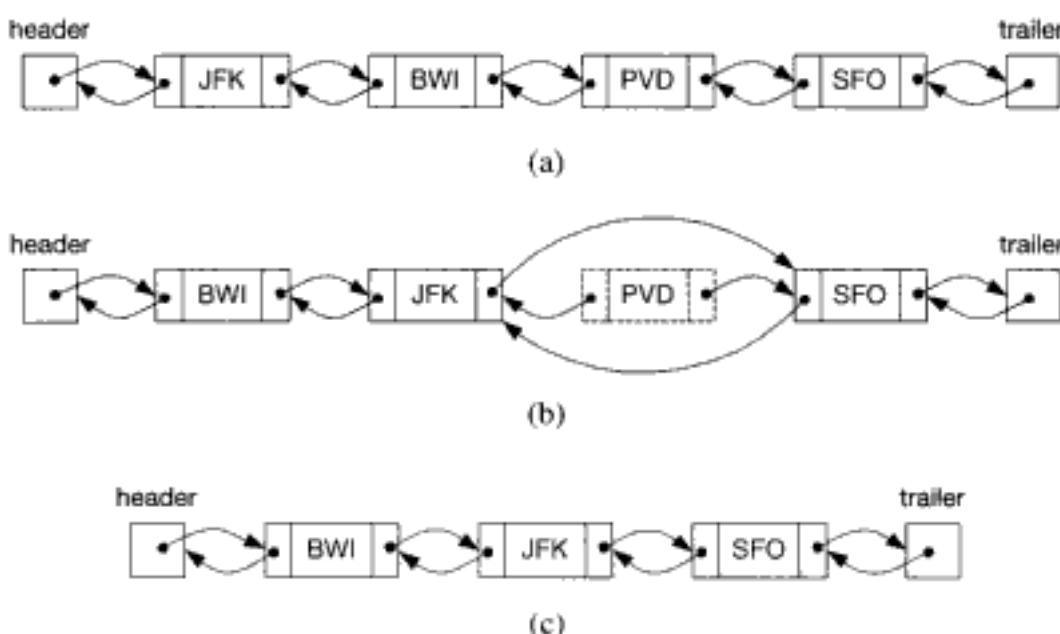
**Algoritmo 3.18:** `remove(v)`:

```

 $u \leftarrow v.getPrev()$            {nodo antes de  $v$ }
 $w \leftarrow v.getNext()$           {nodo depois de  $v$ }
 $w.setPrev(u)$                  {desconectando  $v$ }
 $u.setNext(w)$ 
 $v.setPrev(null)$               {anulando os campos de  $v$ }
 $v.setNext(null)$ 
 $size \leftarrow size - 1$           {decrementando o contador de nodos}

```

**Trecho de código 3.21** Remoção do nodo  $v$  de uma lista duplamente encadeada. Este método funciona mesmo que  $v$  seja o primeiro, o último ou um nodo não-sentinela.



**Figura 3.18** Removendo o nodo que armazena PDV: (a) antes da remoção; (b) desconectando o nodo antigo; (c) depois da remoção (coleta de lixo).

### 3.3.3 Implementação de uma lista duplamente encadeada

Nos Trechos de código 3.22-3.24, apresenta-se a implementação de uma lista duplamente encadeada com nodos que armazenam strings.

```

/* Lista duplamente encadeada com nodos do tipo DNode que armazenam strings */
public class DList {
    protected int size;                  // quantidade de elementos
    protected DNode header, trailer;    // sentinelas
    /* Construtor que cria uma lista vazia */
    public DList() {


```

```

size = 0;
header = new DNode(null, null, null); // cria o cabeçalho
trailer = new DNode(null, header, null); // cria o final
header.setNext(trailer); // faz o cabeçalho e o final apontarem um para o outro
}
/** Retorna o número de elementos na lista */
public int size() { return size; }
/** Informa se a lista está vazia */
public boolean isEmpty() { return (size == 0); }
/** Retorna o primeiro nodo da lista */
public DNode getFirst() throws IllegalStateException {
    if (isEmpty()) throw new IllegalStateException("List is empty");
    return header.getNext();
}
/** Retorna o último nodo da lista */
public DNode getLast() throws IllegalStateException {
{
    if (isEmpty()) throw new IllegalStateException("List is empty");
    return trailer.getPrev();
}
/** Retorna o nodo que antecede um dado nodo v. Gera erro se v é o cabeçalho */
public DNode getPrev(DNode v) throws IllegalArgumentException {
    if (v == header) throw new IllegalArgumentException
        ("Cannot move back past the header of the list");
    return v.getPrev();
}
/** Retorna o nodo que segue um dado nodo v. Gera erro se v é o final */
public DNode getNext(DNode v) throws IllegalArgumentException {
    if (v == trailer) throw new IllegalArgumentException
        ("Cannot move forward past the trailer of the list");
    return v.getNext();
}
}

```

**Trecho de código 3.22** Classe Java DList que implementa uma lista duplamente encadeada cujos nodos são objetos da classe DNode (ver Trecho de código 3.17) que armazenam strings (continua no Trecho de código 3.23).

```

/** Insere um dado nodo z antes de um dado nodo v. Gera um erro se v é o cabeçalho */
public void addBefore(DNode v, DNode z) throws IllegalArgumentException {
    DNode u = getPrev(v); // Deve lançar uma IllegalArgumentException
    z.setPrev(u);
    z.setNext(v);
    v.setPrev(z);
    u.setNext(z);
    size++;
}
/** Insere um dado nodo z depois de uma dado nodo v. Gera um erro se v é o final */
public void addAfter(DNode v, DNode z) {
    DNode w = getNext(v); // Deve lançar uma IllegalArgumentException
    z.setPrev(v);
    z.setNext(w);
    w.setPrev(z);
    v.setNext(z);
    size++;
}

```

```

}

/** Insere o nodo fornecido no início da lista */
public void addFirst(DNode v) {
    addAfter(header, v);
}

/** Insere o nodo fornecido no fim da lista */
public void addLast(DNode v) {
    addBefore(trailer, v);
}

/** Remove um dado nodo v da lista. Gera um erro se v é o cabeçalho ou o final */
public void remove(DNode v) {
    DNode u = getPrev(v);      // Deve lançar uma IllegalArgumentException
    DNode w = getNext(v);     // Deve lançar uma IllegalArgumentException
    // Desconecta o nodo da lista
    w.setPrev(u);
    u.setNext(w);
    v.setPrev(null);
    v.setNext(null);
    size--;
}

```

**Trecho de código 3.23** Classe Java DList que implementa uma lista duplamente encadeada (continua no Trecho de código 3.24).

```

/** Indica se o nodo indicado possui um antecessor */
public boolean hasPrev(DNode v) { return v != header; }
/** Indica se o nodo indicado possui um sucessor */
public boolean hasNext(DNode v) { return v != trailer; }
/** Retorna uma representação string da lista */
public String toString() {
    String s = "[";
    DNode v = header.getNext();
    while (v != trailer) {
        s += v.getElement();
        v = v.getNext();
        if (v != trailer)
            s += ",";
    }
    s += "]";
    return s;
}

```

**Trecho de código 3.24** Classe de uma lista duplamente encadeada (continuação do Trecho de código 3.23).

Podem-se fazer as seguintes observações a cerca da classe DList.

- Objetos da classe DNode, que armazenam elementos string, são usados para todos os nodos da lista, incluindo os sentinelas cabeçalho e final.
- A classe DList pode ser usada apenas para uma lista duplamente encadeada de strings. Para construir uma lista encadeada para outros tipos de objetos, é necessário usar uma declaração genérica que será discutida no capítulo 5.
- Os métodos `getFirst` e `getLast` provêm acesso direto ao primeiro e último nodos da lista.
- Os métodos `getPrev` e `getNext` permitem percorrer a lista.

- Os métodos `getPrev` e `getNext` detectam os limites da lista.
- Os métodos `addFirst` e `addLast` acrescentam nodos no início e no fim da lista.
- Os métodos `addBefore` e `addAfter` acrescentam um nodo novo antes ou depois de um nodo existente.
- A existência de um único método de remoção, `remove`, não chega a ser uma restrição uma vez que é possível remover do início ou do fim de uma lista encadeada  $L$  executando  $L.remove(L.getFirst())$  ou  $L.remove(L.getLast())$ , respectivamente.
- O método `toString` que converte a lista inteira em uma string é útil para propósitos de depuração e teste.

### 3.4 Listas encadeadas circulares e ordenação de listas encadeadas

Nesta seção, serão estudadas algumas aplicações e extensões de listas encadeadas.

#### 3.4.1 Listas encadeadas circulares e a brincadeira do “Pato, Pato, Ganso”

A brincadeira de criança “**Pato, Pato, Ganso**”\* existe em muitas culturas. As crianças de Minnesota praticam uma versão chamada “Duck, Duck, Grey Duck”\*\*\*, mas não perguntam o porquê. Em Indiana, este jogo é chamado “The Mosh Pot”. As crianças da República Checa e de Gana jogam versões cantadas do jogo, conhecidas, respectivamente, como “Pesek” e “Antoakyire”. Uma variação das listas encadeadas, chamada de lista encadeada circular, é usada em várias aplicações que envolvem jogos de roda tais como “Pato, Pato, Ganso”. Este tipo de lista e as aplicações dos jogos de roda serão analisados a seguir.

Uma **lista encadeada circular** tem o mesmo tipo de nodos que uma lista encadeada simples. Isto é, cada nodo em uma lista encadeada circular tem um ponteiro para o próximo nodo e uma referência para um elemento. Entretanto, não existe cabeça ou cauda em uma lista circular. Em vez do último nodo de uma lista circular apontar para `null`, ele aponta para o primeiro nodo. Assim, não existe nodo inicial ou final. Se forem percorridos os nodos de uma lista circular a partir de qualquer nodo seguindo os ponteiros `next`, serão percorridos todos os nodos.

Mesmo que uma lista circular não tenha início ou fim, sempre será necessário que algum nodo seja marcado de forma especial, sendo chamado de **cursor**. O nodo cursor serve como ponto de partida sempre que for necessário percorrer a lista circular. E se for possível lembrar do ponto de partida, então é possível saber quando se completou a volta – o caminhamento sobre uma lista encadeada circular está completo quando se retorna ao nodo marcado como cursor quando se começou.

Pode-se definir alguns métodos de atualização simples para uma lista encadeada circular:

`add(v)`: Insere um nodo novo,  $v$ , imediatamente após o cursor; se a lista está vazia, então  $v$  torna-se o cursor e seu ponteiro `next` aponta para si mesmo.

`remove()`: remove e retorna o nodo  $v$  que se encontra imediatamente após o cursor (não o cursor propriamente dito a menos que ele seja o único nodo); se a lista ficar vazia, o cursor é definido como `null`.

`advance()`: avança o cursor para o próximo nodo da lista.

\* N. de T. Em inglês, “Duck, Duck, Goose”.

\*\* N. de T. “Pato, Pato, Pato Cinza”.

No Trecho de código 3.25, apresenta-se uma implementação Java para a lista encadeada circular que usa a classe `Nodo` do Trecho de código 3.12 e inclui um método `toString` para produzir uma representação da lista.

```
/** Lista encadeada circular com nodos do tipo Node que armazenam strings */
public class CircleList {
    protected Node cursor;          // o cursor corrente
    protected int size;             // a quantidade de nodos da lista
    /** Construtor que cria uma lista vazia */
    public CircleList() { cursor = null; size = 0; }
    /** Retorna o tamanho corrente */
    public int size() { return size; }
    /** Retorna o cursor */
    public Node getCursor() { return cursor; }
    /** Move o cursor adiante */
    public void advance() { cursor = cursor.getNext(); }
    /** Acrescenta um nodo depois do cursor */
    public void add(Node newNode) {
        if (cursor == null) {           // list is empty
            newNode.setNext(newNode);
            cursor = newNode;
        }
        else {
            newNode.setNext(cursor.getNext());
            cursor.setNext(newNode);
        }
        size++;
    }
    /** Remove o nodo que segue o cursor */
    public Node remove() {
        Node oldNode = cursor.getNext(); // o nodo sendo removido
        if (oldNode == cursor)
            cursor = null; // a lista se torna vazia
        else {
            cursor.setNext(oldNode.getNext()); // desconecta o nodo antigo
            oldNode.setNext(null);
        }
        size--;
        return oldNode;
    }
    /** Retorna uma representação string da lista, iniciando pelo cursor */
    public String toString() {
        if (cursor == null) return "[ ] ";
        String s = " [ . . . " + cursor.getElement();
        Node oldCursor = cursor;
        for (advance(); oldCursor != cursor; advance())
            s += ", " + cursor.getElement();
        return s + " . . . ] ";
    }
}
```

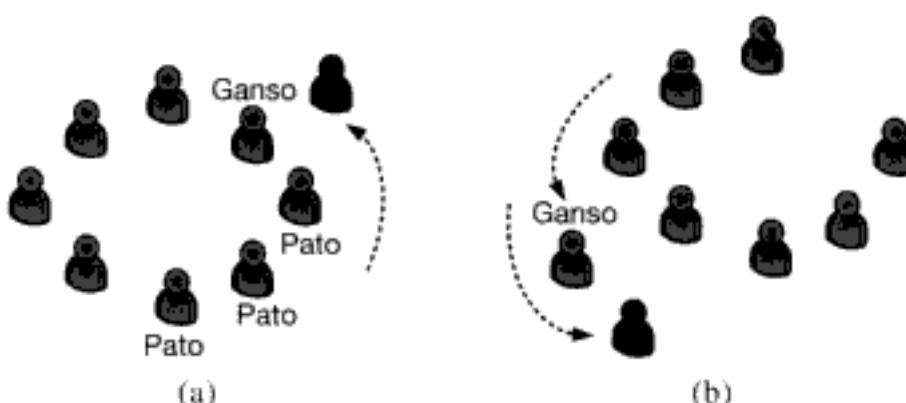
**Trecho de código 3.25** Uma lista encadeada circular com nodos simples.

### Algumas observações sobre a classe CircleList

Existem algumas observações que podem ser feitas a respeito da classe `CircleList`. É um programa simples que pode fornecer enorme funcionalidade para simular jogos de roda tais como "Pato, Pato, Ganso", como será visto. Deve-se observar, entretanto, que não é um programa robusto. Em especial, se a lista circular está vazia, então chamar `advance` ou `remove` sobre a lista irá gerar uma exceção. (Qual?) O exercício R-3.5 lida com este comportamento gerador de exceção e com maneiras de lidar melhor com esta condição de lista vazia.

### Pato, Pato, Ganso

No jogo de "Pato, Pato, Ganso", um grupo de crianças senta em círculo. Uma delas é eleita para ser o "pegador" e caminhar por fora do círculo. O pegador bate na cabeça de cada criança dizendo "pato" até que identifica uma delas como sendo "ganso". Neste ponto, gera-se uma confusão com o "ganso" e o pegador correndo ao redor do círculo. Quem retornar ao lugar do "ganso" primeiro, permanece no círculo. O perdedor da corrida será o pegador na próxima rodada. O jogo continua assim até que as crianças se aborreçam ou um adulto diga que acabou o tempo (ver a Figura 3.19).



**Figura 3.19** O jogo "Pato, Pato, Ganso": (a) selecionando o "ganso"; (b) corrida para o lugar do "ganso" entre o "ganso" e o pegador.

A simulação deste jogo é uma aplicação ideal para listas encadeadas circulares. As crianças podem representar os nós da lista. O pegador pode ser identificado como a pessoa sentada após o cursor, e pode ser removida da lista para simular a marcha ao redor. Avança-se o cursor para cada "pato" que o pegador identifica, o que pode ser simulado com uma decisão aleatória. Uma vez que um "ganso" é identificado, pode-se remover este nó da lista, fazer um sorteio aleatório para decidir quem irá ganhar a corrida e inserir o vencedor de volta na lista. Então, avança-se o cursor e insere-se o pegador de volta, repetindo o processo (ou encerra-se, se esta foi a última rodada).

### Usando uma lista encadeada circular para simular o Pato, Pato, Ganso

O Trecho de código 3.26 apresenta o código Java que simula o jogo do Pato, Pato, Ganso.

```
/** Simulação do Pato, Pato, Ganso usando uma lista encadeada circular */
public static void main(String[] args) {
    CircleList C = new CircleList();
    int N = 3; // Quantidade de iterações do jogo
    Node it; // jogador que é o "pegador"
    Node goose; // ganso
    Random rand = new Random();
```

```

rand.setSeed(System.currentTimeMillis()); // usa o tempo corrente como semente
// Os jogadores
String[] names = {"Bob", "Jen", "Pam", "Tom", "Ron", "Vic", "Sue", "Joe"};
for (int i = 0; i < names.length; i++) {
    C.add(new Node(names[i], null));
    C.advance();
}
for (int i = 0; i < N; i++) {           // joga pato, pato, ganso N vezes
    System.out.println("Playing Duck, Duck, Goose for " + C.toString());
    it = C.remove();
    System.out.print(it.getElement() + " is it.");
    while (rand.nextBoolean() || rand.nextBoolean()) { // anda ao redor do círculo
        C.advance(); // avança com probabilidade de 1/4
        System.out.print(C.getCursor().getElement() + " is a duck.");
    }
    goose = C.remove();
    System.out.println(goose.getElement() + " is the goose!");
    if (rand.nextBoolean()) {
        System.out.println("The goose won!");
        C.add(goose); // coloca o ganso de volta no seu lugar antigo
        C.advance(); // agora o cursor é o ganso
        C.add(it); // o pegador será o mesmo na próxima rodada
    }
    else {
        System.out.println("The goose lost!");
        C.add(it); // coloca o pegador no lugar do ganso
        C.advance(); // agora o cursor está no pegador
        C.add(goose); // o ganso será o pegador na próxima rodada
    }
}
System.out.println("Final circle is " + C.toString());
}

```

**Trecho de código 3.26** Método principal de uma programa que usa uma lista encadeada circular para simular o jogo de criança Pato, Pato, Ganso.

### Um exemplo de saída

A Figura 3.20 mostra um exemplo de saída resultante de uma execução do programa do Pato, Pato, Ganso.

Observa-se que cada iteração nesta execução particular do programa produz um resultado diferente, em virtude das configurações iniciais diferentes e do uso de escolhas aleatórias para identificar os patos e gansos. Da mesma forma, se o “pato” ou o “ganso” ganha a corrida também varia, dependendo de escolhas aleatórias. Esta execução mostra uma situação em que a próxima criança após o pegador é imediatamente identificada como “ganso”, bem como uma outra em que o pegador caminha ao redor de todo o grupo de crianças antes de identificar o “ganso”. Tais situações demonstram a utilidade de uma lista encadeada circular na simulação de jogos de roda tais como o Pato, Pato, Ganso.

---

### 3.4.2 Ordenando uma lista encadeada

No Trecho de código 3.27, é apresentado o algoritmo *inserção ordenada* (Seção 3.1.2) para uma lista duplamente encadeada. Uma implementação Java é apresentada no Trecho de código 3.28.

Playing Duck, Duck, Goose for [...Joe, Bob, Jen, Pam, Tom, Ron, Vic, Sue...]  
 Bob is it.  
 Jen is a duck.  
 Pam is a duck.  
 Tom is a duck.  
 Ron is the goose!  
 The goose won!  
 Playing Duck, Duck, Goose for [...Ron, Bob, Vic, Sue, Joe, Jen, Pam, Tom...]  
 Bob is it.  
 Vic is the goose!  
 The goose won!  
 Playing Duck, Duck, Goose for [...Vic, Bob, Sue, Joe, Jen, Pam, Tom, Ron...]  
 Bob is it.  
 Sue is a duck.  
 Joe is a duck.  
 Jen is a duck.  
 Pam is a duck.  
 Tom is a duck.  
 Ron is a duck.  
 Vic is a duck.  
 Sue is the goose!  
 The goose lost!  
 Final circle is [...Bob, Sue, Joe, Jen, Pam, Tom, Ron, Vic...]

**Figura 3.20** Exemplo de saída do programa Pato, Pato, Ganso.

**Algoritmo** InsertionSort(*L*):

**Entrada:** uma lista duplamente encadeada *L* com elementos comparáveis  
**Saída:** a lista *L* com os elementos reorganizados em ordem não decrescente  
**Se** *L.size()* <= 1 **então**  
     **return**  
     *end*  $\leftarrow L.\text{getFirst}()$   
**enquanto** *end* não for o último nodo de *L* **faça**  
         *pivot*  $\leftarrow end.\text{getNext}()$ ;  
         Remove o *pivot* de *L*  
         *ins*  $\leftarrow end$   
**enquanto** *ins* não for o cabeçalho e o elemento *ins* for maior que o *pivot* **faça**  
             *ins*  $\leftarrow ins.\text{getPrev}()$   
             Acrescenta o *pivot* após *ins* em *L*  
         **se** *ins* = *end* **então** {recém adicionou-se o *pivot* após *end*, neste caso}  
             *end*  $\leftarrow end.\text{getNext}()$

**Trecho de código 3.27** Descrição em pseudocódigo de alto nível do algoritmo de inserção ordenada sobre uma lista duplamente encadeada.

```
/** Inserção ordenada sobre uma lista duplamente encadeada da classe DList */
public static void sort(DList L) {
    se (L.size() <= 1) retorne; // L já está ordenado neste caso
    DNode pivot; // nodo pivô
    DNode ins; // ponto de inserção
    DNode end = L.getFirst(); // fim da execução
    enquanto (end != L.getLast()) {
        pivot = end.getNext(); // obtém o próximo nodo pivô
```

```

L.remove(pivot);           // remove o mesmo
ins = end;                 // inicia a pesquisa pelo fim da ordenação
while (L.hasPrev(ins) &&
      ins.getElement().compareTo(pivot.getElement()) > 0)
    ins = ins.getPrev();   // move para a esquerda
L.addAfter(ins,pivot);    // coloca o pivô de volta após o ponto de inserção
if (ins == end)            // acrescenta o pivô no final, neste caso
    end = end.getNext();  // incrementa o indicador de fim
}
}

```

**Trecho de código 3.28** Implementação Java do algoritmo de inserção ordenada sobre uma lista duplamente encadeada representada pela classe DList (ver o Trecho de código 3.22-3.24).

### 3.5 Recursão

Já foi visto que repetições podem ser obtidas escrevendo-se laços, tais como laços **for** ou laços **while**. Outra forma de se obter repetição é por meio da **recursão**, que ocorre quando uma função chama a si mesma. Já foram vistos exemplos de métodos que chamam outros métodos, de maneira que não deve ser surpresa que a maioria das linguagens de programação modernas, incluindo Java, permite que um método chame a si mesmo. Nesta seção, será visto por que esta capacidade provê uma alternativa elegante e poderosa para executar tarefas repetitivas.

#### A função factorial

Para demonstrar recursão, começa-se com um exemplo simples que computa o valor da **função factorial**. O factorial de um inteiro positivo  $n$ , denotado  $n!$ , é definido como sendo o produto dos inteiros de 1 até  $n$ . Se  $n = 0$ , então  $n!$  é definido como 1 por convenção. De uma maneira mais formal, para qualquer inteiro  $n \geq 0$ ,

$$n! = \begin{cases} 1 & \text{se } n = 0 \\ n \cdot (n-1) \cdot (n-2) \cdots 3 \cdot 2 \cdot 1 & \text{se } n \geq 1 \end{cases}$$

Por exemplo,  $5! = 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1 = 120$ . Para fazer a ligação com métodos mais clara, será usada a notação  $\text{factorial}(n)$  para denotar  $n!$ .

A função factorial pode ser definida de uma forma que sugere uma formulação recursiva. Para entender, observa-se que

$$\text{factorial}(5) = 5 \cdot (4 \cdot 3 \cdot 2 \cdot 1) = 5 \cdot \text{factorial}(4).$$

Assim, pode-se definir  $\text{factorial}(5)$  em termos de  $\text{factorial}(4)$ . Normalmente, para um inteiro positivo  $n$ , pode-se definir  $\text{factorial}(n)$  como sendo  $n \cdot \text{factorial}(n-1)$ . Isso leva a seguinte **definição recursiva**.

$$\text{factorial}(n) = \begin{cases} 1 & \text{se } n = 0 \\ n \cdot \text{factorial}(n-1) & \text{se } n \geq 1 \end{cases}$$

Esta definição é típica de muitas definições recursivas. Primeiro, ela contém um ou mais **casos base**, que são definidos de forma não-recursiva em termos de quantidades fixas. Neste caso,  $n = 0$  é o caso base. Ela também contém um ou mais **casos recursivos**, que são definidos apelando para a definição da sua função. Observa-se que esta definição não é circular porque cada vez que a função é chamada seu argumento é diminuído de um.

### Implementação recursiva da função factorial

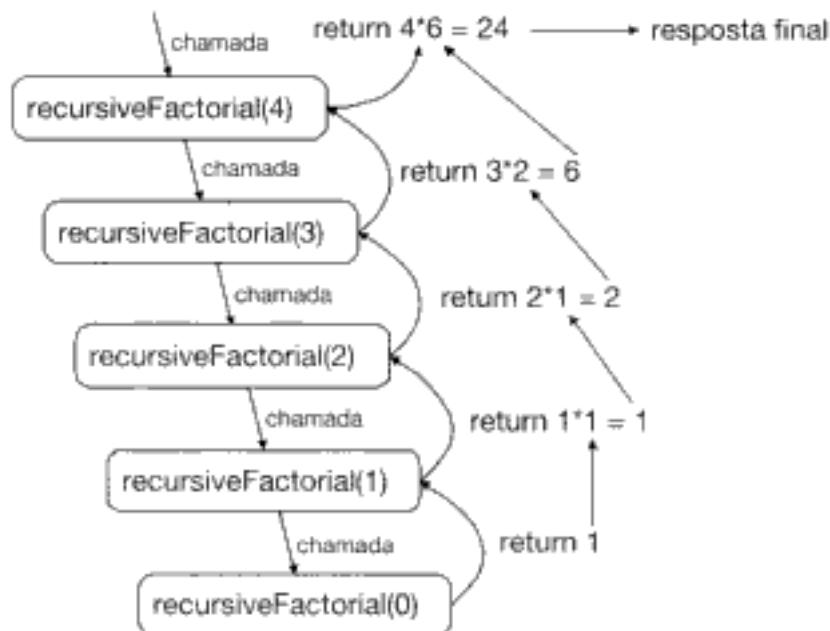
Considere-se a implementação Java da função factorial apresentada no Trecho de código 3.29 sob o nome de `recursiveFactorial()`. Observa-se que nenhum laço é necessário neste caso. As repetidas invocações recursivas da função substituem o laço.

```
public static int recursiveFactorial(int n) {      // função factorial recursiva
    if (n == 0) return 1;                          // caso base
    else return n * recursiveFactorial(n - 1);     // caso recursivo
}
```

**Trecho de código 3.29** Implementação recursiva da função factorial.

Pode-se ilustrar a execução da definição recursiva de uma função por meio de um **rastreamento recursivo**. Cada entrada do rastreamento corresponde a uma chamada recursiva. Cada nova chamada recursiva é indicada por uma seta apontando a nova função ativada. Quando a função retorna, uma seta indicando este retorno é desenhada, e o valor de retorno é indicado na mesma. A Figura 3.21 apresenta um exemplo de rastreamento.

Qual é a vantagem do uso da recursão? Embora a implementação recursiva da função factorial seja mais simples que a versão iterativa, neste caso não existe nenhuma razão determinante para se preferir a versão recursiva sobre a iterativa. Para alguns problemas, entretanto, a implementação recursiva pode ser consideravelmente mais simples e mais fácil de entender do que a versão iterativa. Segue um exemplo destes.

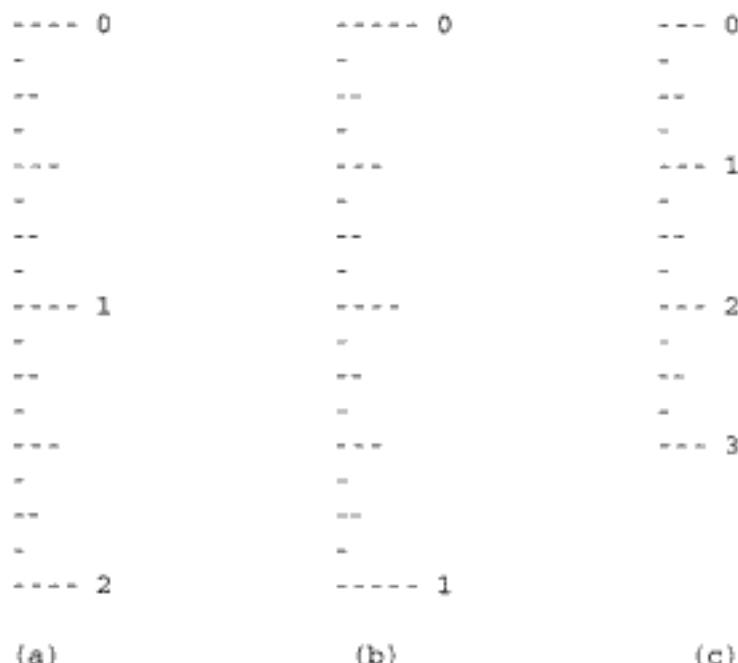


**Figura 3.21** Rastreamento recursivo para a chamada `recursiveFactorial(4)`.

### Desenhando uma régua inglesa

Como um exemplo mais complexo de recursão, pode-se desenhar as marcas de uma régua inglesa típica. A régua é dividida em intervalos de 1 polegada e cada intervalo consiste de um conjunto de **marcas** dispostas a intervalos de  $\frac{1}{2}$  polegada,  $\frac{1}{4}$  de polegada e assim por diante. A medida que o intervalo é reduzido à metade, o comprimento da marca é reduzido em uma unidade. (Ver Figura 3.22.)

Cada múltiplo de polegada tem um rótulo numérico. O maior comprimento de marca é chamado de **comprimento de marca principal**. Entretanto, não existe preocupação com as distâncias reais, sendo impressa apenas uma marca por linha.



**Figura 3.22** Três exemplos de saída da função de desenho da régua: (a) régua de 2 polegadas com a maior marca de comprimento 4; (b) régua de 1 polegada com a maior marca de comprimento 5; (c) uma régua de 3 polegadas com a maior marca de comprimento 3.

### Abordagem recursiva para o desenho da régua

A abordagem para o desenho da régua consiste em três funções. A função principal, `drawRuler()`, desenha a regra inteira. Seus argumentos são o número total de polegadas da régua, `nInches`, e o comprimento de marca principal, `majorLength`. A função utilitária, `drawOneTick()`, desenha uma única marca de um certo comprimento. Ela também pode receber um rótulo inteiro opcional que é impresso se não for negativo.

O trabalho mais interessante é feito pela função recursiva, `drawTicks()`, que desenha a seqüência de marcas de um intervalo. Seu único argumento é o tamanho da marca associada com a marca central do intervalo. Considere-se a régua de 1 polegada com marca principal de tamanho 5, apresentada na Figura 3.22(b). Ignorando as linhas que contém 0 e 1, procura-se desenhar a seqüência de marcas que ocorrem entre essas duas. A marca central ( $\frac{1}{2}$  polegada) tem tamanho 4. Nota-se que os dois padrões de marcas, acima e abaixo da marca central, são idênticos, e que cada um tem uma marca central de tamanho 3. Normalmente, um intervalo com uma marca central de tamanho  $L \geq 1$  é composto da seguinte forma:

- Um intervalo com uma marca central de comprimento  $L - 1$ .
- Uma única marca de tamanho  $L$ .
- Um intervalo com uma marca central de tamanho  $L - 1$ .

A cada chamada recursiva, o comprimento diminui de um. Quando o comprimento chega a zero, simplesmente retorna-se. Como resultado, este processo recursivo sempre terminará. Isso sugere um processo recursivo no qual o primeiro e o último passo são executados chamando-se `drawTicks( $L - 1$ )` recursivamente. O passo do meio é executado chamando a função `drawOneTick( $L$ )`. Esta formulação recursiva é apresentada no Trecho de código 3.30. Como no exemplo do fatorial, o código tem um caso base (quando  $L = 0$ ). Nesta instância, são feitas duas chamadas recursivas para a função.

```
// desenha uma marca sem rótulo
public static void drawOneTick(int tickLength) { drawOneTick(tickLength, -1); }
// desenha uma marca
```

```

public static void drawOneTick(int tickLength, int tickLabel) {
    for (int i = 0; i < tickLength; i++)
        System.out.print(" - ");
    if (tickLabel >= 0) System.out.print(" " + tickLabel);
    System.out.print("\n");
}

public static void drawTicks(int tickLength) { // desenha marcas de um certo comprimento
    if (tickLength > 0) { // para quando o comprimento chega a 0
        drawTicks(tickLength - 1); // desenha recursivamente as marcas da esquerda
        drawOneTick(tickLength); // desenha a marca central
        drawTicks(tickLength - 1); // recursivamente desenha as marcas da direita
    }
}

public static void drawRuler(int nInches, int majorLength) { // desenha a régua
    drawOneTick(majorLength, 0); // desenha a marca 0 e seu rótulo
    for (int i = 1; i <= nInches; i++) {
        drawTicks(majorLength - 1); // desenha as marcas para esta polegada
        drawOneTick(majorLength, i); // desenha a marca i e seu rótulo
    }
}

```

**Trecho de código 3.30** Uma implementação recursiva de uma função que desenha uma régua.

### Ilustrando o desenho da régua usando rastreamento recursivo

A execução recursiva da função recursiva `drawTicks`, definida anteriormente, pode ser visualizada usando-se rastreamento recursivo.

Entretanto, o rastreamento para `drawTicks` é mais complexo que no caso do exemplo do fatorial, porque cada instância faz duas chamadas recursivas. Para ilustrar este fato, apresenta-se o rastreamento recursivo de uma forma que lembra o esboço de um documento. (Ver Figura 3.23.)

Ao longo deste livro, serão vistos vários outros exemplos de como a recursão pode ser usada no projeto de estruturas de dados e algoritmos.

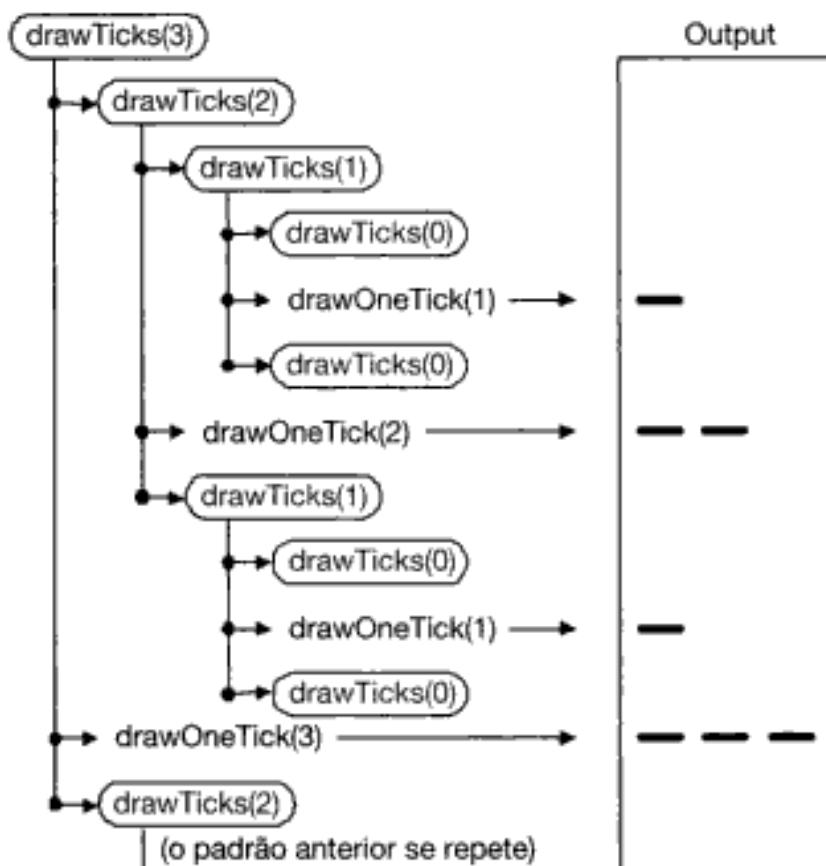
### Outras demonstrações de recursão

Como discutido anteriormente, **recursão** é um conceito que define um método que chama a si mesmo. Quando isso ocorre, denomina-se de **chamada recursiva**. Também se considera como recursivo um método *M*, se ele chama outro método que chama *M* de volta.

O maior benefício da abordagem recursiva no projeto de algoritmos é que nos permite tirar vantagem da estrutura repetitiva presente em muitos problemas. Fazendo a descrição dos algoritmos explorarem esta estrutura repetitiva de uma forma recursiva, pode-se evitar a análise de casos complexos e o uso de laços aninhados. Esta abordagem pode levar a descrições de algoritmos mais legíveis e ainda manter a eficiência.

Além disso, o uso da recursão é útil para definir objetos que tenham uma estrutura repetitiva similar, como nos exemplos que seguem.

**Exemplo 3.1** Sistemas operacionais modernos definem os diretórios do sistema de arquivos (também conhecidos como “pastas”) de uma forma recursiva. Na verdade, um sistema de arquivos consiste em um diretório de mais alto nível e o conteúdo deste diretório compreende arquivos e outros diretórios, que podem, por sua vez, conter arquivos e diretórios, e assim por diante. Os diretórios base de um sistema de arquivos contêm apenas arquivos, mas usando esta definição recursiva, o sistema operacional permite que os diretórios sejam aninhados em qualquer profundidade (enquanto houver espaço na memória).



**Figura 3.23** Rastreamento recursivo parcial para a chamada `drawTicks(3)`. O segundo padrão de chamadas para `drawTicks(2)` não é mostrado, mas é idêntico ao primeiro.

**Exemplo 3.2** Grande parte da sintaxe das linguagens de programação modernas são definidas de forma recursiva. Por exemplo, pode-se definir uma lista de argumentos em Java usando a seguinte notação:

*lista de argumentos:*

*argumento*

*lista de argumentos, argumento*

Em outras palavras, uma lista de argumentos consiste tanto em (i) um argumento ou (ii) uma lista de argumentos, seguida de uma vírgula e um argumento. Isto é, uma lista de argumentos é uma lista com os elementos separados por vírgula. Da mesma forma, expressões aritméticas podem ser definidas recursivamente em termos de primitivas (tais como variáveis e constantes) e expressões aritméticas.

**Exemplo 3.3** Existem vários exemplos de recursão na arte e na natureza. Um dos exemplos mais clássicos de recursão usado na arte são as bonecas russas Matryoshka. Cada boneca é feita de madeira sólida ou oca, contendo outra boneca Matryoshka dentro de si.

### 3.5.1 Recursão linear

A forma mais simples de recursão é a **recursão linear**, na qual um método é definido de maneira a fazer, no máximo, uma chamada recursiva cada vez que é ativado. Este tipo de recursão é útil quando se analisam os problemas de algoritmo em termos do primeiro ou do último elemento mais um conjunto restante que tem a mesma estrutura do conjunto original.

Somando os elementos de um arranjo de maneira recursiva

Supondo, por exemplo, um dado arranjo  $A$  cujos  $n$  inteiros se deseja somar. Pode-se resolver este problema usando recursão linear, observando-se que a soma de todos os  $n$  inteiros em  $A$  é igual a  $A[0]$ , se  $n = 1$ , ou a soma dos primeiros  $n - 1$  inteiros de  $A$  mais o último elemento de  $A$ . Particularmente, pode-se resolver este problema de somatório usando o algoritmo recursivo descrito no Trecho de código 3.31.

**Algoritmo** LinearSum( $A, n$ ):

**Entrada:** um arranjo inteiro  $A$  e um inteiro  $n \geq 1$ , tal que  $A$  tenha pelo menos  $n$  elementos.

**Saída:** o somatório dos primeiros  $n$  elementos de  $A$ .

**se**  $n = 1$  **então**

**retorne**  $A[0]$

**senão**

**retorne** LinearSum( $A, n - 1$ ) +  $A[n - 1]$

**Trecho de código 3.31** Somando os elementos de um arranjo usando recursão linear.

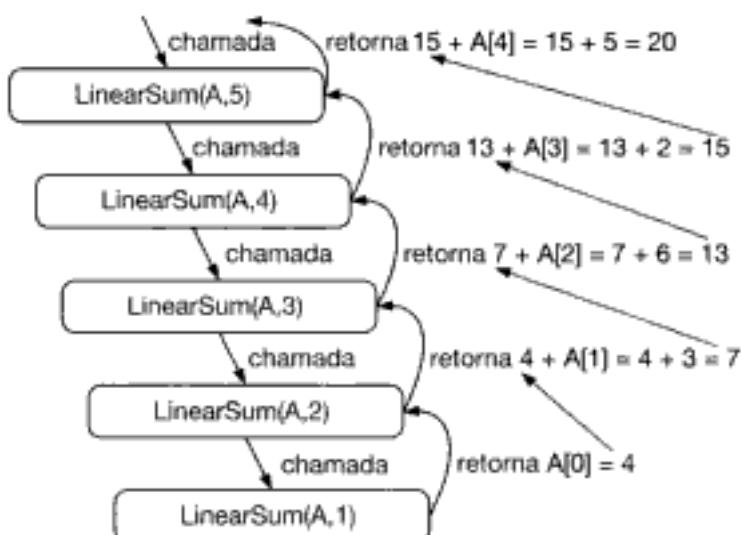
Este exemplo demonstra uma propriedade importante que todo o método recursivo deve respeitar – o método termina. Garante-se esta propriedade escrevendo uma sentença não-recursiva para o caso  $n = 1$ . Além disso, sempre se executa a chamada recursiva sobre um valor menor para o parâmetro ( $n - 1$ ) do que o fornecido ( $n$ ), de maneira que em algum ponto (na “base” da recursão), será executada a parte não-recursiva da computação (retornando  $A[0]$ ). Normalmente, um algoritmo que usa recursão linear tem a seguinte forma típica:

- **Testes para os casos base:** inicia-se testando o conjunto de casos base (pode ser apenas um). Estes casos base devem ser definidos de maneira que toda possível cadeia de recursão eventualmente atinja um deles, e o tratamento dos mesmos não pode usar recursão.
- **Recursão:** após os testes dos casos base, executa-se a chamada recursiva. Este passo recursivo deve envolver o teste que decide qual das diferentes possibilidades de chamada recursiva fazer, mas deve escolher apenas uma destas chamadas por vez que executar este passo. Além disso, deve-se definir cada chamada recursiva possível de maneira que leve em direção a um caso base.

Analizando algoritmos recursivos usando rastreamento recursivo

Pode-se analisar um algoritmo recursivo usando uma ferramenta visual conhecida como **rastreamento recursivo**. Usou-se rastreamento recursivo, por exemplo, para analisar e visualizar a função Fibonacci, da Seção 3.5, da mesma maneira que é usado com os algoritmos de ordenação recursivos das Seções 11.1 e 11.2.

Para desenhar um rastreamento recursivo, cria-se uma caixa para cada instância de método e rotula-se o mesmo com seus parâmetros. Então, visualiza-se as chamadas recursivas desenhando uma seta que parte da caixa correspondente ao método chamador em direção a caixa correspondente ao método chamado. Por exemplo, demonstra-se o rastreamento recursivo do algoritmo LinearSum do Trecho de código 3.31 na Figura 3.24. Rotula-se cada caixa neste rastreamento com os parâmetros usados para fazer a chamada. Cada vez que se faz uma chamada recursiva, desenha-se uma linha para a caixa que representa a mesma. Também é possível usar este diagrama para visualizar os sucessivos passos do algoritmo, uma vez que ele avança da chamada de  $n$  para a chamada de  $n - 1$ , para a chamada de  $n - 2$ , e assim por diante, até chegar a chamada de 1. Quando a chamada final termina, ele retorna o valor de volta para a chamada de 2, que por sua



**Figura 3.24** Rastreamento recursivo para uma execução de  $\text{LinearSum}(A, n)$  com parâmetros de entrada  $A = \{4, 3, 6, 2, 5\}$  e  $n = 5$ .

vez adiciona o valor e retorna a soma parcial para a chamada de 3, e assim por diante, até que a chamada de  $n - 1$  retorne sua soma parcial para a chamada de  $n$ .

A partir da Figura 3.24, deve ficar claro que para um arranjo de entrada de tamanho  $n$ , o algoritmo  $\text{LinearSum}$  faz  $n$  chamadas. Conseqüentemente, ele irá consumir uma quantidade de tempo que é aproximadamente proporcional a  $n$ , uma vez que o tempo gasto com a parte não-recursiva de cada chamada é constante. Além disso, pode-se perceber que a quantidade de memória usada pelo algoritmo (além do arranjo  $A$ ), também é proporcional a  $n$ , uma vez que se necessita de uma quantidade constante de memória para cada uma das  $n$  caixas do rastreamento no momento em que se faz a chamada recursiva final (para  $n = 1$ ).

### Invertendo um arranjo por recursão

Na seqüência, será analisado o problema de inverter os  $n$  elementos de um arranjo,  $A$ , de maneira que o primeiro elemento se torne o último, o segundo se torne o penúltimo, e assim por diante. Pode-se resolver esse problema usando recursão linear, na medida em que se observa que o inverso de um arranjo pode ser obtido trocando-se o primeiro e o último elemento, e então invertendo recursivamente os elementos restantes. Descrevem-se os detalhes deste algoritmo no Trecho de código 3.32, usando a convenção de que a primeira vez que este algoritmo é chamado usa-se a chamada  $\text{ReverseArray}(A, 0, n - 1)$ .

#### Algoritmo $\text{ReverseArray}(A, i, j)$ :

**Entrada:** um arranjo  $A$  e índices inteiros não negativos  $i$  e  $j$

**Saída:** os elementos de  $A$  invertidos começando no índice  $i$  e terminando em  $j$

se  $i < j$  então

Inverter  $A[i]$  e  $A[j]$

$\text{ReverseArray}(A, i + 1, j - 1)$

**retorna**

**Trecho de código 3.32** Invertendo os elementos de um arranjo usando recursão linear.

Observa-se que neste algoritmo na verdade existem dois casos base, a saber, quando  $i = j$  e quando  $i > j$ . Em qualquer um destes casos simplesmente encerra-se o algoritmo, uma vez que uma seqüência com zero elementos ou uma seqüência com um elemento é trivialmente igual a

seu inverso. Além disso, observa-se que no passo recursivo garante-se o progresso em direção a um destes dois casos base. Se  $n$  é ímpar, o caso  $i = j$  será atingido inevitavelmente, e se  $n$  for par, certamente será atingido o caso  $i > j$ . Este argumento implica que o algoritmo recursivo do Trecho de código 3.32 garantidamente termina.

### Definindo problemas de maneira a facilitar a recursão

Para projetar um algoritmo recursivo para um dado problema, é útil pensar em diferentes maneiras de subdividi-lo, definindo subproblemas que tenham a mesma estrutura geral que o original. Este processo significa que algumas vezes é necessário redefinir o problema original para facilitar a obtenção de subproblemas similares. Por exemplo, no algoritmo `ReverseArray`, foram acrescentados os parâmetros  $i$  e  $j$  de maneira que a chamada recursiva para inverter a parte interna do arranjo  $A$  tivesse a mesma estrutura (e mesma sintaxe) que a chamada para inverter todo arranjo  $A$ . Em função disso, em vez de chamar inicialmente `ReverseArray(A)`, chama-se `ReverseArray(A, 0, n - 1)`. Normalmente, se existe dificuldade para se encontrar a estrutura repetitiva necessária para projetar um algoritmo recursivo, pode ser útil trabalhar o problema sobre alguns exemplos concretos de maneira a entender como os subproblemas podem ser definidos.

### Recursão final

Em diversas situações, a recursão é uma ferramenta útil para projetar algoritmos que tem definições curtas e elegantes. Mas essa utilidade tem um custo. Quando se usa um algoritmo recursivo para resolver um problema, se gasta uma certa quantidade de memória para manter o estado de cada chamada recursiva ativa. Quando a memória do computador está escassa, em alguns casos é interessante ser capaz de derivar um algoritmo não-recursivo a partir de um recursivo.

Pode-se usar uma estrutura de dados do tipo pilha, discutida na Seção 5.1, para converter um algoritmo recursivo em um algoritmo não-recursivo, mas existem algumas situações em que esta conversão pode ser feita de maneira mais simples e eficiente. Em particular, pode-se converter facilmente algoritmos que usem **recursão final**. Um algoritmo usa recursão final\* se ele usa recursão linear e o algoritmo faz uma chamada recursiva como sua última operação. Por exemplo, o Trecho de código 3.32 usa recursão final para inverter os elementos do arranjo.

Entretanto, não é suficiente que o último comando da definição do método inclua uma chamada recursiva. Para que um método use recursão final, a chamada recursiva deve ser realmente a última coisa que o método faz (caso contrário, seria um caso base, é claro). Por exemplo, o algoritmo do Trecho de código 3.31 não usa recursão final mesmo que seu último comando inclua uma chamada recursiva. Esta chamada recursiva não é, na verdade, a última coisa que o método faz. Após receber o valor retornado da chamada recursiva, ele adiciona este valor em  $A[n - 1]$  e retorna esta soma. Isto é, a última coisa que este algoritmo faz é uma soma, não uma chamada recursiva.

Quando um algoritmo usa recursão final, pode-se converter o mesmo em um algoritmo não recursivo iterando através das chamadas recursivas em vez de chamá-las explicitamente. Demonstra-se esse tipo de conversão revisitando o problema de inverter os elementos de um arranjo. No Trecho de código 3.33, apresenta-se um algoritmo não-recursivo que executa esta tarefa iterando sobre as chamadas recursivas do algoritmo do Trecho de código 3.32. Inicialmente, chama-se este algoritmo usando `IterativeReverseArray(A, 0, n - 1)`.

#### Algoritmo `IterativeReverseArray(A, i, j)`:

**Entrada:** Um arranjo  $A$  e índices inteiros não negativos  $i$  e  $j$

**Saída:** Os elementos de  $A$  invertidos, começando no índice  $i$  e terminando no índice  $j$

\* N. de T. No original, "Tail Recursion".

```

enquanto  $i < j$  faça
    Inverta  $A[i]$  e  $A[j]$ 
     $i \leftarrow i + 1$ 
     $j \leftarrow j - 1$ 
retorna

```

**Trecho de código 3.33** Invertendo os elementos de um arranjo usando iteração.

### 3.5.2 Recursão binária

Quando um algoritmo faz duas chamadas recursivas, diz-se que usa **recursão binária**. Estas chamadas podem, por exemplo, ser usadas para resolver duas metades do mesmo problema, como foi feito na Seção 3.5 para desenhar a régua inglesa. Como outra aplicação de recursão binária, será revisitado o problema de somar os  $n$  elementos de um arranjo de inteiros  $A$ . Neste caso, serão somados os elementos de  $A$  da seguinte forma: (i) somando recursivamente os elementos da primeira metade de  $A$ ; (ii) somando recursivamente os elementos da segunda metade de  $A$ ; e (iii) somando esses dois valores juntos. Os detalhes do algoritmo são fornecidos no Trecho de código 3.34, que deve ser chamado usando  $\text{BinarySum}(A, 0, n)$ .

**Algoritmo**  $\text{BinarySum}(A, i, n)$ :

**Entrada:** um arranjo  $A$  e os inteiros  $i$  e  $n$

**Saída:** A soma dos  $n$  inteiros de  $A$  iniciando pelo índice  $i$

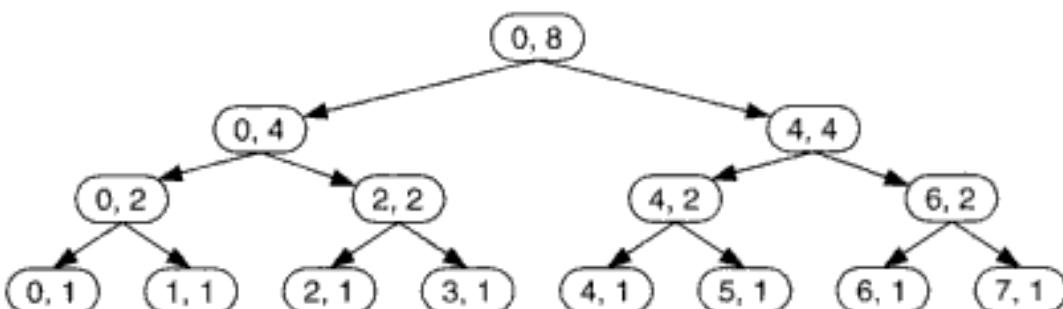
se  $n = 1$  então

**retorna**  $A[i]$

**retorna**  $\text{BinarySum}(A, i, \lceil n/2 \rceil) + \text{Binarysum}(A, i + \lceil n/2 \rceil, \lfloor n/2 \rfloor)$

**Trecho de código 3.34** Somando os elementos de um arranjo usando recursão binária.

Para analisar o algoritmo  $\text{BinarySum}$ , será considerado, por simplicidade, o caso onde  $n$  é potência de 2. O caso geral para  $n$  arbitrário é considerado no Exercício R-4.4. A Figura 3.25 apresenta o rastreamento recursivo de uma execução do método  $\text{BinarySum}(0,8)$ . Rotula-se cada caixa com os valores dos parâmetros  $i$  e  $n$ , que representam o índice inicial e o comprimento da seqüência de elementos a serem revertidos, respectivamente. Observa-se que as setas do rastreamento partem de uma caixa rotulada  $(i, n)$  para outra rotulada  $(i, n/2)$  ou  $(i+n/2, n/2)$ . Isto é, o valor do parâmetro  $n$  é dividido a cada chamada recursiva. Assim, a profundidade da recursão, isto é, o número máximo de instâncias de métodos que estão ativas ao mesmo tempo é  $1 + \log_2 n$ . Assim, o algoritmo  $\text{BinarySum}$  usa uma quantidade de espaço adicional aproximadamente proporcional a este valor. Isso é um grande aperfeiçoamento em relação ao espaço necessário pelo método  $\text{LinearSum}$  do Trecho de código 3.31. Entretanto, o tempo de execução do algoritmo  $\text{BinarySum}$  é também proporcional a  $n$ , uma vez que cada caixa é visitada em um tempo constante quando se avança pelo algoritmo e existem  $2n - 1$  caixas.



**Figura 3.25** Rastreamento recursivo para a execução de  $\text{BinarySum}(0,8)$ .

## Calculando números Fibonacci através de recursão binária

Considera-se o problema de calcular o  $k$ -ésimo número Fibonacci. Na Seção 2.2.3 viu-se que os números Fibonacci são recursivamente definidos como segue:

$$\begin{aligned}F_0 &= 0 \\F_1 &= 1 \\F_i &= F_{i-1} + F_{i-2} \quad \text{for } i > 1.\end{aligned}$$

Aplicando-se diretamente esta definição, o algoritmo `BinaryFib`, apresentado no Trecho de código 3.35, calcula a seqüência de números Fibonacci usando recursão binária.

**Algoritmo** `BinaryFib( $k$ )`:

```
Entrada: inteiro não negativo  $k$ 
Saída: o  $k$ -ésimo número Fibonacci  $F_k$ 
se  $k \leq 1$  então
    retorna  $k$ 
senão
    retorna BinaryFib( $k - 1$ ) + BinaryFib( $k - 2$ )
```

**Trecho de código 3.35** Cálculo do  $k$ -ésimo número de Fibonacci usando recursão binária.

Infelizmente, apesar da definição de Fibonacci se parecer com uma recursão binária, usar esta técnica é ineficiente neste caso. Na verdade, desta forma ele usa uma quantidade exponencial de chamadas para calcular o  $k$ -ésimo número Fibonacci. Mais especificamente, faça  $nk$  denotar o número de chamadas acionadas na execução de `BinaryFib( $k$ )`. Então, tem-se os seguintes valores para  $nk$ .

$$\begin{aligned}n_0 &= 1 \\n_1 &= 1 \\n_2 &= n_1 + n_0 + 1 = 1 + 1 + 1 = 3 \\n_3 &= n_2 + n_1 + 1 = 3 + 1 + 1 = 5 \\n_4 &= n_3 + n_2 + 1 = 5 + 3 + 1 = 9 \\n_5 &= n_4 + n_3 + 1 = 9 + 5 + 1 = 15 \\n_6 &= n_5 + n_4 + 1 = 15 + 9 + 1 = 25 \\n_7 &= n_6 + n_5 + 1 = 25 + 15 + 1 = 41 \\n_8 &= n_7 + n_6 + 1 = 41 + 25 + 1 = 67.\end{aligned}$$

Acompanhando o padrão, percebe-se que o número de chamadas mais que dobra para cada dois índices consecutivos. Isto é,  $n_4$  é mais de duas vezes  $n_2$ ,  $n_5$  é mais de duas vezes  $n_3$ ,  $n_6$  é mais de duas vezes  $n_4$  e assim por diante. Assim  $n_k > 2^{k/2}$ , o que significa que `BinaryFib` faz um número de chamadas que é exponencial em relação a  $k$ . Em outras palavras, o uso de recursão binária para calcular números Fibonacci é muito ineficiente.

## Computando Fibonacci usando recursão linear

O maior problema com a abordagem anterior, baseada em recursão binária, é que o cálculo de números Fibonacci é, na verdade, um problema linearmente recursivo. Ele não é um bom candidato para se usar recursão binária. Fica-se tentado a usar recursão binária por causa da maneira pela qual o  $k$ -ésimo número Fibonacci,  $F_k$ , depende dos dois valores anteriores,  $F_{k-1}$  e  $F_{k-2}$ . Mas pode-se computar  $F_k$  de maneira muito mais eficiente usando recursão linear.

Entretanto, para se usar recursão linear, é necessário redefinir o problema ligeiramente. Uma maneira de obter esta conversão é definir uma função recursiva que calcule um par de números Fibonacci ( $F_k, F_{k-1}$ ) usando como convenção que  $F_{-1} = 0$ . Então se pode usar o algoritmo linearmente recursivo apresentado no Trecho de código 3.36.

**Algoritmo** LinearFibonacci( $k$ ):

```

Entrada: um inteiro não negativo  $k$ 
Saída: um par de números Fibonacci ( $F_k, F_{k-1}$ )
se  $k \leq 1$  então
    retorna ( $k, 0$ )
senão
     $(i, j) \leftarrow \text{LinearFibonacci}(k - 1)$ 
    retorna ( $i + j, i$ )

```

**Trecho de código 3.36** Calculando o  $k$ -ésimo número de Fibonacci usando recursão linear.

O algoritmo apresentado no Trecho de código 3.36 mostra que usar regressão linear para calcular números Fibonacci é muito mais eficiente do que usar recursão binária. Uma vez que cada chamada recursiva para LinearFibonacci decremente o argumento  $k$  de 1, a chamada LinearFibonacci( $k$ ) resulta em uma série de  $k - 1$  chamadas adicionais. Isto é, calcular o  $k$ -ésimo número através de recursão linear requer  $k$  chamadas de métodos. Esta performance é significativamente mais rápida que o tempo exponencial exigido pelo algoritmo baseado em recursão binária, apresentado no Trecho de código 3.35. Por essa razão, quando se usa recursão binária, deve-se primeiro tentar partitionar completamente o problema em dois (como foi feito com a soma dos elementos do arranjo) ou se pode estar convencido de que sucessivas chamadas recursivas são realmente necessárias.

Normalmente pode-se eliminar chamadas recursivas que se sobrepõe usando mais memória para manter os valores anteriores. Na verdade, esta abordagem é a parte central de uma técnica chamada de *programação dinâmica*, que está relacionada à recursão e é discutida na Seção 12.5.2.

### 3.5.3 Recursão múltipla

Generalizando a partir da recursão binária, usa-se **recursão múltipla** quando um método pode fazer várias chamadas recursivas, em um número potencialmente maior que dois. Uma das aplicações mais comuns deste tipo de recursão é usado quando se deseja enumerar várias configurações visando resolver um quebra-cabeças combinatório. Os exemplos que seguem são instâncias de *quebra-cabeças de soma*:

$$\begin{aligned} pot + pan &= bib \\ dog + cat &= pig \\ boy + girl &= baby \end{aligned}$$

Para resolver este tipo de quebra-cabeças, é necessário atribuir um dígito único (isto é 0, 1, ..., 9) para cada letra da equação, de maneira a torná-la verdadeira. Tipicamente, resolvem-se quebra-cabeças com base nas observações “humanas” que se faz de um quebra-cabeças, em particular para eliminar configurações (isto é, possíveis atribuições parciais de dígitos para letras) até que restem apenas configurações válidas para trabalhar, testando-se a correção de cada uma.

Se o número de configurações possíveis não for muito grande, entretanto, pode-se usar um computador para simplesmente enumerar todas as possibilidades e testar cada uma delas sem empregar observações “humanas”. Além disso, tal algoritmo pode usar recursão múltipla para trabalhar com as configurações de uma forma sistemática. Apresenta-se o pseudo-código para este algoritmo no Trecho de código 3.37. Para manter a descrição suficientemente genérica para ser usada com outros quebra-cabeças, o algoritmo enumera e testa todas as sequências de comprimento  $k$  sem repetições de elementos de um dado conjunto  $U$ . Constróem-se as sequências de  $k$  elementos através dos seguintes passos:

1. Geram-se recursivamente as sequências de  $k - 1$  elementos.
2. Acrescenta-se a cada sequência um elemento que ela ainda não contenha.

Por meio da execução do algoritmo usa-se o conjunto  $U$  para manter os elementos que não estão contidos na seqüência corrente, de maneira que um elemento  $e$  ainda não foi usado se e somente se  $e$  estiver em  $U$ .

Outra forma de analisar o algoritmo do Trecho de código 3.37 é que ele enumera todos os tamanhos possíveis para os subconjuntos ordenados de  $U$  de tamanho  $k$ , e testa cada subconjunto como sendo uma possível solução para o quebra-cabeças.

Para quebra-cabeças de soma,  $U = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$  e cada posição na seqüência corresponde a uma dada letra. Por exemplo, a primeira posição pode ser o lugar de um  $b$ , a segunda de um  $o$ , a terceira de um  $y$  e assim por diante.

**Algoritmo** `PuzzleSolve( $k, S, U$ ):`

**Entrada:** um inteiro  $k$ , uma seqüência  $S$  e um conjunto  $U$ .

**Saída:** uma enumeração de todas as extensões de  $S$  de tamanho  $k$  que usam elementos de  $U$  sem repetições.

**Para cada**  $e$  em  $U$  **faça**

    Remova  $e$  de  $U$  { $e$  agora está sendo usado}

    Aumente  $e$  no final de  $S$

**se**  $k = 1$  **então**

        Teste se  $S$  é uma configuração que resolve o quebra-cabeças

**se**  $S$  resolve o quebra-cabeças **então**

**retorna** "Solução encontrada: "  $S$

**senão**

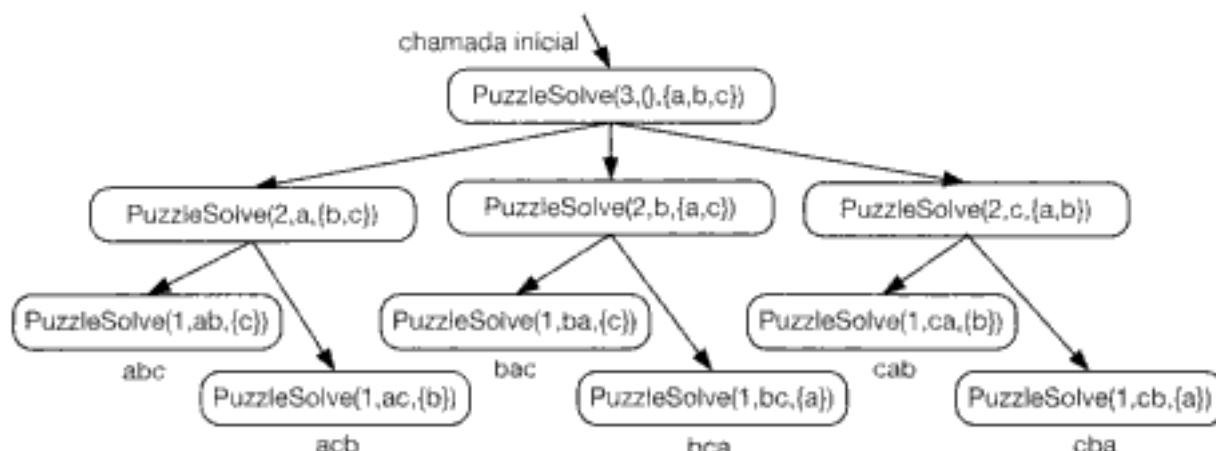
        PuzzleSolve( $k - 1, S, U$ )

    Coloque  $e$  de volta em  $U$  ( $e$  agora não está sendo usado)

    Remova  $e$  do final de  $S$

**Trecho de código 3.37** Resolvendo um quebra-cabeças combinatório pela enumeração e teste de todas as configurações possíveis.

Na Figura 3.26, apresenta-se o rastreamento recursivo de uma chamada para `PuzzleSolve(3, S, U)`, onde  $S$  está vazio e  $U = \{a, b, c\}$ . Durante a execução, todas as permutações dos 3 caracteres são geradas e testadas. Observa-se que a chamada inicial faz três chamadas recursivas, cada uma das quais, por sua vez, faz mais duas. Se `PuzzleSolve(3, S, U)` for executado sobre um conjunto  $U$  consistindo em quatro elementos, a chamada inicial teria feito quatro chamadas recursivas, cada uma das quais teria um rastreamento parecido com o da Figura 3.26.



**Figura 3.26** Rastreamento recursivo para uma execução de `PuzzleSolve(3, S, U)`, onde  $S$  está vazio e  $U = \{a, b, c\}$ . Esta execução gera e testa todas as permutações de  $a, b$  e  $c$ . As permutações geradas são vistas logo abaixo das respectivas caixas.

## 3.6 Exercícios

Para obter ajuda e o código-fonte dos exercícios, visite [Java.datastructures.net](http://Java.datastructures.net).

### Reforço

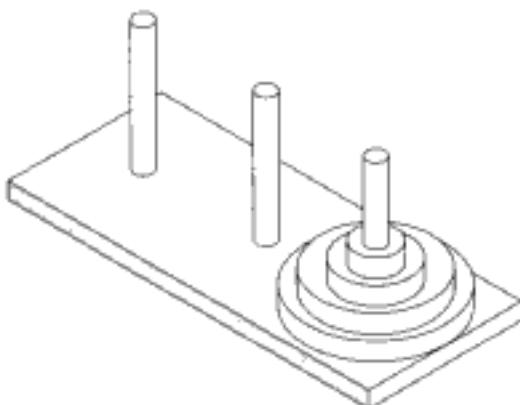
- R-3.1 Os métodos `add` e `remove` do Trecho de código 3.3 e 3.4 não mantêm o número  $n$  de entradas não-nulas do arranjo  $a$ . Em vez disso, as células não usadas apontam para um objeto `null`. Mostre como alterar estes métodos de maneira que eles mantenham o tamanho atual de  $a$  em uma variável de instância  $n$ .
- R-3.2 Descreva uma forma de usar recursão para acrescentar todos elementos de  $a$  em um arranjo  $n \times n$  (bidimensional) de inteiros.
- R-3.3 Explique como modificar o programa da Cifra de César (Trecho de código 3.9) de maneira que ele execute codificação e decodificação ROT13, que usa 13 como valor de deslocamento do alfabeto. Como você pode simplificar o código de maneira que o corpo do método de decodificação tenha apenas uma única linha?
- R-3.4 Explique as alterações que devem ser feitas no programa do Trecho de código 3.9, de maneira que ele possa executar a Cifra de César para mensagens que são escritas em línguas baseadas em alfabetos diferentes do inglês, tais como grego, russo ou hebraico.
- R-3.5 Qual a exceção que é lançada quando `advance` ou `remove` do Trecho de código 3.25 são chamados sobre uma lista vazia? Explique como modificar estes métodos de maneira a fornecer um nome de exceção mais instrutivo para esta condição.
- R-3.6 Apresente uma definição recursiva para uma lista simplesmente encadeada.
- R-3.7 Descreva um método para inserir um elemento no início de uma lista simplesmente encadeada. Assuma que a lista *não* usa um nodo sentinel `e`, em vez disso, usa a variável `head` para referenciar o primeiro nodo da lista.
- R-3.8 Forneça um algoritmo para encontrar o penúltimo nodo em uma lista simplesmente encadeada onde o último elemento é indicado por uma referência `next` nula.
- R-3.9 Descreva um método não-recursivo para encontrar, avançando pelos encadeamentos, o nodo do meio de uma lista duplamente encadeada com sentinelas de início e fim. (Observe: este método deve usar apenas navegação pelos encadeamentos; não deve usar um contador). Qual é o tempo de execução deste método?
- R-3.10 Descreva um algoritmo recursivo para encontrar o maior elemento em um arranjo  $A$  de  $n$  elementos. Qual é o tempo de execução e a memória utilizada?
- R-3.11 Desenhe o rastreamento recursivo da execução do método `ReverseArray(A, 0, 4)` (Trecho de código 3.32) sobre o arranjo  $A = \{4, 3, 6, 2, 5\}$ .
- R-3.12 Desenhe o rastreamento recursivo para a execução do método `PuzzleSolve(3, S, U)` (Trecho de código 3.37) onde  $S$  está vazio e  $U = \{a, b, c, d\}$ .

- R-3.13 Escreva um pequeno método Java que repetidamente seleciona e remove aleatoriamente uma entrada de um arranjo até que ele não armazene mais entradas.
- R-3.14 Escreva um pequeno método Java para contar o número de nodos em uma lista encadeada circular.

### Criatividade

- C-3.1 Forneça o código Java dos métodos `add(e)` e `remove(i)` para entradas de jogos em um arranjo  $A$ , como nos Trechos de código 3.3 e 3.4, exceto que agora as entradas não são mantidas em ordem. Assuma que ainda é necessário manter  $n$  entradas armazenadas nos índices de 0 a  $n - 1$ . Tente implementar os métodos `add` e `remove` sem usar laços, de forma que o número de passos que eles executem não dependa de  $n$ .
- C-3.2 Faça  $A$  ser um arranjo de tamanho  $n \geq 2$  contendo inteiros de 1 a  $n - 1$ , inclusive, com exatamente um repetido. Descreva um algoritmo rápido para encontrar o inteiro de  $A$  que está repetido.
- C-3.3 Seja  $B$  um arranjo de tamanho  $n \geq 6$  contendo inteiros de 1 a  $n - 5$ , inclusive, com exatamente 5 repetidos. Descreva um bom algoritmo para encontrar os 5 inteiros de  $B$  que estão repetidos.
- C-3.4 Suponha que você está projetando um jogo para vários jogadores que tem  $n \geq 1000$  jogadores, numerados de 1 a  $n$ , interagindo em uma floresta encantada. O vencedor deste jogo é o primeiro jogador que puder encontrar todos os demais pelo menos uma vez (cordas são permitidas). Assumindo que existe um método `meet(i, j)` que é chamado cada vez que o jogador  $i$  encontra o jogador  $j$  (com  $i \neq j$ ), descreva uma maneira de manter os pares de jogadores que se encontram e quem é o vencedor.
- C-3.5 Apresente um algoritmo recursivo para calcular o produto de dois inteiros positivos,  $m$  e  $n$ , usando apenas adição e subtração.
- C-3.6 Descreva um algoritmo recursivo rápido para inverter uma lista simplesmente encadeada  $L$ , de maneira que a ordem dos nodos seja o oposto do que era antes. Se a lista só tem uma posição, então não há nada a ser feito; a lista já está invertida. Em qualquer outro caso, remova.
- C-3.7 Descreva um bom algoritmo para concatenar duas listas simplesmente encadeadas,  $L$  e  $M$ , com sentinelas cabeça, em uma única lista  $L'$  que contém todos os nodos de  $L$  seguidos por todos os nodos de  $M$ .
- C-3.8 Forneça um algoritmo rápido para concatenar duas listas duplamente encadeadas  $L$  e  $M$ , com nodos sentinela cabeça e final, em uma única lista  $L'$ .
- C-3.9 Descreva em detalhes como trocar dois nodos  $x$  e  $y$  de uma lista simplesmente encadeada  $L$  fornecidas apenas referências para  $x$  e  $y$ . Repita este exercício para o caso onde  $L$  é uma lista duplamente encadeada. Qual algoritmo consome mais tempo?
- C-3.10 Descreva em detalhes um algoritmo para inverter uma lista simplesmente encadeada  $L$  usando apenas uma quantidade constante de espaço adicional e sem usar recursão.
- C-3.11 No quebra-cabeça das *Torres de Hanói*, existe uma plataforma com três pinos,  $a$ ,  $b$  e  $c$  fincados na mesma. No pino  $a$  temos uma pilha de discos,

um maior que o outro, de maneira que o menor está no topo e o maior na base. O desafio é mover todos os discos do pino *a* para o pino *c*, movendo um disco de cada vez, de maneira que nunca seja colocado um disco maior sobre um disco menor. Veja a Figura 3.27 como exemplo do caso de  $n = 4$ . Descreva um algoritmo recursivo para resolver o quebra-cabeças das Torres de Hanói para qualquer valor de  $n$ . (Dica: considere primeiro o subproblema de mover todos menos o  $n$ -ésimo disco do pino *a* para outro pino usando o terceiro como “armazenamento temporário”.)



**Figura 3.27** Desenho do quebra-cabeças das Torres de Hanói.

- C-3.12 Descreva um método recursivo para converter um string de dígitos no inteiro que ele representa. Por exemplo, "13531" representa o inteiro 13.531.
- C-3.13 Descreva um algoritmo recursivo que conte o número de nodos em uma lista simplesmente encadeada.
- C-3.14 Escreva um programa Java recursivo que resultará em todos os suconjuntos de um conjunto de  $n$  elementos (sem repetir nenhum subconjunto).
- C-3.15 Escreva um pequeno programa Java recursivo que encontre o menor e o maior valor de um arranjo de valores `int` sem usar nenhum laço.
- C-3.16 Descreva um algoritmo recursivo que irá verificar se um arranjo  $A$  de inteiros contém um inteiro  $A[i]$  que é a soma de dois inteiros que aparecem antes em  $A$ , isto é, tais que  $A[i] = A[j] + A[k]$  para  $j, k < i$ .
- C-3.17 Escreva um pequeno método Java recursivo que irá reorganizar um arranjo de valores `int` de maneira que todos os valores pares apareçam antes de todos os valores ímpares.
- C-3.18 Escreva um pequeno método Java recursivo que pega um caracter string *s* e exibe seu inverso. Assim, por exemplo, o inverso de "post&pans" será "snap&stop".
- C-3.19 Escreva um pequeno método Java recursivo que determina se uma string *s* é um palíndromo, isto é, se é igual ao seu inverso. Por exemplo "racecar" e "gohangasalamimalasagnahog" são palíndromos.
- C-3.20 Use recursão para escrever um método Java para determinar se uma string *s* tem mais vogais que consoantes.
- C-3.21 Suponha que são fornecidas duas listas circulares, *L* e *M*, isto é, duas listas de nodos de forma que cada nodo possui uma referência `next` não-nula. Descreva um algoritmo rápido para dizer se *L* e *M* são na verdade a mesma lista de nodos, mas com diferentes (cursors) pontos de partida.

- C-3.22 Dada uma lista circular encadeada  $L$  contendo um número par de nodos, descreva como dividir  $L$  em duas listas encadeadas circulares com a metade do tamanho.

---

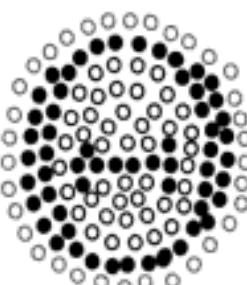
### Projetos

- P-3.1 Escreva um programa Java para uma classe matriz que possa adicionar e multiplicar arranjos bidimensionais de inteiros quaisquer.
- P-3.2 Execute o projeto anterior, mas use tipos genéricos de maneira que as matrizes envolvidas possam conter tipos arbitrários de números.
- P-3.3 Escreva uma classe que mantém os dez maiores escores para uma aplicação de jogo, implementando os métodos `add` e `remove` da Seção 3.11, mas usando uma lista simplesmente encadeada em vez de um arranjo.
- P-3.4 Execute o projeto anterior, mas use uma lista duplamente encadeada. Além disso, sua implementação de `remove(i)` deve fazer o menor número de deslocamentos sobre as conexões para obter a entrada sob o índice  $i$ .
- P-3.5 Execute o projeto anterior mas use uma lista que encadeada que seja tanto circular como duplamente encadeada.
- P-3.6 Escreva um programa para resolver os quebra-cabeças de soma pela enumeração e teste de todas as soluções possíveis. Usando seu programa, resolva os três quebra-cabeças fornecidos na Seção 3.5.3.
- P-3.7 Escreva um programa que criptografe e decriptografe usando uma cifra de substituição arbitrária. Neste caso, o arranjo de criptografia é uma mistura aleatória de letras do alfabeto. Seu programa deve gerar o array aleatório de criptografia, seu arranjo correspondente de decodificação e use estes para codificar e decodificar uma mensagem.
- P-3.8 Escreva um programa que possa executar a Cifra de César para mensagens em inglês que incluem tanto caracteres minúsculos como maiúsculos.

---

### Observações sobre o capítulo

As estruturas de dados fundamentais de arranjos e listas encadeadas, bem como recursão, discutidas neste capítulo pertencem ao folclore da Ciência da Computação. Elas foram descritas pela primeira vez na literatura de Ciência da Computação por Knuth no seu original livro sobre *Fundamentos de Algoritmos* [62].



## Conteúdo

---

<b>4.1 As sete funções usadas neste livro .....</b>	<b>150</b>
4.1.1 A função constante .....	150
4.1.2 A função logaritmo .....	150
4.1.3 A função linear.....	151
4.1.4 A função $n \cdot \log n$ .....	151
4.1.5 A função quadrática .....	152
4.1.6 A função cúbica e outras polinomiais .....	152
4.1.7 A função exponencial .....	154
4.1.8 Comparando taxas de crescimento .....	155
<b>4.2 Análise de algoritmos .....</b>	<b>156</b>
4.2.1 Estudos experimentais .....	157
4.2.2 Operações primitivas.....	158
4.2.3 Notação assintótica.....	159
4.2.4 Análise assintótica..	163
4.2.5 Usando a notação $O$ .....	164
4.2.6 Um algoritmo recursivo para calcular potência.	167
<b>4.3 Técnicas simples de justificativa .....</b>	<b>168</b>
4.3.1 Através de exemplos.....	168
4.3.2 O ataque “contra” .....	168
4.3.3 Indução e invariantes em laços.....	169
<b>4.4 Exercícios .....</b>	<b>171</b>

## 4.1 As sete funções usadas neste livro

Esta seção discute brevemente as sete funções mais importantes usadas em análise de algoritmos. São usadas apenas sete funções simples para a maioria das análises feitas neste livro. As seções que usam alguma outra função serão marcadas com uma estrela (\*), indicando que são opcionais. O Apêndice A contém uma lista de outros fatos matemáticos úteis que se aplicam ao contexto de análise de algoritmos e estruturas de dados.

### 4.1.1 A função constante

A função mais simples que se pode imaginar é a *função constante*. Esta é a função,

$$f(n) = c,$$

para alguma constante fixa  $c$ , tal que  $c = 5$ ,  $c = 27$  ou  $c = 2^{10}$ . Isto é, para qualquer argumento  $n$ , a função constante  $f(n)$  atribui um valor  $c$ . Em outras palavras, não importa qual é o valor de  $n$ ;  $f(n)$  será sempre igual ao valor da constante  $c$ .

Uma vez que se está mais interessado em funções inteiros, a função constante mais fundamental é  $g(n) = 1$ , e esta é a típica função constante que será usada neste livro. Observa-se que qualquer outra função constante,  $f(n) = c$ , pode ser escrita como uma constante  $c$  que multiplica  $g(n)$ . Isto é,  $f(n) = cg(n)$  neste caso.

Por sua simplicidade, a função constante é útil na análise de algoritmos porque caracteriza o número de passos necessários para executar uma operação básica em um computador, tal como adicionar dois valores, atribuir um valor para alguma variável ou comparar dois valores.

### 4.1.2 A função logaritmo

Um dos aspectos mais interessantes e talvez mais surpreendentes da análise de estruturas de dados e algoritmos é a onipresença da *função logaritmo*,  $f(n) = \log_b n$  para alguma constante  $b > 1$ . Esta função é definida como segue:

$$x = \log_b n \quad \text{se e somente se} \quad b^x = n.$$

Por definição,  $\log_b 1 = 0$ . O valor  $b$  é conhecido como a *base* do logaritmo.

Calcular a função logaritmo exata para qualquer inteiro  $n$  envolve o uso de cálculo, mas pode-se usar uma aproximação que é boa o suficiente para o que se propõe, sem cálculo. Pode-se calcular facilmente o menor inteiro maior ou igual a  $\log_b n$ , pois este número é igual ao número de vezes que se pode dividir  $n$  por  $b$  até que se obtenha um número menor ou igual a 1. Por exemplo, a avaliação de  $\log_2 27$  é 3, uma vez que  $27/3/3/3 = 1$ . Da mesma forma, a avaliação de  $\log_4 64$  é 4, uma vez que  $64/4/4/4 = 1$ , e a aproximação para  $\log_2 12$  é 4, uma vez que  $12/2/2/2/2 = 0.75 \leq 1$ . A aproximação de base 2 surge na análise de algoritmos porque uma operação comum a muitos algoritmos é dividir repetidamente uma entrada pela metade.

Na verdade, uma vez que computadores armazenam inteiros em binário, a base mais comum para a função logaritmo em Ciência da Computação é 2. De fato, esta base é tão comum que tipicamente não é indicada se for 2. Então, será considerado

$$\log n = \log_2 n.$$

Observa-se que a maioria das calculadoras portáteis tem um botão marcado LOG, mas que normalmente é usado para calcular o logaritmo de base 10, e não de base 2.

Existem algumas regras importantes sobre logaritmos, parecidas com as regras de expoente.

**Proposição 4.1 (Regras de logaritmo)** Dados números reais  $a > 0$ ,  $b > 1$ ,  $c > 0$  e  $d > 1$ , tem-se que:

1.  $\log_b ac = \log_b a + \log_b c$
2.  $\log_b a/c = \log_b a - \log_b c$
3.  $\log_b a^c = c \log_b a$
4.  $\log_b a = (\log_d a) / \log_d b$
5.  $b^{\log_d a} = a^{\log_d b}$ .

Também, como uma abreviatura notacional, será usado  $\log^n n$  para denotar a função  $(\log n)^n$ . Em vez de mostrar como se pode derivar cada uma das identidades acima, que derivam todas da definição de logaritmos e expoentes, as mesmas serão demonstradas com alguns exemplos.

**Exemplo 4.2** Demonstra-se algumas aplicações interessantes das regras de logaritmos da Proposição 4.1 (usando a convenção usual de que a base do logaritmo é 2, se omitida).

- $\log(2n) = \log 2 + \log n + 1 + \log n$ , pela regra 1
- $\log(n/2) = \log n - \log 2 = \log n - 1$ , pela regra 2
- $\log n^3 = 3 \log n$ , pela regra 3
- $\log_2 n = n \log 2 = n \cdot 1 = n$ , pela regra 3
- $\log_4 n = (\log n) / \log 4 = (\log n)/2$ , pela regra 4
- $2^{\log n} = n^{\log 2} = n^1 = n$ , pela regra 5

De uma forma prática, observa-se que a regra 4 fornece uma maneira de calcular o logaritmo de base 2 em uma calculadora que tenha o botão de logaritmo da base 10, LOG, pois

$$\log_2 n = \text{LOG } n / \text{LOG } 2$$

#### 4.1.3 A função linear

Outra função simples, mas importante, é a *função linear*,

$$f(n) = n.$$

Isto é, dado um valor de entrada  $n$ , a função linear  $f$  atribui o valor  $n$  para si mesma. Esta função aparece na análise de algoritmos sempre que se tem de executar uma operação básica sobre cada um de  $n$  elementos. Por exemplo, comparar um número  $x$  com cada elemento de um arranjo de tamanho  $n$  requer  $n$  comparações. A função linear também representa o melhor tempo de execução que se pode desejar obter para qualquer algoritmo que processa uma coleção de  $n$  objetos que não estão na memória do computador, uma vez que a própria leitura dos  $n$  objetos requer  $n$  operações.

#### 4.1.4 A função $n \cdot \log n$

A próxima função a ser discutida nesta seção é a *função  $n \cdot \log n$* ,

$$f(n) = n \log n,$$

ou seja, a função que atribui para uma entrada  $n$  o valor de  $n$  multiplicado pelo logaritmo de base 2 de  $n$ . Esta função cresce um pouco mais rápido que a função linear e muito mais devagar que a função quadrática. Assim, como será mostrado em várias ocasiões, desejando-se melhorar o tempo de execução da solução de um problema de quadrático para  $n \cdot \log n$ , ter-se-á um algoritmo que executa muito mais rápido no geral.

#### 4.1.5 A função quadrática

Outra função que aparece com freqüência na análise de algoritmos é a *função quadrática*,

$$f(n) = n^2.$$

Isto é, dado um valor de entrada  $n$ , a função  $f$  atribui o produto de  $n$  por si mesmo (em outras palavras, “ $n$  ao quadrado”).

A principal razão de a função quadrática aparecer na análise de algoritmos é que existem vários algoritmos que possuem laços aninhados, onde o laço mais interno executa uma quantidade linear de operações e o laço mais externo é executado um número linear de vezes. Assim, nestes casos, o algoritmo executa  $n \cdot n = n^2$  operações.

#### Laços aninhados e a função quadrática

A função quadrática também pode surgir no contexto de laços aninhados onde a primeira iteração do laço usa uma operação, a segunda usa duas, a terceira usa três e assim por diante. Isto é, o número de operações é

$$1 + 2 + 3 + \dots + (n - 2) + (n - 1) + n.$$

Em outras palavras, este será o total de operações executadas pelo laço mais interno se o número de operações dentro do laço crescer uma unidade a cada iteração do laço mais externo. Esta quantidade também está relacionada a uma história interessante.

Em 1787, um professor alemão de colégio decidiu manter seus pupilos de nove e dez anos ocupados adicionando os inteiros de 1 a 100. Mas quase imediatamente, uma das crianças manifestou ter encontrado a resposta! O professor ficou cismado porque o aluno tinha apenas a resposta em sua lousa. Mas estava correta — 5050 — e o estudante, Carl Gauss, cresceu e se tornou um dos maiores matemáticos de seu tempo. Suspeita-se que o jovem Gauss usou a seguinte identidade.

**Proposição 4.3** *Para qualquer inteiro  $\geq 1$  tem-se que:*

$$1 + 2 + 3 + \dots + (n - 2) + (n - 1) + n = \frac{n(n+1)}{2}.$$

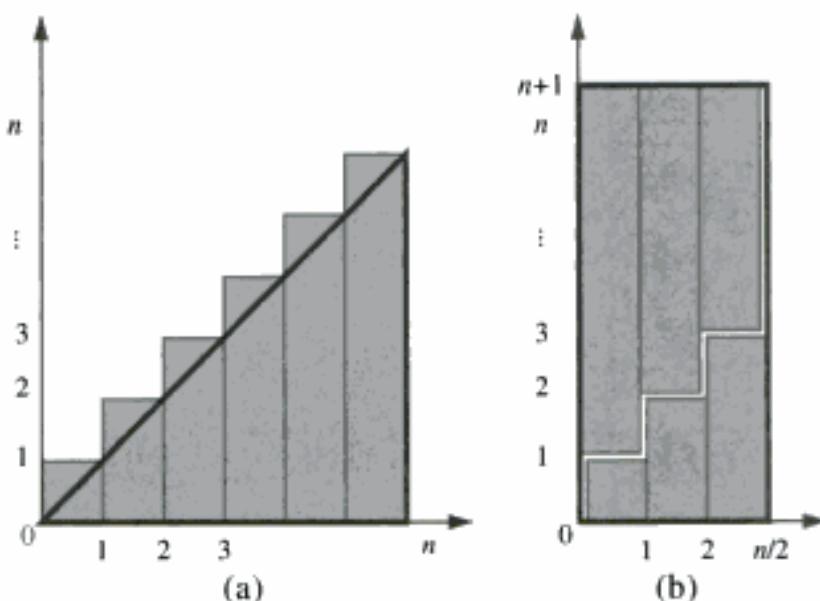
São fornecidas duas justificativas “visuais” para a Proposição 4.3 na Figura 4.1.

A lição aprendida a partir da Proposição 4.3 é que se executa um algoritmo com laços aninhados de maneira que as operações do laço mais interno crescem uma de cada vez, então a quantidade total de operações é quadrática em relação ao número de vezes,  $n$ , que o laço mais externo é executado. Mais especificamente, o número de operações é  $n^2/2 + n/2$ . Neste caso, o que é um pouco mais do que o fator constante ( $1/2$ ) vezes a função quadrática  $n^2$ . Em outras palavras, tal algoritmo é ligeiramente melhor que um algoritmo que usa  $n$  operações cada vez que o laço interno é executado. Esta observação inicialmente aparenta não ser intuitiva, mas é sempre verdade, como mostra a Figura 4.1.

#### 4.1.6 A função cúbica e outras polinomiais

Continuando a discussão sobre funções que são potência de sua entrada, considera-se a *função cúbica*,

$$f(n) = n^3.$$



**Figura 4.1** Justificativas visuais da Proposição 4.3. Ambas ilustrações visualizam a identidade em termos do total de área coberto por  $n$  retângulos unitários com alturas 1, 2, ...,  $n$ . Em (a) os retângulos são mostrados de maneira a cobrir um grande triângulo com área  $n^2/2$  (base  $n$  e altura  $n$ ) mais  $n$  pequenos triângulos de área  $\frac{1}{2}$  cada (base 1 e altura 1). Em (b), que se aplica apenas quando  $n$  é ímpar, os retângulos são mostrados cobrindo um grande retângulo de base  $n/2$  e altura  $n + 1$ .

que atribui para um valor de entrada  $n$  o produto de  $n$  por ele mesmo três vezes. Esta função aparece com menos freqüência no contexto da análise de algoritmos do que as funções constante, linear ou quadrática previamente mencionadas, mas aparece de vez em quando.

## Polinomiais

De maneira interessante, as funções listadas até agora podem ser vistas todas como sendo parte de uma classe maior de funções, os **polinômios**.

Uma função polinomial é uma função da forma,

$$f(n) = a_0 + a_1 n + a_2 n^2 + a_3 n^3 + \cdots + a_d n^d,$$

onde  $a_0, a_1, \dots, a_d$  são constantes chamadas de **coeficientes** do polinômio e  $a_d \neq 0$ . O inteiro  $d$ , que indica a maior potência do polinômio, é chamado de **grau** do polinômio.

Por exemplo, as funções a seguir são polinomiais:

- $f(n) = 2 + 5n + n^2$
- $f(n) = 1 + n^3$
- $f(n) = 1$
- $f(n) = n$
- $f(n) = n^2$ .

Por essa razão, pode-se argumentar que este livro apresenta apenas quatro funções importantes usadas na análise de algoritmos, mas, teimosamente, insistiremos que são sete, uma vez que as funções constante, linear e quadrática são muito importantes para serem agrupadas com as demais polinomiais. Tempos de execução que são polinomiais com um grau  $d$ , em geral são melhores que tempos de execução polinomiais com um grau mais alto.

## Somatários

Uma notação que aparece com freqüência em análise de estruturas de dados e algoritmos é o **somatário**, que é definido como segue:

$$\sum_{i=a}^b f(i) = f(a) + f(a+1) + f(a+2) + \dots + f(b),$$

onde  $a$  e  $b$  são inteiros e  $a \leq b$ . Somatórios surgem na análise de estruturas de dados e algoritmos porque os tempos de execução dos laços correspondem naturalmente à somatórios. Usando a notação de somatório, pode-se reescrever a fórmula da Proposição 4.3 como

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

Da mesma forma, pode-se escrever uma polinomial  $f(n)$  de grau  $d$  com coeficientes  $a_0, \dots, a_d$  como

$$f(n) = \sum_{i=0}^d a_i n^i.$$

Assim, a notação de somatório fornece um atalho para expressar somas de termos crescentes que tem uma estrutura regular.

### 4.1.7 A função exponencial

Outra função usada na análise de algoritmos é a **função exponencial**,

$$f(n) = b^n,$$

onde  $b$  é uma constante positiva, chamada **base**, e o argumento  $n$  é o **expoente**. Isto é, a função  $f(n)$  atribui ao argumento de entrada  $n$  o valor obtido pela multiplicação da base  $b$  por si mesma  $n$  vezes. Na análise de algoritmos, a base mais comum para a função exponencial é  $b = 2$ . Por exemplo, se existe um laço que começa executando uma operação e dobra o número de operações executadas a cada iteração, então o número de operações executadas pela  $n$ -ésima iteração é  $2^n$ . Além disso, uma palavra inteira contendo  $n$  bits pode representar todos os inteiros não negativos menores que  $2^n$ . Assim, a função exponencial de base 2 é muito comum. A função exponencial também será chamada de **função expoente**.

Entretanto, algumas vezes ocorrem outros expoentes além de  $n$ ; consequentemente, é útil conhecer algumas regras práticas para se trabalhar com expoentes. Em especial, as seguintes **Regras de Expoentes** são muito úteis.

**Proposição 4.4 (Regras de Expoente):** *Dados os inteiros positivos  $a$ ,  $b$ ,  $c$  tem-se que*

1.  $(b^a)^c = b^{ac}$
2.  $b^a b^c = b^{a+c}$
3.  $b^a / b^c = b^{a-c}$ .

Por exemplo, tem-se o seguinte:

- $256 = 16^2 = (2^4)^2 = 2^{4 \cdot 2} = 2^8 = 256$  (Regra de Expoente 1)
- $243 = 3^5 = 3^{2+3} = 3^2 3^3 = 9 \cdot 27 = 243$  (Regra de Expoente 2)
- $16 = 1024/64 = 2^{10}/2^6 = 2^{10-6} = 2^4 = 16$  (Regra de Expoente 3).

Pode-se estender a função exponencial a expoentes que são frações ou números reais e para expoentes negativos como segue. Dado um inteiro positivo  $k$ , define-se  $b^{1/k}$  como sendo a  $k$ -ésima raiz de  $b$ , isto é, o número  $r$  tal que  $r^k = b$ . Por exemplo,  $25^{1/2} = 5$ , uma vez que  $5^2 = 25$ . Da mesma forma,  $27^{1/3} = 3$  e  $16^{1/4} = 2$ . Esta abordagem permite definir qualquer potência cujo expoente possa ser expresso como uma fração, pois  $b^{a/c} = (b^{1/c})^a$ , pela Regra dos Exponentes 1. Por exemplo,  $9^{3/2} = (9^3)^{1/2} = 729^{1/2} = 27$ . Assim,  $b^{a/c}$  é na verdade a  $c$ -ésima raiz da integral exponencial  $b^a$ .

Pode-se também estender a função exponencial para definir  $b^x$  para qualquer número real  $x$ , calculando uma série de números da forma  $b^{a/c}$  para frações  $a/c$  que se aproximam cada vez mais de  $x$ . Qualquer número real  $x$  pode ser aproximado arbitrariamente por uma fração  $a/c$ ; consequentemente, pode-se usar a fração  $a/c$  como expoente de  $b$  para ficar muito próximo de  $b^x$ . Assim, por exemplo, o número  $2^{\pi}$  está bem definido. Finalmente, dado um expoente negativo  $d$ , define-se  $b^d = 1/b^{1-d}$ , o que corresponde à aplicação da regra de expoente 3 com  $a = 0$  e  $c = -d$ .

### Somas geométricas

Suponha que existe um laço onde cada iteração usa um fator multiplicativo maior que o anterior. Este laço pode ser analisado usando a seguinte proposição.

**Proposição 4.5** *Para qualquer inteiro  $n \geq 0$  e qualquer número real tal que  $a > 0$  e  $a \neq 1$ , considera-se o somatório*

$$\sum_{i=0}^n a^i = 1 + a + a^2 + \cdots + a^n$$

(lembrando que  $a^0 = 1$  se  $a > 0$ ). Este somatório é igual a

$$\frac{a^{n+1} - 1}{a - 1}.$$

Somatório, como o mostrado na Proposição 4.5, é chamado de somatório **geométrico**, porque cada termo é geometricamente maior que o anterior se  $a > 1$ . Por exemplo, todo mundo que trabalha em computação deve saber que

$$1 + 2 + 4 + 8 + \cdots + 2^{n-1} = 2^n - 1,$$

porque este é o maior inteiro que pode ser representado em notação binária usando  $n$  bits.

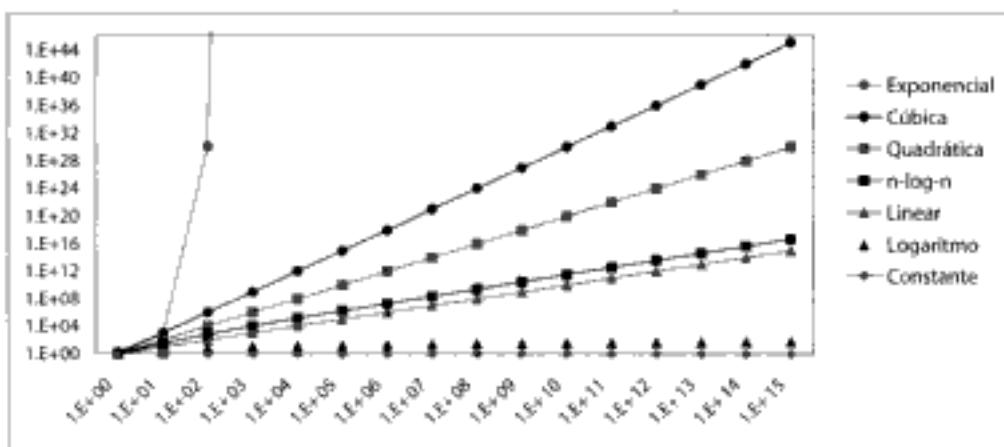
#### 4.1.8 Comparando taxas de crescimento

Resumindo, a Tabela 4.1 mostra cada uma das sete funções mais comuns usadas em análise de algoritmos, descritas anteriormente, pela ordem.

constante	logaritmo	linear	n-log-n	quadrática	cúbica	exponencial
1	$\log n$	$n$	$n \log n$	$n^2$	$n^3$	$a^n$

**Tabela 4.1** Classes de funções. Assume-se que  $a > 1$  é uma constante.

Idealmente, se gostaria que as operações de estruturas de dados executassem em tempos de execução proporcional as funções constante ou logarítmica e seria desejável que os algoritmos executassem em tempo linear ou n-log-n. Algoritmos com tempos de execução quadráticos ou cúbicos são pouco práticos, mas algoritmos com tempos de execução exponenciais são impraticáveis a não ser para pequenas entradas. O gráfico das sete funções pode ser visto na Figura 4.2.



**Figura 4.2** Taxas de crescimento para as sete funções fundamentais usadas em análise de algoritmos. Usa-se a base  $a = 2$  para a função exponencial. As funções são traçadas em um quadro di-log para comparar as taxas de crescimento principalmente pelas inclinações. Mesmo assim, para mostrar todos os seus valores no gráfico, a função exponencial cresce muito rapidamente. Também se usa notação científica para os números, onde  $aE+b$  denota  $a10^b$ .

As funções de arredondamento para cima e arredondamento para baixo

Um comentário adicional em relação às funções vistas se aplica. O valor de um logaritmo tipicamente não é um inteiro, enquanto que o tempo de execução de um algoritmo é normalmente expresso em termos de quantidades inteiras, tais como a quantidade de operações executadas. Assim, a análise de um algoritmo pode algumas vezes envolver o uso das funções **arredondamento para cima** e **arredondamento para baixo**, que são definidas, respectivamente, como segue:

- $\lfloor x \rfloor$  = ao maior inteiro menor ou igual a  $x$ .
- $\lceil x \rceil$  = a menor inteiro maior ou igual a  $x$ .

## 4.2 Análise de algoritmos

Em uma história clássica, o famoso matemático Arquimedes foi chamado para determinar se uma coroa de ouro solicitada pelo rei era de ouro puro e não com parte de prata, como um informante estava apregoando. Arquimedes descobriu uma maneira de fazer esta análise enquanto entrava em uma banheira (grega). Ele notou que a água que espirrava para fora da banheira era proporcional a ele que estava entrando. Percebendo as implicações deste fato, ele imediatamente saiu da banheira e correu nu pela cidade gritando: “Eureka, eureka!”, pois tinha descoberto uma ferramenta de análise (deslocamento), que combinada com uma simples escala, podia determinar se a coroa do rei era boa ou não. Isto é, Arquimedes podia mergulhar a coroa e uma massa com o mesmo peso de ouro em uma bacia com água e verificar se ambos deslocavam a mesma quantidade. Esta descoberta foi uma infelicidade para o ourives, entretanto, porque quando Arquimedes fez sua análise, a coroa deslocou mais água que a massa de ouro com o mesmo peso, indicando que a coroa não era, de fato, de ouro puro.

Neste livro o interesse está no projeto de “bons” algoritmos e estruturas de dados. De uma forma simples, uma **estrutura de dados** é uma forma sistemática de organizar e acessar dados, e um **algoritmo** é um procedimento passo a passo para executar alguma tarefa em tempo finito. Estes conceitos são fundamentais para computação, mas para ser capaz de classificar uma estrutura de dados ou algoritmo como sendo “bom”, são necessárias formas de analisar os mesmos.

A ferramenta principal de análise que será usada neste livro envolve a caracterização dos tempos de execução dos algoritmos e das operações sobre as estruturas de dados, sendo que o

espaço utilizado também é importante. Tempo de execução é uma medida natural de “qualidade”, uma vez que o tempo é um recurso precioso – soluções computacionais devem executar o mais rápido possível.

Normalmente, o tempo de execução de um algoritmo ou método de estrutura de dados cresce com o tamanho da entrada, embora, possa variar para diferentes entradas do mesmo tamanho. Além disso, o tempo de execução pode ser afetado pelo ambiente de hardware (reflexo do processador, velocidade do clock, memória, disco, etc) e de software (reflexo do sistema operacional, linguagem de programação, compilador, interpretador etc) no qual o algoritmo é implementado, compilado e executado. Se todos os demais fatores forem iguais, o tempo de execução do mesmo algoritmo sobre o mesmo conjunto de entradas será menor se o computador tiver um processador mais rápido ou se a implementação foi feita em um programa compilado em código nativo da máquina em vez de uma execução interpretada em uma máquina virtual. Todavia, apesar das possibilidades de variação que se originam em diferentes fatores de ambiente, deseja-se focar no relacionamento entre o tempo de execução de um algoritmo e o tamanho da entrada.

Deseja-se caracterizar o tempo de execução de um algoritmo como função do tamanho da entrada. Mas qual é a maneira adequada de medir isso?

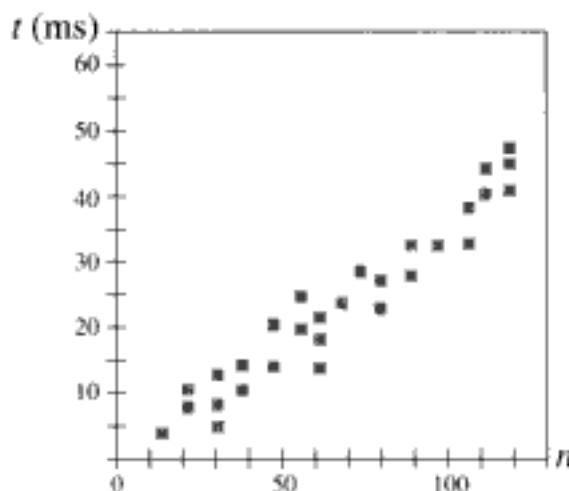
#### 4.2.1 Estudos experimentais

Se um algoritmo foi implementado, pode-se estudar seu tempo de execução, executando o mesmo sobre diferentes conjuntos de entradas e armazenando o tempo real gasto em cada execução. Felizmente, tais medidas podem ser feitas de forma bastante precisa, usando chamadas do sistema incluídas na linguagem ou no sistema operacional (por exemplo, usando o método `System.currentTimeMillis()` ou chamando o ambiente de execução com o perfil habilitado). Este teste atribui um tempo de execução para um tamanho de entrada específico, mas se o interesse for determinar a dependência geral do tempo de execução sobre o tamanho da entrada. Para determinar esta dependência, devem-se executar vários experimentos sobre diferentes conjuntos de entrada, com diferentes tamanhos. Então, se podem visualizar os resultados destes experimentos plotando a performance de cada execução do algoritmo como um ponto com coordenada  $x$  igual ao tamanho da entrada,  $n$ , e com a coordenada  $y$  igual ao tempo de execução,  $t$ . (Ver a Figura 4.3.) A partir desta visualização e dos dados que a suporta, pode-se fazer uma análise estatística que procura ajustar a melhor função para o tamanho dos dados experimentais. Para ser mais clara, esta análise requer que se escolham boas amostras e testes suficientes para que possam ser capazes de fazer afirmações estatísticas razoáveis sobre o tempo de execução do algoritmo.

Apesar dos estudos experimentais sobre os tempos de execução serem úteis, eles têm três grandes limitações:

- Experimentos só podem ser feitos sobre um conjunto limitado de entradas de teste; consequentemente, são deixados de fora os tempos de execução das entradas não incluídas nos experimentos (e estas entradas podem ser importantes).
- É difícil comparar os tempos de execução de dois algoritmos, a menos que os experimentos sejam executados nos mesmos ambientes de hardware e software.
- É necessário implementar e executar o algoritmo de maneira a estudar seu tempo de execução experimentalmente.

Este último requisito é óbvio, mas provavelmente é o aspecto que mais consome tempo na execução de uma análise experimental de um algoritmo. As outras limitações também impõem barreiras sérias, é claro. Assim, idealmente deve-se dispor de uma ferramenta de análise que permita evitar a execução de experimentos.



**Figura 4.3** Resultados de um estudo experimental sobre o tempo de execução de um algoritmo. Um ponto de coordenadas  $(n, t)$  indica que sobre uma entrada de tamanho  $n$ , o tempo de execução do algoritmo é  $t$  milissegundos (ms).

No restante deste capítulo, desenvolve-se uma maneira geral de analisar os tempos de execução de algoritmos que:

- considera todas as entradas possíveis;
- permite que se avalie a eficiência relativa de quaisquer dois algoritmos de uma forma independente dos ambientes de hardware e software;
- pode ser executada estudando-se descrições de alto-nível de algoritmos sem ter de implementá-lo ou executar experimentos.

Esta metodologia visa associar, com cada algoritmo, uma função  $f(n)$  que caracteriza o tempo de execução do algoritmo como uma função do tamanho da entrada  $n$ . Funções típicas que serão encontradas incluem as sete funções mencionadas anteriormente neste capítulo.

#### 4.2.2 Operações primitivas

Como já foi visto, a análise experimental é importante, mas tem suas limitações. Desejando-se analisar um algoritmo em particular sem realizar experimentos para medir seu tempo de execução, pode-se fazer uma análise diretamente sobre o pseudocódigo de alto nível. Define-se um conjunto de **operações primitivas** como as que seguem:

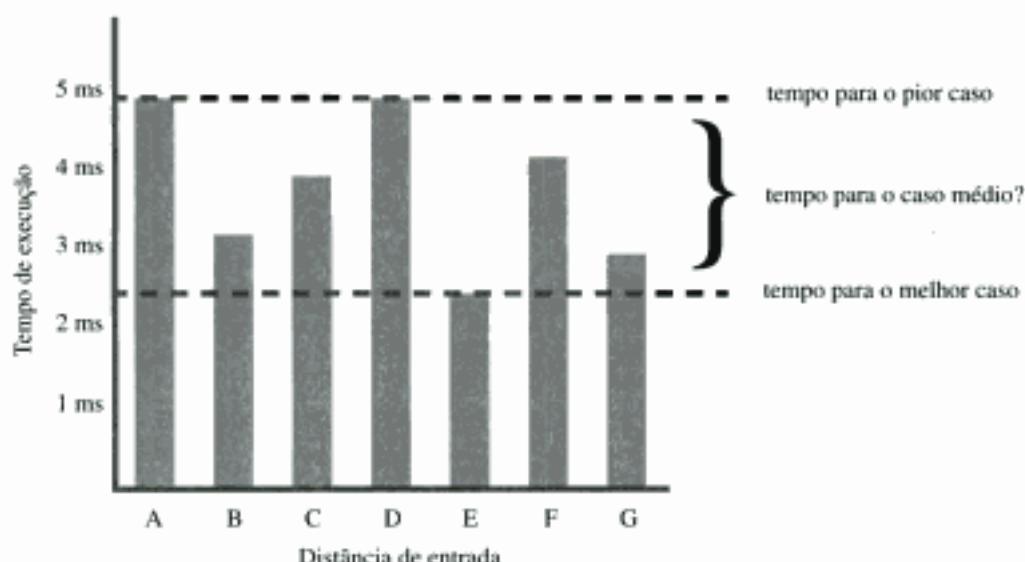
- atribuição de valores a variáveis;
- chamadas de métodos;
- operações aritméticas (por exemplo, adição de dois números);
- comparação de dois números;
- acesso a um arranjo;
- seguir uma referência para um objeto;
- retorno de um método.

#### Contando operações primitivas

Mais especificamente, uma operação primitiva corresponde a uma instrução de baixo nível com um tempo de execução constante. Em vez de tentar determinar o tempo de execução específico de cada operação primitiva, simplesmente **conta-se** quantas operações primitivas são executadas e usa-se este número  $t$  como uma estimativa do tempo de execução do algoritmo.

Esta contagem de operações está relacionada com o tempo de execução em um computador específico, pois cada operação corresponde a uma instrução realizada em tempo constante e existe um número fixo de operações primitivas. Nesta abordagem, assume-se implicitamente que os tempos de execução de operações primitivas diferentes serão similares. Assim, o número  $t$  de operações primitivas que um algoritmo realiza será proporcional ao tempo de execução daquele algoritmo.

Um algoritmo pode executar mais rapidamente sobre algumas entradas do que sobre outras do mesmo tamanho. Assim, deseja-se expressar o tempo de execução de um algoritmo como uma função do tamanho da entrada obtido pela média de todas as possíveis entradas do mesmo tamanho. Infelizmente, esse tipo de análise do *caso médio* costuma ser desafiadora. Ela requer a determinação da distribuição de probabilidade do conjunto de entrada, o que normalmente não é tarefa simples. A Figura 4.4 mostra de maneira esquemática como, dependendo da distribuição de entrada, o tempo de execução de um algoritmo pode estar em qualquer ponto entre o tempo para o pior caso e o tempo para o melhor caso. Por exemplo, o que acontece se as entradas forem apenas dos tipos "A" e "D"?



**Figura 4.4** Diferença entre os tempos para o melhor e o pior caso. Cada barra representa o tempo de execução de um algoritmo sobre uma entrada diferente.

### Focando no pior caso

A análise do caso médio normalmente requer que se calcule os tempos de execução esperados com base em uma distribuição de entrada, o que, em geral, envolve teoria de probabilidade sofisticada. Em função disso, no restante deste livro, a menos que seja especificado, no entanto, os tempos de execução serão caracterizados em termos do *pior caso*, como função do tamanho da entrada,  $n$ , do algoritmo.

A análise do pior caso é muito mais fácil do que a análise do caso médio, pois requer apenas a habilidade de identificar a entrada do pior caso, o que normalmente é simples. Além disso, tipicamente, essa abordagem conduz a algoritmos melhores. O padrão é que para um algoritmo executar bem no pior caso, é necessário que ele execute melhor para as demais entradas. Isto é, projetar para o pior caso conduz a algoritmos com mais "músculos", assim como um especialista em trilhas, que sempre pratica subindo um plano inclinado.

### 4.2.3 Notação assintótica

Em geral, cada passo em uma descrição em pseudocódigo ou implementação em linguagem de alto nível corresponde a um pequeno número de operações primitivas (exceto para chamadas de

métodos, naturalmente). Assim, podemos realizar uma análise simplificada de um algoritmo escrito em pseudocódigo que estima o número de operações primitivas executadas, exceto por um fator constante, contando os passos do pseudocódigo (mas deve-se ter cuidado uma vez que uma linha de pseudocódigo pode denotar vários passos em alguns casos).

### Simplificando a análise

Na análise de algoritmos, é importante concentrar-se na taxa de crescimento do tempo de execução como uma função do tamanho da entrada  $n$ , obtendo-se um quadro geral do comportamento, em vez de concentrar-se nos detalhes menores. Frequentemente, basta saber que o tempo de execução de um algoritmo como `arrayMax`, apresentado na Seção 1.9.2, **cresce proporcionalmente** a  $n$ , com o verdadeiro tempo de execução sendo  $n$  vezes um fator constante que depende de um computador específico.

Estrutura de dados e algoritmos serão analisados usando-se uma notação matemática para funções que desconsidera fatores constantes. Desta forma, o tempo de execução de um algoritmo será caracterizado usando funções que mapeiam o tamanho da entrada,  $n$ , para valores que correspondem a um fator principal que determina a taxa de crescimento em termos de  $n$ . Entretanto, não será definido formalmente o que  $n$  significa; em vez disso, deixar-se-á que  $n$  se refira a uma medida do tamanho da entrada, que pode ser definida de forma diferente para cada algoritmo que está sendo analisado. Esta abordagem permite focar a atenção nos aspectos gerais da função de tempo de execução. Além disso, a mesma abordagem permite caracterizar o espaço gasto com estruturas de dados e algoritmos, onde se define **espaço gasto** como a quantidade total de células de memória utilizadas.

### Notação $O$

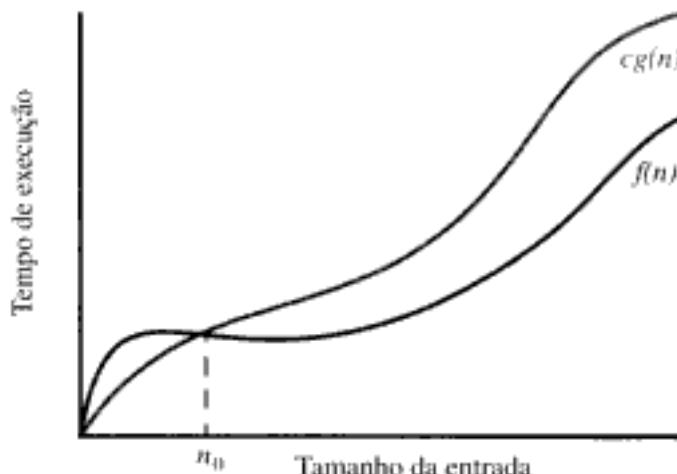
Sejam  $f(n)$  e  $g(n)$  funções mapeando inteiros não-negativos em números reais. Diz-se que  $f(n)$  é  $O(g(n))$  se existe uma constante real  $c > 0$  e uma constante inteira  $n_0 \geq 1$  tais que

$$f(n) \leq cg(n) \quad \text{para todo inteiro } n \geq n_0.$$

Esta definição é geralmente chamada de notação  $O$ , pois geralmente se diz “ $f(n)$  é  $O$  de  $g(n)$ .” Outra opção é dizer que “ $f(n)$  é **da ordem** de  $g(n)$ .” (Esta definição é ilustrada na Figura 4.5.)

**Exemplo 4.6** A função  $8n - 2$  é  $O(n)$ .

**Justificativa** Pela definição da notação  $O$ , é necessário encontrar uma constante  $c > 0$  e uma constante inteira  $n_0 \geq 1$  tais que  $8n - 2 \leq cn$  para todo inteiro  $n \geq n_0$ . É fácil perceber que uma escolha poderia ser  $c = 8$  e  $n_0 = 1$ . De fato, esta é uma das infinitas escolhas possíveis porque



**Figura 4.5** Ilustrando a notação  $O$ . A função  $f(n)$  é  $O(g(n))$ , pois  $f(n) \leq c \cdot g(n)$  quando  $n \geq n_0$ .

qualquer número real maior ou igual a 8 será uma escolha possível para  $c$  e qualquer inteiro maior ou igual a 1 é uma escolha possível para  $n_0$ . ■

A notação  $O$  permite dizer que uma função de  $f(n)$  é “menor que ou igual a” outra função  $g(n)$  até um fator constante e de uma maneira **assintótica** à medida que  $n$  cresce para infinito. Esta habilidade vem do fato de que a definição usa “ $\leq$ ” para comparar  $f(n)$  com  $g(n)$  vezes uma constante,  $c$ , para o caso assintótico quando  $n \geq n_0$ .

### Caracterizando tempos de execução usando a notação $O$

A notação  $O$  é usada largamente para caracterizar o tempo de execução e limites espaciais em função de um parâmetro  $n$  que varia de problema para problema, mas é geralmente definido como uma medida escolhida do tamanho do mesmo. Por exemplo, se for necessário encontrar o maior elemento em um arranjo de inteiros, como no algoritmo arrayMax, deve-se fazer  $n$  representar o número de elementos no arranjo. Usando a notação  $O$  pode-se escrever a seguinte afirmação matematicamente precisa sobre o tempo de execução do algoritmo arrayMax em **qualquer** computador.

**Proposição 4.7** *O algoritmo arrayMax para determinar o maior elemento de um arranjo de  $n$  inteiros executa em tempo  $O(n)$ .*

**Justificativa** O número de operações primitivas executadas pelo algoritmo arrayMax em cada iteração é constante. Conseqüentemente, como cada operação primitiva executa em um tempo constante, pode-se dizer que o tempo de execução do algoritmo arrayMax para uma entrada de tamanho  $n$  é no máximo uma constante, vezes  $n$ , isto é, pode-se concluir que o tempo de execução do algoritmo arrayMax é  $O(n)$ . ■

### Algumas propriedades da notação $O$

A notação  $O$  permite que se ignorem os fatores constantes e os termos de menor ordem, mantendo o foco nos principais componentes da função que afetam seu crescimento.

**Exemplo 4.8**  $5n^4 + 3n^3 + 2n^2 + 4n + 1$  é  $O(n^4)$ .

**Justificativa** observe que  $5n^4 + 3n^3 + 2n^2 + 4n + 1 \leq (5 + 3 + 2 + 4 + 1)n^4 = cn^4$ , para  $c = 15$ , quando  $n \geq n_0 = 1$ . ■

Na verdade, pode-se caracterizar a taxa de crescimento de qualquer função polinomial.

**Proposição 4.9** *Se  $f(n)$  é um polinômio de grau  $d$ , isto é,*

$$f(n) = a_0 + a_1 n + \cdots + a_d n^d,$$

e  $a_d > 0$ , então  $f(n)$  é  $O(nd)$ .

**Justificativa** Nota-se que, para  $n \geq 1$ , tem-se que  $1 \leq n \leq n^2 \leq \dots \leq n^d$ ; consequentemente,

$$a_0 + a_1 n + a_2 n^2 + \cdots + a_d n^d \leq (a_0 + a_1 + a_2 + \cdots + a_d)n^d.$$

Em função disso, pode-se mostrar que  $f(n)$  é  $O(n^d)$  pela definição  $c = a_0 + a_1 + \cdots + a_d$  e  $n_0 = 1$ . ■

Assim, o termo de mais alto grau em um polinômio é o termo que determina a taxa de crescimento assintótico do polinômio. Propriedades adicionais da notação  $O$  são consideradas nos exercícios. Entretanto, serão analisados mais alguns exemplos, focando em combinações das sete funções fundamentais usadas em projeto de algoritmos.

**Exemplo 4.10**  $5n^2 + 3n \log n + 2n + 5$  é  $O(n^2)$ .

**Justificativa**  $5n^2 + 3n \log n + 2n + 5 \leq (5 + 3 + 2 + 5)n^2 = cn^2$ , para  $c = 15$ , quando  $n \geq n_0 = 2$  (observa-se que  $n \log n$  é zero para  $n = 1$ ). ■

**Exemplo 4.11**  $20n^3 + 10n \log n + 5$  é  $O(n^3)$ .

**Justificativa**  $20n^3 + 10n \log n + 5 \leq 35n^3$ , para  $n \geq 1$ . ■

**Exemplo 4.12**  $3 \log n + 2$  é  $O(\log n)$ .

**Justificativa**  $3 \log n + 2 \leq 5 \log n$  para  $n \geq 2$ . Observa-se que  $\log n$  é zero para  $n = 1$ . Por isso usa-se,  $n \geq n_0 = 2$  neste caso. ■

**Exemplo 4.13**  $2^{n+2}$  é  $O(2^n)$ .

**Justificativa**  $2^{n+2} = 2^n 2^2 = 4 \cdot 2^n$ ; consequentemente, pode-se usar  $c = 4$  e  $n_0 = 1$  neste caso. ■

**Exemplo 4.14**  $2n + 100 \log n$  é  $O(n)$ .

**Justificativa**  $2n + 100 \log n \leq 102n$ , para  $n \geq n_0 = 2$ ; consequentemente, pode-se usar  $c = 102$  neste caso. ■

### Caracterizando funções em termos mais simples

Em geral, deve-se usar a notação  $O$  para caracterizar uma função tão detalhadamente quanto possível. Mesmo sendo verdade que a função  $f(n) = 4n^3 + 3n^2$  é  $O(n^3)$  ou mesmo  $O(n^4)$ , é mais preciso dizer que  $f(n)$  é  $O(n^3)$ . Considere, por analogia, um cenário em que um viajante com fome dirige por uma estrada do interior e passa por um fazendeiro que está indo para casa voltando do mercado. Quando o viajante pergunta quanto ele ainda tem de dirigir até achar comida, pode ser correto o fazendeiro responder: "Com certeza são menos de doze horas". No entanto, é muito mais preciso (e útil) para ele dizer: "Você encontra um mercado dirigindo mais alguns minutos por esta estrada". Da mesma forma, na notação  $O$ , deve-se fazer um esforço para na medida do possível dizer toda a verdade.

Também é considerado de mau gosto incluir fatores constantes de termos de menor ordem na notação  $O$ . Por exemplo, não é elegante dizer que a função  $2n^2$  é  $O(4n^2 + 6n \log n)$ , ainda que isso esteja perfeitamente correto. Deve-se fazer um esforço, porém, para descrever a função  $O$  em *termos simples*.

As sete funções listadas na Seção 4.1 são as funções mais comumente usadas em conjunto com a notação  $O$  para caracterizar os tempos de execução e consumo de memória dos algoritmos. Na verdade, comumente usam-se os nomes destas funções para referenciar o tempo de execução dos algoritmos que elas caracterizam. Assim, por exemplo, pode-se dizer que um algoritmo que executa no pior caso em tempo  $4n^2 + n \log n$  como um algoritmo de **tempo quadrático**, uma vez que ela executa em tempo  $O(n^2)$ . Da mesma forma, um algoritmo executando em um tempo máximo  $5n + 20 \log n + 4$  será dito um algoritmo de **tempo linear**.

### Notação omega

Assim como a notação  $O$  fornece uma maneira assintótica de dizer que uma função é "menor que ou igual a" outra função, a notação a seguir fornece uma maneira assintótica de dizer que uma função cresce a uma taxa que é "maior ou igual" a de uma outra função.

Sejam  $f(n)$  e  $g(n)$  funções mapeando números inteiros em números reais. Diz-se que  $f(n)$  é  $\Omega(g(n))$  (pronuncia-se "f(n) é ômega de g(n)") se  $g(n)$  é  $O(f(n))$ ; ou seja, se existe uma constante  $c > 0$  e uma constante inteira  $n_0 \geq 1$  tais que

$$f(n) \geq cg(n) \quad \text{para } n \geq n_0.$$

Esta definição nos permite dizer que uma função é assintoticamente maior que ou igual a outra, exceto por um fator constante.

**Exemplo 4.15**  $3n \log n + 2n$  é  $\Omega(n \log n)$ .

**Justificativa**  $3n \log n + 2 \geq 3n \log n$  para  $n \geq 2$ . ■

### Notação theta

Além disso, existe uma notação que permite dizer que duas funções crescem à mesma taxa, até fatores constantes. Diz-se que  $f(n) \in \Theta(g(n))$  (pronuncia-se “ $f(n)$  é theta de  $g(n)$ ”) se  $f(n) \in O(g(n))$  e  $f(n) \in \Omega(g(n))$ ; ou seja, existem constantes reais  $c' > 0$  e  $c'' > 0$  e uma constante inteira  $n_0 \geq 1$  tais que

$$c'g(n) \leq f(n) \leq c''g(n), \quad \text{para } n \geq n_0.$$

**Exemplo 4.16**  $3n \log n \leq 4n + 5 \log n \in \Theta(n \log n)$ .

**Justificativa**  $3n \log n \leq 3n \log n + 4n + 5 \log n \leq (3+4+5)n \log n$  para  $n \geq 2$ . ■

### 4.2.4 Análise assintótica

Suponha que dois algoritmos podem resolver o mesmo problema: um algoritmo  $A$  que tem um tempo de execução  $O(n)$  e um algoritmo  $B$  com tempo de execução  $O(n^2)$ . Qual deles é melhor? Sabe-se que  $n \in O(n^2)$ , e isso implica que o algoritmo  $A$  é **assintoticamente melhor** do que o algoritmo  $B$ , embora para algum dado valor (pequeno) de  $n$  seja possível que  $B$  tenha um tempo de execução menor do que  $A$ .

Pode-se usar a notação  $O$  para ordenar classes de funções por seu crescimento assintótico. As sete funções estão ordenadas por sua taxa de crescimento na seqüência que segue, isto é, se uma função  $f(n)$  precede uma função  $g(n)$  na seqüência, então  $f(n) \in O(g(n))$ :

$$1 \quad \log n \quad n \log n \quad n^2 \quad n^3 \quad 2^n.$$

A Figura 4.2 apresenta as taxas de crescimento de algumas funções importantes.

$n$	$\log n$	$n$	$n \log n$	$n^2$	$n^3$	$2^n$
8	3	8	24	64	512	256
16	4	16	64	256	4.096	65.536
32	5	32	160	1.024	32.768	4.294.967.296
64	6	64	384	4.096	262.144	$1.84 \times 10^{19}$
128	7	128	896	16.384	2.097.152	$3.40 \times 10^{38}$
256	8	256	2.048	65.536	16.777.216	$1.15 \times 10^{77}$
512	9	512	4.608	262.144	134.217.728	$1.34 \times 10^{154}$

**Tabela 4.2** Valores selecionados de funções fundamentais para análise de algoritmos

Pode-se demonstrar a importância da notação assintótica na Tabela 4.3. Essa tabela explora o maior tamanho permitido para os dados de entrada que são processados por um algoritmo em 1 segundo, 1 minuto e 1 hora. Ela mostra a importância do bom projeto de um algoritmo, pois um algoritmo assintoticamente demorado é facilmente batido para problemas grandes por um algoritmo com tempo assintoticamente mais rápido, mesmo que o fator constante do algoritmo assintoticamente mais rápido seja pior.

Tempo de execução(s)	Tamanho máximo de problemas(n)		
	1 segundo	1 minuto	1 hora
$400n$	2.500	150.000	9.000.000
$2n^2$	707	5.477	42.426
$2^n$	19	25	31

**Tabela 4.3** Tamanho máximo de problemas que podem ser resolvidos em um segundo, um minuto e uma hora para vários tempos de execução medidos em microsegundos.

A importância do bom projeto de algoritmos, entretanto, vai além do que pode ser resolvido eficientemente em um dado computador. Como mostrado na Tabela 4.4, mesmo se o hardware for drasticamente acelerado, ainda assim não se pode superar o problema representado por um algoritmo assintoticamente lento. A tabela mostra o novo tamanho máximo de problema que pode ser resolvido em um computador 256 vezes mais rápido do que o anterior.

Tempo de execução	Novo tamanho máximo de problema
$400n$	$256m$
$2n^2$	$16m$
$2^n$	$m + 8$

**Tabela 4.4** Aumento no tamanho máximo do problema que pode ser resolvido em um dado tempo usando-se um computador 256 vezes mais rápido que o anterior. Cada entrada é dada em função de  $m$ , o tamanho máximo do problema dado anteriormente.

#### 4.2.5 Usando a notação $O$

Após estudar a notação  $O$  para analisar algoritmos, serão discutidos brevemente alguns tópicos relacionados a seu uso. Considera-se pouco elegante dizer “ $f(n) \leq O(g(n))$ ”, já que a notação  $O$  por si mesma transmite a idéia de “menor ou igual”. Da mesma forma, embora comum, não é correto escrever “ $f(n) = O(g(n))$ ” (com a relação de “=” mantendo seu sentido usual) já que não faz sentido a declaração “ $O(g(n)) = f(n)$ ”. Além disso, é errado dizer “ $f(n) \geq O(g(n))$ ” ou “ $f(n) > O(g(n))$ ”, pois  $g(n)$  na notação  $O$  expressa um limite superior para  $f(n)$ . O mais apropriado é dizer

“ $f(n) \in O(g(n))$ ”.

Para o leitor com maior pendor para a matemática, também é correto dizer,

“ $f(n) \in O(g(n))$ ,”

pois a notação  $O$  denota, tecnicamente, um conjunto de funções. Neste livro, as sentenças usando notação  $O$  serão apresentadas da forma “ $f(n) \in O(g(n))$ ”. Mesmo sob esta interpretação, existe considerável liberdade para se usar operações aritméticas com a notação  $O$ , e com esta liberdade exige-se uma certa dose de responsabilidade.

#### Palavras de cautela

Algumas palavras de cautela sobre a notação assintótica podem ser apresentadas neste ponto. Primeiro, observa-se que o uso da notação  $O$  e suas parentes pode ser um pouco confusa se os fatores constantes que elas “escondem” for muito alto. Por exemplo, enquanto é verdade que a função  $10^{100} n$  é  $O(n)$ , se este for o tempo de execução de um algoritmo sendo comparado a um outro cujo tempo de execução é  $10n \log n$ , será preferido o algoritmo com tempo de execução  $O(n \log n)$ , mesmo que o primeiro algoritmo seja linear, e, portanto, assintoticamente mais rápido. Esta preferência se justifica pelo fator constante,  $10^{100}$ , que é chamado de “um googol”, e que muitos astrônomos acreditam ser o limite para o número de átomos no universo observável. Assim, é improvável que exista algum problema do mundo real com este tamanho de entrada. Mesmo assim, usando a notação  $O$ , deve-se estar consciente dos fatores constantes e dos termos de mais baixa ordem que estão “escondidos”.

A observação anterior conduz à discussão do que seja um algoritmo “rápido”. De forma geral, qualquer algoritmo rodando em tempo  $O(n \log n)$  (e com um fator constante razoável) pode ser considerado eficiente. Mesmo um método com tempo  $O(n^2)$  pode ser suficientemente rápido em alguns contextos, ou seja, quando  $n$  é pequeno. Por outro lado, um algoritmo rodando em tempo  $O(2^n)$  não pode nunca ser considerado eficiente.

## Tempos de execução exponenciais

Existe uma história famosa sobre o inventor do jogo de xadrez. Ele pediu que seu rei lhe pagasse um grão de arroz pela primeira casa do tabuleiro, dois pela segunda, quatro pela terceira, oito pela quarta, e assim por diante. É um bom exercício de programação escrever um pequeno programa que calcule o número de grãos de arroz que o rei teria de pagar. De fato, qualquer programa em Java escrito para calcular este número usando uma variável inteira causaria um erro de overflow (embora provavelmente a máquina virtual Java não reclamasse). Para representar esse número exatamente como um inteiro, é preciso usar a classe `BigInteger`.

Na medida em que é preciso diferenciar algoritmos eficientes e ineficientes, é natural fazer esta distinção entre os algoritmos que rodam em tempo polinomial e aqueles que requerem tempo exponencial. Ou seja, faz-se a distinção entre algoritmos que rodam em tempo  $O(n^c)$  para alguma constante  $c > 1$  e aqueles cujo tempo de execução é  $O(b^n)$  para alguma constante  $b > 1$ . Assim como outras noções discutidas nesta seção, esta também deve ser acolhida com certa cautela, pois um algoritmo rodando em tempo  $O(n^{100})$  provavelmente não deveria ser considerado muito eficiente. Mesmo assim, a distinção entre algoritmos de tempo polinomial e algoritmos de tempo exponencial é considerada uma medida robusta de tratabilidade.

Resumindo, as notações  $O$ ,  $\Omega$  e  $\Theta$  fornecem uma linguagem conveniente para a análise de estruturas de dados e algoritmos. Como mencionado anteriormente, estas noções são convenientes porque permitem a concentração nos aspectos gerais, em vez dos detalhes.

## Dois exemplos de análise assintótica de algoritmos

Encerra-se esta seção analisando dois algoritmos que resolvem o mesmo problema, mas têm tempos de execução bastante diferentes. O problema em questão é fazer o cálculo das *médias prefixadas* de uma seqüência de números. Ou seja, se dispondo de um arranjo  $X$  armazenando  $n$  números, deseja-se compor um arranjo  $A$  tal que  $A[i]$  seja a média dos elementos  $X[0], \dots, X[i]$  para  $i = 0, \dots, n - 1$ . Ou seja,

$$A[i] = \frac{\sum_{j=0}^i X[j]}{i+1}.$$

As médias prefixadas tem várias aplicações em economia e estatística. Por exemplo, dados os retornos anuais de um fundo de investimento, pode-se avaliar o retorno médio anual do fundo no último ano e nos últimos três, cinco ou dez anos. Da mesma forma, dados os logs de uso diário da Web, um gerenciador de sites pode querer rastrear a tendência da média de uso em diferentes períodos de tempo.

### Um algoritmo de tempo quadrático

O primeiro algoritmo para o problema das médias prefixadas, chamado de `prefixAverages1`, é mostrado no Trecho de código 4.1. Ele calcula cada elemento de  $A$  separadamente, de acordo com a definição.

**Algoritmo** `prefixAverages1`( $X$ ):

**Entrada:** um arranjo  $X$  com  $n$  elementos.

**Saída:** um arranjo  $A$  com  $n$  elementos tal que  $A[i]$  é a média de  $X[0], \dots, X[i]$ .

Seja  $A$  um arranjo de  $n$  números.

**para**  $i \leftarrow 0$  até  $n - 1$  **faca**

$a \leftarrow 0$

**para**  $j \leftarrow 0$  até  $i$  **faca**

```

 $a \leftarrow a + X[j]$ 
 $A[i] \leftarrow a/(i+1)$ 
retorne arranjo A

```

#### Trecho de código 4.1 Algoritmo prefixAverages1.

Esta é a análise do algoritmo prefixAverages1.

- Inicializar o arranjo  $A$  no início e retorná-lo ao final são ações que podem ser feitas com um número constante de operações primitivas por elemento de  $A$ , e custa tempo  $O(n)$ .
- Existem dois laços **para** aninhados, controlados pelos contadores  $i$  e  $j$ . O corpo do laço externo (controlado por  $i$ ) é executado  $n$  vezes para valores  $i = 0, \dots, n-1$ . Assim, os comandos  $a = 0$  e  $A[i] = a/(i+1)$  são executados  $n$  vezes cada. Isso implica que esses dois comandos, mais o incremento e teste do contador  $i$ , contribuem com um número de operações primitivas proporcional a  $n$ , ou seja,  $O(n)$ .
- O corpo do laço interno (controlado por  $j$ ) é executado  $i+1$  vez, dependendo do contador  $i$  do laço externo. Assim, o comando  $a = a + X[j]$  no laço interno é executado  $1+2+3+\dots+n$  vezes. Pela Proposição 4.3, sabe-se que  $1+2+3+\dots+n = n(n+1)/2$ , o que implica que o comando do laço interno contribui com tempo  $O(n^2)$ . Um argumento similar pode ser feito para as operações primitivas associadas ao incremento e teste de  $j$ , que também custam tempo  $O(n^2)$ .

O tempo de execução de prefixAverages1 é dado pela soma dos três termos. O primeiro e segundo termos são  $O(n)$  e o terceiro é  $O(n^2)$ . Aplicando a Proposição 4.9, tem-se que o tempo de execução de prefixAverages1 é  $O(n^2)$ .

#### Um algoritmo de tempo linear

Para calcular médias prefixadas mais eficientemente, pode-se observar que duas médias consecutivas  $A[i-1]$  e  $A[i]$  são similares:

$$\begin{aligned} A[i-1] &= (X[0] + X[1] + \dots + X[i-1])/i \\ A[i] &= (X[0] + X[1] + \dots + X[i-1] + X[i])/(i+1). \end{aligned}$$

Denotando com  $S_i$  a **soma prefixada**  $X[0] + X[1] + \dots + X[i]$ , as médias prefixadas podem ser calculadas como sendo  $A[i] = S_i/(i+1)$ . É fácil manter o controle da soma prefixada corrente enquanto se faz a varredura do arranjo  $X$  em um laço. Está-se em condição de apresentar o algoritmo prefixAverages2 no Trecho de código 4.2.

#### Algoritmo prefixAverages2( $X$ ):

**Entrada:** um arranjo  $X$  com  $n$  elementos.

**Saída:** um arranjo  $A$  com  $n$  elementos tal que  $A[i]$  é a média de  $X[0], \dots, X[i]$ .

Seja  $A$  um arranjo de  $n$  números.

$s \leftarrow 0$

**para**  $i \leftarrow 0$  até  $n-1$  **faça**

$s \leftarrow s + X[i]$

$A[i] \leftarrow s/(i+1)$

**retorne** arranjo  $A$

#### Trecho de código 4.2 Algoritmo prefixAverages2.

Segue a análise do tempo de execução do algoritmo prefixAverages2:

- Inicializar o arranjo  $A$  no início e fim pode ser feito com um número constante de operações primitivas por elemento e leva tempo  $O(n)$ .

- Inicializar a variável  $s$  no início, leva tempo  $O(1)$ .
- Há um simples laço **para** que é controlado pelo contador  $i$ . O corpo do laço é executado  $n$  vezes para  $i = 0, \dots, n - 1$ . Assim, os comandos  $s = s + X[i]$  e  $A[i] = s/(i + 1)$  são executados  $n$  vezes, cada. Isso implica que estes dois comandos, mais o incremento e teste do contador  $i$ , contribuem para um número de operações primitivas proporcionais a  $n$ , isto é,  $O(n)$  vezes.

O tempo de execução do algoritmo `prefixAverages2` é dado pela soma dos três termos. O primeiro e o terceiro termo são  $O(n)$ , e o segundo termo é  $O(1)$ . Aplicando a Proposição 4.19, tem-se que o tempo de execução de `prefixAverages2` é  $O(n)$ , muito melhor do que o tempo quadrático de `prefixAverages1`.

#### 4.2.6 Um algoritmo recursivo para calcular potência

Como um exemplo mais interessante de análise de algoritmos, será considerado o problema de aumentar um número  $x$  para um inteiro arbitrário não-negativo,  $n$ . Isto é, deseja-se calcular a **função potência**  $p(x, n)$ , definida como  $p(x, n) = x^n$ . Esta função tem uma definição recursiva, baseada em recursão linear:

$$p(x, n) = \begin{cases} 1 & \text{se } n = 0 \\ x \cdot p(x, n-1) & \text{caso contrário} \end{cases}$$

Esta definição leva a um algoritmo recursivo que usa  $O(n)$  chamadas de métodos para calcular  $p(x, n)$ . Entretanto, pode-se calcular a função potência de forma muito mais rápida, usando a seguinte definição alternativa, também baseada em recursão linear, que emprega a seguinte técnica:

$$p(x, n) = \begin{cases} 1 & \text{se } n = 0 \\ x \cdot p(x, (n-1)/2)^2 & \text{se } n \text{ é ímpar} \\ p(x, n/2)^2 & \text{se } n \text{ é par} \end{cases}$$

Para demonstrar como esta definição funciona, consideram-se os seguintes exemplos:

$$\begin{aligned} 2^4 &= 2^{(4/2)2} = (2^{4/2})^2 = (2^2)^2 = 4^2 = 16 \\ 2^5 &= 2^{1+(4/2)2} = 2(2^{4/2})^2 = 2(2^2)^2 = 2(4^2) = 32 \\ 2^6 &= 2^{(6/2)2} = (2^{6/2})^2 = (2^3)^2 = 8^2 = 64 \\ 2^7 &= 2^{1+(6/2)2} = 2(2^{6/2})^2 = 2(2^3)^2 = 2(8^2) = 128. \end{aligned}$$

Esta definição sugere o algoritmo do Trecho de código 4.3.

**Algoritmo** Power( $x, n$ ):

**Entrada:** um número  $x$  e um inteiro  $n \geq 0$

**Saída:** o valor de  $x^n$

**se**  $n = 0$  **então**

**retorna** 1

**se**  $n$  é ímpar **então**

$y \leftarrow \text{Power}(x, (n-1)/2);$

**retorna**  $x \cdot y \cdot y$

**senão**

$y \leftarrow \text{Power}(x, n/2)$

**retorna**  $y \cdot y$

**Trecho de código 4.3** Calculando a função potência usando recursão linear.

Para analisar o tempo de execução do algoritmo, observa-se que cada chamada recursiva do método  $\text{Power}(x, n)$  divide o expoente,  $n$ , por dois. Conseqüentemente, existem  $O(\log n)$  chamadas recursivas, e não  $O(n)$ . Isto é, usando recursão linear e a técnica do quadrado, reduz-se o tempo de cálculo da função potência de  $O(n)$  para  $O(\log n)$ , o que é uma melhoria significativa.

## 4.3 Técnicas simples de justificativa

Algumas vezes, deseja-se fazer afirmações sobre um algoritmo, tais como mostrar que ele é correto ou executa mais rápido. Para fazer tais afirmações de forma rigorosa, deve-se usar uma argumentação matemática, justificando ou *provando* nossas afirmações. Felizmente, existem várias maneiras simples de fazer isso.

### 4.3.1 Através de exemplos

Algumas afirmações têm uma forma genérica: “Existe um elemento  $x$  no conjunto  $S$  que tem a propriedade  $P$ ”. Para justificar tal afirmação, precisa-se apenas encontrar um  $x$  em  $S$  que tenha a propriedade  $P$ . Outras afirmações são da forma: “Todo elemento  $x$  no conjunto  $S$  tem a propriedade  $P$ ”. Para indicar que esta afirmação é falsa, é necessário apenas mostrar um  $x$  do conjunto  $S$  que não tenha a propriedade  $P$ . Um tal  $x$  é chamado de *contra-exemplo*.

**Exemplo 4.17** *Um certo professor Amongus afirma que todo número da forma  $2^i - 1$  é primo, se  $i$  for maior do que 1. O professor está errado.*

**Justificativa** Para provar que o professor está errado, precisa-se achar um contra-exemplo. Felizmente não se precisa procurar muito, pois  $2^4 - 1 = 15 = 3 \cdot 5$ . ■

### 4.3.2 O ataque “contra”

Outro conjunto de técnicas envolve o uso de negações. Os dois métodos básicos são o uso de *contrapositivos* e da *contradição*. O uso do contrapositivo é como olhar em um espelho negativo: para justificar a afirmação “se  $p$  é verdade, então  $q$  é verdade”, estabelece-se a verdade da afirmação “se  $q$  não é verdade então  $p$  não é verdade”. Logicamente, essas duas afirmações são equivalentes, mas a segunda, que é a *contrapositiva* da primeira, pode ser mais fácil de demonstrar.

**Exemplo 4.18** *Sejam  $a$  e  $b$  inteiros. Se  $ab$  é par, então  $a$  é par ou  $b$  é par.*

**Justificativa** Para justificar essa afirmação, considere seu contrapositivo “se  $a$  é ímpar e  $b$  é ímpar, então  $ab$  é ímpar”. Assim, suponha  $a = 2i + 1$  e  $b = 2j + 1$  para inteiros  $i$  e  $j$ . Então  $ab = 4ij + 2i + 2j + 1 = 2(2ij + i + j) + 1$ ; Portanto,  $ab$  é ímpar. ■

Além de mostrar o uso da técnica de justificativa contrapositiva, o exemplo anterior também contém uma aplicação da *Lei de DeMorgan*. Essa lei ajuda a lidar com negações, pois ela afirma que a negação de “ $p$  ou  $q$ ” assume a forma “não  $p$  e não  $q$ ”. Da mesma forma, estabelece que a negação de uma sentença da forma “ $p$  e  $q$ ” é “não  $p$  ou não  $q$ ”.

#### Contradição

Outra técnica de justificativa por negação envolve o uso da *contradição*, que freqüentemente também envolve o uso da Lei de DeMorgan. Aplicando a técnica, fica estabelecido que uma afirmação  $q$  é verdadeira supondo primeiro que ela é falsa e mostrando que esta suposição leva a uma contradição (tal como  $2 \neq 2$  ou  $1 > 3$ ). Chegando a uma contradição, mostra-se que se  $q$

for falsa existirá uma situação inconsistente, e portanto  $q$  deve ser verdadeira. Naturalmente, para chegar a essa conclusão, deve-se ter certeza da existência de uma situação consistente ainda antes de supor que  $q$  é falsa.

**Exemplo 4.19** Sejam  $a$  e  $b$  inteiros. Se  $ab$  é ímpar, então  $a$  é ímpar e  $b$  é ímpar.

**Justificativa** Supõe-se que  $ab$  é ímpar. Deseja-se mostrar que  $a$  é ímpar e que  $b$  é ímpar. Assim, tentar-se-á obter uma contradição assumindo o oposto, ou seja, que  $a$  é par ou  $b$  é par. De fato, pode-se assumir que  $a$  é par (uma vez que o caso de  $b$  é simétrico). Então  $a = 2i$  para algum inteiro  $i$ . Portanto  $ab = (2i)b = 2(ib)$ , ou seja,  $ab$  é par. Mas isso é uma contradição;  $ab$  não pode ser simultaneamente ímpar e par. Conseqüentemente,  $a$  é ímpar e  $b$  é ímpar. ■

### 4.3.3 Indução e invariantes em laços

A maior parte das afirmações que foram feitas sobre o tempo de execução ou consumo de memória de um algoritmo dizem respeito a um parâmetro inteiro  $n$  (em geral, representando uma noção intuitiva do “tamanho” do problema). Além disso, a maior parte dessas afirmações equivalem a dizer que determinada afirmação  $q(n)$  é verdadeira “para todo  $n \geq 1$ ”. Como isso, equivale a fazer uma afirmação sobre um conjunto infinito de números: não se pode justificá-la de forma exaustiva de forma direta.

#### Indução

Entretanto, freqüentemente pode-se justificar afirmações como as acima apresentadas como verdadeiras, se for feito uso da técnica da *indução*. Esta técnica se resume em mostrar que para qualquer  $n \geq 1$  existe uma seqüência finita de implicações que inicia com um fato verdadeiro e leva à confirmação de que  $q(n)$  é verdadeiro. Especificamente, começa-se uma justificativa por indução mostrando que  $q(n)$  é verdadeiro para  $n = 1$  (e possivelmente outros valores  $n = 2, 3, \dots, k$  para alguma constante  $k$ ). A seguir, justifica-se que o “passo” indutivo é verdadeiro para  $n > k$ , ou seja, mostra-se que “se  $q(i)$  é verdadeiro para  $i < n$  então  $q(n)$  é verdadeiro”. A combinação dessas duas partes completa a justificativa por indução.

**Proposição 4.20** Considere função Fibonacci,  $F(n)$  onde se define  $F(1) = 1$ ,  $F(2) = 2$ , e  $F(n) = F(n - 1) + F(n - 2)$  para  $n \geq 2$ . (ver Seção 2.2.3) Afirma-se que  $F(n) < 2^n$ .

**Justificativa** Mostra-se que essa afirmação é correta por indução.

**Caso base:** ( $n \leq 2$ ).  $F(1) = 1 < 2 = 2^1$  e  $F(2) = 2 < 4 = 2^2$ .

**Passo da indução:** ( $n > 2$ ). Supondo que a afirmação é verdadeira para  $n' < n$ . Considere-se  $F(n)$ . Já que  $n > 2$ , então  $F(n) = F(n - 1) + F(n - 2)$ . Além disso, já que  $n - 1 < n$  e  $n - 2 < n$ , pode-se aplicar a suposição da indução (às vezes chamada de “hipótese de indução”) para implicar que  $F(n) < 2^{n-1} + 2^{n-2}$ , uma vez que

$$2^{n-1} + 2^{n-2} < 2^{n-1} + 2^{n-2} = 2 \cdot 2^{n-1} = 2^n.$$

Outro argumento indutivo será mostrado, desta vez para um fato já visto antes.

**Proposição 4.21** (que equivale à Proposição 4.3)

$$\sum_{i=1}^n i = \frac{n(n+1)}{2}.$$

**Justificativa** A justificativa será feita por indução.

**Caso base:**  $n = 1$ . É trivial, pois  $1 = n(n + 1)/2$ , se  $n = 1$ .

**Passo da indução:**  $n \geq 2$ . Supõe-se que a afirmação é verdadeira para  $n' < n$ , e considera-se  $n$ .

$$\sum_{i=1}^n i = n + \sum_{i=1}^{n-1} i.$$

Pela hipótese de indução, tem-se

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

que pode ser simplificada para

$$n + \frac{(n-1)n}{2} = \frac{2n + n^2 - n}{2} = \frac{n^2 + n}{2} = \frac{n(n+1)}{2}$$

■

Justificar algo verdadeiro para *todo*  $n \geq 1$  às vezes pode ser uma sobrecarga. É necessário lembrar, no entanto, que a técnica de indução é bastante concreta: ela mostra que, para qualquer  $n$  em particular, existe uma seqüência de implicações que se inicia com um fato verdadeiro e leva a uma verdade sobre  $n$ . Resumindo, o argumento indutivo é uma fórmula para construir uma seqüência de justificativas diretas.

### Invariante de laços

A última técnica de justificativa que será discutida nesta seção é o *laço invariante*. Para provar que uma afirmação  $S$  sobre um laço é correta, defina  $S$  como uma seqüência de afirmações menores  $S_0, S_1, \dots, S_k$ , onde:

1. a afirmação *inicial*  $S_0$  seja verdadeira antes que o laço se inicie;
2. se  $S_{i-1}$  é verdadeira antes da iteração  $i$ , então se pode mostrar que  $S_i$  será verdadeira depois que a iteração  $i$  terminar;
3. a afirmação final  $S_k$  implica que a afirmação  $S$  que se deseja provar é verdadeira.

De fato, já se viu a técnica da justificativa por invariantes de laços em operação na Seção 1.9.2 (discutindo a correção do algoritmo arrayMax), mas será visto mais um exemplo aqui. Em particular, vamos usar o método para mostrar a correção do algoritmo arrayFind, mostrado no Trecho de código 3.3, que resolve o problema de encontrar um elemento  $x$  em um arranjo  $A$ .

#### Algoritmo arrayFind( $x, A$ ):

**Entrada:** Um elemento  $x$  e um arranjo  $A$  com  $n$  elementos.

**Saída:** O índice  $i$  tal que  $x = A[i]$ , ou  $-1$  se nenhum elemento de  $A$  é igual a  $x$ .

$i \leftarrow 0$

**enquanto**  $i < n$  **faça**

**se**  $x = A[i]$  **então**

**retorna**  $i$

**senão**

$i \leftarrow i + 1$

**retorna**  $-1$

**Trecho de código 4.4** Algoritmo arrayFind para encontrar um determinado elemento em um arranjo.

Para mostrar que o algoritmo `arrayFind` está correto, define-se uma série de afirmações  $S_i$  que demonstrarão a correção do algoritmo. Especificamente, afirma-se que o seguinte fato é verdadeiro no início da iteração  $i$  do laço `enquanto`:

$$S_i: x \text{ não é igual a nenhum dos primeiros elementos } i \text{ de } A.$$

Essa afirmação é verdadeira no início da primeira iteração do laço, já que não há elementos entre o primeiro 0 de  $A$ . (diz-se que este tipo de afirmação trivial é vazia) Na iteração  $i$ , compara-se o elemento  $x$  com o elemento  $A[i]$  e retorna-se o índice  $i$  se eles forem iguais, o que é claramente correto e completa o algoritmo neste caso. Se os elementos  $x$  e  $A[i]$  não são iguais, então se encontrou mais um elemento diferente de  $x$  e incrementa-se o índice  $i$ . Assim, a afirmação  $S_i$  será verdadeira para este novo valor de  $i$ ; consequentemente, será verdadeira no início da próxima iteração. Se o laço termina sem nunca retornar um índice em  $A$ , então provavelmente tem-se  $i = n$ . Ou seja,  $S_n$  é verdadeiro – não há elementos em  $A$  que sejam iguais a  $x$ . Portanto, o algoritmo está correto ao retornar o valor  $-1$ , indicando que  $x$  não está em  $A$ .

## 4.4 Exercícios

Para obter auxílio e o código fonte dos exercícios, visite [java.datastructures.net](http://java.datastructures.net).

### Reforço

- R-4.1 Forneça uma descrição em pseudocódigo de um algoritmo de tempo  $O(n)$  para calcular a função potência  $p(x,n)$ . Desenhe também o rastreamento recursivo deste algoritmo para o cálculo de  $p(2,5)$ .
- R-4.2 Forneça uma descrição em Java do algoritmo `Power` para calcular a função potência  $p(x,n)$ . (Trecho de código 4.3)
- R-4.3 Desenhe o rastreamento recursivo do algoritmo `Power`. (Trecho de código 4.3), que calcula a função potência  $p(x,n)$  para  $p(2,9)$ .
- R-4.4 Analise o tempo de execução do algoritmo `BinarySum` (Trecho de código 3.34) usando valores arbitrários para o parâmetro  $n$ .
- R-4.5 Desenhe o gráfico das funções  $8n$ ,  $4n \log n$ ,  $2n^2$ ,  $n^3$  e  $2^n$  usando uma escala logarítmica para os eixos  $x$  e  $y$ , isto é, se o valor da função  $f(n)$  é  $y$ , desenhe este ponto com a coordenada  $x$  em  $\log x$  em  $\log n$  e a coordenada  $y$  em  $\log y$ .
- R-4.6 O número de operações executadas pelos algoritmos  $A$  e  $B$  é  $8n \log n$  e  $2n^2$ , respectivamente. Determine  $n_0$  tal que  $A$  seja melhor que  $B$  para  $n \geq n_0$ .
- R-4.7 O número de operações executadas pelos algoritmos  $A$  e  $B$  é  $40n^2$  e  $2n^3$ , respectivamente. Determine  $n_0$  de maneira que  $A$  seja melhor que  $B$  para  $n \geq n_0$ .
- R-4.8 Apresente um exemplo de função cujo desenho seja o mesmo tanto em uma escala logarítmica como em uma escala padrão.
- R-4.9 Explique porque o desenho da função  $n^c$  é uma linha reta com inclinação  $c$  em uma escala logarítmica.
- R-4.10 Qual é a soma de todos os números pares de 0 a  $2n$ , para qualquer inteiro positivo?
- R-4.11 Mostre que as duas afirmações a seguir são equivalentes:
  - O tempo de execução do algoritmo  $A$  é  $O(f(n))$ .
  - No pior caso, o tempo de execução do algoritmo  $A$  é  $O(f(n))$ .

R-4.12 Ordene as funções a seguir por sua taxa assintótica de crescimento.

$$\begin{array}{lll} 4n \log n + 2n & 2^{10} & 2^{\log n} \\ 3n + 100n \log n & 4n & 2^n \\ n^2 + 10n & n^3 & n \log n \end{array}$$

- R-4.13 Mostre que se  $d(n) \in O(f(n))$ , então  $ad(n) \in O(f(n))$ , então  $ad(n) \in O(f(n))$ , para qualquer constante  $a > 0$ .
- R-4.14 Mostre que se  $d(n) \in O(f(n))$  e  $e(n) \in O(g(n))$ , então o produto  $d(n)e(n) \in O(f(n)g(n))$ .
- R-4.15 Forneça uma caracterização  $O$  em termos de  $n$  do tempo de execução do método Ex1 apresentado no Trecho de código 4.5.
- R-4.16 Forneça uma caracterização em termos de  $n$  do tempo de execução do método Ex2 apresentado no Trecho de código 4.5.
- R-4.17 Forneça uma caracterização em termos de  $n$  do tempo de execução do método Ex3 apresentado no Trecho de código 4.5.
- R-4.18 Forneça uma caracterização em termos de  $n$  do tempo de execução do método Ex4 apresentado no Trecho de código 4.5.
- R-4.19 Forneça uma caracterização em termos de  $n$  do tempo de execução do método Ex5 apresentado no Trecho de código 4.5.
- R-4.20 Bill dispõe de um algoritmo, find2D, para encontrar um elemento  $x$  em um arranjo  $A$   $n \times n$ . O algoritmo find2D itera sobre  $s$  linhas de  $A$  e chama o algoritmo arrayFind, do Trecho de código 4.4, para cada linha, até que  $x$  seja encontrado ou todas as linhas de  $A$  tenham sido pesquisadas. Qual o tempo para o pior caso de find2D em termos de  $n$ ? Qual o tempo para o pior caso de find2D em termos de  $N$ , onde  $N$  é o tamanho total de  $A$ ? É correto dizer que find2D é um algoritmo de tempo linear? Por que sim ou por que não?
- R-4.21 Para cada função  $f(n)$  e tempo  $t$  da tabela a seguir, determine o maior tamanho de  $n$  para um problema  $P$  que pode ser resolvido em tempo  $t$  se o algoritmo para resolver  $P$  consome  $f(n)$  microsegundos (uma das entradas já foi feita).

	1 segundo	1 hora	1 mês	1 século
$\log n$	$\approx 10^{300000}$			
$n$				
$n \log n$				
$n^2$				
$2^n$				

- R-4.22 Mostre que se  $d(n) \in O(f(n))$  e  $e(n) \in O(g(n))$ , então  $d(n) + e(n) \in O(f(n) + g(n))$ .
- R-4.23 Mostre que se  $d(n) \in O(f(n))$  e  $e(n) \in O(g(n))$ , então  $d(n) - e(n)$  **não é necessariamente**  $O(f(n) - g(n))$ .
- R-4.24 Mostre que se  $d(n) \in O(f(n))$  e  $f(n) \in O(g(n))$ , então  $d(n) \in O(g(n))$ .
- R-4.25 Mostre que  $O(\max\{f(n), g(n)\}) = O(f(n) + g(n))$ .
- R-4.26 Mostre que se  $f(n) \in O(g(n))$  se e somente se  $g(n) \in \Omega(f(n))$ .

R-4.27 Mostre que se  $p(n)$  é polinomial em relação a  $n$ , então  $\log p(n)$  é  $O(\log n)$ .

R-4.28 Mostre que  $(n + 1)^5$  é  $O(n^5)$ .

**Algoritmo Ex1( $A$ ):**

*Entrada:* um arranjo  $A$  que armazena  $n \geq 1$  elementos

*Saída:* a soma dos elementos de  $A$ .

$s \leftarrow A[0]$

**para**  $i \leftarrow 1$  até  $n - 1$  **faça**

$s \leftarrow s + A[i]$

**retorna**  $s$

**Algoritmo Ex2( $A$ ):**

*Entrada:* um arranjo  $A$  que armazena  $n \geq 1$  elementos

*Saída:* a soma dos elementos das células ímpares de  $A$ .

$s \leftarrow A[0]$

**para**  $i \leftarrow 2$  até  $n - 1$  em incrementos de 2 **faça**

$s \leftarrow s + A[i]$

**retorna**  $s$

**Algoritmo Ex3( $A$ ):**

*Entrada:* um arranjo  $A$  que armazena  $n \geq 1$  elementos

*Saída:* a soma da soma dos prefixos de  $A$ .

$s \leftarrow 0$

**para**  $i \leftarrow 0$  até  $n - 1$  **faça**

$s \leftarrow s + A[0]$

**para**  $j \leftarrow 1$  até  $i$  **faça**

$s \leftarrow s + A[j]$

**retorna**  $s$

**Algoritmo Ex4( $A$ ):**

*Entrada:* um arranjo  $A$  que armazena  $n \geq 1$  elementos

*Saída:* a soma da soma dos prefixos de  $A$ .

$s \leftarrow A[0]$

$t \leftarrow s$

**para**  $i \leftarrow 1$  até  $n - 1$  **faça**

$s \leftarrow s + A[i]$

$t \leftarrow t + s$

**retorna**  $t$

**Algoritmo Ex5( $A, B$ ):**

*Entrada:* Arranjos  $A$  e  $B$  cada um armazenando  $n \geq 1$  elementos

*Saída:* a quantidade de elementos de  $B$  iguais à soma da soma dos prefixos de  $A$ .

$c \leftarrow 0$

**para**  $i \leftarrow 1$  até  $n - 1$  **faça**

$s \leftarrow 0$

**para**  $j \leftarrow 1$  até  $n - 1$  **faça**

$s \leftarrow s + A[0]$

**para**  $k \leftarrow 1$  até  $j$  **faça**

$s \leftarrow s + A[k]$

**se**  $B[i] = s$  **então**

$c \leftarrow c + 1$

**retorna**  $c$

**Trecho de código 4.5** Alguns algoritmos.

- R-4.29 Mostre que  $2^{n-1}$  é  $O(2^n)$ .
- R-4.30 Mostre que  $n$  é  $O(n \log n)$ .
- R-4.31 Mostre que  $n^2$  é  $\Omega(n \log n)$ .
- R-4.32 Mostre que  $n \log n$  é  $\Omega(n)$ .
- R-4.33 Mostre que  $\lceil O(f(n)) \rceil$  é  $O(f(n))$ , se  $f(n)$  é uma função positiva não decrescente que é sempre maior que 1.
- R-4.34 O algoritmo  $A$  executa uma computação em tempo  $O(\log n)$  para cada entrada de um arranjo de  $n$  elementos. Qual o pior caso em relação ao tempo de execução de  $A$ ?
- R-4.35 Dado um arranjo  $X$  de  $n$  elementos, o algoritmo  $B$  escolhe  $\log n$  elementos de  $x$ , aleatoriamente, e executa um cálculo em tempo  $O(n)$  para cada um. Qual o pior caso em relação ao tempo de execução de  $B$ ?
- R-4.36 Dado um arranjo  $X$  de  $n$  elementos inteiros, o algoritmo  $C$  executa uma computação em tempo  $O(n)$  para cada número par de  $X$  e uma computação em tempo  $O(\log n)$  para cada elemento ímpar de  $X$ . Qual o melhor caso e o pior caso em relação ao tempo de execução de  $C$ ?
- R-4.37 Dado um arranjo  $X$  de  $n$  elementos, o algoritmo  $D$  chama o algoritmo  $E$  para cada elemento  $X[i]$ . O algoritmo  $E$  executa em tempo  $O(i)$  quando é chamado sobre um elemento  $X[i]$ . Qual o pior caso em relação ao tempo de execução do algoritmo  $D$ ?
- R-4.38 Al e Bob estão discutindo sobre seus algoritmos. Al afirma que seu método de tempo  $O(n \log n)$  é *sempre* mais rápido que o método de Bob de tempo  $O(n^2)$ . Para decidir a questão, eles executaram um conjunto de experimentos. Para o espanto de Al, eles encontraram que se  $n < 100$ , o algoritmo  $O(n^2)$  executa mais rápido, e somente quando  $n \geq 100$  é que o algoritmo  $O(n \log n)$  é um pouco melhor. Explique como isso é possível.

---

### Criatividade

- C-4.1 Descreva um algoritmo recursivo para calcular a parte inteira do logaritmo de base 2 de  $n$  usando apenas somas e divisões inteiras.
- C-4.2 Descreva como implementar um TAD fila usando duas pilhas. Qual o tempo de execução dos métodos `enqueue()` e `dequeue()` neste caso?
- C-4.3 Suponha que seja fornecido um arranjo  $A$  de  $n$  elementos contendo inteiros distintos que são listados em ordem crescente. Dado um número  $k$ , descreva um algoritmo recursivo para encontrar dois inteiros em  $A$  cuja soma seja  $k$ , se tal par existir. Qual o tempo de execução do seu algoritmo?
- C-4.4 Dado um arranjo  $A$  de  $n$  elementos inteiros não ordenado e um inteiro  $k$ , descreva um algoritmo recursivo para reorganizar os elementos de  $A$  de maneira que os elementos menores ou iguais a  $k$  antecedam qualquer elemento maior que  $k$ . Qual é o tempo de execução do seu algoritmo?
- C-4.5 Mostre que  $\sum_{i=1}^n i^2$  é  $O(n^3)$ .
- C-4.6 Mostre que  $\sum_{i=1}^n i / 2^i < 2$ . (Dica: tente limitar esta soma, termo a termo, usando uma progressão geométrica).
- C-4.7 Mostre que  $\log_b f(n)$  é  $\Theta(\log f(n))$  se  $b > 1$  é uma constante.

- C-4.8 Descreva um método para encontrar tanto o mínimo como o máximo entre  $n$  números usando menos que  $3n/2$  comparações. (Dica: primeiro crie um grupo de candidatos a mínimo e um grupo de candidatos a máximo).
- C-4.9 Bob construiu um site Web e forneceu a URL apenas para os seus  $n$  amigos, que ele numerou de 1 a  $n$ . Ele disse ao amigo número  $i$  que ele pode visitar o site no máximo  $i$  vezes. Agora Bob tem um contador,  $C$ , que mantém o total de visitas ao site (mas não as identidades dos visitantes). Qual é o valor mínimo de  $C$  tal que Bob possa ficar sabendo que um de seus amigos está visitando o site mais do que o número permitido de vezes?
- C-4.10 Considere a seguinte “justificativa” para o fato da função Fibonacci,  $F(n)$ , (veja a Proposição 4.20) ser  $O(n)$ :  
*Caso Base* ( $n \leq 2$ ):  $F(1) = 1$  e  $F(2) = 2$ ;  
*Passe de indução* ( $n > 2$ ): Assume-se a afirmação como verdadeira para  $n < n$ . Considere  $n$ .  $F(n) = F(n - 1) + F(n - 2)$ . Por indução,  $F(n - 1)$  é  $O(n' - 1)$  e  $F(n - 2)$  é  $O(n - 2)$ . Então  $F(n)$  é  $O((n - 1) + (n - 2))$ , pela identidade apresentada no Exercício R-4.22. Consequentemente,  $F(n)$  é  $O(n)$ . O que está errado nesta justificativa?
- C-4.11 Seja  $p(x)$  uma polinomial de grau  $n$ , isto é,  $\sum_{i=0}^n a_i x^i$ .
- Descreva um método simples de tempo  $O(n^2)$  para calcular  $p(x)$ .
  - Agora considere re-escrever  $p(x)$  como
- $$p(x) = a_0 + x(a_1 + x(a_2 + x(a_3 + \cdots + x(a_{n-1} + x a_n) \cdots))),$$
- o que é conhecido como **método de Horner**. Usando a notação  $O$ , caractere a quantidade de operações aritméticas que este método executa.
- C-4.12 Considere a função Fibonacci,  $F(n)$  (veja Proposição 4.20). Mostre por indução que  $F(n)$  é  $\Omega((3/2)^n)$ .
- C-4.13 Dado um conjunto  $A = \{a_1, a_2, \dots, a_n\}$  de  $n$  inteiros, descreva em pseudocódigo um método eficiente para calcular cada uma das somas parciais  $s_k = \sum_{i=1}^k a_i$  para  $k = 1, 2, \dots, n$ . Qual o tempo de execução deste método?
- C-4.14 Desenhe uma justificativa visual para a Proposição 4.3 análoga a da Figura 4.1(b) para o caso onde  $n$  é ímpar.
- C-4.15 Um arranjo  $A$  contém  $n - 1$  inteiros únicos no intervalo  $[0, n - 1]$ , isto é, existe um número neste arranjo que não está em  $A$ . Projete um algoritmo de tempo  $O(n)$  para encontrar este número. Pode-se usar somente  $O(1)$  espaço adicional além do arranjo  $A$  propriamente dito.
- C-4.16 Seja  $S$  um conjunto de  $n$  linhas no plano tal que não existem duas paralelas a não existe um trio que se encontre no mesmo ponto. Mostre, por indução, que as linhas de  $S$  determinam  $\Theta(n^2)$  pontos de intersecção.
- C-4.17 Mostre que o somatório  $\sum_{i=1}^n \lceil \log_2 i \rceil$  é  $O(n \log n)$ .
- C-4.18 Um rei malvado tem  $n$  garrafas de vinho e um espião envenenou apenas uma delas. Infelizmente ele não sabe qual. O veneno é extremamente mortal: apenas uma gota diluída em um bilhão ainda mata. Mesmo assim, leva um mês para o veneno fazer efeito. Desenhe um esquema para determinar exatamente qual das garrafas de vinho foi envenenada, em apenas um mês, usando apenas  $O(\log n)$  testadores.
- C-4.19 Suponha que cada linha de um arranjo  $A$ ,  $n \times n$ , consiste de zeros e uns tais que, em qualquer linha de  $A$ , todos os uns antecedem todos os zeros.

Assumindo que  $A$  ainda está na memória, descreva um método que rode em tempo  $O(n)$  (não  $O(n^2)$ ) para encontrar a linha de  $A$  que tem o maior número de uns.

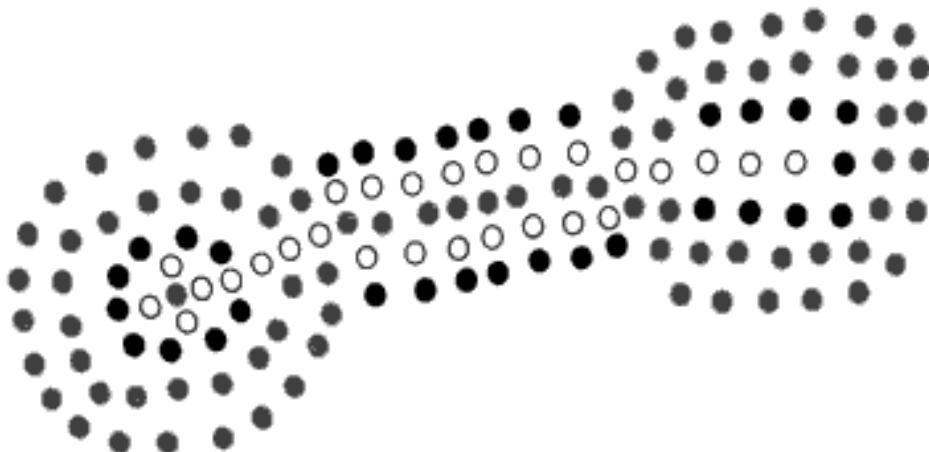
- C-4.20 Descreva em pseudocódigo um método para multiplicar uma matriz  $A$   $n \times m$  e uma matriz  $B$   $m \times p$ . Lembre que o produto  $C = AB$  é definido como  $C[i][j] = \sum_{k=1}^m A[i][k] \cdot B[k][j]$ . Qual o tempo de execução de seu método?
- C-4.21 Suponha que cada linha de um arranjo  $A$ ,  $n \times n$ , consiste em zeros e uns tais que, em qualquer linha de  $A$ , todos os uns antecedem todos os zeros. Também suponha que o número de uns na linha  $i$  é pelo menos o número na linha  $i + 1$ , para  $i = 0, 1, \dots, n - 2$ . Assumindo que  $A$  está na memória, descreva um método que execute em tempo  $O(n)$  (não  $O(n^2)$ ) para contar o número de uns em  $A$ .
- C-4.22 Descreva um método recursivo para calcular o  $n$ -ésimo **número harmônico**,  $H_n = \sum_{i=1}^n 1/i$ .
- 

## Projetos

- P-4.1 Implemente `prefixAverages1` e `prefixAverages2`, da Seção 4.2.5, e execute uma análise experimental dos seus tempos de execução. Visualize seus tempos de execução como uma função do tamanho da entrada usando um gráfico di-log.
- P-4.2 Execute uma análise experimental cuidadosa que compare os tempos relativos de execução dos métodos apresentados no Trecho de código 4.5.
- 

## Observações sobre o capítulo

A notação  $O$  tem gerado vários comentários sobre seu uso [16, 47, 61]. Knuth [62,61] a define usando a notação  $f(n) = O(g(n))$ , mas diz que esta igualdade funciona apenas em um sentido. Foi escolhida uma visão mais tradicional de igualdade e considerou-se a notação  $O$  como um conjunto, seguindo Brassard [16]. O leitor interessado em estudar análise do caso médio pode procurar o capítulo do livro de Vitter e Flajolet [97]. A história de Arquimedes é encontrada em [77]. Para algumas ferramentas matemáticas adicionais, consulte o Apêndice A.

**Conteúdo**

---

<b>5.1 Pilhas</b> .....	<b>178</b>
5.1.1 O tipo abstrato de dados pilha .....	178
5.1.2 Uma implementação baseada em arranjos.....	181
5.1.3 Implementando uma pilha usando uma lista encadeada genérica.....	185
5.1.4 Invertendo um arranjo usando uma pilha .....	187
5.1.5 Verificando parênteses e tags HTML .....	188
<b>5.2 Filas</b> .....	<b>191</b>
5.2.1 O tipo abstrato de dados fila .....	191
5.2.2 Uma implementação simples baseada em arranjos .....	193
5.2.3 Implementando uma fila usando uma lista encadeada genérica .....	195
5.2.4 Escalonadores round-robin .....	196
<b>5.3 Filas com dois finais</b> .....	<b>198</b>
5.3.1 O tipo abstrato de dados deque .....	198
5.3.2 Implementando um deque .....	199
<b>5.4 Exercícios</b> .....	<b>201</b>

## 5.1 Pilhas

Uma *pilha* é uma coleção de objetos que são inseridos e retirados de acordo com o princípio de que *o último que entra é o primeiro que sai (LIFO)*\*. É possível inserir objetos em uma pilha a qualquer momento, mas somente o objeto inserido mais recentemente (ou seja, o último que “entrou”) pode ser removido a qualquer momento. O nome “pilha” deriva-se da metáfora de uma pilha de pratos em uma cantina. Neste caso, as operações fundamentais envolvem a colocação e retirada de pratos da pilha. Quando um novo prato se faz necessário, retira-se o prato do topo da pilha (*pop*) e quando se acrescenta um prato, este é colocado sobre os já empilhados (*push*), passando a ser o novo topo\*\*. Talvez uma metáfora mais divertida pudesse ser uma máquina PEZ® fornecedora de doces: estas máquinas guardam doces empilhados sobre uma mola, que oferece o doce no topo da pilha quando a tampa da máquina é erguida (ver Figura 5.1). As pilhas são uma estrutura de dados fundamental: elas são usadas em muitas aplicações, incluindo as seguintes.



**Figura 5.1** Esquema de um dispensador PEZ®; uma implementação física do TAD pilha. (PEZ® é uma marca registrada da PEZ Candy Inc).

**Exemplo 5.1** Navegadores para a Internet armazenam os endereços mais recentemente visitados em uma pilha. Cada vez que o navegador visita um novo site, o endereço do site é armazenado na pilha de endereços. O navegador permite que o usuário retorne a site previamente visitados (“*pop*”) usando o botão “*back*”.

**Exemplo 5.2** Editores de texto geralmente oferecem um mecanismo de reversão de operações (“*undo*”) que cancela operações recentes e reverte um documento a estados anteriores. A operação de reversão é implementada mantendo-se as alterações no texto em uma pilha.

---

### 5.1.1 O tipo abstrato de dados pilha

Pilhas são as mais simples de todas as estruturas de dados, apesar de estar entre uma das mais importantes, na medida em que são usadas em uma gama de aplicações diferentes que incluem

\* N. de T. Em inglês, *last-in, first-out*.

\*\* N. de T. Em inglês, estas operações de colocação e retirada de uma pilha são chamadas de *push* e *pop*, respectivamente, e esta nomenclatura será mantida neste livro.

estruturas de dados muito mais sofisticadas. Formalmente, uma pilha  $S$  é um tipo abstrato de dados (TAD) que suporta os dois métodos que seguem:

`push( $e$ )`: Insere o objeto  $e$  no topo da pilha.

`pop()`: Remove o elemento no topo da pilha e o retorna; ocorre um erro se a pilha estiver vazia.

Adicionalmente, podem-se definir os seguintes métodos:

`size()`: Retorna o número de elementos na pilha.

`isEmpty()`: Retorna um booleano indicando se a pilha está vazia.

`top()`: Retorna o elemento no topo da pilha, sem retirá-lo; ocorre um erro se a pilha estiver vazia.

**Exemplo 5.3** A tabela a seguir mostra uma série de operações de pilha e seus efeitos sobre uma pilha  $S$  de inteiros, inicialmente vazia.

Operação	Saída	Conteúdo da pilha
<code>push(5)</code>	—	(5)
<code>push(3)</code>	—	(5,3)
<code>pop()</code>	3	(5)
<code>push(7)</code>	—	(5,7)
<code>pop()</code>	7	(5)
<code>top()</code>	5	(5)
<code>pop()</code>	5	0
<code>pop()</code>	"error"	0
<code>isEmpty()</code>	<code>true</code>	0
<code>push(9)</code>	—	(9)
<code>push(7)</code>	—	(9,7)
<code>push(3)</code>	—	(9,7,3)
<code>push(5)</code>	—	(9,7,3,5)
<code>size()</code>	4	(9,7,3,5)
<code>pop()</code>	5	(9,7,3)
<code>push(8)</code>	—	(9,7,3,8)
<code>pop()</code>	8	(9,7,3)
<code>pop()</code>	3	(9,7,3)

### Uma interface para pilhas em Java

Por sua importância, a estrutura de dados pilha é uma classe “embutida” no pacote `java.util` de Java. A classe `java.util.Stack` é uma estrutura de dados que armazena objetos Java genéricos e inclui, entre outros, os métodos `push()`, `pop()`, `peek()` (equivalente a `top()`), `size()` e `empty()` (equivalente a `isEmpty()`). Os métodos `pop()` e `peek()` lançam a exceção `EmptyStackException` se a pilha estiver vazia quando eles forem chamados. Embora seja conveniente usar a classe `java.util.Stack`, é instrutivo aprender como projetar e implementar uma pilha desde o início.

Implementar um tipo abstrato de dados em Java envolve dois passos. O primeiro passo é a definição de uma **Application Programming Interface** (API) ou simplesmente **interface** que descreve os nomes dos métodos que o TAD suporta e como eles são declarados e usados.

Além disso, devem-se definir exceções para qualquer condição de erro que possa ocorrer. Por exemplo, a condição de erro que ocorre quando se chama os métodos `pop()` ou `top()` sobre uma pilha vazia é sinalizada pelo lançamento de uma exceção do tipo `EmptyStackException`, que é definida no Trecho de código 5.1.

```


/*
 * Exceção de tempo de execução lançada quando alguém tenta executar uma operação top
 * ou pop sobre uma pilha vazia.
 */

public class EmptyStackException extends RuntimeException {
    public EmptyStackException(String err) {
        super(err);
    }
}


```

**Trecho de código 5.1** Exceção lançada pelos métodos `pop()` e `top()` da interface `Stack` quando ativados sobre uma pilha vazia.

Uma interface Java completa para o TAD pilha é fornecida no Trecho de código 5.2. Observa-se que esta interface é bastante geral, pois ela especifica que elementos de quaisquer classes (e suas derivadas) podem ser colocados na pilha. Esta generalidade é obtida usando o conceito de **genéricos** (Seção 2.5.2).

Para que um TAD seja útil, é necessário providenciar uma classe concreta que implemente os métodos da interface associada com aquele TAD. Uma implementação simples para a interface `Stack` é apresentada na próxima subseção.

```


/**
 * Interface para uma pilha: uma coleção de objetos
 * que são inseridos e removidos de acordo com o princípio do último que entra é o
 * primeiro que sai. Esta interface inclui os principais métodos de Java.util.Stack
 *
 * @author Roberto Tamassia
 * @author Michael Goodrich
 * @see EmptyStackException
 */

public interface Stack<E> {
    /**
     * Retorna o número de elementos na pilha
     * @return número de elementos na pilha.
     */
    public int size();

    /**
     * Indica quando a pilha está vazia
     * @return true se a pilha é vazia, false em caso contrário.
     */
    public boolean isEmpty();

    /**
     * Inspeciona o elemento no topo da pilha
     * @return o elemento do topo da pilha.
     * @exception EmptyStackException se a pilha estiver vazia.
     */
    public E top()
        throws EmptyStackException;

    /**
     * Insere um elemento no topo da pilha.
     * @param elemento a ser inserido.
     */


```

```

public void push (E element);
/*
 * Remove o elemento do topo da pilha.
 * @return elemento a ser removido
 * @exception EmptyStackException se a pilha estiver vazia.
 */
public E pop()
    throws EmptyStackException;
}

```

**Trecho de código 5.2** Interface Stack documentada com comentários em estilo Javadoc. (Ver Seção 1.9.3.) Observe também o uso do tipo genérico parametrizado, E, o que implica que a pilha pode conter elementos de qualquer classe.

### 5.1.2 Uma implementação baseada em arranjos

Pode-se implementar uma pilha armazenando-se seus elementos em um arranjo. Mais especificamente, a pilha desta implementação consiste em um arranjo  $S$  de  $N$  elementos mais uma variável inteira  $t$  que fornece o índice do elemento topo no arranjo  $S$ . (Ver Figura 5.2.)



**Figura 5.2** Implementação de uma pilha através de um arranjo  $S$ . O elemento do topo de  $S$  está armazenado na célula  $S[t]$ .

Lembrando que os índices para um arranjo começam no valor 0 em Java, inicializa-se  $t$  com  $-1$  e usa-se esse valor para identificar quando a pilha está vazia. Da mesma forma, pode-se usar esta variável para determinar a quantidade de elementos ( $t + 1$ ). Introduz-se um novo tipo de exceção chamada `FullStackException`, que sinalizará uma condição de erro ao se tentar inserir um novo elemento em um arranjo cheio. A exceção `FullStackException` é específica para esta implementação e não está definida no TAD pilha. Os detalhes desta implementação de pilha baseada em arranjo são fornecidos no Trecho de código 5.3.

```

Algoritmo size():
    return  $t + 1$ 
Algoritmo isEmpty():
    return ( $t < 0$ )
Algoritmo top():
    se isEmpty() então
        lançar uma EmptyStackException
    retorna  $S[t]$ 
Algoritmo push( $e$ ):
    se size() =  $N$  então
        lançar uma FullStackException
     $t \leftarrow t + 1$ 
     $S[t] \leftarrow e$ 
Algoritmo pop():
    se isEmpty() então
        lançar uma EmptyStackException

```

```

 $e \leftarrow S[t]$ 
 $S[t] \leftarrow \text{nulo}$ 
 $t \leftarrow t - 1$ 
retorna  $e$ 

```

**Trecho de código 5.3** Implementação de uma pilha através de um arranjo de tamanho fixo,  $N$ .

Analizando a implementação da pilha baseada em arranjo

A correção dos métodos da implementação baseada em arranjo resulta imediatamente da definição dos próprios métodos. Ainda assim, há um ponto interessante na implementação do método *pop*.

Observa-se que se poderia evitar resetar  $S[t]$  para **nulo** e ainda resultaria um método correto. Existe um acordo em se evitar esta atribuição quando se pensa em implementar este algoritmo em Java. Este acordo envolve o sistema de *coleta de lixo* de Java que procura na memória por objetos que não estão mais sendo referenciados por objetos ativos, e libera o espaço para uso futuro. (Para mais detalhes, ver a Seção 14.1.3) Seja  $e = S[t]$  o objeto no topo da pilha antes que o método *pop* seja chamado. Fazendo com que  $S[t]$  seja nulo, indica-se que a pilha não precisa mais guardar uma referência ao objeto  $e$ . Assim, se não existem outras referências ativas para o objeto  $e$ , então o espaço de memória ocupado por  $e$  será liberado pelo coletor de lixo.

A Tabela 5.1 apresenta os tempos de execução dos métodos de uma implementação de pilha usando arranjo. Na implementação usando arranjo, cada um dos métodos executa uma quantidade constante de comandos que envolvem operações aritméticas, comparações e atribuições. Além disso, *pop* também chama *isEmpty*, que também executa em um tempo constante. Logo, resta implementação do TAD pilha cada método executa em tempo constante, isto é, executa em tempo  $O(1)$ .

Método	Tempo
<i>size</i>	$O(1)$
<i>isEmpty</i>	$O(1)$
<i>top</i>	$O(1)$
<i>push</i>	$O(1)$
<i>pop</i>	$O(1)$

**Tabela 5.1** O desempenho de uma pilha implementada com arranjo. O uso de espaço é  $O(N)$ , onde  $N$  é o número máximo de elementos que a pilha pode conter, determinado quando a pilha é instanciada. Observa-se que o espaço é independente do número  $n \leq N$  de elementos que estão realmente na pilha.

Uma implementação concreta em Java da especificação em pseudocódigo do Trecho de código 5.3 com a classe *ArrayStack*, implementando a interface *Stack*, é mostrada nos Trechos de código 5.4 e 5.5. Infelizmente, por questões de espaço, omitiu-se grande parte dos comentários “javadocs” deste e da maioria dos demais trechos de código Java apresentados no restante deste livro. Observa-se que foi usado um nome simbólico, *CAPACITY*, para especificar a capacidade do arranjo. Isso permite que a capacidade do arranjo seja especificada em um local do código e tenha seu valor disponível em todo código.

```
/**
```

```
* Implementação da interface Stack usando um arranjo de tamanho fixo.
```

```
* Uma exceção é lançada ao tentar realizar uma operação de push quando o
```

```

* tamanho da pilha é igual ao tamanho do arranjo. Esta classe inclui os principais
* métodos da classe Java pré-definida java.util.Stack.
*/
public class ArrayStack<E> implements Stack<E> {
    protected int capacity; // capacidade real do arranjo da pilha
    public static final int CAPACITY = 1000; // capacidade default do arranjo
    protected E S[ ]; // Arranjo genérico usado para implementar a pilha
    protected int top = -1; // índice para o topo da pilha
    public ArrayStack( ) {
        this(CAPACITY); // capacidade default
    }

    public ArrayStack(int cap) {
        capacity = cap;
        S = (E[ ]) new Object[capacity]; // o compilador deve gerar um aviso, mas está ok
    }
    public int size( ) {
        return (top + 1);
    }
    public boolean isEmpty( ) {
        return (top < 0);
    }
    public void push(E element) throws FullStackException {
        if (size( ) == capacity)
            throw new FullStackException("Stack is full.");
        S[++ top] = element;
    }
    public E top( ) throws EmptyStackException {
        if (isEmpty( ))
            throw new EmptyStackException("Stack is empty.");
        return S[top];
    }
    public E pop( ) throws EmptyStackException {
        E element;
        if (isEmpty( ))
            throw new EmptyStackException("Stack is empty.");
        element = S[top];
        S[top--] = null; // desreferencia S[top] para o sistema de coleta de lixo
        return element;
    }
}

```

**Trecho de código 5.4** Uma implementação em Java para a interface Stack. (Continua no Trecho de código 5.5.)

```

public String toString( ) {
    String s;
    s = " [ ";
    if (size( ) > 0) s += S[0];
    if (size( ) > 1)
        for (int i = 1; i <= size( ) - 1; i++)
            s += ", " + S[i];
    }
    return s + "] ";
}
// Imprime informação de estado sobre uma operação recente da pilha

```

```

public void status(String op, Object element) {
    System.out.print("-----> " + op); // imprime esta operação
    System.out.println(", returns " + element); // o que foi retornado
    System.out.print("result: size = " + size() + ", isEmpty = " + isEmpty());
    System.out.println(", stack: " + this); // conteúdo da pilha
}
/** 
 * Testa o programa executando uma série de operações sobre pilhas,
 * imprimindo as operações executadas, os elementos retornados e o conteúdo da pilha
 * após cada operação
 */
public static void main(String[] args) {
    Object o;
    ArrayStack<Integer> A = new ArrayStack<Integer>();
    A.status("new ArrayStack<Integer> A", null);
    A.push(7);
    A.status("A.push(7)", null);
    o = A.pop();
    A.status("A.pop()", o);
    A.push(9);
    A.status("A.push(9)", null);
    o = A.pop();
    A.status("A.pop()", o);
    ArrayStack<String> B = new ArrayStack<String>();
    B.status("new ArrayStack<String> B", null);
    B.push("Bob");
    B.status("B.push(\"Bob\")", null);
    B.push("Alice");
    B.status("B.push(\"Alice\")", null);
    o = B.pop();
    B.status("B.pop()", o);
    B.push("Eve");
    B.status("B.push(\"Eve\")", null);
}
}

```

**Trecho de código 5.5** Pilha baseada em arranjo. (Continuação do Trecho de código 5.4.)

### Exemplo de saída

A seguir, a saída do programa `ArrayStack` já visto é apresentada. Observa-se que por meio do uso de tipos genéricos é possível criar um `ArrayStack A` que armazena inteiros e outro `ArrayStack B` que armazena strings.

```

-----> new ArrayStack<Integer> A, returns null
result: size = 0, isEmpty = true, stack: []
-----> A.push(7), returns null
result: size = 1, isEmpty = false, stack: [7]
-----> A.pop(), returns 7
result: size = 0, isEmpty = true, stack: []
-----> A.push(9), returns null
result: size = 1, isEmpty = false, stack: [9]
-----> A.pop(), returns 9
result: size = 0, isEmpty = true, stack: []
-----> new ArrayStack<String> B, returns null

```

```

result: size = 0, isEmpty = true, stack: []
-----> B.push("Bob"), returns null
result: size = 1, isEmpty = false, stack: [Bob]
-----> B.push("Alice"), returns null
result: size = 2, isEmpty = false, stack: [Bob, Alice]
-----> B.pop(), returns Alice
result: size = 1, isEmpty = false, stack: [Bob]
-----> B.push("Eve"), returns null
result: size = 2, isEmpty = false, stack: [Bob, Eve]

```

### Um problema com a implementação da pilha baseada em arranjo

A implementação de uma pilha com arranjos é simples e eficiente. Mesmo assim, esta implementação tem um aspecto negativo – ela deve assumir um limite superior fixo, CAPACITY, para o tamanho máximo da pilha. No Trecho de código 5.4, escolheu-se o valor 1000 de forma mais ou menos arbitrária. Uma aplicação real pode precisar de muito menos espaço e, nesse caso, ocorreria desperdício de memória. Por outro lado, uma aplicação pode precisar de mais espaço e, neste caso, a implementação da pilha poderia gerar uma exceção tão logo o programa cliente tente armazenar o objeto 1001 na pilha. Por isso, mesmo com esta simplicidade e eficiência, a implementação de pilha baseada em arranjos não é necessariamente a ideal.

Felizmente, existem outras implementações, discutidas a seguir, que não sofrem limitações de tamanho e usam memória proporcionalmente ao número de elementos armazenados na pilha. Nos casos em que se tem uma boa estimativa do número de elementos que serão colocados na pilha, entretanto, a implementação baseada em arranjos é difícil de superar. As pilhas são uma função vital de muitas aplicações, e por isso é muito útil dispor de uma implementação veloz do TAD pilha tal como a implementação baseada em arranjos.

### 5.1.3 Implementando uma pilha usando uma lista encadeada genérica

Nesta seção, serão exploradas as listas simplesmente encadeadas para implementar o TAD pilha. No projeto de tal implementação, será necessário decidir se o topo da pilha estará localizado na cabeça ou na cauda da lista. Entretanto, a melhor escolha é óbvia, uma vez que se pode inserir e remover elementos em tempo constante apenas na cabeça. Assim, é mais eficiente ter o topo da pilha localizado na cabeça da lista. Além disso, de maneira a executar a operação `size` em um tempo constante, manter-se-á o número corrente de elementos em uma variável de instância.

Em vez de se usar uma lista encadeada que armazena apenas um tipo de objeto, como mostrado na Seção 3.2, optou-se neste caso, por implementar uma pilha genérica usando uma lista encadeada **genérica**. Assim, se faz necessário usar um tipo genérico de nodo para implementar esta lista encadeada. Apresenta-se tal classe `Node` no Trecho de código 5.6.

```

public class Node<E> {
    // Variáveis de instância
    private E element;
    private Node<E> next;
    /** Cria um nodo com referencias nulas para os seus elementos e o próximo nodo */
    public Node() {
        this(null, null);
    }
    /** Cria um nodo com um dado elemento e o próximo nodo */
    public Node(E e, Node<E> n) {
        element = e;
        next = n;
    }
}

```

```

// Métodos de acesso:
public E getElement() {
    return element;
}
public Node<E> getNext() {
    return next;
}
// Métodos modificadores:
public void setElement(E newElem) {
    element = newElem;
}
public void setNext(Node<E> newNext) {
    next = newNext;
}
}

```

**Trecho de código 5.6** Classe Node, que implementa um nodo genérico para uma lista simplesmente encadeada.

### A classe genérica NodeStack

Uma implementação Java de uma pilha, usando uma lista simplesmente encadeada genérica é fornecida no Trecho de código 5.7. Todos os métodos da interface Stack são executados em tempo constante. Além de ser eficiente em relação ao tempo, esta implementação de lista encadeada tem uma necessidade de memória que é  $O(n)$ , onde  $n$  é o número de elementos na pilha. Assim, esta implementação não requer que uma nova exceção seja criada para lidar com o problema de estouro do tamanho. Usa-se uma variável de instância, top, para referenciar a cabeça da lista (que irá apontar para o objeto null se a lista estiver vazia). Quando se insere um novo elemento  $e$  na pilha, simplesmente cria-se um novo nodo  $v$  para  $e$ , referencia-se  $e$  a partir de  $v$ , e insere-se  $v$  na cabeça da lista. Da mesma forma, quando se retira um elemento da pilha, simplesmente remove-se o nodo da cabeça da lista e retorna-se seu elemento. Assim, executam-se todas as inserções e remoções de elementos na cabeça da lista.

```

public class NodeStack<E> implements Stack<E> {
    protected Node<E> top;           //referencia para o nodo cabeça
    protected int size;              //quantidade de elementos na pilha
    public NodeStack() { //constrói uma pilha vazia
        top = null;
        size = 0;
    }
    public int size() { return size; }
    public boolean isEmpty() {
        if (top == null) return true;
        return false;
    }
    public void push(E elem) {
        Node<E> v = new Node<E>(elem, top); //cria e encadeia um nodo novo
        top = v;
        size++;
    }
    public E top() throws EmptyStackException {
        if (isEmpty()) throw new EmptyStackException("Stack is empty.");
        return top.getElement();
    }
    public E pop() throws EmptyStackException {

```

```

if (isEmpty( )) throw new EmptyStackException("Stack is empty.");
E temp = top.getElement();
top = top.getNext();           //desencadeia o nodo topo
size--;
return temp;
}
}

```

**Trecho de código 5.7** Classe NodeStack que implementa a interface Stack usando uma lista simplesmente encadeada cujos nodos são objetos da classe Node, do Trecho de código 5.6.

### 5.1.4 Invertendo um arranjo usando uma pilha

Pode-se usar uma pilha para inverter os elementos de um arranjo através da geração de um algoritmo não-recursivo para o problema da inversão de um arranjo introduzido na Seção 3.5.1. A idéia básica consiste em inserir todos os elementos do arranjo em ordem na pilha. O Trecho de código 5.8 fornece uma implementação Java deste algoritmo. Por acaso, este método demonstra também como se podem usar tipos genéricos em uma aplicação simples que usa uma pilha genérica. Em especial, quando os elementos são retirados da pilha neste exemplo, eles são automaticamente retornados como elementos do tipo E; consequentemente, eles podem ser imediatamente retornados para o arranjo de entrada. Apresenta-se um exemplo de uso deste método no Trecho de código 5.9.

```

/** Um método genérico não recursivo para inverter um arranjo */
public static <E> void reverse(E[ ] a) {
    Stack<E> S = new ArrayStack<E>(a.length);
    for (int i=0; i < a.length; i++)
        S.push(a[i]);
    for (int i=0; i < a.length; i++)
        a[i] = S.pop();
}

```

**Trecho de código 5.8** Um método genérico que inverte os elementos do arranjo de tipo E usando uma pilha declarada através da interface Stack<E>.

```

/** Rotina de teste para a inversão de arranjo */
public static void main(String args[ ]) {
    Integer[ ] a = {4, 8, 15, 16, 23, 42};      // o autoboxing permite isso
    { }String[ ] s = {"Jack", "Kate", "Hurley", "Jin", "Boone"};
    System.out.println("a = " + Arrays.toString(a));
    System.out.println("s = " + Arrays.toString(s));
    System.out.println("Reversing . . .");
    reverse(a);
    reverse(s);
    System.out.println("a = " + Arrays.toString(a));
    System.out.println("s = " + Arrays.toString(s));
}

```

A saída do método é a seguinte:

```

a = [4, 8, 15, 16, 23, 42]
s = [Jack, Kate, Hurley, Jin, Michael]
Reversing...
a = [42, 23, 16, 15, 8, 4]
s = [Michael, Jin, Hurley, Kate, Jack]

```

**Trecho de código 5.9** Teste do método de inversão usando dois arranjos.

### 5.1.5 Verificando parênteses e tags HTML

Nesta subseção exploram-se duas aplicações relacionadas com pilhas, sendo que a primeira lida com verificação de parênteses e o agrupamento de símbolos em expressões aritméticas.

As expressões aritméticas podem conter vários pares de símbolos agrupados, tais como

- Parênteses: "(" e ")"
- Chaves: "{" e "}"
- Colchetes: "[" e "]"
- Símbolos de truncamento: "[" e "]"
- Símbolos de arredondamento por excesso: "f" e "l"

e para cada símbolo de abertura deve corresponder um símbolo de fechamento. Por exemplo, um abre colchetes "[", deve corresponder a um fecha colchetes "]", como na expressão que segue:

$$[(5 + x) - (y + z)].$$

Os exemplos a seguir ilustram esse conceito:

- Correto: ( )(( )){([ ( )])}
- Correto: ((( )( )){([ ( )])})
- Incorreto: )(( )){([ ( )])}
- Incorreto: {[ ])}
- Incorreto: (.

Deixa-se a definição mais precisa da verificação de agrupamento de símbolos para o Exercício R-5.5.

#### Um algoritmo para verificação de parênteses

Um problema importante no processamento de expressões aritméticas é ter certeza que os grupos de símbolos estão casados corretamente. Pode-se usar uma pilha  $S$  para executar a verificação de grupos de símbolos em expressões aritméticas com uma varredura simples da esquerda para direita. O algoritmo testa se os símbolos de abertura e fechamento casam e se são do mesmo tipo.

Supondo uma seqüência  $X = x_0 x_1 x_2 \dots x_{n-1}$ , onde cada  $x_i$  é um *token* que pode ser um conjunto de símbolos, um nome de variável, um operador aritmético ou um número. A idéia básica por trás da verificação de que os símbolos em  $S$  casam corretamente é processar os tokens de  $X$  em ordem. Cada vez que se encontra um símbolo de abertura insere-se o símbolo na pilha  $S$  (assumindo-se que a pilha não está vazia) e verifica-se se os dois são do mesmo tipo. Se a pilha estiver vazia após ter sido processada toda a seqüência, então os símbolos em  $X$  casam. Assumindo que as operações push e pop são implementadas para executar em tempo constante, este algoritmo executa em tempo  $O(n)$  que é linear. O Fragmento de código 5.10 apresenta a descrição em pseudocódigo deste algoritmo.

#### Algoritmo ParenMatch( $X, n$ ):

**Entrada:** Um arranjo  $X$  de  $n$  tokens, cada um dos quais é um grupo de símbolos, uma variável, um operador aritmético ou um número.

**Saída:** true se e somente se todos os grupos de símbolos de  $X$  casam corretamente

Seja  $S$  uma pilha vazia

**para**  $i \leftarrow 0$  até  $n-1$  **fazer**

    se  $X[i]$  é um símbolo de abertura **então**

$S.push(X[i]);$

```

senão se  $X[i]$  é um símbolo de fechamento então
    se  $S.isEmpty()$  então
        retorna false {nada para casar}
    se  $S.pop()$  não casa com o tipo de  $X[i]$  então
        retorna false {tipo errado}
    se  $S.isEmpty()$  então
        retorna true {todos os símbolos casam}
    senão
        retorna false {alguns símbolos não casam}

```

**Trecho de código 5.10** Algoritmo para verificar o agrupamento de símbolos em expressões aritméticas.

### Verificando tags em um documento HTML

Outra aplicação na qual a verificação de agrupamento é importante é na validação de documentos HTML. HTML é um formato padrão para hiperdocumentos na Internet. Em um documento HTML, porções de texto são delimitadas por **tags HTML**. Uma tag de abertura simples tem a forma “<nome>” e a tag de fechamento correspondente tem a forma “</nome>”. As tags HTML mais usadas incluem

- body: o corpo do documento
- h1: seção de cabeçalho
- center: texto centralizado
- p: parágrafo
- ol: lista numerada (ordenada)
- li: item de lista

No caso ideal, todas as tags de um documento HTML devem casar, embora alguns navegadores tolerem algumas tags que não casam.

A Figura 5.3 apresenta um exemplo de documento HTML e uma possibilidade de execução para o mesmo.

```

<body>
<center>
<h1> The Little Boat </h1>
</center>
<p> The storm tossed the little
boat like a cheap sneaker in an
old washing machine. The three
drunken fishermen were used to
such treatment, of course, but
not the tree salesman, who even as
a stowaway now felt that he
had overpaid for the voyage. </p>
<ol>
<li> Will the salesman die? </li>
<li> What color is the boat? </li>
<li> And what about Naomi? </li>
</ol>
</body>

```

(a)

### The Little Boat

The storm tossed the little boat  
like a cheap sneaker in an old  
washing machine. The three  
drunken shermen were used to  
such treatment, of course, but not  
the tree salesman, who even as  
a stowaway now felt that he had  
overpaid for the voyage.

1. Will the salesman die?
2. What color is the boat?
3. And what about Naomi?

(b)

**Figura 5.3** Demonstrando tags HTML. (a) um documento HTML; (b) sua execução.

Felizmente, mais ou menos o mesmo algoritmo do Trecho de código 5.10 pode ser usado para verificar as tags de um documento HTML. Nos Trechos de código 5.11 e 5.12, é fornecido o programa Java que verifica tags em um documento HTML lido a partir da entrada padrão. Por simplicidade, assume-se que todas as tags são simples, de abertura ou fechamento, definidas anteriormente e que nenhuma tag está mal formada.

```
import java.io.*;
import java.util.Scanner;
import net.datastructures.*;
/** Verificação simplificada de tags em um arquivo HTML */
public class HTML {
    /** Retira o primeiro e o último caracter de uma <tag> string */
    public static String stripEnds(String t) {
        if (t.length() <= 2) return null; // esta é uma tag degenerada
        return t.substring(1,t.length() - 1);
    }
    /** Testa se uma <tag> string retirada é vazia ou é uma tag de abertura verdadeira */
    public static boolean isOpeningTag(String tag) {
        return (tag.length() == 0) || (tag.charAt(0) != '/');
    }
}
```

**Trecho de código 5.11** Um programa Java completo para verificar as tags de um documento HTML. (Continua no Trecho de código 5.12)

```
/** Testa se a tag1 casa com a tag2 de fechamento (o primeiro caracter é um '/') */
public static boolean areMatchingTags(String tag1, String tag2) {
    return tag1.equals(tag2.substring(1)); // test against name after '/'
}
/** Testa se toda tag de abertura tem uma tag de fechamento */
public static boolean isHTMLMatched(String[] tag) {
    Stack<String> S = new NodeStack<String>(); // Pilha para verificar tags
    for (int i = 0; (i < tag.length) && (tag[i] != null); i++) {
        if (isOpeningTag(tag[i]))
            S.push(tag[i]); // tag de abertura; coloca-o de volta na pilha.
        else {
            if (S.isEmpty())
                return false; // nada para casar
            if (!areMatchingTags(S.pop(), tag[i]))
                return false; // casamento errado
        }
    }
    if (S.isEmpty()) return true; // tudo casa
    return false; // algumas tags não casam
}
public final static int CAPACITY = 1000; // Tamanho do arranjo de tags
/* Transforma um documento HTML em um arranjo de tags HTML */
public static String[] parseHTML(Scanner s) {
    String[] tag = new String[CAPACITY]; // o arranjo de tags (inicialmente todas nulas)
    int count = 0; // contador de tags
    String token; // token retornado pelo scanner s
    while (s.hasNextLine()) {
        while ((token = s.findInLine("<[^>]*>")) != null) // encontra a próxima tag
            tag[count++] = stripEnds(token); // retira o fim desta tag
        s.nextLine(); // vai para a próxima linha
    }
    return tag; // o arranjo de tags (retiradas)
}
```

```

}
public static void main(String[] args) throws IOException { // testador
    if (isHTMLMatched(parseHTML(new Scanner(System.in))))
        System.out.println("The input file is a matched HTML document.");
    else
        System.out.println("The input file is not a matched HTML document.");
}
}

```

**Trecho de código 5.12** Programa Java para testar o casamento de tags em um documento HTML. (Continuação do Trecho de código 5.11.) O método `isHTMLMatched` usa uma pilha para armazenar os nomes das tags de abertura, vistas anteriormente, de maneira semelhante ao que foi usado no Trecho de código 5.10. O método `parseHTML` usa um `Scanner`s para extrair as tags do documento HTML usando o padrão “`<[^>]*>`”, que denota uma string que começa por ‘`<`’ seguida por zero ou mais caracteres que não são ‘`>`’, seguidos por um ‘`>`’.

## 5.2 Filas

Outra estrutura de dados fundamental é a *fila*. Ela é uma “prima” próxima da pilha, pois uma fila é uma coleção de objetos que são inseridos e removidos de acordo com o princípio de que “*o primeiro que entra é o primeiro que sai*” (*FIFO*\*). Isto é, os elementos podem ser inseridos a qualquer momento, mas somente o elemento que está na fila há mais tempo pode ser retirado em um dado momento.

Geralmente, diz-se que os elementos entram na fila *por trás* e saem da fila *pela frente*. A metáfora para esta terminologia é uma fila de pessoas esperando para andar em um brinquedo de parque de diversões. As pessoas esperando para andar juntam-se à fila por trás e conseguem andar quando estão na frente.

### 5.2.1 O tipo abstrato de dados fila

Formalmente, o tipo abstrato de dados fila define uma coleção que mantém objetos em uma sequência, na qual o acesso aos elementos e sua remoção são restritos ao primeiro elemento da sequência, que é chamado de *índice* da fila e a inserção de elementos é restrita ao fim da sequência, que é chamada de *fim* da fila. Essa restrição garante a regra de que se inserem e se deletam itens em uma fila de acordo com o princípio de que o primeiro que entra é o primeiro que sai (FIFO).

O tipo abstrato de dados *fila* suporta os dois métodos fundamentais que seguem:

`enqueue(o)`: Insere o elemento *e* no fim da fila.

`dequeue()`: Retira e retorna o objeto da frente da fila. Ocorre um erro se a fila estiver vazia.

Adicionalmente, de forma semelhante ao tipo abstrato de dados Stack, o TAD fila inclui os seguintes métodos auxiliares:

`size()`: Retorna o número de objetos na fila.

`isEmpty()`: Retorna um booleano indicando se a fila está vazia.

`front()`: Retorna, mas não remove, o objeto na frente da fila. Ocorre um erro se a fila estiver vazia.

\* N. de T. *First in, first out*, em inglês.

**Exemplo 5.4** A tabela a seguir mostra uma série de operações e seus efeitos sobre uma fila  $Q$ , inicialmente vazia, de objetos inteiros. Para simplificar, serão usados inteiros em vez de objetos inteiros como argumentos das operações.

Operação	Saída	$frente \leftarrow Q \leftarrow fim$
enqueue(5)	—	(5)
enqueue(3)	—	(5,3)
dequeue()	5	(3)
enqueue(7)	—	(3,7)
dequeue()	3	(7)
front()	7	(7)
dequeue()	7	(0)
dequeue()	"error"	(0)
isEmpty()	true	(0)
enqueue(9)	—	(9)
enqueue(7)	—	(9,7)
size()	2	(9,7)
enqueue(3)	—	(9,7,3)
enqueue(5)	—	(9,7,3,5)
dequeue()	9	(7,3,5)

### Aplicações exemplo

Existem várias possibilidades de aplicações para filas. Lojas, teatros, centrais de reserva e outros serviços similares normalmente processam as requisições dos clientes de acordo com o princípio FIFO. Uma fila pode, consequentemente, ser a escolha lógica para a estrutura de dados que trata o processamento de transações de tais aplicações. Por exemplo, pode ser a escolha natural para tratar as chamadas de uma central de reservas de uma companhia aérea ou da bilheteria de um cinema.

### Uma interface de fila em Java

Uma interface em Java para o TAD fila é fornecida no Trecho de código 5.13. Esta interface genérica especifica que objetos de classes arbitrárias podem ser inseridos na fila. Assim, não existe a necessidade de se usar coerção explícita quando da retirada de elementos.

Observa-se que os métodos `size` e `isEmpty` têm o mesmo significado que seus equivalentes no TAD pilha. Estes dois métodos, bem como o método `front`, são conhecidos como métodos de **acesso**, pois retornam um valor e não alteram o conteúdo da estrutura de dados.

```
public interface Queue<E> {
    /**
     * Retorna o número de elementos na fila.
     * @return número de elementos na fila.
     */
    public int size();
    /**
     * Retorna se a fila está vazia.
     * @return true se a fila estiver vazia, false em caso contrário.
     */
    public boolean isEmpty();
    /**
     * Inspeciona o elemento à frente da fila.
     */
```

```

    * @return o elemento à frente da fila.
    * @exception EmptyQueueException se a fila estiver vazia
public E front( ) throws EmptyQueueException;
/**
 * Insere elemento no final da fila.
 * @param element, o novo elemento a ser inserido
 */
public void enqueue (E element);
/**
 * Remove o elemento à frente da fila.
 * @return elemento à frente da fila.
 * @exception EmptyQueueException se a fila estiver vazia.
 */
public E dequeue( ) throws EmptyQueueException;
}

```

**Trecho de código 5.13** Interface Queue documentada com comentários em estilo Javadoc.

### 5.2.2 Uma implementação simples baseada em arranjos

Nesta subseção, será apresentado como implementar uma fila usando um arranjo  $Q$  de tamanho fixo para armazenar seus elementos. Já que a regra principal com o tipo abstrato de dados fila é que os elementos são inseridos e deletados de acordo com o princípio FIFO, deve-se decidir como manter o controle da frente e do fim da fila.

Uma possibilidade seria adaptar a abordagem usada para a implementação da pilha, fazendo com que  $Q[0]$  seja a frente da fila e deixando a fila crescer a partir daí. Entretanto, esta não é uma solução eficiente porque exige que se movam todos os elementos para a frente uma posição, a cada vez que se efetuar uma operação `dequeue`. Uma implementação assim requereria tempo  $O(n)$  para executar o método `dequeue`, onde  $n$  é o número corrente de objetos na fila. Se for desejável tempo constante para cada método da fila, será necessária uma abordagem diferente.

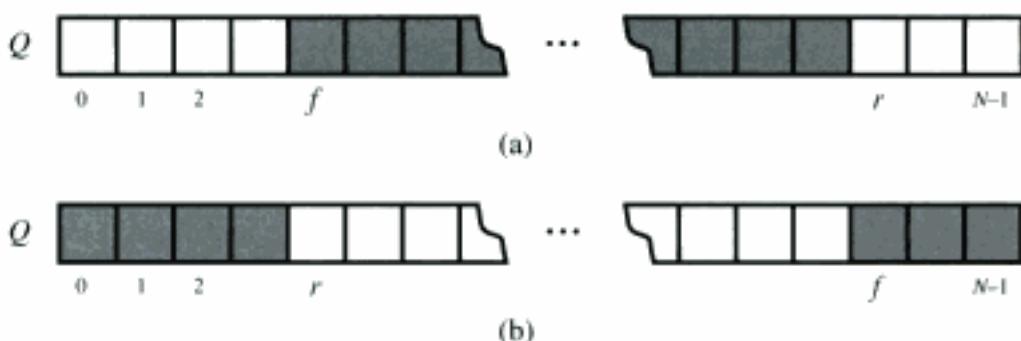
#### Usando um arranjo de maneira circular

Para evitar mover objetos uma vez que eles tenham sido colocados em  $Q$ , definem-se duas variáveis  $f$  e  $r$  que possuem os seguintes significados:

- $f$  é um índice de uma célula de  $Q$  que guarda o primeiro elemento da fila (que é o próximo candidato à remoção no caso de uma operação `dequeue`), a não ser que a fila esteja vazia (e neste caso  $f = r$ ).
- $r$  é um índice para a próxima posição livre em  $Q$ .

Inicialmente, atribui-se  $f = r = 0$ , indicando que a fila está vazia. Quando se remove um elemento da frente da fila, incrementa-se  $f$  para indicar a próxima célula. Da mesma forma, quando se acrescenta um elemento, armazena-se o mesmo em  $Q[r]$  e incrementa-se  $r$  para indicar a próxima célula livre em  $Q$ . Esse esquema permite implementar os métodos `front`, `enqueue` e `dequeue` em tempo constante, isto é,  $O(1)$ . Entretanto, ainda existe um problema com esta abordagem.

Considere-se, por exemplo, o que acontece se um mesmo elemento for inserido e retirado  $N$  vezes. Neste caso, tem-se  $f = r = N$ . Ao se tentar inserir o elemento apenas mais uma vez, irá ocorrer um erro de índice fora de faixa (pois as  $N$  posições válidas de  $Q$  vão de  $Q[0]$  a  $Q[N-1]$ ), mesmo que, neste caso, haja bastante espaço na fila. Para evitar este problema e poder utilizar todo o arranjo  $Q$ , faz-se com que os índices  $f$  e  $r$  “façam a volta” ao final de  $Q$ . Isto é, entende-se  $Q$  como um “arranjo circular” que vai de  $Q[0]$  a  $Q[N-1]$  e recomeça em  $Q[0]$  outra vez. (Ver Figura 5.4.)



**Figura 5.4** Usando o arranjo  $Q$  de forma circular: (a) a configuração “normal” com  $f \leq r$ ; (b) a configuração “reversa” com  $r < f$ . As posições armazenando elementos da fila estão salientadas.

### Usando o módulo operador para implementar um arranjo circular

Implementar esta visão circular de  $Q$  é bastante fácil. Cada vez que se incrementa  $f$  ou  $r$ , simplesmente calcula-se este incremento como “ $(f + 1) \bmod N$ ” ou “ $(r + 1) \bmod N$ ”, respectivamente.

Deve-se lembrar que o operador “mod” é o operador **módulo** que é calculado avaliando-se o resto de uma divisão inteira. Por exemplo, 14 dividido por 3 é 4, com resto 2, de forma que  $14 \bmod 4 = 2$ . Mais especificamente, dados os inteiros  $x$  e  $y$  tal que  $x \geq 0$  e  $y > 0$ , tem-se que  $x \bmod y = x - \lfloor x/y \rfloor y$ . Isto é, se  $r = x \bmod y$ , então há um inteiro não negativo  $q$ , de tal modo que  $x = qy + r$ . Java usa “%” para denotar o operador módulo. Usando este operador, pode-se ver  $Q$  como um arranjo circular e implementar cada método de uma fila em um tempo constante (ou seja, tempo  $O(1)$ ). Apresenta-se como usar esta abordagem para implementar uma fila no Trecho de código 5.14.

#### Algoritmo size():

```
    retorna  $(N - f + r) \bmod N$ 
```

#### Algoritmo isEmpty():

```
    retorna  $(f == r)$ 
```

#### Algoritmo front():

```
    se isEmpty() então
        lançar uma QueueEmptyException
    retorna  $Q[f]$ 
```

#### Algoritmo dequeue():

```
    se isEmpty() então
        lançar uma QueueEmptyException
    temp  $\leftarrow Q[f]$ 
     $Q[f] \leftarrow \text{null}$ 
     $f \leftarrow (f + 1) \bmod N$ 
    retorna temp
```

#### Algoritmo enqueue( $e$ ):

```
    se size() =  $N - 1$  então
        lançar uma FullQueueException
     $Q[r] \leftarrow e$ 
     $r \leftarrow (r + 1) \bmod N$ 
```

**Trecho de código 5.14** Implementação de uma fila usando um arranjo circular. A implementação usa o operador módulo para “reverter” índices após o final do arranjo, e inclui duas variáveis de instância,  $f$  e  $r$ , que indexam a frente da fila e a primeira posição vazia após o fim da fila, respectivamente.

A implementação apresentada tem um detalhe importante que pode passar despercebido a princípio. Considere-se o que ocorre se forem enfileirados  $N$  objetos em  $Q$  sem que nenhum seja retirado da fila. Resultaria  $f = r$ , que é a mesma condição que acontece quando a fila está vazia. Assim, não é possível determinar a diferença entre uma fila cheia e uma vazia. Felizmente, este não é um grande problema, e existem várias maneiras de resolvê-lo.

A solução que será descrita consiste em exigir que  $Q$  nunca contenha mais que  $N - 1$  objetos. Esta regra simples para tratar de uma fila cheia contorna o último problema desta implementação e leva ao pseudocódigo mostrado no Trecho de código 5.14. Observa-se que foi introduzida uma exceção chamada `FullQueueException`, que é específica desta implementação para sinalizar que não se pode mais inserir elementos na fila. Também observa-se a forma usada para calcular o tamanho da fila através da expressão  $(N - f + r) \bmod N$ , que fornece o resultado correto tanto na configuração "normal" (quando  $f \leq r$ ) como na configuração "reversa" (quando  $r < f$ ). A implementação em Java de uma fila usando arranjos é similar à implementação de uma pilha, e é deixada como exercício (P-5.4).

A Tabela 5.2 mostra os tempos de execução dos métodos em uma implementação de fila feita com um arranjo. Assim como a implementação de pilha baseada em arranjo apresentada, cada um dos métodos da fila realiza um número constante de instruções consistindo em operações aritméticas, comparações e atribuições. Assim, cada método nesta implementação é executado em tempo  $O(1)$ .

Método	Tempo
<code>size</code>	$O(1)$
<code>isEmpty</code>	$O(1)$
<code>front</code>	$O(1)$
<code>enqueue</code>	$O(1)$
<code>dequeue</code>	$O(1)$

**Tabela 5.2** Desempenho de uma fila implementada através de arranjo. O espaço utilizado é  $O(N)$ , onde  $N$  é o tamanho do arranjo, determinado quando a fila é criada. Observa-se que o uso de espaço é independente do número  $n < N$  de elementos que estão na fila.

Da mesma forma que a implementação de pilha baseada em arranjo, a única desvantagem real da implementação de fila baseada em arranjo é que se define artificialmente a capacidade da fila em um valor fixo. Em uma aplicação real pode-se precisar de mais ou menos capacidade na fila, mas se a estimativa do número de elementos que deverão estar na fila em um dado momento é boa, então a implementação baseada em arranjo é bastante eficiente.

### 5.2.3 Implementando uma fila usando uma lista encadeada genérica

Pode-se implementar de forma eficiente o TAD fila usando uma lista simplesmente encadeada. Por razões de eficiência, definiu-se que a frente da fila seja o início da lista, e que o final da fila seja o final da lista. (Por que seria ruim inserir no início e remover no final?) Observa-se que é necessário manter referências para os nós do início e do final da lista. Em vez de descrever todos os detalhes da implementação, será mostrada uma implementação Java para os métodos fundamentais para filas no Trecho de código 5.15.

```
public void enqueue(E elem) {
    Node<E> node = new Node<E>();
    node.setElement(elem);
```

```
node.setNext(null); // nodo será o novo nodo do final
if (size == 0)
    head = node; // caso especial de uma lista previamente vazia
else
    tail.setNext(node); // adiciona nodo no final da lista
tail = node; // atualiza referência ao nodo do final
size++;
}

...
public E dequeue() throws EmptyQueueException {
    if (size == 0)
        throw new EmptyQueueException("Queue is empty.");
    E tmp = head.getElement();
    head = head.getNext();
    size--;
    if (size == 0)
        tail = null; // a fila está vazia agora
    return tmp;
}
```

**Trecho de código 5.15** Métodos enqueue e dequeue na implementação do TAD fila usando uma lista simplesmente encadeada, usando nodos da classe Node do Trecho de código 5.6.

Cada um dos métodos da implementação com lista simplesmente encadeada do TAD fila é executado em tempo  $O(1)$ . Não é necessário especificar um tamanho máximo para a fila como se fez na implementação baseada em arranjos, mas este benefício tem o preço de usar mais espaço de memória por elemento. Os métodos usados na implementação com lista encadeada são mais complicados do que se gostaria, pois se deve ter cuidado em lidar com casos especiais em que a fila está vazia antes de um enqueue, ou quando a fila fica vazia depois de um dequeue.

---

#### 5.2.4 Escalonadores round-robin

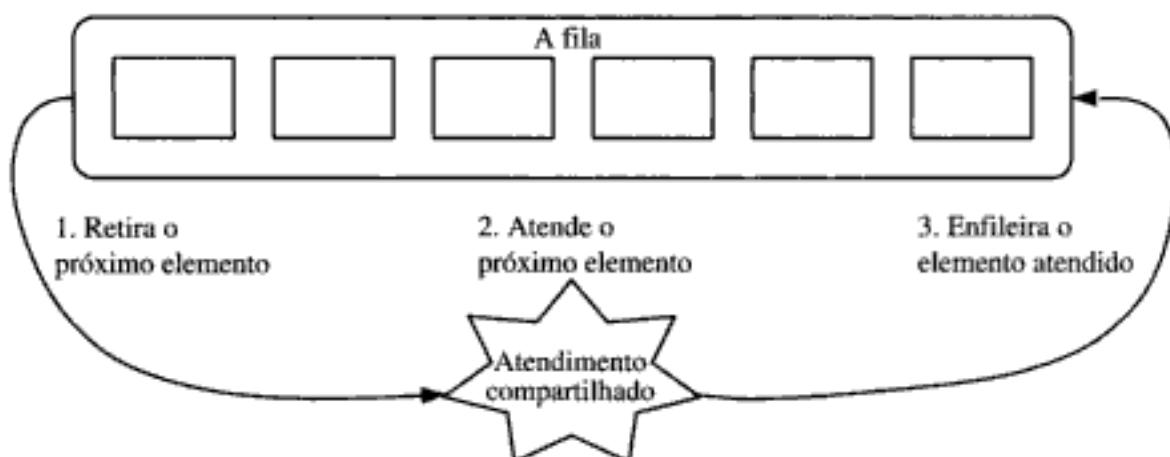
Um uso popular da estrutura de dados fila é implementar um escalonador **round-robin**, no qual se itera através de uma coleção de elementos de forma circular e “atende-se” cada elemento executando uma certa ação sobre ele. Tal escalonador é usado, por exemplo, para fazer uma alocação justa de um recurso que tem de ser compartilhado por uma coleção de clientes. Por exemplo, pode-se usar um escalonador round-robin para alocar uma fatia do tempo da CPU para várias aplicações que estão executando concorrentemente em um computador.

Pode-se implementar um escalonador round-robin usando uma fila,  $Q$ , executando de forma repetitiva os seguintes passos (ver Figura 5.5):

1.  $e \leftarrow Q.dequeue();$
2. Atende o elemento  $e$
3.  $Q.enqueue(e)$

#### O problema de Josephus

No jogo infantil “batata quente”, um grupo de  $n$  crianças senta em círculo e passa um objeto, chamado de “batata”, ao redor do círculo. A batata começa com uma das crianças do círculo, e as outras crianças continuam passando a batata até que um líder toque um sino, momento no



**Figura 5.5** Os três passos iterativos quando se usa uma fila para implementar um escalonador round-robin.

qual a criança que estiver com a batata deve sair do jogo, após deixar a batata com a próxima criança do círculo. Após a criança selecionada sair, as demais fecham a roda. Este processo continua até que a criança remanescente é declarada a vencedora. Se o líder sempre usa a estratégia de tocar o sino após a batata ter sido passada  $k$  vezes, para algum valor fixo  $k$ , então a determinação do vencedor para uma dada lista de crianças é conhecida como o *problema de Josephus*.

### Resolvendo o problema de Josephus usando uma fila

Pode-se resolver o problema de Josephus para uma coleção de  $n$  elementos usando uma fila, associando a batata com o elemento na frente da fila, e armazenando os elementos na fila de acordo com sua disposição ao redor do círculo. Assim, passar a batata é equivalente a retirar um elemento da fila e enfileirá-lo novamente. Após esse processo ter sido executado  $k$  vezes, se remove o elemento do início, retirando-o da fila e descartando-o. Um programa Java completo para resolver o problema de Josephus usando esta abordagem é apresentado no Trecho de código 5.16, que descreve uma solução que executa em tempo  $O(nk)$  (este problema pode ser resolvido mais rapidamente usando técnicas que estão além do escopo deste livro).

```
import net.datastructures.*;
public class Josephus {
    /** Solução para o problema de Josephus usando uma fila */
    public static <E> E Josephus(Queue<E> Q, int k) {
        if (Q.isEmpty()) return null;
        while (Q.size() > 1) {
            System.out.println(" Queue: " + Q + " k = " + k);
            for (int i=0; i < k; i++)
                Q.enqueue(Q.dequeue()); // move o elemento do início para o fim
            E e = Q.dequeue(); // remove o elemento da frente da coleção
            System.out.println(" " + e + " is out");
        }
        return Q.dequeue(); // o vencedor
    }
    /** Cria uma fila a partir de um arranjo de objetos */
    public static <E> Queue<E> buildQueue(E a[]) {
        Queue<E> Q = new NodeQueue<E>();
        for (int i=0; i < a.length; i++)
            Q.enqueue(a[i]);
    }
}
```

```
    return Q;
}
/** Método de teste */
public static void main(String[ ] args) {
    String[] a1 = {"Alice", "Bob", "Cindy", "Doug", "Ed", "Fred"};
    String[] a2 = {"Gene", "Hope", "Irene", "Jack", "Kim", "Lance"};
    String[] a3 = {"Mike", "Roberto"};
    System.out.println("First winner is " + Josephus(buildQueue(a1), 3));
    System.out.println("Second winner is " + Josephus(buildQueue(a2), 10));
    System.out.println("Third winner is " + Josephus(buildQueue(a3), 7));
}
```

**Trecho de código 5.16** Um programa Java completo para resolver o problema de Josephus usando uma fila. A classe NodeQueue é apresentada no Trecho de código 5.15.

---

## 5.3 Filas com dois finais

Considere-se agora uma estrutura de dados similar a uma fila que suporta inserção e remoção tanto em seu final, quanto em seu início. Esta extensão das filas é chamada de **fila com dois finais ou deque**, que normalmente pronuncia-se “deck” para evitar confusão com o método dequeue de um TAD fila normal, o qual pronuncia-se da mesma forma que a abreviatura “D.Q.”\*.

---

### 5.3.1 O tipo abstrato de dados deque

O tipo abstrato de dados deque é mais rico do que os tipos TAD pilha e fila. Os métodos fundamentais para o TAD deque são os que seguem:

- addFirst(*e*): Insere um novo elemento *e* no começo do deque.
- addLast(*e*): Insere um novo elemento *e* no final do deque.
- removeFirst(): Remove e retorna o primeiro elemento do deque; ocorre um erro se o deque estiver vazio.
- removeLast(): Remove e retorna o último elemento do deque; ocorre um erro se o deque estiver vazio.

Adicionalmente, o TAD deque pode incluir os seguintes métodos auxiliares:

- getfirst(): Retorna o primeiro elemento do deque; ocorre um erro se o deque estiver vazio.
- getLast(): Retorna o último elemento do deque; ocorre um erro se o deque estiver vazio.
- size(): Retorna o número de elementos do deque.
- isEmpty(): Determina se o deque está vazio.

**Exemplo 5.5** A tabela a seguir mostra uma série de operações e seus efeitos em um deque *D*, inicialmente vazio, de objetos inteiros. Para simplificar, usam-se inteiros em vez de objetos inteiros como argumentos das operações.

---

\* N. do T. Considere a pronúncia em inglês.

Operação	Saída	D
insertFirst(3)	-	(3)
insertFirst(5)	-	(5,3)
removeFirst()	5	(3)
insertLast(7)	-	(3,7)
removeFirst()	3	(7)
removeLast()	7	0
removeFirst()	"error"	0
isEmpty()	true	0

### 5.3.2 Implementando um deque

Já que o deque requer inserção e remoção em ambos os extremos da lista, usar uma lista simplesmente encadeada para implementar um deque seria ineficiente. Pode-se usar uma lista duplamente encadeada, entretanto, para implementar um deque de forma eficiente. Como foi analisado na Seção 3.3, inserir ou remover elementos nos dois extremos de uma lista encadeada pode ser feito de forma direta em tempo  $O(1)$ , se forem usados nodos sentinela para a cabeça e a cauda.

Para inserir um novo elemento  $e$ , deve-se ter acesso ao nodo  $p$  anterior ao local onde  $e$  deve ser colocado e ao nodo  $q$  posterior ao local onde  $e$  deve ser colocado. Para inserir um novo elemento entre  $p$  e  $q$  (que podem ser sentinelas), cria-se um novo nodo  $t$ , acertam-se as conexões next e prev de  $t$  para que apontem para  $q$  e  $p$ , respectivamente, depois faz-se com que o next de  $p$  aponte para  $t$  e o prev de  $q$  aponte para  $t$ .

Da mesma forma, para remover um elemento localizado no nodo  $t$ , podem-se acessar os nodos  $p$  e  $q$  em cada lado de  $t$  (e estes nodos devem existir, desde que se estejam usando sentinelas).

Para remover o nodo  $t$  entre  $p$  e  $q$ , simplesmente faz-se com que  $p$  e  $q$  apontem um para o outro, em vez de apontar para  $t$ . Não é necessário alterar as informações em  $t$ , pois agora  $t$  será detectado pelo algoritmo de coleta de lixo, pois ninguém está apontando para  $t$ .

Método	Tempo
size, isEmpty	$O(1)$
getFirst, getLast	$O(1)$
addFirst, addLast	$O(1)$
removeFirst, removeLast	$O(1)$

**Tabela 5.3** Performance de um deque implementado usando uma lista duplamente encadeada.

Deste modo, uma lista duplamente encadeada pode ser usada para implementar cada método do TAD deque, em tempo constante. Os detalhes de uma implementação Java eficiente do TAD deque ficam como exercício (P-5.7).

Casualmente, todos os métodos do TAD deque, como descritos acima, estão incluídos na classe `java.util.LinkedList<E>`. Assim, se for necessário usar um deque e não for o caso implementar um desde o início, pode-se simplesmente usar a classe predefinida `java.util.LinkedList<E>`.

Em qualquer caso, apresenta-se a interface `Deque` no Trecho de código 5.17, e a implementação desta interface no Trecho de código 5.18.

```

/*
 * Interface para um deque: uma coleção de objetos que são inseridos e removidos em
 * ambas as extremidades; um subconjunto dos métodos de Java.util.LinkedList.
 *
 * @author Roberto Tamassia
 * @author Michael Goodrich
 */
public interface Deque<E>
{
    /**
     * Retorna o número de elementos no deque
     */
    public int size();

    /**
     * Retorna se o deque está vazio
     */
    public boolean isEmpty();

    /**
     * Retorna o primeiro elemento; uma exceção é lançada se o deque está vazio.
     */
    public E getFirst() throws EmptyDequeException;

    /**
     * Retorna o último elemento; uma exceção é lançada se o deque está vazio.
     */
    public E getLast() throws EmptyDequeException;

    /**
     * Insere um elemento para ser o primeiro do deque.
     */
    public void addFirst (E element);

    /**
     * Insere um elemento para ser o último do deque.
     */
    public void addLast (E element);

    /**
     * Remove o primeiro elemento; uma exceção é lançada se o deque está vazio.
     */
    public E removeFirst() throws EmptyDequeException;

    /**
     * Remove o último elemento; uma exceção é lançada se o deque está vazio.
     */
    public E removeLast() throws EmptyDequeException;
}

```

**Trecho de código 5.17** Interface Deque documentada com comentários em estilo Javadocs (Seção 1.9.3). Nota-se também o uso do parâmetro de tipo genérico, E, o que implica que o deque pode armazenar elementos de qualquer classe.

```

public class NodeDeque<E> implements Deque<E> {
    protected DLNode<E> header, trailer;      // Sentinelas
    protected int size;             // Número de elementos
    public NodeDeque() {           // Inicializa um deque vazio
        header = new DLNode<E>();
        trailer = new DLNode<E>();
        header.setNext(trailer); // faz a cabeça apontar para a cauda
        trailer.setPrev(header); // faz a cauda apontar para a cabeça
    }
}

```

```

        size = 0;
    }
    public int size() {
        return size;
    }
    public boolean isEmpty() {
        if (size == 0)
            return true;
        return false;
    }
    public E getFirst() throws EmptyDequeException {
        if (isEmpty())
            throw new EmptyDequeException("Deque is empty.");
        return header.getNext().getElement();
    }
    public void addFirst(E o) {
        DLNode<E> second = header.getNext();
        DLNode<E> first = new DLNode<E>(o, header, second);
        second.setPrev(first);
        header.setNext(first);
        size++;
    }
    public E removeLast() throws EmptyDequeException {
        if (isEmpty())
            throw new EmptyDequeException("Deque is empty.");
        DLNode<E> last = trailer.getPrev();
        E o = last.getElement();
        DLNode<E> secondtolast = last.getPrev();
        trailer.setPrev(secondtolast);
        secondtolast.setNext(trailer);
        size--;
        return o;
    }
}

```

**Trecho de código 5.18** Classe NodeDeque implementando a interface Deque, não sendo mostrados nem a classe DLNode, que corresponde a um nodo genérico de lista duplamente encadeada, nem os métodos getLast e addLast ou removeFirst.

## 5.4 Exercícios

Para obter ajuda e o código fonte dos exercícios, visite [java.datastructures.net](http://java.datastructures.net).

### Reforço

- R-5.1 Suponha que uma lista inicialmente vazia  $S$  tenha executado um total de 25 operações push, 12 operações top e 10 operações pop, 3 das quais geraram StackEmptyExceptions, que foram capturadas e ignoradas. Qual é o tamanho corrente de  $S$ ?
- R-5.2 Se implementarmos a pilha  $S$  do problema anterior usando um arranjo, como descrito neste capítulo, então qual será o valor corrente da variável de instância top?

- R-5.3 Descreva a saída resultante da seguinte série de operações de pilha: push(5), push(3), pop(), push(2), push(8), pop(), pop(), push(9), push(1), pop(), push(7), push(6), pop(), pop(), push(4), pop(), pop().
- R-5.4 Apresente um método recursivo para remover todos os elementos de uma pilha.
- R-5.5 Apresente uma definição precisa e completa do conceito de verificação de grupos de símbolos em uma expressão aritmética.
- R-5.6 Descreva a saída resultante da seguinte seqüência de operações sobre uma fila: enqueue(5), enqueue(3), dequeue(), enqueue(2), enqueue(8), dequeue(), dequeue(), enqueue(9), enqueue(1), dequeue(), enqueue(7), enqueue(6), dequeue(), dequeue(), enqueue(4), dequeue(), dequeue().
- R-5.7 Suponha que uma fila  $Q$  inicialmente vazia tenha executado um total de 32 operações enqueue, 10 operações front e 15 operações dequeue, 5 das quais geraram exceções QueueEmptyException, que foram tratadas e ignoradas. Qual o tamanho atual de  $Q$ ?
- R-5.8 Se a fila do problema anterior foi implementada com um arranjo de capacidade  $N = 30$ , como descrito neste capítulo, e nunca gerou uma FullQueueException, quais podem ser os valores atuais de  $f$  e  $r$ ?
- R-5.9 Descreva a saída para a seguinte seqüência de operações sobre o TAD deque: addFirst(3), addLast(8), addLast(9), addFirst(5), removeFirst(), removeLast(), first(), addLast(7), removeFirst(), last(), removeLast().
- R-5.10 Suponha que você tem um deque  $D$  contendo os números (1,2,3,4,5,6,7,8), nesta ordem. Suponha, além disso, que você tem uma fila inicialmente vazia  $Q$ . Forneça uma descrição em pseudocódigo de um método que usa apenas  $D$  e  $Q$  (e nenhuma outra variável ou objeto) e resulta  $D$  armazenando os elementos (1,2,3,4,5,6,7,8) nesta ordem.
- R-5.11 Repita o problema anterior usando o deque  $D$  e uma pilha inicialmente vazia  $S$ .

---

### Criatividade

- C-5.1 Suponha que você tem uma pilha  $S$  contendo  $n$  elementos e uma fila  $Q$  que está inicialmente vazia. Descreva como você pode usar  $Q$  para percorrer  $S$  para ver se ela contém um certo elemento  $x$ , com a restrição adicional que seu algoritmo deve retornar os elementos de volta para  $S$  em sua ordem original. Você não pode usar um arranjo ou uma lista encadeada – apenas  $S$  e  $Q$  e um número fixo de variáveis de referência.
- C-5.2 Apresente uma descrição em pseudocódigo de uma implementação baseada em arranjo de um TAD lista encadeada. Qual o tempo de execução de cada operação?
- C-5.3 Suponha que Alice selecionou 3 inteiros diferentes e os colocou em uma pilha  $S$  em qualquer ordem. Escreva um pequeno trecho de pseudocódigo (sem laços ou recursão) que use apenas uma comparação e apenas uma variável  $x$ , garantindo com probabilidade de 2/3 que ao final deste código a variável  $x$  irá armazenar o maior dos 3 inteiros de Alice. Argumente porque seu método está correto.

- C-5.4 Descreva como implementar o TAD pilha usando duas filas. Qual o tempo de execução dos métodos *push* e *pop* neste caso?
- C-5.5 Mostre como usar uma pilha *S* e uma fila *Q* para gerar todos os possíveis subconjuntos de um conjunto *T* de *n* elementos de maneira não-recursiva.
- C-5.6 Suponha que se dispõe de um arranjo bidimensional *A*,  $n \times n$ , que se deseja usar para armazenar números inteiros, mas não se pretende despender um esforço  $O(n^2)$  para inicializá-lo com zeros (da forma que Java faz), porque sabe-se de antemão que serão usadas no máximo *n* células neste algoritmo que executa em tempo  $O(n)$  (sem contar o tempo de inicialização). Mostre como usar uma pilha *S* baseada em um arranjo que armazena triplas  $(i, j, k)$  para permitir o uso do arranjo *A* sem inicializá-lo e ainda implementar o algoritmo em tempo  $O(n)$ , mesmo que os valores iniciais das células de *A* sejam um lixo completo.
- C-5.7 Descreva um algoritmo não-recursivo para enumerar todas as permutações de números  $\{1, 2, \dots, n\}$ .
- C-5.8 *Notação pós-fixada* é uma forma não ambígua de escrever expressões aritméticas sem usar parênteses. É definida de maneira que se " $(exp_1)op(exp_2)$ " é uma expressão normal completamente entre parênteses cujo operador é *op*, então a versão pós-fixada da mesma é " $pexp_1 pexp_2 op$ ", onde *pexp<sub>1</sub>* é a versão pós-fixada de *exp<sub>1</sub>*, e *pexp<sub>2</sub>* é a versão pós-fixada de *exp<sub>2</sub>*. A versão pós-fixada de um único número ou variável é o próprio número ou variável. Então, por exemplo, a versão pós-fixada de " $((5+2)*(8-3))/4$ " é " $(5\ 2 + 8\ 3 - * 4 /)$ ". Descreva uma maneira não recursiva de avaliar uma expressão em notação pós-fixada.
- C-5.9 Suponha que você tem duas pilhas não vazias *S* e *T* e um deque *D*. Descreva como usar *D* de maneira que *S* armazene todos os elementos de *T* abaixo de seus elementos originais, mantendo os dois conjuntos de elementos em sua ordem original.
- C-5.10 Alice tem três pilhas baseadas em arranjo *A*, *B* e *C*, tais que *A* tem capacidade 100, *B* tem capacidade 5 e *C* tem capacidade 3. Inicialmente, *A* está cheio e *B* e *C* estão vazios. Infelizmente, as pessoas que programaram a classe para estas pilhas fizeram os métodos *push* e *pop* privados. O único método que Alice pode usar é um método estático, *transfer(S,T)*, que transfere (aplicando iterativamente os métodos *push* e *pop*) os elementos da pilha *S* para a pilha *T* até que *S* fique vazio ou *T* esteja cheio. Então, por exemplo, começando na configuração inicial e executando *transfer(A,C)* resulta em *A* armazenando 97 elementos e *C* armazenando 3. Descreva uma sequência de operações de transferência que comece da configuração inicial e resulte em *B* armazenando 4 elementos no final.
- C-5.11 Alice tem duas filas, *S* e *T*, que podem armazenar inteiros. Bob fornece para Alice 50 inteiros ímpares e 50 inteiros pares e insiste que ela armazene todos os 100 inteiros em *S* e *T*. Eles então iniciam um jogo onde Bob seleciona *S* ou *T* aleatoriamente e aplica o escalonador round-robin, descrito neste capítulo, sobre a fila escolhida um número aleatório de vezes. Se o número que sair da fila ao final do jogo for ímpar, Bob ganha. Caso contrário Alice ganha. Como Alice pode distribuir os inteiros pelas filas de maneira a otimizar suas chances de vitória? Qual sua chance de vitória?

- C-5.12 Suponha que Bob tem quatro vacas e que ele quer levá-las através da ponte. Mas ele possui apenas um cambão, que une apenas duas vacas lado a lado. O cambão é muito pesado para que ele possa carregá-lo pela ponte, mas ele pode amarrar (e soltar) as vacas no mesmo, rapidamente. De suas quatro vacas, Mazie pode atravessar a ponte em 2 minutos, Dayse pode atravessar em 4 minutos, Crazy leva 10 minutos e Lazy pode fazê-lo em 20 minutos. Naturalmente, quando duas vacas estão presas ao cambão, elas devem andar na velocidade da vaca mais lenta. Descreva como Bob pode atravessar suas vacas pela ponte em 34 minutos.

---

## Projetos

- P-5.1 Implemente o TAD pilha usando uma lista duplamente encadeada.
- P-5.2 Implemente o TAD pilha usando a classe `ArrayList` de Java (sem usar a classe predefinida de Java, `Stack`).
- P-5.3 Implemente um programa que possa receber uma expressão em notação pós-fixada (ver Exercício C-5.8) e exibir seu valor.
- P-5.4 Implemente o TAD fila usando um arranjo.
- P-5.5 Implemente todo o TAD fila usando uma lista simplesmente encadeada.
- P-5.6 Projete um TAD para uma pilha dupla de duas cores que consiste em duas pilhas – uma “vermelha” e outra “azul” – e tem suas versões coloridas das operações normais de um TAD pilha. Por exemplo, este TAD pode admitir tanto uma operação `push azul` como `vermelha`. Apresente uma implementação eficiente deste TAD usando um único arranjo cuja capacidade é definida em um valor  $N$  que se assume ser maior que os tamanhos das pilhas vermelho e azul combinadas.
- P-5.7 Implemente o TAD deque usando uma lista duplamente encadeada.
- P-5.8 Implemente o TAD deque usando um arranjo tratado de forma circular.
- P-5.9 Implemente as interfaces `Stack` e `Queue` com uma única classe que estende a classe `NodeQueue` (Trecho de código 5.8).
- P-5.10 Quando um lote de ações de uma companhia é vendido, o *capital obtido* (ou às vezes perdido) é a diferença entre o preço de venda e o preço pago originalmente pelas ações. Esta regra é fácil de entender para uma única ação, mas se vendemos vários lotes de ações comprados ao longo de um período de tempo, então é necessário identificar as ações que estão sendo vendidas. Um princípio padrão em contabilidade para a identificação de lotes de ações vendidas, neste caso, é o uso de um protocolo FIFO – as ações vendidas são aquelas que foram armazenadas mais tempo (na verdade este é o princípio padrão adotado em vários pacotes de software de finanças pessoais). Por exemplo, suponha que se deseja comprar 100 ações a R\$ 20,00 cada, no dia 1: 20 ações a R\$ 24,00, no dia 2; 200 ações a R\$ 36,00, no dia 3; e então vender 150 ações, no dia 4, a R\$ 30,00 cada. Então, aplicando o princípio FIFO, significa que das 150 ações vendidas, 100 foram compradas no dia 1, 20 no dia 2 e 30 no dia 3. O capital obtido neste caso foi  $100 \cdot 10 + 20 \cdot 6 + 30 \cdot (-6)$ , ou R\$ 940,00. Escreva um programa que recebe como entrada uma seqüência de transações do tipo “compre ações(s) por R\$y cada” ou “venda x ações(s) por

R\$y “cada”, assumindo que as transações ocorrem em dias consecutivos e que os valores de  $x$  e  $y$  são inteiros. Dada a seqüência de entrada, a saída pode ser o capital total ganho (ou perdido) para a seqüência completa, usando um protocolo FIFO para identificar as ações.

---

## Observações sobre o capítulo

A abordagem de definir primeiro as estruturas de dados em termos de seus TADs e depois de suas implementações concretas foi introduzida pela primeira vez pelos livros clássicos de Aho, Hopcroft e Ullman [4,5], que, não por acaso, são os primeiros trabalhos em que se vê um problema similar ao do exercício C-5.6. Os exercícios C-5.10, C-5.11 e C-5.12 são similares às questões da entrevista ditas originárias de uma companhia de software bem conhecida. Para aprofundar seus estudos de tipos abstratos de dados, veja Liskov e Guttag [69], Cardelli e Wegner [20] ou Demurjian [28].





## Conteúdo

<b>6.1 Listas arranjo .....</b>	<b>208</b>
6.1.1 O tipo abstrato de dados lista arranjo.....	208
6.1.2 O Padrão adaptador .....	209
6.1.3 Uma implementação simples usando arranjo.....	209
6.1.4 A interface simples e a classe Java.util.ArrayList .....	211
6.1.5 Implementando uma lista arranjo usando arranjos extensíveis.....	212
<b>6.2 Listas de nodos .....</b>	<b>215</b>
6.2.1 Operações baseadas em nodos.....	215
6.2.2 Posições .....	216
6.2.3 O tipo abstrato de dados lista de nodos.....	216
6.2.4 Implementação usando lista duplamente encadeada.....	219
<b>6.3 Iteradores .....</b>	<b>224</b>
6.3.1 Os tipos abstratos de dados iterador e iterável .....	224
6.3.2 O laço de Java para-cada .....	226
6.3.3 Implementando iteradores .....	226
6.3.4 Iteradores de lista em Java .....	228
<b>6.4 Os TADs de lista e o framework de coleções .....</b>	<b>229</b>
6.4.1 O framework de coleções do Java .....	229
6.4.2 A classe java.util.LinkedList.....	231
6.4.3 Seqüências .....	231
<b>6.5 Estudo de caso: a heurística mover-para-frente.....</b>	<b>233</b>
6.5.1 Usando uma lista ordenada e uma classe aninhada.....	233
6.5.2 Usando uma lista com a heurística mover-para-frente .....	235
6.5.3 Possíveis usos de uma lista de favoritos.....	236
<b>6.6 Exercícios .....</b>	<b>238</b>

## 6.1 Listas arranjo

Suponha que se dispõe de uma coleção  $S$  de  $N$  elementos armazenados em uma certa ordem linear, de maneira que é possível se referir aos elementos de  $S$  como primeiro, segundo, terceiro e assim por diante. Tal coleção é conhecida genericamente como uma *lista* ou *seqüência*. É possível fazer uma referência individual a cada elemento  $e$  de  $S$  usando um inteiro no intervalo  $[0, n - 1]$  que é igual ao número de elementos de  $S$  que precede  $e$  em  $S$ . O *índice* de um elemento  $e$  em  $S$  é o número de elementos que estão antes de  $e$  em  $S$ . Consequentemente, o primeiro elemento de  $S$  tem índice 0, e o último tem índice  $n - 1$ . Além disso, se um elemento de  $S$  tem índice  $i$ , o elemento anterior (se existir) tem índice  $i - 1$ , e o elemento seguinte (se existir) tem índice  $i + 1$ . O conceito de índice está relacionado ao conceito de *colocação*\* de um elemento em uma lista, que normalmente é definido como sendo um a mais do que seu índice; assim, o primeiro elemento está na primeira colocação, o segundo está na segunda colocação e assim por diante.

Uma seqüência que suporte acesso a todos os seus elementos através de seus índices é chamada de *lista arranjo* (ou *vetor*, usando um termo mais antigo). Uma vez que a definição de índice é mais consistente com a maneira pela qual os arranjos são indexados em Java e outras linguagens de programação (tais como C e C++), o lugar onde um elemento é armazenado em uma lista será referido como “índice”, e não “colocação” (apesar de ser usada a letra  $r^*$  para denotar este índice se a letra  $i$  estiver sendo usada como contador de um laço de **for**).

Este conceito de índice é uma notação simples, porém, poderosa, uma vez que pode ser usada para especificar onde inserir um novo elemento em uma lista ou onde remover um elemento antigo.

---

### 6.1.1 O tipo abstrato de dados lista arranjo

Como um TAD, uma *lista arranjo* tem os seguintes métodos (além dos métodos padrão `size()` e `isEmpty()`):

- `get(i)`: retorna o elemento de  $S$  com índice  $i$ ; uma condição de erro ocorre se  $i < 0$  ou  $i > \text{size}() - 1$ .
- `set(i, e)`: substitui por  $e$  o elemento de índice  $i$ ; uma condição de erro ocorre se  $i < 0$  ou  $i > \text{size}() - 1$ .
- `add(i, e)`: insere um elemento novo  $e$  em  $S$  para que tenha o índice  $i$ ; uma condição de erro ocorre se  $i < 0$  ou  $i > \text{size}()$ .
- `remove(i)`: remove de  $S$  o elemento de índice  $i$ ; uma condição de erro ocorre se  $i < 0$  ou  $i > \text{size}() - 1$ .

*Não* se afirma que um arranjo deva ser usado para implementar uma lista arranjo e que, neste caso, o elemento de índice 0 deva ser armazenado no índice 0 do arranjo, embora esta possa ser uma possibilidade (muito natural). A definição de índice oferece uma forma de referir o “lugar” onde o elemento está armazenado em uma seqüência, sem preocupações com a implementação exata desta seqüência. O índice de um elemento pode se alterar sempre que a seqüência é atualizada, como ilustrado no exemplo a seguir.

---

\* N. de T. Em inglês, *rank*.

**Exemplo 6.1** Apresentam-se a seguir algumas operações sobre uma lista arranjo  $S$  inicialmente vazia.

Operação	Saída	$S$
add(0, 7)		(7)
add(0, 4)		(4, 7)
get(1)	7	(4, 7)
add(2, 2)		(4, 7, 2)
get(3)	error	(4, 7, 2)
remove(1)	7	(4, 2)
add(1, 5)		(4, 5, 2)
add(1, 3)		(4, 3, 5, 2)
add(4, 9)		(4, 3, 5, 2, 9)
get(2)	5	(4, 3, 5, 2, 9)
set(3, 8)	2	(4, 3, 5, 8, 9)

### 6.1.2 O padrão adaptador

Com freqüência, escrevem-se classes que provêm funcionalidades similares a outras classes. O padrão **adaptador** se aplica a qualquer contexto em que se deseja modificar uma classe existente de maneira que seus métodos combinem com os de uma classe ou interface relacionada mas diferente. Uma forma geral de aplicar o padrão adaptador é definir uma classe nova de maneira que ela contenha uma instância da classe velha como um campo escondido, e implemente cada método da nova classe usando os métodos desta variável de instância escondida. O resultado da aplicação do padrão adaptador é que se cria uma nova classe que executa praticamente as mesmas funções da classe anterior, mas de uma forma mais conveniente.

Em relação à discussão sobre o TAD lista arranjo, percebe-se que este TAD é suficiente para definir uma classe adaptadora para o TAD deque, como pode ser visto na Tabela 6.1 (ver também o Exercício C-6.8).

Método de deque	Implementação com métodos de lista arranjo
size(), isEmpty()	size(), isEmpty()
getFirst()	get(0)
getLast()	get(size() - 1)
addFirst( $e$ )	add(0, $e$ )
addLast( $e$ )	add(size(), $e$ )
removeFirst()	remove(0)
removeLast()	remove(size() - 1)

Tabela 6.1 Implementação de um deque como uma lista arranjo.

### 6.1.3 Uma implementação simples usando arranjo

Uma escolha óbvia para implementar o TAD lista arranjo é usar um arranjo  $A$ , onde  $A[i]$  armazena (uma referência para) o elemento de índice  $i$ . Escolhe-se o tamanho  $N$  do arranjo  $A$  grande o suficiente, e se mantém a quantidade de elementos em uma variável de instância  $n < N$ .

Os detalhes de implementação dos métodos do TAD lista arranjo são simples. Para implementar a operação  $\text{get}(i)$ , por exemplo, apenas retorna-se  $A[i]$ . A implementação dos métodos  $\text{add}(i, e)$  e  $\text{remove}(i)$  é fornecida no Trecho de código 6.1. Uma parte importante desta implementação (e que demanda tempo) envolve o deslocamento de elementos para cima ou para baixo, para manter contíguas as células ocupadas do arranjo. Essas operações de deslocamento são necessárias para manter a regra de sempre armazenar o elemento de índice  $i$  no índice  $i$  do arranjo  $A$ . (Ver a Figura 6.1 e também o Exercício R-6.12).

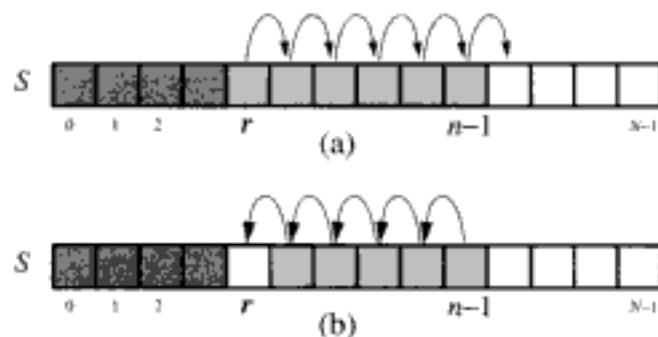
**Algoritmo**  $\text{add}(i, e)$ :

```
para  $j = n - 1, n - 2, \dots, i$  faça
     $A[j + 1] \leftarrow A[j]$  {Abre espaço para o novo elemento}
     $A[i] \leftarrow e$ 
     $n \leftarrow n + 1$ 
```

**Algoritmo**  $\text{remove}(i)$ :

```
 $e \leftarrow A[i]$  { $e$  é uma variável temporária}
para  $j = i, i + 1, \dots, n - 2$  faça
     $A[j] \leftarrow A[j + 1]$  {substitui pelo elemento removido}
     $n \leftarrow n - 1$ 
return  $e$ 
```

**Trecho de código 6.1** Métodos  $\text{add}(i, e)$  e  $\text{remove}(i)$  da implementação de um TAD lista arranjo. Denota-se por  $n$  a variável de instância que armazena a quantidade de elementos na lista arranjo.



**Figura 6.1** Implementação baseada em um arranjo de uma lista arranjo  $S$ , armazenando  $n$  elementos: (a) deslocando uma posição para cima para inserir no índice  $i$ ; (b) deslocamento para baixo para remover do índice  $i$ .

### Performance da implementação simples baseada em arranjo

A Tabela 6.2 indica os tempos de execução para o pior caso dos métodos de uma lista arranjo de  $n$  elementos, implementada usando um arranjo. Os métodos `isEmpty`, `size`, `get` e `set` claramente executam em um tempo  $O(1)$ , mas os métodos de inserção e remoção podem consumir muito mais tempo. Especialmente, o método  $\text{add}(i, e)$ , executa em tempo  $O(n)$ . Na verdade, o pior caso para esta operação ocorre quando  $i = 0$ , uma vez que todos os  $n$  elementos terão de ser deslocados para frente. Argumento similar se aplica ao método  $\text{remove}(i)$ , que executa em tempo  $O(n)$  porque é necessário mover  $n - 1$  elementos uma posição para trás, no pior caso ( $i = 0$ ). De fato, assumindo que todos os índices têm igual probabilidade de serem passados por parâmetro, para estas operações o tempo de execução médio é  $O(n)$ , pois é necessário deslocar  $n/2$  elementos, em média.

Método	Tempo
<code>size()</code>	$O(1)$
<code>isEmpty()</code>	$O(1)$
<code>get(<i>i</i>)</code>	$O(1)$
<code>set(<i>i, e</i>)</code>	$O(1)$
<code>add(<i>i, e</i>)</code>	$O(n)$
<code>remove(<i>i</i>)</code>	$O(n)$

**Tabela 6.2** Performance de uma lista arranjo de  $n$  elementos, implementada usando um arranjo. O espaço usado é  $O(N)$ , onde  $N$  é o tamanho do arranjo.

Analizando com mais cuidado `add(i, e)` e `remove(i)` percebe-se que executam em um tempo  $O(n - i + 1)$ , pois apenas os elementos da posição  $i$  e superior deverão ser deslocados. Logo, a inserção ou remoção de um item no fim de uma lista arranjo usando os métodos `add(i, e)` e `remove(i - 1)`, respectivamente, consome tempo  $O(1)$ . Além disso, esta observação tem uma consequência interessante na adaptação do TAD lista arranjo para o TAD deque, apresentado na Seção 6.1.1. Se o TAD lista arranjo, neste caso, é implementado sobre um arranjo como antes descrito, então os métodos `addLast` e `removeLast` do deque executam cada um em tempo  $O(1)$ . Entretanto, os métodos `addFirst` e `removeFirst` do deque executam cada um em tempo  $O(n)$ .

Na verdade, com um pequeno esforço, pode-se criar uma implementação baseada em arranjo para o TAD lista arranjo que resulte em tempo  $O(1)$  para inserções e deleções na colocação 0, bem como nas inserções e deleções no fim da lista arranjo. Obter isso implica abandonar a regra que determina que um elemento de índice  $i$  deve ser armazenado no índice  $i$  do arranjo, e usar uma abordagem baseada em um arranjo circular semelhante à usada na Seção 5.2 para implementar uma fila. Os detalhes dessa implementação são deixados como exercício (C-6.9).

#### 6.1.4 A interface simples e a classe `java.util.ArrayList`

Para preparar a construção de uma implementação Java do TAD lista arranjo, apresenta-se, no Trecho de código 6.2, uma interface Java, `IndexList`, que captura os principais métodos do TAD lista arranjo. Neste caso, usa-se uma `IndexOutOfBoundsException` para sinalizar um argumento de índice inválido.

```
public interface IndexList<E> {
    /** Retorna a quantidade de elementos desta lista. */
    public int size();
    /** Retorna se a lista está vazia. */
    public boolean isEmpty();
    /** Insere um elemento e de maneira que o mesmo ocupe o índice i, deslocando todos os
     * elementos depois deste. */
    public void add(int i, E e)
        throws IndexOutOfBoundsException;
    /** Retorna o elemento no índice i, sem removê-lo. */
    public E get(int i)
        throws IndexOutOfBoundsException;
    /** Remove e retorna o elemento no índice i, deslocando os elementos após este. */
    public E remove(int i)
        throws IndexOutOfBoundsException;
```

```
    /** Substitui o elemento no índice i por e, retornando o elemento anterior em i. */
    public E set(int i, E e)
        throws IndexOutOfBoundsException;
}
```

**Trecho de código 6.2** A interface IndexList para o TAD lista arranjo.

### A classe java.util.ArrayList

Java oferece uma classe, `java.util.ArrayList`, que implementa todos os métodos fornecidos anteriormente para o TAD lista arranjo. Isto é, inclui todos os métodos apresentados no Trecho de código 6.2 da interface `IndexList`. Mais que isso, a classe `Java.util.ArrayList` tem recursos além dos do TAD lista arranjo simplificado. Por exemplo, a classe `Java.util.ArrayList` também inclui um método `clear`, que remove todos os elementos da lista arranjo e um método `toArray()`, que retorna um arranjo contendo todos os elementos da lista arranjo na mesma ordem. Adicionalmente, a classe `java.util.ArrayList` dispõe de métodos para pesquisa na lista, incluindo o método `indexOf(e)` que retorna o índice da primeira ocorrência do elemento igual a `e` na lista arranjo e o método `lastIndexOf(e)`, que retorna o índice da última ocorrência do elemento igual a `e` na lista arranjo. Os dois métodos retornam o índice inválido  $-1$  se um elemento igual a `e` não for encontrado.

---

#### 6.1.5 Implementando uma lista arranjo usando arranjos extensíveis

Além de implementar os métodos da interface `IndexList` (e alguns outros métodos úteis), a classe `java.util.ArrayList` provê um recurso interessante que sobrepõe a fraqueza da implementação simples baseada em arranjo.

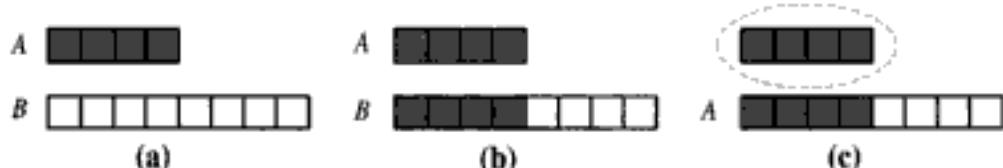
Especificamente, o ponto mais fraco da implementação simples usando um arranjo do TAD lista arranjo, fornecida na Seção 6.1.3, é que esta exige a especificação antecipada de uma capacidade fixa,  $N$ , para o número total de elementos que podem ser armazenados na lista arranjo. Se o número real de elementos  $n$  da lista arranjo for muito menor que  $N$ , então a implementação irá desperdiçar espaço. Pior ainda, se  $n$  for maior que  $N$ , então a implementação irá falhar.

Em vez disso, a classe `java.util.ArrayList` usa uma técnica interessante de arranjo extensível, de maneira que não é necessário se preocupar com estouros do arranjo quando se usa esta classe.

Da mesma forma que a classe `java.util.ArrayList`, será providenciada uma forma de aumentar o arranjo  $A$  que armazena os elementos da lista arranjo  $S$ . É claro que em Java (e outras linguagens de programação) não se pode realmente aumentar o arranjo  $A$ ; sua capacidade é fixa para um determinado valor  $N$ , como já foi visto. Em vez disso, quando ocorre uma situação de *overflow*, ou seja, quando  $n = N$  e o método `add` é ativado, executam-se os seguintes passos:

1. Alocar um novo arranjo  $B$  com capacidade  $2N$
2. Fazer  $A[i] \leftarrow B[i]$  para  $i = 0, \dots, N - 1$
3. Fazer  $A \leftarrow B$ , ou seja, usar  $B$  como sendo o arranjo que suporta  $S$
4. Inserir o novo elemento em  $A$

Esta estratégia de substituição de um vetor é conhecida como *arranjo extensível*, na medida em que pode ser vista como a ampliação do arranjo base para abrir mais espaço para novos elementos (ver Figura 6.2). Intuitivamente, esta estratégia é semelhante à de um bernardo-ermitão, que se muda para uma concha maior quando fica maior que a anterior.



**Figura 6.2** Representação dos três passos para “fazer crescer” um arranjo extensível: (a) criar um novo arranjo  $B$ ; (b) copiar os elementos de  $A$  para  $B$ ; (c) atribuir o novo arranjo para a referência  $A$ . Não é mostrado que o arranjo antigo será eliminado pelo sistema de coleta de lixo.

### Implementando a interface IndexList usando um arranjo extensível

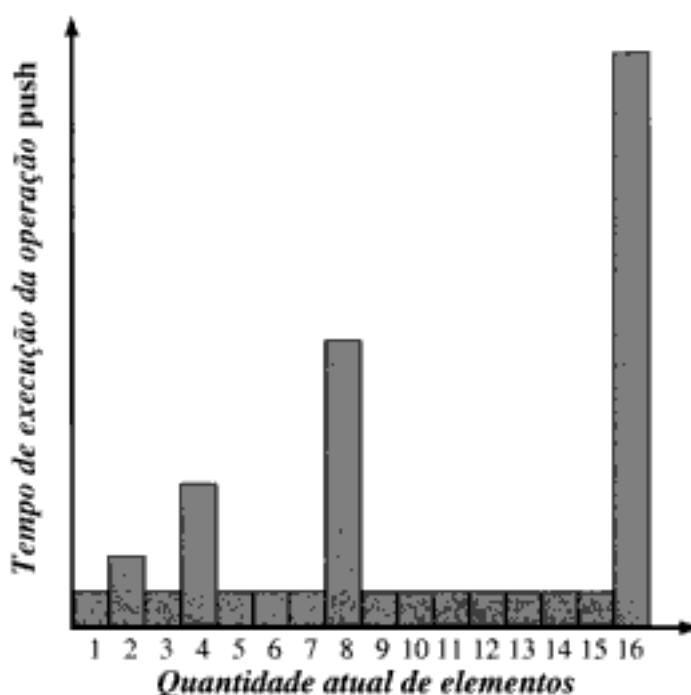
Partes da implementação em Java do TAD lista arranjo usando um arranjo extensível são apresentadas no Trecho de código 6.3. Esta classe provê apenas a capacidade de expansão do vetor. O Exercício C-6.2 explora uma implementação que também pode encolher.

```
/** Implementação de uma lista encadeada usando um arranjo cujo tamanho é duplicado
 * quando o tamanho da lista indexada excede a capacidade do arranjo.
 */
public class ArrayList<E> implements IndexList<E> {
    private E[] A; // arranjo que armazena os elementos da lista indexada
    private int capacity = 16; // tamanho inicial do arranjo A
    private int size = 0; // número de elementos armazenados na lista indexada
    /** Cria a lista indexada com capacidade inicial 16 */
    public ArrayList() {
        A = (E[]) new Object[capacity]; // o compilador vai avisar, mas está ok
    }
    /** Insere um elemento no índice especificado. */
    public void add(int r, E e)
        throws IndexOutOfBoundsException {
        checkIndex(r, size() + 1);
        if (size == capacity) // um overflow
            capacity *= 2;
        E[] B = (E[]) new Object[capacity];
        for (int i=0; i<size; i++)
            B[i] = A[i];
        A = B;
    }
    for (int i=size-1; i>=r; i--) // desloca um elemento para cima
        A[i+1] = A[i];
    A[r] = e;
    size++;
}
/** Remove o elemento armazenado no índice especificado. */
public E remove(int r)
    throws IndexOutOfBoundsException {
    checkIndex(r, size());
    E temp = A[r];
    for (int i=r; i<size-1; i++) // desloca um elemento para baixo
        A[i] = A[i+1];
    size--;
    return temp;
}
```

**Trecho de código 6.3** Partes da classe `ArrayList` que implementa o TAD lista arranjo usando um arranjo extensível. O método `checkIndex(r,n)` (não apresentado) verifica se um índice  $r$  pertence ao intervalo  $[0, n - 1]$ .

### Análise amortizada de um arranjo extensível

A estratégia de substituição de arranjos pode parecer lenta, à primeira vista, pois a execução de uma única substituição, necessária em certas operações de inserção, levará tempo  $O(n)$ . Observa-se, porém, que após uma substituição, o novo arranjo permite a inserção de outros  $n$  elementos novos antes que o arranjo tenha de ser substituído outra vez. Este simples fato permite mostrar que o tempo de uma série de operações executadas sobre um vetor inicialmente vazio é realmente bastante eficiente. Usando uma notação abreviada, a operação de inserir um elemento no final de um vetor será chamada de ***push*** (ver Figura 6.3).



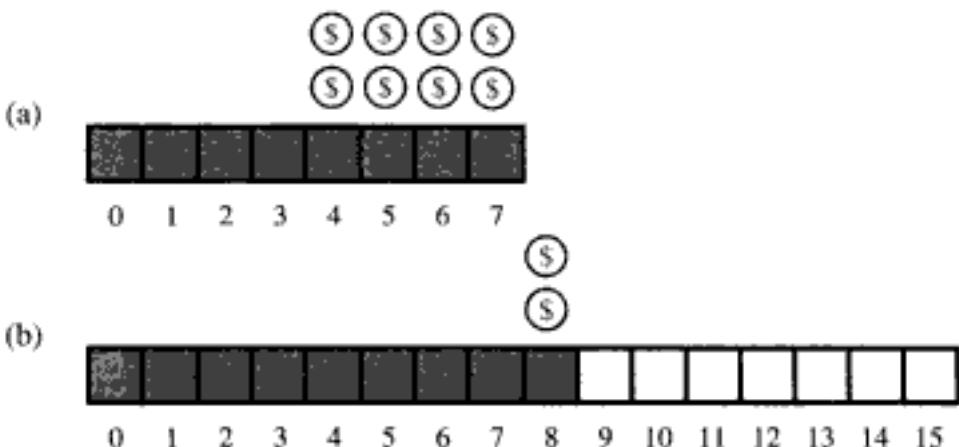
**Figura 6.3** Tempos de execução de uma série de operações push sobre um `java.util.ArrayList` de tamanho inicial 1.

Utilizando um padrão de projeto chamado de ***amortização***, pode-se mostrar que executar uma seqüência de operações push em um vetor implementado sobre um arranjo extensível é realmente muito eficiente. Para fazer uma ***análise amortizada***, será usada uma técnica de conta-corrente na qual um computador será visto como uma máquina que necessita a inserção de moedas para funcionar, e que requer o pagamento de um ***ciberdólar*** para cada período determinado de uso. Quando uma operação é executada, é necessário ter ciberdólares suficientes na “conta de ciberdólares” para pagar o tempo de execução da operação. Então, o total de ciberdólares gastos para qualquer cálculo será proporcional ao tempo gasto naquele cálculo. A vantagem deste método de análise é que se pode supervalorizar certas operações de maneira a poupar ciberdólares para pagar outras.

**Proposição 6.2** *Seja  $S$  uma lista implementada sobre um arranjo extensível de tamanho 1. O tempo total para executar uma série  $n$  de operações push sobre  $S$ , iniciando com  $S$  vazio é  $O(n)$ .*

**Justificativa** Assume-se que um ciberdólar é suficiente para pagar a execução de cada operação push sobre  $S$ , desconsiderando o tempo para fazer o arranjo crescer. Supõe-se também que aumentar o arranjo de um tamanho  $k$  para  $2k$ , requer  $k$  ciberdólares para o tempo gasto copiando os elementos. Pode-se cobrar por cada operação push 3 ciberdólares. Desta forma, se está sobre-taxando cada operação push que não causa overflow, em dois ciberdólares. Considere-se que os

dois ciberdólares de lucro obtidos nas inserções que não aumentam o arranjo são “armazenados” junto ao elemento inserido. Um overflow ocorre quando o vetor  $S$  tem  $2^i$  elementos, para um inteiro  $i \geq 0$ , e o tamanho do arranjo usado para representá-lo tem tamanho  $2^i$ . Então, dobrar o tamanho do arranjo requer  $2^i$  ciberdólares. Felizmente, estes ciberdólares podem ser encontrados nos elementos armazenados nas células  $2^{i-1}$  a  $2^i - 1$  (ver a Figura 6.4). Observa-se que o overflow anterior ocorreu quando o número de elementos ficou maior que  $2^{i-1}$  pela primeira vez e os ciberdólares armazenados nas células  $2^{i-1}$  a  $2^i - 1$  não foram gastos. Desta forma, dispõe-se de um esquema de amortização no qual se cobram 3 ciberdólares por cada operação e todo o tempo de cálculo é pago. Ou seja, se paga pela execução de  $n$  operações push usando  $3n$  ciberdólares. Em outras palavras, o tempo de execução amortizado de cada operação é  $O(1)$ ; assim, o tempo total de execução de  $n$  operações push é  $O(n)$ . ■



**Figura 6.4** Representação de uma série de operações push sobre uma lista arranjo: (a) um arranjo de tamanho 8 cheio, com dois ciberdólares “armazenados” nos índices de 4 a 7; (b) uma operação push causa um overflow e duplica a capacidade. A cópia dos 8 elementos antigos para o novo arranjo é paga pelos ciberdólares armazenados; a inserção de novos elementos é paga por um dos ciberdólares cobrados pela operação push; os dois ciberdólares de lucro são armazenados na célula 8.

## 6.2 Listas de nodos

Usar um índice não é a única maneira de se referir ao lugar onde um elemento aparece em uma seqüência. Se existe uma seqüência  $S$  implementada sobre uma lista (simples ou duplamente) encadeada, então é mais natural e eficiente usar um nodo em vez de um índice como forma de identificar onde acessar ou atualizar essa lista. Nesta seção, define-se o TAD lista de nodos, que abstrai a estrutura de dados concreta em uma lista encadeada (Seções 3.2 e 3.3) usando um TAD com posições relativas que abstrai o conceito de “lugar” em uma lista de nodos.

### 6.2.1 Operações baseadas em nodos

Seja  $S$  uma lista (simplesmente ou duplamente) encadeada. Gostaria-se de definir métodos para  $S$  que recebessem nodos da lista como parâmetros e que resultassem como tipo de retorno. Tais métodos poderiam ser significativamente mais rápidos em relação a métodos baseados em índices para localizar um elemento em uma lista encadeada, pois para localizar um elemento em uma lista encadeada é necessário pesquisar através da mesma de forma incremental a partir do início ou fim, contando os elementos à medida que se avança.

Como exemplo, pode-se definir um método hipotético `remove(v)`, que remove o elemento de  $S$  armazenado no nodo  $v$  da lista. Usar o nodo como parâmetro permite remover o elemento em tempo  $O(1)$  simplesmente indo direto ao lugar onde o nodo está armazenado e, então, desconectando-o da lista através de uma atualização dos campos `next` e `prev` de seus vizinhos. Da mesma forma, pode-se inserir, em tempo  $O(1)$ , um elemento novo  $e$  em  $S$  com uma operação tal como `addAfter(v,e)`, que especifica o nodo  $v$  depois do qual o novo elemento deve ser inserido. Neste caso, apenas se encadeia o nodo novo.

Definir os métodos de um TAD lista acrescentando operações baseadas em nodos, reforça a questão relativa à quanta informação sobre a implementação da lista pode ser exposta. Certamente é desejável ser capaz de usar tanto uma lista simples, como duplamente encadeada, sem revelar estes detalhes para o usuário. Por outro lado, não seria desejável que o usuário modificasse a estrutura interna da lista. Tais modificações seriam possíveis, entretanto, se fosse passada para o usuário uma referência para um nodo da lista, de forma que o mesmo tivesse acesso à estrutura interna do nodo (tais como os campos `next` ou `prev`).

Para abstrair e unificar diferentes formas de armazenar elementos nas possíveis implementações de uma lista, introduz-se o conceito de *posição*, formalizando a noção intuitiva de “lugar” de um elemento em relação aos outros na lista.

### 6.2.2 Posições

Para expandir de forma segura o conjunto de operações sobre listas, abstrai-se a noção de “posição”, o que permite aproveitar a eficiência de implementações baseadas em listas simples ou duplamente encadeadas, sem violar os princípios de projeto orientado a objetos. Neste esquema, vê-se uma lista como um repositório de elementos armazenados em posições que são mantidas organizadas em uma ordem linear. Uma posição também é um tipo abstrato de dados que suporta o seguinte método simples:

`element()`: Retorna o elemento armazenado nesta posição.

Uma posição é sempre definida de forma *relativa*, isto é, em relação aos vizinhos. Em uma lista, uma posição  $p$  estará sempre “depois” de uma posição  $q$  e “antes” de uma posição  $s$  (a menos que  $p$  seja a primeira ou a última posição). Uma posição  $p$ , associada com um elemento  $e$  em uma lista  $S$ , não se altera mesmo se o índice de  $e$  se modificar em  $S$ , a menos que  $e$  seja explicitamente removido (destruindo a posição  $p$ ). Por outro lado, a posição  $p$  não se modifica, mesmo quando se substitui ou se permuta o elemento  $e$  armazenado em  $p$  por outro. Esses fatos permitem definir um conjunto rico de métodos de lista baseados em posições que as recebem dos objetos como parâmetro e fornecem objetos de posição como valores de retorno.

### 6.2.3 O tipo abstrato de dados lista de nodos

Usando o conceito de posição para encapsular a idéia de “nodo” em uma lista, pode-se definir outro tipo de TAD seqüência chamado de TAD *lista de nodos*. Este TAD suporta os seguintes métodos para uma lista  $S$ :

`first()`: Retorna a posição do primeiro elemento de  $S$ ; ocorre um erro se  $S$  está vazio.

`last()`: Retorna a posição do último elemento de  $S$ ; ocorre um erro se  $S$  está vazio.

`prev(p)`: Retorna a posição do elemento de  $S$  que precede o que se encontra na posição  $p$ ; ocorre um erro se  $p$  for a primeira posição.

`next(p)`: Retorna a posição do elemento de *S* que segue o que se encontra na posição *p*; ocorre um erro se *p* for a última posição.

Os métodos acima permitem fazer referência a posições relativas de uma lista, começando no início ou no fim e deslocando-se por incremento para cima ou para baixo. As posições podem ser intuitivamente entendidas como sendo os nós da lista, porém, observa-se que não existem referências específicas a objetos nodo. Além do mais, se for fornecida uma posição como argumento para um método da lista, então ela deverá representar uma posição válida da lista.

### Métodos de atualização de uma lista de nós

Além dos métodos listados e dos genéricos `size` e `isEmpty`, o TAD lista inclui, também, os seguintes métodos de atualização que recebem um objeto posição como parâmetro e/ou fornecem objetos posição como valores de retorno.

`set(p,e)`: Substitui o elemento que se encontra na posição *p* por *e*, retornando o elemento que se encontrava antes na posição *p*.

`addFirst(e)`: Insere o novo elemento *e* em *S* como primeiro elemento.

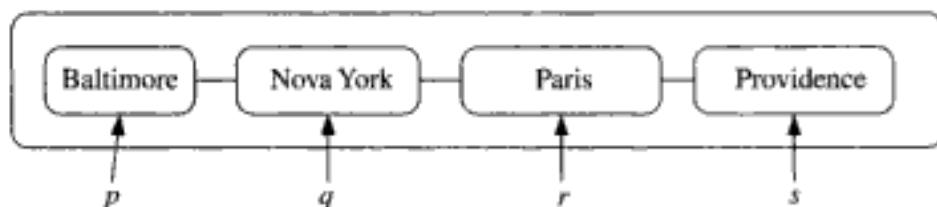
`addLast(e)`: Insere o novo elemento *e* em *S* como último elemento.

`addBefore(e)`: Insere um novo elemento *e* em *S* antes da posição *p*.

`addAfter(e)`: Insere um novo elemento *e* em *S* depois da posição *p*.

`remove(p)`: Remove e retorna o elemento na posição *p* de *S*, invalidando esta posição de *S*.

O TAD lista de nós permite que se entenda uma coleção ordenada de objetos em função de seus lugares, sem se preocupar com a maneira exata pela qual esses locais são representados (ver Figura 6.5).



**Figura 6.5** Uma lista de nós. As posições na ordem atual são *p, q, r e s*.

Em um primeiro momento, parece haver redundância no repertório de operações do TAD lista de nós, uma vez que se pode executar a operação `addFirst(e)`, usando `addBefore(first(),e)` e a operação `addLast(e)`, usando `addAfter(getLast(),e)`. Mas essas substituições só podem ser feitas para uma lista não-vazia.

Observa-se que uma condição de erro ocorre se uma posição passada por parâmetro para uma das operações da lista for inválida. As razões que podem levar uma posição a ser inválida incluem:

- *p = null*
- *p* foi previamente eliminado da lista
- *p* é uma posição de uma lista diferente.
- *p* é a primeira posição da lista e chama-se `prev(p)`
- *p* é a última posição da lista e chama-se `next(p)`.

As operações de um TAD lista de nós são demonstradas no exemplo que segue.

**Exemplo 6.3** Apresenta-se na seqüência uma série de operações sobre uma lista  $S$  inicialmente vazia. Usam-se as variáveis  $p_1$ ,  $p_2$ , e assim por diante, para denotar as diferentes posições, e identifica-se o objeto atualmente armazenado em tal posição entre parênteses.

Operação	Saída	$S$
addFirst(8)		(8)
first()	$p_1(8)$	(8)
addAfter( $p_1$ , 5)		(8,5)
next( $p_1$ )	$p_2(5)$	(8,5)
addBefore( $p_2$ , 3)		(8,3,5)
prev( $p_2$ )	$p_3(3)$	(8,3,5)
addFirst(9)		(9,8,3,5)
last()	$p_2(5)$	(9,8,3,5)
remove(first())	9	(8,3,5)
set( $p_3$ , 7)	3	(8,7,5)
addAfter(first(), 2)		(8,2,7,5)

O TAD lista de nodos, com sua idéia de posição embutida, é útil em um grande número de configurações. Por exemplo, um programa que modele várias pessoas jogando cartas pode representar a mão de cada jogador como uma lista de nodos. Uma vez que a maioria das pessoas gosta de manter as cartas do mesmo naipe juntas, inserir e remover as cartas da mão de uma pessoa pode ser implementado usando os métodos do TAD lista de nodos, com as posições sendo determinadas pela ordem natural dos naipes. De forma semelhante, um simples editor de texto embute a noção de inserção e remoção baseadas em posição, uma vez que normalmente os editores executam suas operações relativas a um *cursor*, que representa a posição atual na lista dos caracteres do texto que está sendo editado.

Uma interface Java representando o TAD posição é apresentada no Trecho de código 6.4.

```
public interface Position<E> {
    /** Retorna o elemento armazenado nesta posição. */
    E element();
}
```

**Trecho de código 6.4** Interface Java do TAD posição.

Uma interface para o TAD lista de nodos, chamada `PositionList`, é fornecida no Trecho de código 6.5. Essa interface usa as seguintes exceções para indicar condições de erro.

`BoundaryViolationException`: lançada se for feita uma tentativa de acessar um elemento cuja posição esta fora do intervalo de posições da lista (por exemplo, chamando-se o método `next` sobre a última posição da seqüência).

`InvalidPositionException`: lançada se a posição fornecida como argumento não é válida (por exemplo, se é uma referência nula ou não tem lista associada).

```
public interface PositionList<E>{
    /** Retorna o número de elementos desta lista. */
    public int size();
    /** Retorna quando a lista está vazia. */
    public boolean isEmpty();
    /** Retorna o primeiro nodo da lista. */
    public Position<E> first();
    /** Retorna o último nodo da lista. */
    public Position<E> last();
```

```

public Position<E> last();
/** Retorna o nodo que segue um dado nodo da lista. */
public Position<E> next(Position<E> p)
    throws InvalidPositionException, BoundaryViolationException;
/** Retorna o nodo que antecede um dado nodo da lista. */
public Position<E> prev(Position<E> p)
    throws InvalidPositionException, BoundaryViolationException;
/** Insere um elemento no inicio da lista, retornando uma posição nova. */
public void addFirst(E e);
/** Insere um elemento na última posição, retornando uma posição nova */
public void addLast(E e);
/** Insere um elemento após um dado elemento da lista. */
public void addAfter(Position<E> p, E e)
    throws InvalidPositionException;
/** Insere um elemento antes de um dado elemento da lista. */
public void addBefore(Position<E> p, E e)
    throws InvalidPositionException;
/** Remove um nodo da lista, retornando o elemento lá armazenado */
public E remove(Position<E> p) throws InvalidPositionException;
/** Substitui o elemento armazenado em um determinado nodo, retornando o elemento que
    estava lá armazenado */
public E set(Position<E> p, E e) throws InvalidPositionException;
}

```

**Trecho de código 6.5** Interface de Java para o TAD lista de nodos

### Outro adaptador de deque

Com respeito à discussão relativa ao TAD lista de nodos, observa-se que este TAD é suficiente para definir uma classe adaptadora para o TAD deque, como se pode ver na Tabela 6.3.

<i>Método do deque</i>	<i>Implementação com métodos da lista nodo</i>
size( ), isEmpty()	size( ), isEmpty()
getFirst()	first( ).element()
getLast()	last( ).element()
addFirst( <i>e</i> )	addFirst( <i>e</i> )
addLast( <i>e</i> )	addLast( <i>e</i> )
removeFirst()	remove(first( ))
removeLast()	remove(last( ))

**Tabela 6.3** Implementação de um deque usando uma lista nodo.

#### 6.2.4 Implementação usando lista duplamente encadeada

Supondo que se deseja implementar um TAD lista, baseado em uma lista duplamente encadeada (Seção 3.3), pode-se simplesmente fazer com que os nodos da lista implementem o TAD posição. Isto é, cada nodo implementa a interface Position e, por consequência, um método chamado element( ), que retorna o elemento armazenado no nodo. Assim, os próprios nodos atuam como posições. Eles são vistos internamente pela lista encadeada como nodos, mas do ponto de vista externo são vistos apenas como posições. Do ponto de vista interno, cada nodo tem as variáveis de instância prev e next, que se referem, respectivamente, aos nodos ante-

cessor e sucessor de  $v$  (que podem, de fato, ser os nodos sentinela inicial ou final, que marcam o início e o fim da lista). Em vez de usar as variáveis  $prev$  e  $next$  diretamente, definem-se os métodos  $getPrev$ ,  $setPrev$ ,  $getNext$  e  $setNext$  para o nodo, de maneira a acessar e modificar estas variáveis.

No Trecho de código 6.6, apresenta-se a classe Java `DNode` para os nodos de uma lista duplamente encadeada que implementa o TAD posição. Essa classe é similar à classe `DNode`, apresentada no Trecho de código 3.17, exceto porque agora os nodos armazenam um elemento genérico em vez de uma string. Observa-se que as variáveis de instância  $prev$  e  $next$  desta classe são referências privadas para outros objetos `DNode`.

```
public class DNode<E> implements Position<E> {
    private DNode<E> prev, next; // Referencia para os nodos anterior e posterior
    private E element; // Elemento armazenado nesta posição
    /** Construtor */
    public DNode(DNode<E> newPrev, DNode<E> newNext, E elem) {
        prev = newPrev;
        next = newNext;
        element = elem;
    }
    // Método da interface Position
    public E element() throws InvalidPositionException {
        if ((prev == null) && (next == null))
            throw new InvalidPositionException("Position is not in a list!");
        return element;
    }
    // Métodos de acesso
    public DNode<E> getNext() { return next; }
    public DNode<E> getPrev() { return prev; }
    // Métodos de atualização
    public void setNext(DNode<E> newNext) { next = newNext; }
    public void setPrev(DNode<E> newPrev) { prev = newPrev; }
    public void setElement(E newElement) { element = newElement; }
}
```

**Trecho de código 6.6** Classe `DNode` representando um nodo de uma lista duplamente encadeada que implementa a interface `Position` (TAD).

Dada uma posição  $p$  em  $S$ , pode-se “desempacotar”  $p$  para revelar o nodo  $v$ . Isso é possível **convertendo** a posição para um nodo. Uma vez que se tem um nodo  $v$ , pode-se, por exemplo, implementar o método  $prev(p)$ , usando  $v.getPrev()$  (a menos que o nodo retornado por  $v.getPrev()$  seja o nodo inicial, caso em que se sinaliza um erro). Consequentemente, posições em uma lista duplamente encadeada podem ser suportadas em um estilo orientado a objetos sem necessidade de tempo ou espaço adicional.

Considere-se, a seguir, como se pode implementar o método  $addAfter(p,e)$  para inserir um elemento  $e$  depois da posição  $p$ . Da mesma forma que discutido na Seção 3.3.1, cria-se um novo nodo  $v$  para abrigar o elemento  $e$ , liga-se  $v$  no seu lugar da lista, e então atualizam-se as referências  $next$  e  $prev$  de  $v$  com seus novos vizinhos. Este método é apresentado no Trecho de código 6.7 e ilustrado (novamente) na Figura 6.6. Lembrando o uso das sentinelas (Seção 3.3), observa-se que este algoritmo funciona mesmo se  $p$  for a última posição real.

**Algoritmo**  $addAfter(p,e)$ :

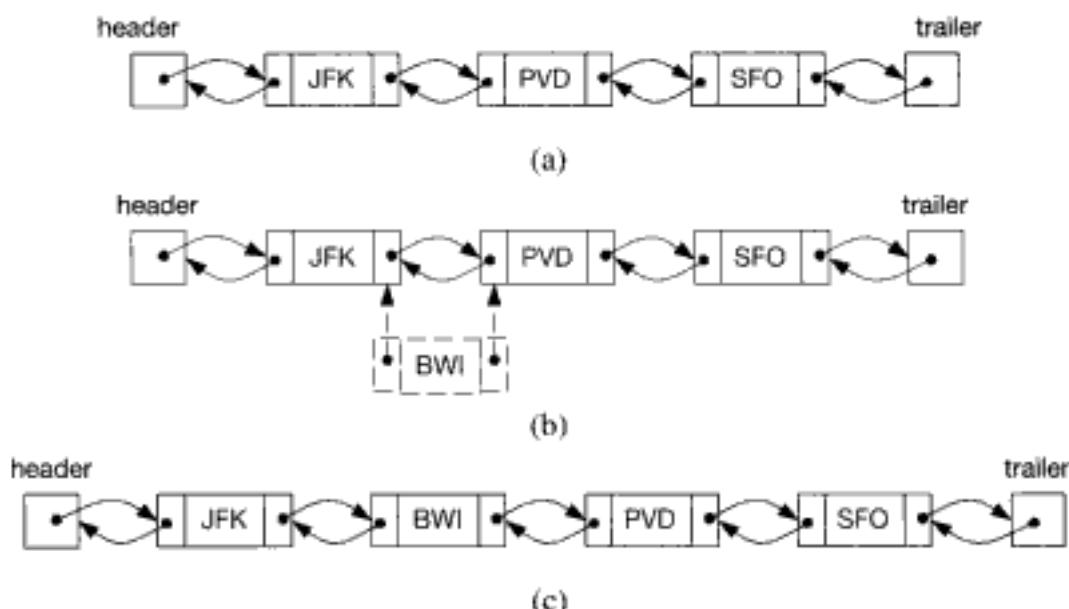
Cria um nodo novo  $e$   
 $v.setElement(e)$

```

v.setPrev(p)           {conecta v com seu antecessor}
v.setNext(p.getNext()) {conecta v com seu sucessor}
(p.getNext()).setPrev(v) {conecta o antigo sucessor de v com p}
p.setNext(v)           {conecta p com seu novo sucessor, v}

```

**Trecho de código 6.7** Inserção de um elemento *v* após uma posição *p* em uma lista encadeada.



**Figura 6.6** Acrescentando um novo nodo após a posição “JFK”: (a) antes da inserção; (b) criação do novo nodo *v* com o elemento “BWI” e concatenação do mesmo; (c) após a inserção.

Os algoritmos para os métodos *addBefore*, *addFirst* e *addLast* são similares aos do método *addAfter*. O detalhamento fica para o Exercício R-6.5.

A seguir, considera-se o método *remove(p)*, que remove o elemento e armazenado na posição *p*. Da mesma forma que apresentado na Seção 3.3.2, para executar essa operação, conectam-se os dois vizinhos de *p* de maneira que os mesmos se referenciem entre si como novos vizinhos – desconectando *p*. Observa-se que depois que *p* é desconectado, nenhum nodo estará apontando para o mesmo, logo o sistema de coleta de lixo pode recuperar o espaço de *p*. Este algoritmo é apresentado no Trecho de código 6.8 e representado na Figura 6.7. Lembrando o uso de sentinelas, salienta-se que este algoritmo trabalha mesmo que *p* seja a primeira, a última ou a única posição real da lista.

**Algoritmo** *remove(p)*:

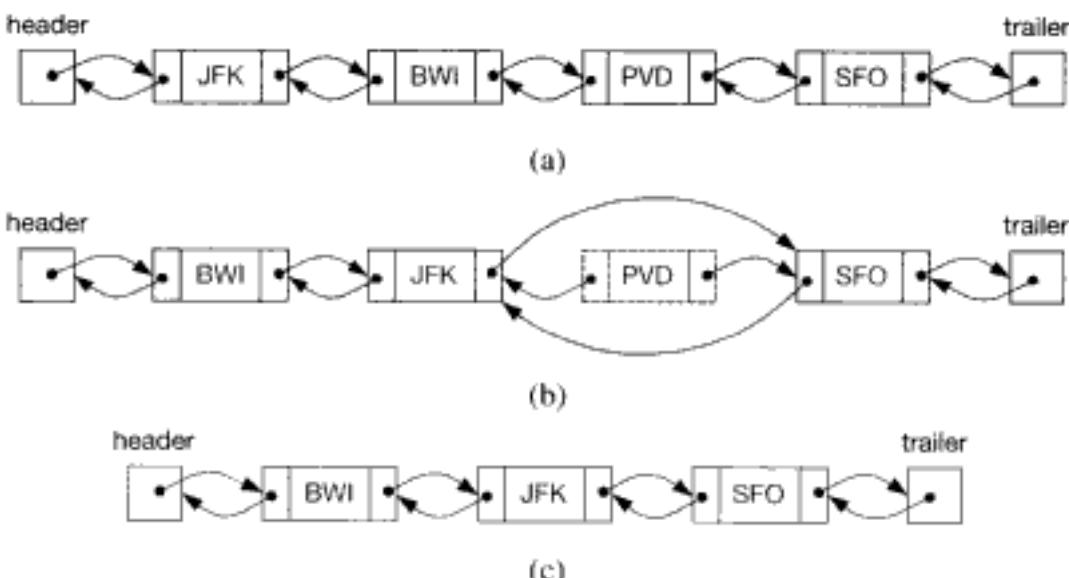
```

t ← p.element           {uma variável temporária para abrigar o valor de retorno}
(p.getPrev()).setNext(p.getNext())      {desconectando p}
(p.getNext()).setPrev(p.getPrev())
p.setPrev(null)          {invalidando a posição p}
p.setNext(null)
return t

```

**Trecho de código 6.8** Removendo um elemento *v* armazenado na posição *p* de uma lista encadeada.

Concluindo, usando uma lista duplamente encadeada, podem ser executados todos os métodos do TAD lista em tempo  $O(1)$ . Logo, uma lista duplamente encadeada é uma implementação eficiente do TAD lista.



**Figura 6.7** Removendo o objeto armazenado na posição “PVD”: (a) antes da remoção; (b) desconectando o nodo velho; (c) depois da remoção (e da coleta de lixo).

### Implementação de uma lista de nodos em Java

Partes do código da classe Java `NodePositionList<E>` que implementa o TAD lista de nodos usando uma lista duplamente encadeada, são apresentadas nos Trechos de código 6.9-6.11. O Trecho de código 6.9 apresenta as variáveis de instância de `NodePositionList`, seu construtor, e o método `checkPosition`, que executa algumas verificações de segurança e “desempacota” uma posição, convertendo-a novamente em um objeto `DNode`. O Trecho de código 6.10 apresenta métodos de acesso e atualização adicionais. O Trecho de código 6.11 apresenta métodos de atualização adicionais.

```

public class NodePositionList<E> implements PositionList<E> {
    protected int numElts; // Número de elementos na lista
    protected DNode<E> header, trailer; // Sentinelas especiais
    /** Construtor que cria uma lista vazia; tempo O(1) */
    public NodePositionList() {
        numElts = 0;
        header = new DNode<E>(null, null, null); // cria a cabeça
        trailer = new DNode<E>(header, null, null); // cria a cauda
        header.setNext(trailer); // faz a cabeça e a cauda apontarem um para o outro
    }
    /** Verifica se a posição é válida para esta lista e a converte para
     * DNode se for válida; tempo O(1) */
    protected DNode<E> checkPosition(Position<E> p)
        throws InvalidPositionException {
        if (p == null)
            throw new InvalidPositionException
                ("Null position passed to NodeList");
        if (p == header)
            throw new InvalidPositionException
                ("The header node is not a valid position");
        if (p == trailer)
            throw new InvalidPositionException
                ("The trailer node is not a valid position");
        try {

```

```

DNode<E> temp = (DNode<E>) p;
if ((temp.getPrev() == null) || (temp.getNext() == null))
    throw new InvalidPositionException
    ("Position does not belong to a valid NodeList");
return temp;
} catch (ClassCastException e) {
    throw new InvalidPositionException
    ("Position is of wrong type for this list");
}
}
}

```

**Trecho de código 6.9** Partes da implementação da classe NodePositionList que implementam o TAD lista de nodos usando uma lista duplamente encadeada. (Continua nos Trechos de código 6.10 e 6.11.)

```

/** Retorna a quantidade de elementos na lista; tempo O(1) */
public int size() { return numElts; }
/** Retorna quando a lista está vazia; tempo O(1) */
public boolean isEmpty() { return (numElts == 0); }
/** Retorna a primeira posição da lista; tempo O(1) */
public Position<E> first()
    throws EmptyListException {
    if (isEmpty())
        throw new EmptyListException("List is empty");
    return header.getNext();
}
/** Retorna a posição que antecede a fornecida; tempo O(1) */
public Position<E> prev(Position<E> p)
    throws InvalidPositionException, BoundaryViolationException {
    DNode<E> v = checkPosition(p);
    DNode<E> prev = v.getPrev();
    if (prev == header)
        throw new BoundaryViolationException
        ("Cannot advance past the beginning of the list");
    return prev;
}
/** Insere o elemento antes da posição fornecida, retornando
 * a nova posição; tempo O(1) */
public void addBefore(Position<E> p, E element)
    throws InvalidPositionException { // 
    DNode<E> v = checkPosition(p);
    numElts++;
    DNode<E> newNode = new DNode<E>(v.getPrev(), v, element);
    v.getPrev().setNext(newNode);
    v.setPrev(newNode);
}

```

**Trecho de código 6.10** Partes da implementação da classe NodePositionList que implementam o TAD lista de nodos usando uma lista duplamente encadeada. (Continuação do Trecho de código 6.9. Continua no Trecho de código 6.11.)

```

/** Insere o elemento dado no inicio da lista, retornando
 * a nova posição; tempo O(1) */
public void addFirst(E element) {
    numElts++;

```

```
DNode<E> newNode = new DNode<E>(header, header.getNext( ), element);
header.getNext( ).setPrev(newNode);
header.setNext(newNode);
}
/** Remove da lista a posição fornecida; tempo O(1) */
public E remove(Position<E> p)
    throws InvalidPositionException {
DNode<E> v = checkPosition(p);
numElts--;
DNode<E> vPrev = v.getPrev( );
DNode<E> vNext = v.getNext( );
vPrev.setNext(vNext);
vNext.setPrev(vPrev);
E vElem = v.element( );
// Desconecta a posição da lista e marca-a como inválida
v.setNext(null);
v.setPrev(null);
return vElem;
}
/** Substitui o elemento da posição fornecida por um novo
 * e retorna o elemento velho; tempo O(1) */
public E set(Position<E> p, E element)
    throws InvalidPositionException {
DNode<E> v = checkPosition(p);
E oldElm = v.element();
v.setElement(element);
return oldElm;
}
```

**Trecho de código 6.11** Partes da classe `NodePositionList` que implementam o TAD lista de nodos usando uma lista duplamente encadeada. (Continuação dos Trechos de código 6.9 e 6.10.) Observa-se que o mecanismo usado para tornar inválida uma posição no método `remove` é consistente com aquele utilizado nas verificações da função de conveniência `checkPosition`.

---

## 6.3 Iteradores

Uma operação típica sobre um vetor, uma lista ou uma seqüência é percorrer seus elementos em ordem, um de cada vez, para, por exemplo, procurar um elemento específico.

---

### 6.3.1 Os tipos abstratos de dados iterador e iterável

Um *iterador* é um padrão de projeto de software que abstrai o processo de busca sobre uma coleção de elementos, um de cada vez. Um iterador consiste em uma seqüência  $S$ , um elemento corrente de  $S$  e uma forma de avançar para o próximo elemento de  $S$ , tornando-o o elemento corrente. Logo, um iterador estende o conceito do TAD posição, introduzido na Seção 6.2. De fato, uma posição pode ser entendida como um iterador que não é capaz de se deslocar. Um iterador encapsula os conceitos de “lugar” e “próximo” em uma coleção de objetos.

Define-se o TAD *iterador* como suportando os dois métodos que seguem:

hasNext: Testa a existência de elementos remanescentes no iterador.

nextObject: Retorna o próximo elemento do iterador.

Observa-se que o TAD iterador usa a noção de elemento corrente quando percorre uma sequência. O primeiro elemento de um iterador é fornecido pela primeira chamada ao método next, supondo, é claro, que o iterador contenha pelo menos um elemento.

Um iterador oferece um esquema unificado para acessar todos os elementos de uma coleção de objetos de uma forma independente da organização interna da coleção. Um iterador para uma lista arranjo, lista ou sequência deve retornar os elementos de acordo com sua ordenação linear.

### Iteradores simples em Java

Java fornece um iterador através de sua interface `java.util.Iterator`. Observa-se que a classe `java.util.Scanner` (Seção 1.6) implementa esta interface. Esta interface suporta um método adicional (opcional) para remover da coleção elementos previamente retornados. Essa funcionalidade (remoção de elementos por meio de um iterador) é bastante controversa do ponto de vista de orientação a objetos; logo, não é de surpreender que sua implementação por classes seja opcional. Java também oferece a interface `java.util.Enumeration`, historicamente mais antiga que a interface `Iterator`, e que usa os nomes `hasMoreElements()` e `nextElement()`.

### O tipo abstrato de dados iterável

Com o objetivo de fornecer um mecanismo genérico unificado para percorrer uma estrutura de dados, os TADs que armazenam coleções de objetos devem suportar o seguinte método:

`iterator()`: Retorna um iterador para os elementos da coleção.

Esse método é oferecido pela interface `java.util.ArrayList`. Na verdade, este método é tão importante, que existe uma interface inteira, `java.lang.Iterable`, que contém apenas este método. Este método torna simples especificar computações que necessitem percorrer os elementos de uma lista. Para garantir que a lista suporta os métodos anteriores, por exemplo, pode-se adicionar este método à interface `PositionList`, como mostrado no Trecho de código 6.12. Neste caso, pode-se querer também declarar que `PositionList` estende `Iterable`. Consequentemente, assume-se que as listas arranjo e as listas de nodos suportam o método `iterator()`.

```
public interface PositionList<E> extends Iterable<E> {
    // ...todos os outros métodos do TAD lista ...
    /** Retorna um iterador sobre todos os elementos da lista. */
    public Iterator<E> iterator();
}
```

**Trecho de código 6.12** Acrescentando o método `iterator` na interface `PositionList`.

Fornecida tal definição para `PositionList`, pode-se usar o iterador retornado pelo método `iterator()` para criar uma representação string da lista de nodos, como mostrado no Trecho de código 6.13.

```
/** Retorna a representação textual de uma lista de nodos */
public static <E> String toString(PositionList<E> l) {
    Iterator<E> it = l.iterator();
    String s = "[";
    while (it.hasNext()) {
        s += it.next(); // coerção implícita do próximo elemento para uma string
        if (it.hasNext())
            s += ", ";
    }
    s += "]";
    return s;
}
```

**Trecho de código 6.13** Exemplo de iterador Java usado para converter uma lista de nodos em uma string.

---

### 6.3.2 O laço de Java para-cada

Uma vez que executar um laço sobre os elementos retornados por um iterador é uma construção muito comum, Java provê uma notação simplificada para tais laços, chamada de *laço para-cada*. A sintaxe de tal laço é a que segue:

```
for (Tipo nome : expressão)
    comandos do laço
```

onde *expressão* corresponde a uma coleção que implementa a interface `java.lang.Iterable`, *Tipo* é o tipo do objeto retornado pelo iterador desta classe e *nome* é o nome de uma variável que irá receber os valores dos elementos deste iterador nos *comandos do laço*. Esta notação é apenas uma simplificação do que segue:

```
for (Iterator<Tipo> it = expressão.iterator(); it.hasNext();) {
    Type name = it.next();
    comandos do laço
}
```

Por exemplo, se existe uma lista, `values`, de objetos `Integer`, e `values` implementa `java.lang.Iterable`, então se podem somar todos os inteiros de `values` da seguinte forma:

```
List<Integer> values;
// ... comandos que criam uma nova lista de valores e a preenchem com Integers ...
int sum = 0;
for (Integer i : values)
    sum += i; // unboxing permite isso
```

Pode-se ler o laço acima como “para cada inteiro *i* em `values`, execute o corpo do laço” (neste caso, somar *i* em `sum`).

Além desta forma de laço recém descrita, Java também permite que laços para-cada sejam definidos quando a *expressão* é um arranjo do tipo *Tipo*, o qual, neste caso, pode ser tanto um tipo base como um objeto. Por exemplo, podem-se totalizar os inteiros de um arranjo, `v`, que armazena os primeiros dez inteiros positivos como segue:

```
int[] v = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
int total = 0;
for (int i : v)
    total += i;
```

---

### 6.3.3 Implementando iteradores

Uma maneira de implementar um iterador para uma coleção de elementos é criar uma “foto” da coleção e iterar sobre a mesma. Esta abordagem irá envolver o armazenamento da coleção em uma estrutura de dados separada que suporte acesso seqüencial a seus elementos. Por exemplo, podem se inserir todos os elementos de uma coleção em uma pilha, caso em que o método `hasNext()` irá corresponder a `isEmpty()`, e `next` irá corresponder a `enqueue()`. Usando esta abordagem, o método `iterator()` irá consumir um tempo  $O(n)$  para uma coleção de tamanho  $n$ . Uma vez que o custo da cópia é relativamente alto, prefere-se, na maioria dos casos, fazer os iteradores operarem sobre a própria coleção, e não sobre uma cópia.

Quando se implementa esta abordagem direta, é necessário apenas manter o ponto da coleção para onde o cursor do iterador aponta. Logo, criar um iterador novo, neste caso, envolve apenas a criação de um objeto iterador que represente um cursor posicionado antes do primeiro elemento

da coleção. Da mesma forma, executar o método `next()` envolve o retorno do próximo elemento, se existir, e a movimentação do cursor para que deixe para trás a posição deste elemento. Assim, nesta abordagem, a criação de um iterador consome tempo  $O(1)$ , da mesma forma que cada um dos métodos do iterador. Apresenta-se uma classe que implementa tal iterador no Trecho de código 6.14 e no Trecho de código 6.15, e como este iterador pode ser usado para implementar o método `iterator` da classe `NodePositionList`.

```
public class ElementIterator<E> implements Iterator<E> {
    protected PositionList<E> list; // a lista subjacente
    protected Position<E> cursor; // a próxima posição
    /** Cria o elemento iterador sobre a lista fornecida. */
    public ElementIterator(PositionList<E> L) {
        list = L;
        cursor = (list.isEmpty( ))? null : list.first();
    }
    public boolean hasNext() { return (cursor != null); }
    public E next() throws NoSuchElementException {
        if (cursor == null)
            throw new NoSuchElementException("No next element");
        E toReturn = cursor.element();
        cursor = (cursor == list.last( ))? null : list.next(cursor);
        return toReturn;
    }
}
```

**Trecho de código 6.14** Classe de um elemento iterador para `PositionList`.

```
/** Retorna um iterador sobre todos os elementos da lista. */
public Iterator<E> iterator() { return new ElementIterator<E>(this); }
```

**Trecho de código 6.15** O método `iterator` da classe `NodePositionList`.

### Iteradores de posição

Para os TADs que suportam a noção de posição, tais como os TADs lista e seqüência, pode-se fornecer o seguinte método:

`positions()`: retorna um objeto `Iterable` (tal como uma lista arranjo ou uma lista de nodos) contendo as posições da coleção como elementos.

Um iterador retornado por este método permite que se percorram as posições de uma lista. Para garantir que uma lista de nodos suporte este método, deve-se acrescentar à mesma a interface `PositionList`, como demonstrado no Trecho de código 6.16. Então, será possível, por exemplo, acrescentar a implementação deste método a `NodePositionList`, como mostrado no Trecho de código 6.17. Este método usa a própria classe `NodePositionList` para criar uma lista que contém as posições da lista original como seus elementos. Retornando esta lista de posições como um objeto `Iterable`, permite que se chame `iterator()` sobre este objeto para obter um iterador sobre as posições da lista original.

```
public interface PositionList<E> extends Iterable<E> {
    // ...todos os outros métodos do TAD lista
    /** Retorna uma coleção iterável de todos os nodos da lista. */
    public Iterable<Position<E>> positions();
}
```

**Trecho de código 6.16** Acrescentando o método `iterator` na interface `PostionList`.

```
/** Retorna uma coleção iterável de todos os nodos da lista. */
public Iterable<Position<E>> positions() {           // cria uma lista de posições
    PositionList<Position<E>> P = new NodePositionList<Position<E>>();
    if (!isEmpty( )) {
        Position<E> p = first( );
        while (true) {
            P.addLast(p); // acrescenta a posição p como último elemento da lista P
            if (p == last( ))
                break;
            p = next(p);
        }
    }
    return P; // retorna P como objeto iterável
}
```

Trecho de código 6.17 O método `positions()` da classe `NodePositionList`.

O método `iterador()` retornado por este e outros objetos `Iterable` define um tipo restrito de iterador que permite apenas uma passagem sobre os elementos. Entretanto, iteradores mais poderosos também podem ser definidos, permitindo o deslocamento para frente e para trás sobre uma certa ordem de elementos.

---

#### 6.3.4 Iteradores de lista em Java

A classe `java.util.LinkedList` não expõe o conceito de posição para os usuários de sua API. Em vez disso, a forma preferida de acessar e atualizar um objeto `LinkedList` em Java, sem usar índices, é usando um `ListIterator`, que é gerado pela lista encadeada através do método `listIterator()`. Tal iterador permite que se percorra a lista para frente e para trás, bem como métodos de atualização. A posição corrente é entendida como sendo antes do primeiro elemento, entre dois elementos ou depois do último elemento. Isto é, ela usa um *cursor* de lista, parecido com a maneira pela qual um cursor de tela é visto, localizado entre dois caracteres da tela. Mais especificamente, a interface `Java.util.ListIterator` inclui os seguintes métodos.

- `add(e)`: acrescenta o elemento *e* na posição corrente do iterador
- `hasNext()`: True se e somente se existe um elemento após a posição corrente do iterador
- `hasPrevious()`: True se e somente se existe um elemento antes da posição corrente do iterador
- `previous()`: retorna o elemento *e* que antecede a posição corrente e faz com que a posição corrente seja a que antecede *e*
- `next()`: retorna o elemento *e* que sucede a posição corrente e faz com que a posição corrente seja a que sucede *e*
- `nextIndex()`: retorna o índice do próximo elemento
- `previousIndex()`: retorna o índice do elemento anterior
- `set(e)`: substitui o elemento retornado pela última operação `next` ou `previous` por *e*
- `remove()`: remove o elemento retornado pela última operação `next` ou `previous`

É um risco usar vários iteradores sobre a mesma lista enquanto se modifica seu conteúdo. Se inserções, deleções ou substituições são requeridas em vários “lugares” de uma lista, é mais se-

guro usar posições para especificar estas localizações. Mas a classe `java.util.LinkedList` não expõe seus objetos posição para o usuário. Assim, para evitar o risco de modificar uma lista que tenha criado vários iteradores (através de chamadas para seu método `iterator()`), os objetos `java.util.Iterator` tem um recurso de “falha rápida”, que imediatamente invalida um iterador se a coleção subjacente for modificada de forma inesperada. Por exemplo, se um objeto `Java.util.LinkedList L` retornou cinco iteradores diferentes, e um deles modifica `L`, então os outros quatro se tornam imediatamente inválidos. Isto é, Java permite que vários iteradores de lista estejam percorrendo uma lista encadeada `L` ao mesmo tempo, mas se um deles modifica `L` (usando os métodos `add`, `set` ou `remove`), então todos os demais iteradores sobre `L` se tornam inválidos. Da mesma forma, se `L` é modificado por um de seus próprios métodos de atualização, então todos os iteradores existentes para `L` imediatamente se tornam inválidos.

### A interface `java.util.List` e sua implementação

Java provê funcionalidades semelhantes aos TADs lista e lista de nodos na interface `Java.util.List`, que é implementada usando-se arranjos em `java.util.ArrayList` e empregando uma lista encadeada em `java.util.LinkedList`. Existem alguns problemas com estas duas implementações que são explorados com maior profundidade na próxima seção. Além disso, Java usa iteradores para obter uma funcionalidade similar aquela que o TAD lista deriva a partir das posições. A Tabela 6.4 mostra os métodos correspondentes entre os TADs de lista (arranjo e de nodos) e as interfaces `java.util.List` e `ListIterator`, com observações a respeito de suas implementações nas classes `java.util.ArrayList` e `java.util.LinkedList`.

## 6.4 Os TADs de lista e o framework de coleções

Nesta seção, discutem-se TADs de listas genéricos, que combinam os métodos dos TADs deque, lista arranjo e ou lista de nodos. Antes de descrever tais TADs, apresenta-se o contexto no qual estão inseridos.

### 6.4.1 O framework de coleções do Java

Java fornece um pacote de interfaces e classes de estruturas de dados que, em conjunto, definem o *framework de coleções de Java*. Este pacote, `java.util`, inclui versões de várias das estruturas de dados discutidas neste livro, algumas das quais já foram apresentadas e outras que serão discutidas no restante do livro. Em especial, o pacote `java.util` inclui as seguintes interfaces:

**Collection:** uma interface genérica para qualquer estrutura de dados que contenha uma coleção de elementos. Estende `java.lang.Iterable`; logo, inclui o método `iterator()`, que retorna um iterador sobre os elementos da coleção.

**Iterator:** uma interface para o TAD iterador simples.

**List:** uma interface que estende `Collection` para incluir o arranjo do TAD lista. Também inclui um método `listIterator` para retornar um objeto `ListIterator` para esta lista.

**ListIterator:** interface de iterador que permite tanto caminhamento para frente como para trás sobre uma lista, bem como métodos de atualização baseados em cursor.

**Map:** uma interface para mapear chaves para valores. Este conceito e interface são discutidos na Seção 9.1.

Método do TAD Lista	Método de java.util.List	Método de ListIterator	Observações
size()	size()		Tempo $O(1)$
isEmpty()	isEmpty()		Tempo $O(1)$
get( <i>i</i> )	get( <i>i</i> )		<i>A</i> é $O(1)$ , <i>L</i> é $O(\min\{i, n - i\})$
first()	listIterator()		O primeiro elemento é o próximo
last()	listIterator(size())		O último elemento é o antecessor
prev( <i>p</i> )		previous()	Tempo $O(1)$
next( <i>p</i> )		next()	Tempo $O(1)$
set( <i>p, e</i> )		set( <i>e</i> )	Tempo $O(1)$
set( <i>i, e</i> )	set( <i>i, e</i> )		<i>A</i> é $O(1)$ , <i>L</i> é $O(\min\{i, n - i\})$
add( <i>i, e</i> )	add( <i>i, e</i> )		Tempo $O(n)$
remove( <i>i</i> )	remove( <i>i</i> )		<i>A</i> é $O(1)$ , <i>L</i> é $O(\min\{i, n - i\})$
addFirst( <i>e</i> )	add(0, <i>e</i> )		<i>A</i> é $O(n)$ , <i>L</i> é $O(1)$
addFirst( <i>e</i> )	addFirst( <i>e</i> )		Existe apenas em <i>L</i> , $O(1)$
addLast( <i>e</i> )	add( <i>e</i> )		Tempo $O(1)$
addLast( <i>e</i> )	addLast( <i>e</i> )		Existe apenas em <i>L</i> , $O(1)$
addAfter( <i>p, e</i> )		add( <i>e</i> )	A inserção é na posição do cursor; <i>A</i> é $O(n)$ , <i>L</i> é $O(1)$
addBefore( <i>p, e</i> )		add( <i>e</i> )	A inserção é na posição do cursor; <i>A</i> é $O(n)$ , <i>L</i> é $O(1)$
remove( <i>p</i> )		remove()	A remoção é na posição do cursor; <i>A</i> é $O(n)$ , <i>L</i> é $O(1)$

**Tabela 6.4** Correspondência entre os métodos dos TADs lista arranjo e lista de nodos, e as interfaces de java.util, List e ListIterator. Usa-se *A* e *L* como abreviatura de java.util.ArrayList e java.util.LinkedList (ou seus tempos de execução).

**Queue:** Uma interface para o TAD fila, mas usando métodos com nomes diferentes. Os métodos incluem peek() (o mesmo que front()), offer(*e*) (o mesmo que enqueue(*e*)) e poll(), o mesmo que dequeue().

**Set:** uma interface que estende Collection para conjuntos.

O framework de coleções de Java também inclui várias classes concretas, implementando várias combinações das interfaces acima. Em vez de se colocar aqui uma lista de cada uma destas classes, elas serão discutidas em locais mais apropriados deste livro. Um tópico que se deseja esgotar agora, porém, é que qualquer classe que implementa a interface java.util.Collection também implementa a interface java.lang.Iterable; assim, ela inclui um método iterator, e pode ser usada em um laço para-cada. Além disso, qualquer classe que implementa a interface java.util.List também inclui o método listIterator. Como observado anteriormente, tais interfaces são úteis para se percorrer os elementos de uma coleção ou lista.

### 6.4.2 A classe `java.util.LinkedList`

A classe `java.util.LinkedList` contém vários métodos, incluindo todos os métodos do TAD deque (Seção 5.3) e todos os métodos do TAD lista arranjo (Seção 6.1). Além disso, como mencionado anteriormente, ele também oferece funcionalidades similares àquelas do TAD lista de nodos por meio do uso de seu iterador de lista.

#### Performance da classe `java.util.LinkedList`

A documentação da classe `java.util.LinkedList` torna claro que esta classe é implementada usando-se uma lista duplamente encadeada. Assim, todos os métodos de atualização do iterador de lista associado executam em tempo  $O(1)$ . Da mesma forma, todos os métodos do TAD deque também executam em  $O(1)$ , uma vez que envolvem apenas a atualização ou consulta da lista em seus extremos. Mas os métodos do TAD lista arranjo também estão incluídos em `java.util.LinkedList` e, em geral, não são adequados para uma implementação baseada em listas duplamente encadeadas.

Em especial, uma vez que uma lista encadeada não permite o acesso indexado a seus elementos, executar a operação `get(i)`, para retornar o elemento de índice  $i$ , requer que se percorra toda a lista a partir de uma das extremidades, contando para cima ou para baixo, até encontrar o nodo que armazena o elemento de índice  $i$ . Uma otimização superficial seria começar a busca a partir da extremidade mais próxima da lista, obtendo então um tempo de execução que é

$$O(\min(i+1, n-i)),$$

onde  $n$  é a quantidade de elementos na lista. O pior caso para este tipo de pesquisa ocorre quando

$$r = \lfloor n/2 \rfloor.$$

Assim, o tempo de execução permanece  $O(n)$ .

As operações `add(i,e)` e `remove(i)` também devem promover uma busca para localizar o nodo armazenando o elemento de índice  $i$ , e então inserir ou deletar um nodo. Os tempos de execução destas implementações de `add(i,e)` e `remove(i)` são da mesma forma

$$O(\min(i+1, n-i+1)),$$

o que é  $O(n)$ . Uma vantagem desta abordagem é que se  $i = 0$  ou  $i = n-1$ , como é o caso na adaptação do TAD lista arranjo para o TAD deque apresentado na Seção 6.1.1, então `add` e `remove` executam em tempo  $O(1)$ . Mas, em geral é ineficiente usar os métodos de uma lista arranjo com um objeto `java.util.LinkedList`.

### 6.4.3 Seqüências

Uma *seqüência* é um TAD que suporta todos os métodos do TAD deque (Seção 5.3) e do TAD lista arranjo (Seção 6.1). Ou seja, fornece acesso explícito aos elementos da lista, tanto por seus índices como por suas posições. Além disso, por ter essa dupla capacidade, também foram incluídos dois métodos de “transição” que relacionam colocações e posições:

`atIndex(r)`: Retorna a posição do elemento de índice  $i$ ; uma condição de erro ocorre se  $i < 0$  ou  $i > \text{size}() - 1$ .

`indexOf(p)`: Retorna o índice do elemento na posição  $p$ .

## Herança múltipla no TAD seqüênciа

A definição do TAD seqüênciа, incluindo todos os métodos de três TADs diferentes, é um exemplo de **herança múltipla** (Seção 2.4.2). Ou seja, o TAD seqüênciа herda métodos de três outros TADs “ancestrais”. Em outras palavras, seus métodos incluem a união dos métodos de seus TADs “ancestrais”. Veja o Trecho de código 6.18 para uma especificação Java do TAD seqüênciа como uma interface Java.

```
/*
 * Interface para uma seqüênciа, uma estrutura de dados que suporta
 * todas as operações de um deque, lista indexada e lista de posições.
 */
public interface Sequence<E>
    extends Deque<E>, IndexList<E>, PositionList<E> {
    /** Retorna a posição contendo o elemento em um dado índice. */
    public Position<E> atIndex(int r) throws BoundaryViolationException;
    /** Retorna o índice do elemento armazenado em uma determinada posição. */
    public int indexOf(Position<E> p) throws InvalidPositionException;
}
```

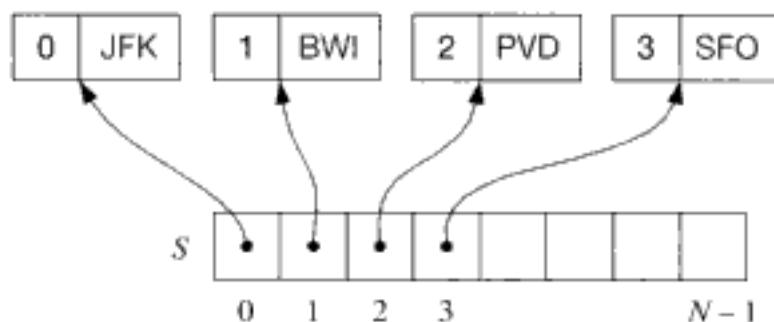
**Trecho de código 6.18** A interface seqüênciа definida usando-se herança múltipla. Inclui todos os métodos das interfaces Deque, IndexList e PositionList (definidas para qualquer tipo genérico E), e tem dois métodos adicionais.

## Implementando uma seqüênciа com um arranjo

Se uma seqüênciа  $S$  for implementada usando-se uma lista duplamente encadeada, será obtida performance semelhante a da classe `java.util.LinkedList`. Suponha, então, que se pretende implementar uma seqüênciа  $S$ , armazenando cada elemento  $e$  de  $S$  em uma célula  $A[i]$  de um arranjo  $A$ . Pode-se definir um objeto posição  $p$  para abrigar um índice  $i$ , e uma referência para o arranjo  $A$ , como variáveis de instância neste caso. O maior problema com esta abordagem, entretanto, é que as células de  $A$  não têm como referenciar suas posições correspondentes. Portanto, após uma operação `addFirst`, não existe maneira de informar as posições existentes em  $S$  de que suas colocações foram acrescidas de 1 (lembre-se que as posições de uma seqüênciа são sempre definidas em relação a seus vizinhos, não em relação a sua colocação). Assim, ao se implementar uma seqüênciа genérica usando um arranjo, precisa-se adotar uma estratégia diferente.

Considere-se uma solução alternativa na qual, em vez de armazenar os elementos de  $S$  no arranjo  $A$ , armazena-se um novo tipo de objeto posição em cada uma das células de  $A$ , e guardam-se os elementos nessas posições. O novo objeto posição  $p$  abriga o índice  $i$  e o elemento  $e$  associado com  $p$ .

Com essa estrutura de dados, apresentada na Figura 6.8, percorre-se facilmente o arranjo para atualizar a variável  $i$  de cada posição, cuja colocação foi alterada em função de inserções ou remoções.



**Figura 6.8** Implementação do TAD seqüênciа baseada em um arranjo.

## Questões de eficiência com uma seqüência baseada em arranjo

Nesta implementação de seqüência, os métodos `addFirst`, `addBefore`, `addAfter` e `remove` consomem um tempo  $O(n)$  porque é preciso deslocar a posição dos objetos para abrir espaço para as novas posições ou para preencher os vazios criados pela remoção de uma posição antiga (da mesma forma que os métodos `insert` ou `remove` baseados em índices). Todos os outros métodos baseados em posições consomem tempo  $O(1)$ .

## 6.5 Estudo de caso: a heurística mover-para-frente

Suponha que se deseja manter uma coleção de elementos, ao mesmo tempo em que se mantém o número de vezes que cada elemento é acessado. Manter esse tipo de contagem permite saber quais elementos estão entre os “dez mais” populares, por exemplo. Exemplos de tais cenários incluem um navegador Web que mantém os endereços Web mais populares (ou URLs) que um usuário visita, ou um programa de álbum de fotos que mantém uma lista das imagens mais populares para um usuário. Além disso, uma lista de favoritos pode ser usada em uma interface gráfica (GUI) para manter os botões mais usados em um menu pull-down, e então apresentar menus pull-down condensados, contendo as opções mais populares.

Em função disso, nesta seção será analisada a implementação do TAD *lista de favoritos*, que suporta os métodos `size()` e `isEmpty()` bem como os que seguem:

- `access(e)`: acessa o elemento  $e$ , incrementando seu contador de acesso e acrescenta o mesmo na lista de favoritos se ainda não estiver presente.
- `remove(e)`: remove o elemento  $e$  da lista de favoritos desde que ele já esteja lá.
- `top( $k$ )`: retorna uma coleção iterável com os  $k$  elementos mais acessados.

### 6.5.1 Usando uma lista ordenada e uma classe aninhada

A primeira implementação da lista de favoritos que será considerada (nos Trechos de código 6.19 – 6.20) é construir uma classe, `FavoritList`, que armazena as referências para os objetos acessados em uma lista encadeada, ordenada pelo número de acessos. Esta classe usa um recurso de Java que permite definir uma classe aninhada relacionada dentro da definição da classe mais externa. Esta **classe aninhada** deve ser declarada **static**, para indicar que sua definição está relacionada à classe mais externa e não a uma instância específica desta classe. O uso de classes aninhadas permite definir classes de “auxílio” ou “suporte” que podem ser protegidas de uso externo.

Neste caso, a classe aninhada `Entry` armazena para cada elemento  $e$  da lista um par  $(c,v)$ , onde  $c$  é o contador de acessos e  $v$  é uma referência de **valor** para o próprio elemento  $e$ . Cada vez que um elemento é acessado, localiza-se o mesmo na lista (inserindo-o se não for encontrado) e incrementa-se seu contador de acessos. A remoção implica na localização do elemento e sua remoção da lista encadeada. Retornar os  $k$  elementos mais acessados implica apenas copiar as entradas de valor em uma lista de resultados de acordo com sua ordem na lista encadeada interna.

```
/** Lista de elementos favoritos com seus contadores de acesso. */
public class FavoriteList<E> {
    protected PositionList<Entry<E>> fList; // Lista de entradas
    /** Construtor; tempo O(1) */
    public FavoriteList() { fList = new NodePositionList<Entry<E>>(); }
    /** Retorna a quantidade de elementos na lista; tempo O(1) */
    public int size() { return fList.size(); }
    /** Indica quando a lista está vazia; tempo O(1) */
}
```

```

public boolean isEmpty() { return fList.isEmpty(); }
/** Remove o elemento indicado desde que ele esteja na lista; tempo O(n) */
public void remove(E obj) {
    Position<Entry<E>> p = find(obj);      // procura por obj
    if (p != null)
        fList.remove(p);                      // remove a entrada
    }
/** Incrementa o contador de acesso de um dado elemento e insere o mesmo se ainda não
 * estiver presente; tempo O(n) */
public void access(E obj) {
    Position<Entry<E>> p = find(obj);      // encontra a posição de obj
    if (p != null)
        p.element().incrementCount(); // incrementa contador de acesso
    else {
        fList.addLast(new Entry<E>(obj)); // acrescenta uma nova entrada no fim
        p = fList.last();
    }
    moveUp(p);      // move a entrada para sua posição final
}
/** Encontra a posição de um dado elemento ou retorna null; tempo O(n) */
protected Position<Entry<E>> find(E obj) {
    for (Position<Entry<E>> p: fList.positions())
        if (value(p).equals(obj))
            return p; // encontrado na posição p
    return null; // não encontrado
}
/** Move a entrada para cima para sua posição correta na lista; tempo O(n) */
protected void moveUp(Position<Entry<E>> cur) {
    Entry<E> e = cur.element();
    int c = count(cur);
    while (cur != fList.first()) {
        Position<Entry<E>> prev = fList.prev(cur); // posição anterior
        if (c <= count(prev)) break; // a entrada está na posição correta
        fList.set(cur, prev.element()); // move para baixo a entrada anterior
        cur = prev;
    }
    fList.set(cur, e); // armazena a entrada em sua posição final
}

```

**Trecho de código 6.19** Classe FavoriteList. (Continua no Trecho de código 6.20)

```

/** Retorna os k elementos mais acessados, dado k; tempo O(k) */
public Iterable<E> top(int k) {
    if (k < 0 || k > size())
        throw new IllegalArgumentException("Invalid argument");
    PositionList<E> T = new NodePositionList<E>(); // lista dos top-k
    int i = 0; // contador de entradas inseridas na lista
    for (Entry<E> e: fList) {
        if (i++ >= k)
            break; // todas as k entradas foram inseridas
        T.addLast(e.value()); // acrescenta uma entrada na lista
    }
    return T;
}

```

```

/** Representação string da lista de favoritos */
public String toString() { return fList.toString(); }

/** Método auxiliar que obtém o valor de uma entrada em uma dada posição */
protected E value(Position<Entry<E>> p) { return (p.element()).value(); }

/** Método auxiliar que obtém o contador de uma entrada em uma dada posição. */
protected int count(Position<Entry<E>> p) { return (p.element()).count(); }

/** Classe aninhada que armazena os elementos e seus contadores de acesso. */
protected static class Entry<E> {
    private E value; // elemento
    private int count; // contador de acessos
    /** Construtor */
    Entry(E v) { count = 1; value = v; }
    /** Retorna o elemento */
    public E value() { return value; }
    /** Retorna o contador de acessos */
    public int count() { return count; }
    /** Incrementa o contador de acessos */
    public int incrementCount() { return ++count; }
    /** Representação string da entrada na forma [contador,valor] */
    public String toString() { return "[" + count + ", " + value + "]"; }
}
} // Fim da classe FavoriteList

```

**Trecho de código 6.20** Classe FavoriteList, incluindo a classe aninhada Entry, para representar os elementos e seus contadores de acesso. (Continuação do Trecho de código 6.19.)

### 6.5.2 Usando uma lista com a heurística mover-para-frente

A implementação anterior da lista de favoritos executava o método `access(e)` em um tempo proporcional ao índice de  $e$  na lista de favoritos. Isto é, se  $e$  é o  $k$ -ésimo elemento mais popular da lista, então acessar o mesmo consome tempo  $O(k)$ . Em seqüências de acesso da vida real, incluindo aqueles gerados pelas visitas que os usuários fazem a uma página da Web, é comum que, uma vez que um elemento foi acessado, o mesmo seja acessado novamente em breve. Diz-se que tais cenários têm **referência de localização**.

Uma **heurística** ou regra que tira vantagem da referência de localização que está presente em uma seqüência de acessos é a **heurística mover-para-frente**. Para aplicar esta heurística, cada vez que um elemento é acessado, move-se o mesmo para frente da lista. O que se espera, naturalmente, é que este elemento seja acessado novamente em breve. Considere, por exemplo, o cenário no qual existem  $n$  elementos e a seguinte série de  $n^2$  acessos:

- elemento 1 é acessado  $n$  vezes
- elemento 2 é acessado  $n$  vezes
- ...
- elemento  $n$  é acessado  $n$  vezes

Se os elementos forem armazenados ordenados pelos seus contadores de acesso, inserindo cada elemento a primeira vez que ele é acessado, então

- Cada acesso ao elemento 1 executa em tempo  $O(1)$
- Cada acesso ao elemento 2 executa em tempo  $O(2)$
- ...
- Cada acesso ao elemento  $n$  executa em tempo  $O(n)$

Logo, o tempo total para executar a série de acessos é proporcional a

$$n + 2n + 3n + \dots + n \cdot n = n(1 + 2 + 3 + \dots + n) = n \cdot \frac{n(n+1)}{2},$$

que é  $O(n^2)$ .

Por outro lado, se for usada à heurística mover-para-frente, inserindo cada elemento a primeira vez que é acessado, então

- Cada acesso ao elemento 1 executa em tempo  $O(1)$
- Cada acesso ao elemento 2 executa em tempo  $O(1)$
- ...
- Cada acesso ao elemento  $n$  executa em tempo  $O(1)$

Assim, o tempo para executar todos os acessos neste caso é  $O(n)$ . A implementação mover-para-frente, portanto, tem um tempo de acesso mais rápido para este cenário. Este benefício tem um custo, entretanto.

### Implementando a heurística mover-para-frente em Java

No Trecho de código 6.21, apresenta-se a implementação de uma lista de favoritos usando a heurística mover-para-frente. Implementa-se a abordagem mover-para-frente, neste caso, definindo-se uma classe nova, `FavoriteListMTF`, que estende a classe `FavoriteList` e sobrecarrega as definições dos métodos `moveUp` e `top`. Neste caso, o método `moveUp` simplesmente remove o elemento acessado da sua posição atual na lista encadeada, e então insere este elemento de volta no início da lista. O método `top`, por outro lado, é mais complicado.

### Problemas com a heurística mover-para-frente

Agora que a lista de favoritos não está mais sendo mantida ordenada pelo valor dos contadores de acesso, quando se buscam os  $k$  elementos mais acessados, é necessário procurar pelos mesmos. Neste caso, pode-se implementar o método `top( $k$ )` como segue:

1. Copiam-se as entradas da lista de favoritos em outra lista,  $C$ , e cria-se uma lista vazia,  $T$ .
2. Percorrem-se a lista  $C$   $k$  vezes. Em cada varredura, procura-se pela entrada de  $C$  com o maior contador de acesso, remove-se esta entrada de  $C$  e insere-se a mesma no fim de  $T$ .
3. Retorna-se a lista  $T$ .

Esta implementação do método `top` consome tempo  $O(kn)$ . Logo, quando  $k$  é uma constante, o método `top` executa em tempo  $O(n)$ . Isso ocorre, por exemplo, quando se deseja obter a lista do “dez maiores”. Entretanto, se  $k$  é proporcional a  $n$ , então `top` executa em tempo  $O(n^2)$ . Isso ocorre, por exemplo, quando se deseja a lista dos “25% maiores”.

Como a abordagem mover-para-frente é apenas uma heurística ou regra, existem seqüências de acesso nas quais o uso desta abordagem é mais lento do que a manutenção simples da lista de favoritos ordenada pelos contadores de acesso. Além disso, ela reduz a velocidade potencial dos acessos que possuem referência de localização, implicando em uma demora maior na geração do relatório dos melhores elementos.

---

#### 6.5.3 Possíveis usos de uma lista de favoritos

No Trecho de código 6.22, é apresentado um exemplo de aplicação da lista de favoritos para resolver o problema de manter as URLs mais populares a partir de uma seqüência simulada de acessos a páginas da Web. Este programa acessa um conjunto de URLs em ordem decrescente, e então exibe uma janela que mostra a página da Web mais popular acessada na simulação.

```

public class FavoriteListMTF<E> extends FavoriteList<E> {
    /** Construtor default */
    public FavoriteListMTF() {}
    /** Move uma entrada para a primeira posição; tempo O(1) */
    protected void moveUp(Position<Entry<E>> pos) {
        fList.addFirst(fList.remove(pos));
    }
    /** Retorna os k elementos mais acessados, para um dado k; tempo: O(kn) */
    public Iterable<E> top(int k) {
        if (k < 0 || k > size())
            throw new IllegalArgumentException("Invalid argument");
        PositionList<E> T = new NodePositionList<E>(); // top-k list
        if (!isEmpty()) {
            // copia as entradas para uma lista temporária C
            PositionList<Entry<E>> C = new NodePositionList<Entry<E>>();
            for (Entry<E> e: fList)
                C.addLast(e);
            // encontra os k primeiros elementos, um de cada vez
            for (int i = 0; i < k; i++) {
                Position<Entry<E>> maxPos = null; // posição do elemento superior
                int maxCount = -1; // contador de acessos do elemento superior
                for (Position<Entry<E>> p: C.positions()) {
                    // examina todas as entradas de C
                    int c = count(p);
                    if (c > maxCount) { // encontrada a entrada com maior contador de acessos
                        maxCount = c;
                        maxPos = p;
                    }
                }
                T.addLast(value(maxPos)); // insere a maior entrada na lista T
                C.remove(maxPos); // remove a maior entrada da lista C
            }
        }
        return T;
    }
}

```

**Trecho de código 6.21** Implementação da classe FavoriteListMTF usando a heurística mover-para-frente. Esta classe estende FavoriteList (Trecho de código 6.19-6.20) e sobrecarrega os métodos moveUp e top.

```

import java.io.*;
import javax.swing.*;
import java.awt.*;
import java.net.*;
import java.util.Random;
/** Programa exemplo para as classes FavoriteList e FavoriteListMTF */
public class FavoriteTester {
    public static void main(String[] args) {
        String[] urlArray = {"http://wiley.com", "http://datastructures.net",
                            "http://algorithmdesign.net", "http://www.brown.edu",
                            "http://uci.edu"};
        FavoriteList<String> L1 = new FavoriteList<String>();
        FavoriteListMTF<String> L2 = new FavoriteListMTF<String>();

```

```
int n = 20; // quantidade de operações de acesso
// Cenário simulado: acessar n vezes uma URL aleatória
Random rand = new Random();
for (int k = 0; k < n; k++) {
    System.out.println("-----");
    int i = rand.nextInt(urlArray.length); // índice randômico
    String url = urlArray[i]; // URL randômica
    System.out.println("Accessing: " + url);
    L1.access(url);
    System.out.println("L1 = " + L1);
    L2.access(url);
    System.out.println("L2 = " + L2);
}
int t = L1.size() / 2;
System.out.println("-----");
System.out.println("Top " + t + " in L1 = " + L1.top(t));
System.out.println("Top " + t + " in L2 = " + L2.top(t));
// Exibe uma janela de navegador mostrando a URL mais popular de L1
try {
    String popular = L1.top(1).iterator().next(); // URL mais popular de L1
    JEditorPane jep = new JEditorPane(popular);
    jep.setEditable(false);
    JFrame frame = new JFrame(popular);
    frame.getContentPane().add(new JScrollPane(jep), BorderLayout.CENTER);
    frame.setSize(640, 480);
    frame.setVisible(true);
} catch (IOException e) { // ignora as exceções de E/S
}
}
}
```

**Trecho de código 6.22** Demonstração do uso das classes FavoriteList e FavoriteListMTF para contar os acessos a páginas da Web. Esta simulação acessa randomicamente várias páginas URL, e então exibe a página mais popular.

---

## 6.6 Exercícios

Para obter auxílio e o código fonte dos exercícios, visite [java.datastructures.net](http://java.datastructures.net).

---

### Reforço

- R-6.1 Desenhe a representação de uma lista arranjo inicialmente vazia *A* depois de executar a seguinte seqüência de operações: add(0,4), add(0,3), add(0,2), add(2,1), add(1,5), add(1,6), add(3,7), add(0,8).
- R-6.2 Apresente uma justificativa para os tempos de execução, apresentados na Tabela 6.2, para os métodos da lista arranjo implementada usando um arranjo (não-extensível).
- R-6.3 Forneça uma classe adaptadora que suporte a interface Stack usando os métodos do TAD lista arranjo.
- R-6.4 Reescreva a justificativa da Proposição 6.2, partindo do princípio de que o custo de aumentar o arranjo de um tamanho *k* para *2k* é 3*k* ciberdólares.

- Quanto se deve cobrar por cada operação de inserção para fazer o esquema de amortização funcionar?
- R-6.5 Forneça descrições em pseudocódigo de algoritmos para os métodos `addBefore(p,e)`, `addFirst(e)` e `addLast(e)` para o TAD lista de nodos, supondo que a lista é implementada usando uma lista duplamente encadeada.
- R-6.6 Desenhe figuras demonstrando cada um dos passos principais dos algoritmos desenvolvidos no exercício anterior.
- R-6.7 Forneça os detalhes de uma implementação baseada em arranjo de um TAD lista de nodos, incluindo como executar os métodos `addBefore` e `addAfter`.
- R-6.8 Forneça trechos de código em Java para os métodos da interface `PositionList`, do Trecho de código 6.5, que não estejam incluídos nos Trechos de código 6.9-6.11.
- R-6.9 Descreva método não-recursivo para inverter uma lista de nodos representada usando-se uma lista duplamente encadeada e que faça uma única passada pela lista (você pode usar os ponteiros internos).
- R-6.10 Dado o conjunto de elementos  $\{a,b,c,d,e,f\}$  armazenado em uma lista, mostre o estado final da lista assumindo que se usa a heurística mover-para-frente e acessam-se os elementos conforme a seguinte seqüência  $(a,b,c,d,e,f,a,c,f,b,d,e)$ .
- R-6.11 Suponha que estejam sendo mantidos contadores de acesso em uma lista  $L$  de  $n$  elementos. Suponha também que foram feitos um total de  $kn$  acessos aos elementos de  $L$ , para algum inteiro  $k \geq 1$ . Qual o número mínimo e máximo de elementos que foram acessados menos de  $k$  vezes?
- R-6.12 Escreva o pseudocódigo que descreve como implementar todas as operações do TAD lista arranjo usando um arranjo de maneira circular. Qual o tempo de execução para cada um desses métodos?
- R-6.13 Usando os métodos da interface `Sequence`, descreva um método recursivo para determinar se uma seqüência  $S$  de  $n$  objetos inteiros contém um dado inteiro  $k$ . O método não pode conter laços. Quanto espaço adicional o método irá precisar além do espaço usado por  $S$ ?
- R-6.14 Descreva brevemente um novo método de seqüência, `makeFirst(p)`, que move o elemento na posição  $p$  de uma seqüência  $S$  para a primeira posição de  $S$ , mantendo a ordem relativa dos demais elementos inalterada. Isto é, `makeFirst(p)` executa um mover-para-frente. O método deve executar em tempo  $O(1)$  considerando que  $S$  seja implementado usando uma lista duplamente encadeada.
- R-6.15 Descreva como usar uma lista arranjo e um campo `int` para implementar um iterador. Inclua trechos de pseudocódigo que descrevam os métodos `hasNext()` e `next()`.
- R-6.16 Descreva como criar um iterador para uma lista de nodos que retorne todos os elementos da lista.
- R-6.17 Suponha que se mantenha uma coleção  $C$  de elementos, tais que, cada vez que se acrescenta um novo elemento na coleção, copia-se o conteúdo de  $C$  em uma nova lista arranjo com exatamente o mesmo tamanho. Qual o tempo de execução para se adicionarem  $n$  elementos a coleção  $C$  inicialmente vazia neste caso?

- R-6.18 Descreva a implementação dos métodos `addLast` e `addBefore` usando apenas os métodos do conjunto `{isEmpty, checkPosition, first, last, prev, next, addAfter, addFirst}`.
- R-6.19 Seja  $L$  uma lista de  $n$  itens ordenados em ordem decrescente de contagem de acesso. Descreva uma série de acessos  $O(n^2)$  que irão inverter  $L$ .
- R-6.20 Seja  $L$  uma lista de  $n$  itens mantidos de acordo com a heurística mover-para-frente. Descreva uma série de acessos  $O(n)$  que irão inverter  $L$ .

### Criatividade

- C-6.1 Forneça o pseudocódigo para os métodos de uma nova classe, `ShrinkingArrayList`, que estenda a classe `ArrayList` apresentada no Trecho de código 6.3, adicionando o método `shrinkToFit()`, que substitui o arranjo base corrente por um cuja capacidade seja exatamente igual ao número de elementos atuais da lista arranjo.
- C-6.2 Descreva as alterações necessárias na implementação de um arranjo extensível apresentado no Trecho de código 6.3, de maneira a comprimir pela metade o tamanho  $N$  do arranjo sempre que o número de elementos do vetor cair abaixo de  $N/4$ .
- C-6.3 Mostre que usando o arranjo extensível que cresce e encolhe, como foi descrito nos exercícios anteriores, a seguinte seqüência de  $2n$  operações consome tempo  $O(n)$ : (i)  $n$  operações `push` sobre uma lista arranjo com capacidade inicial  $N = 1$ ; (ii)  $n$  operações `pop` (remoção do último elemento).
- C-6.4 Mostre como melhorar a implementação do método `add` do Trecho de código 6.3, de maneira que, em caso de overflow, os elementos sejam copiados para seu lugar definitivo no novo arranjo, isto é, nenhum deslocamento é feito neste caso.
- C-6.5 Considere a implementação de um TAD lista arranjo que usa um arranjo extensível mas que, em vez de copiar os elementos para um novo arranjo com o dobro do tamanho (isto é, de  $N$  para  $2N$ ) quando sua capacidade é alcançada, copia os elementos para um arranjo com  $\lceil N/4 \rceil$  células adicionais, aumentando sua capacidade de  $N$  para  $N + \lceil N/4 \rceil$ . Mostre que a execução de uma seqüência de  $n$  operações `push` (isto é, inserções no final) ainda executa em tempo  $O(n)$  neste caso.
- C-6.6 A implementação de `NodePositionList` apresentada nos Trechos de código 6.9-6.11 não faz verificações de erro para testar se uma dada posição  $p$  é realmente membro dessa lista em particular. Por exemplo, se  $p$  é uma posição da lista  $S$ , e chamamos `T.addAfter(p,e)` em uma lista  $T$  diferente, na realidade adicionamos o elemento em  $S$  logo após  $p$ . Descreva como alterar a implementação de `NodePositionList` de uma forma eficiente que impeça esses maus usos.
- C-6.7 Suponha que se deseja estender o tipo abstrato de dados seqüência com os métodos `indexOfElement(e)` e `positionOfElement(e)`, que retornam, respectivamente, o índice e a posição do elemento  $e$  (primeira ocorrência) na seqüência. Mostre como implementar esses métodos os descrevendo em termos de outros métodos da interface `Sequence`.

- C-6.8 Forneça uma adaptação do TAD lista arranjo para o TAD deque que seja diferente da fornecida na Tabela 6.1.
- C-6.9 Descreva a estrutura e o pseudocódigo para uma implementação baseada em arranjo de um TAD lista arranjo que obtém tempo  $O(1)$  para inserções e remoções no índice 0, bem como inserções e remoções no fim da lista arranjo. A implementação deve prever, também, um tempo constante para o método `get`. (Dica: pense em como estender a implementação baseada em arranjo circular do TAD fila apresentado no capítulo anterior).
- C-6.10 Descreva uma forma eficiente de colocar uma lista arranjo representando um conjunto de  $n$  cartas, em uma ordem aleatória. Pode ser usada a função `randomInteger(n)`, que retorna um número aleatório entre 0 e  $n - 1$ , inclusive. O método deve garantir que todas as possíveis ordenações tenham igual probabilidade. Qual é o tempo de execução do método?
- C-6.11 Descreva um método para manter uma lista de favoritos  $L$  tal que todo elemento de  $L$  seja acessado pelo menos uma vez nos últimos  $n$  acessos, onde  $n$  é o tamanho de  $L$ . Este esquema deve acrescentar apenas um tempo  $O(1)$  amortizado em cada operação.
- C-6.12 Suponha que exista uma lista  $L$  de  $n$  elementos mantida pela heurística mover-para-frente. Descreva uma sequência de  $n^2$  acessos que é garantida para consumir tempo  $\Omega(n^3)$  para executar sobre  $L$ .
- C-6.13 Projete um TAD lista de nodos circular que abstrai uma lista encadeada circular da mesma forma que o TAD lista de nodos abstrai uma lista duplamente encadeada.
- C-6.14 Descreva como implementar um iterador para uma lista encadeada circular. Uma vez que `hasNext` sempre irá retornar `true` neste caso, descreva como implementar `hasNewNext()`, que irá retornar `true` se e somente se o próximo nodo da lista ainda não tiver sido retornado pelo iterador.
- C-6.15 Descreva um esquema para criar iteradores de lista que *falham rapidamente*, isto é, tornam-se inválidos tão logo a lista subjacente seja alterada.
- C-6.16 Um arranjo é *esparsão* se a maioria de suas entradas são `null`. Uma lista  $L$  pode ser usada para implementar tal arranjo,  $A$ , de forma eficiente. Em especial, para cada célula não-nula  $A[i]$  pode-se armazenar uma entrada  $(i, e)$  em  $L$ , onde  $e$  é o elemento armazenado em  $A[i]$ . Esta abordagem nos permite representar  $A$  consumindo espaço  $O(m)$ , onde  $m$  é a quantidade de entradas não-nulas de  $A$ . Descreva e analise formas eficientes de executar os métodos do TAD lista arranjo em tal representação. É melhor armazenar as entradas de  $L$  em ordem crescente de índices ou não?
- C-6.17 Existe um algoritmo simples mas ineficiente, chamado *ordenação da bolha*, para ordenar uma sequência  $S$  de  $n$  elementos comparáveis. Este algoritmo percorre a seqüência  $n - 1$  vezes e, em cada varredura, compara o elemento corrente com o próximo, e troca os dois se eles estiverem fora de ordem. Apresente uma descrição em pseudocódigo para a ordenação da bolha que seja tão eficiente quanto possível assumindo que  $S$  é implementado usando-se uma lista duplamente encadeada. Qual o tempo de execução deste algoritmo?
- C-6.18 Responda o Exercício 6.17 assumindo que  $S$  é implementado usando uma lista arranjo.

- C-6.19 Uma operação útil sobre bancos de dados é a *junção natural*\*. Se entendermos um banco de dados como uma lista de pares ordenados de objetos, então a junção natural dos bancos de dados  $A$  e  $B$  é a lista de todas as triplas ordenadas  $(x, y, z)$ , tal que o par  $(x, y)$  se encontra em  $A$  e o par  $(y, z)$ , em  $B$ . Descreva e analise um algoritmo eficiente para computar a junção natural de uma lista  $A$  de  $n$  pares e uma lista  $B$  de  $m$  pares.
- C-6.20 Quando Bob deseja enviar uma mensagem  $M$  para Alice via Internet, ele quebra  $M$  em *pacotes de dados*, numera os pacotes consecutivamente e injeta-os na rede. Quando os pacotes chegam no computador de Alice, eles podem estar fora de ordem, de maneira que Alice precisa remontar a sequência em ordem antes de se certificar de ter recebido toda mensagem. Descreva um esquema eficiente para Alice fazer isso. Qual o tempo de execução deste algoritmo?
- C-6.21 Forneça uma lista  $L$  com  $n$  inteiros positivos, cada um representado usando  $k = \lceil \log n \rceil + 1$  bits, descreva um método de tempo  $O(n)$  para encontrar um inteiro de  $k$  bits que não esteja em  $L$ .
- C-6.22 Argumente porque qualquer solução para o problema anterior deve executar em tempo  $\Omega(n)$ .
- C-6.23 Apresente uma lista  $L$  com  $n$  inteiros arbitrários, projete um método de tempo  $O(n)$  para encontrar um inteiro que não possa ser formado pela soma de dois inteiros que estão em  $L$ .
- C-6.24 Isabel tem uma forma interessante de totalizar a soma dos valores de um arranjo  $A$  de  $n$  inteiros, onde  $n$  é uma potência de dois. Ela criou um arranjo  $B$  com a metade do tamanho de  $A$  e fez  $B[i] = A[2i] + A[2i + 1]$ , para  $i = 0, 1, \dots, (n/2) - 1$ . Se  $B$  tem tamanho 1, então ela retorna  $B[0]$ . Em qualquer outro caso, ela substitui  $A$  por  $B$  e repete o processo. Qual o tempo de execução de seu algoritmo?

---

## Projetos

- P-6.1 Implemente um TAD lista arranjo como um arranjo extensível usado de maneira circular de forma que inserções e deleções no início e no fim do arranjo sejam executadas em tempo constante.
- P-6.2 Implemente o TAD lista arranjo usando uma lista duplamente encadeada. Mostre experimentalmente que esta implementação é pior do que a abordagem baseada em arranjo.
- P-6.3 Escreva um editor de textos simples que armazena e exibe uma string de caracteres usando o TAD lista juntamente com um objeto cursor que destaca a posição de um dos caracteres da string. O editor deve suportar as seguintes operações:
- *left*: move o cursor um caractere para a esquerda (ou não faz nada se estiver no fim do texto).
  - *right*: move o cursor um caractere para a direita (ou não faz nada se estiver no fim do texto).

---

\* N. de T. Em inglês, *natural join*.

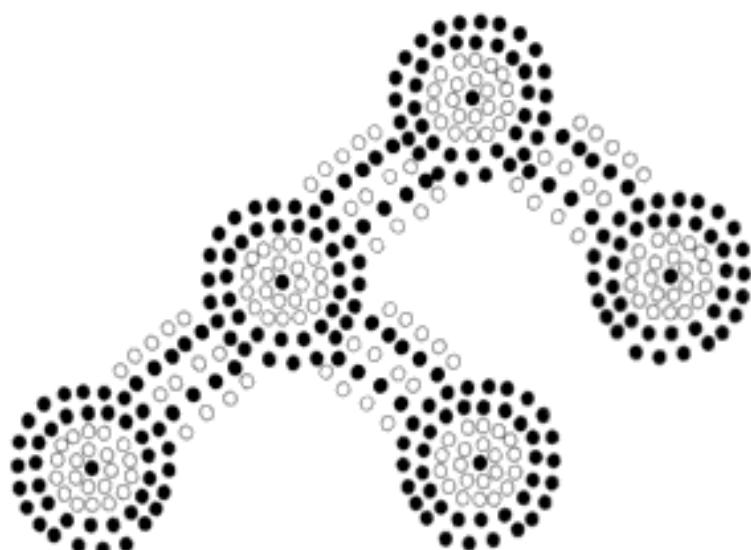
- **cut:** apaga o caractere à direita do cursor (ou não faz nada se estiver no fim do texto).
  - **paste (*c*):** insere o caractere *c* após o cursor.
- P-6.4 Implemente uma *lista de favoritos com fases*. Uma fase consiste em  $N$  acessos na lista para um dado parâmetro  $N$ . Durante uma fase, a lista deve manter seus elementos ordenados em ordem decrescente dos contadores de acesso. Ao final da fase, ela deve limpar os contadores de acesso e iniciar a próxima fase. Experimentalmente, determine quais são os melhores valores de  $N$  para vários tamanhos de lista.
- P-6.5 Escreva uma classe adaptadora completa que implemente o TAD seqüênciia usando um objeto `java.util.ArrayList`.
- P-6.6 Implemente a lista de favoritos usando uma lista arranjo em vez de uma lista. Compare-a, experimentalmente, com uma implementação que use lista.

---

## Observações sobre o capítulo

A concepção de entender estruturas de dados como coleções (e outros princípios de projeto orientado a objetos) podem ser encontrados nos livros de projeto orientado a objetos de Booch [14], Budd[17], Golberg e Robson [40] e Liskov e Guttag[69]. Listas e iteradores são conceitos impregnados no framework de coleções de Java. Nosso TAD lista é derivado da abstração “posição”, introduzida por Aho, Hopcroft e Ullman [5] e o TAD lista de Wood [100]. Implementações de listas usando arranjos e listas encadeadas são discutidas por Knuth [62].



**Conteúdo**

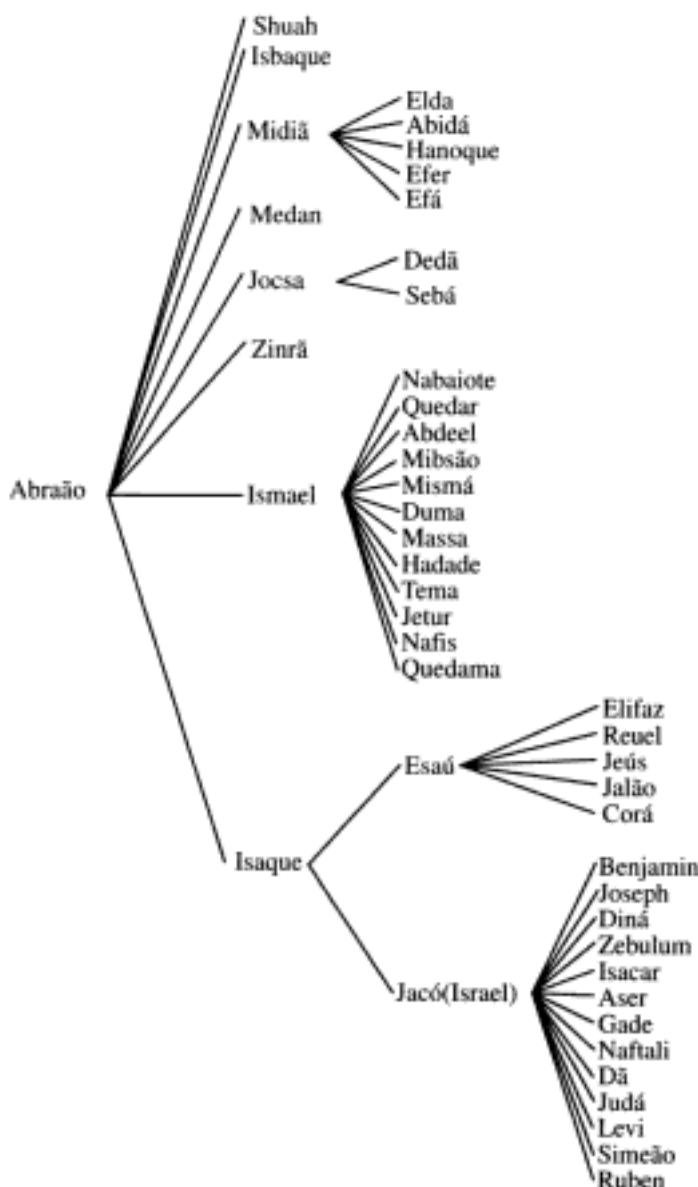
---

<b>7.1 Árvores genéricas . . . . .</b>	<b>246</b>
7.1.1 Definição de árvore e propriedades . . . . .	247
7.1.2 O tipo abstrato de dados árvore . . . . .	249
7.1.3 Implementando uma árvore. . . . .	250
<b>7.2 Algoritmos de caminhamento em árvores. . . . .</b>	<b>251</b>
7.2.1 Altura e profundidade . . . . .	252
7.2.2 Caminhamento prefixado . . . . .	254
7.2.3 Caminhamento pós-fixado . . . . .	256
<b>7.3 Árvores binárias . . . . .</b>	<b>259</b>
7.3.1 O TAD árvore binária . . . . .	260
7.3.2 Uma interface de árvore binária em Java . . . . .	261
7.3.3 Propriedades de árvores binárias . . . . .	261
7.3.4 Estruturas encadeadas para árvores binárias . . . . .	263
7.3.5 Uma estrutura baseada em lista arranjo para árvores binárias . . . . .	270
7.3.6 Caminhamentos sobre árvores binárias . . . . .	272
7.3.7 O padrão do método modelo . . . . .	278
<b>7.4 Exercícios . . . . .</b>	<b>281</b>

## 7.1 Árvores genéricas

Peritos em produtividade dizem que as mudanças se originam em pensamentos “não-lineares”. Neste capítulo, uma das estruturas de dados não-lineares mais importantes da computação será estudada: as **árvores**. Estruturas do tipo árvore são, na verdade, uma ruptura em organização de dados, pois permitem a implementação de uma gama de algoritmos muito mais rápidos do que no uso de estruturas de dados lineares, tais como listas. Árvores também provêm uma forma natural de organizar os dados e, consequentemente, se tornaram estruturas ubíquas em sistemas de arquivos, interfaces gráficas com o usuário, bancos de dados, sites da Web e outros sistemas computacionais.

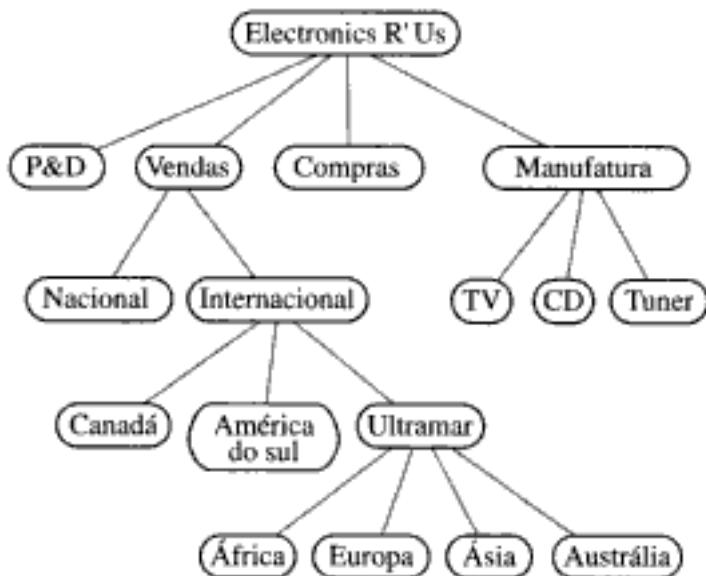
Não é muito claro o que os peritos querem afirmar com pensamento “não-linear”, mas quando se diz que árvores são não-lineares, a referência é feita a um relacionamento organizacional que é mais rico do que simplesmente “antes” e “depois” entre objetos de uma seqüência. Os relacionamentos em uma árvore são **hierárquicos**, com alguns objetos estando “acima” e outros “abaixo” dos outros. Na verdade, a principal terminologia das estruturas de árvore vem das árvores genealógicas, sendo os termos “pai”, “filho”, “ancestral” e “descendente” os mais usados para descrever os relacionamentos. A Figura 7.1 apresenta um exemplo de árvore genealógica.



**Figura 7.1** Uma árvore genealógica que apresenta os descendentes de Abraão como descrito no Gênesis, capítulos 25-36.

### 7.1.1 Definição de árvore e propriedades

Uma **árvore** é um tipo abstrato de dados que armazena elementos de maneira hierárquica. Com exceção do elemento do topo, cada elemento da árvore tem um elemento **pai** e zero ou mais elementos **filhos**. Uma árvore é normalmente desenhada colocando-se os elementos dentro de elipses ou retângulos e conectando pais e filhos com linhas retas. (Ver Figura 7.2.) Normalmente, o elemento topo é chamado de **raiz** da árvore, mas é desenhado como sendo o elemento mais alto, com todos os demais conectados abaixo (exatamente ao contrário de uma árvore real).



**Figura 7.2** Uma árvore com 17 nodos representando a estrutura organizacional uma corporação fictícia. A raiz armazena *Electronics R'Us*. Os filhos da raiz armazenam *P&D*, *Vendas*, *Compras* e *Manufatura*. Os nodos internos armazenam *Vendas*, *Internacional*, *Ultramar*, *Electronics R'Us* e *Manufatura*.

#### Definição formal de árvore

Formalmente, define-se uma **árvore**  $T$  como um conjunto de **nodos** que armazenam elementos em relacionamentos **pai-filho** com as seguintes propriedades:

- Se  $T$  não é vazia, ela tem um nodo especial chamado de **raiz** de  $T$  que não tem pai.
- Cada nodo  $v$  de  $T$  diferente da raiz tem um único nodo **pai**,  $w$ ; todo nodo com pai  $w$  é **filho** de  $w$ .

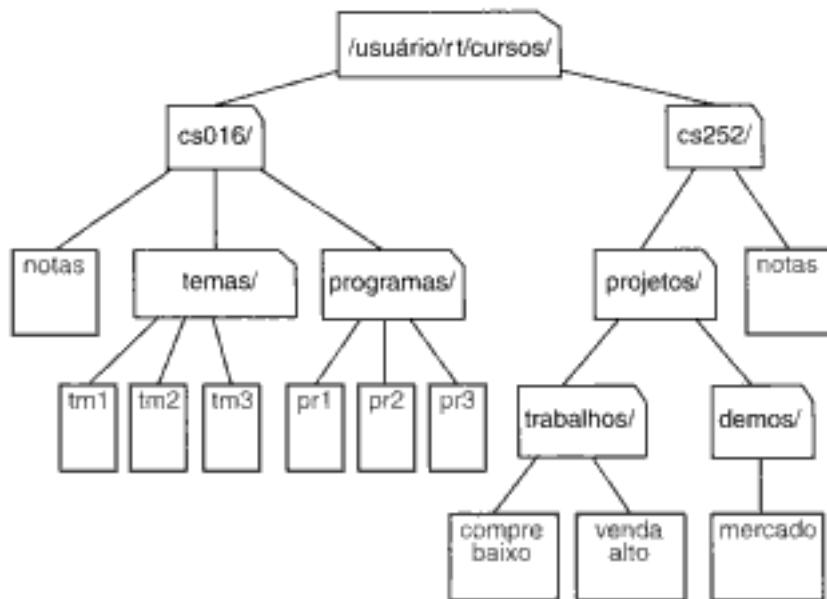
Observa-se que, por esta definição, uma árvore pode ser vazia, o que significa que ela não tem nodos. Esta convenção permite que se defina uma árvore recursivamente, de maneira que uma árvore  $T$  ou está vazia ou consiste em um nodo  $r$ , chamado de raiz de  $T$ , e um conjunto (possivelmente vazio) de árvores cujas raízes são filhas de  $r$ .

#### Outros relacionamentos entre nodos

Dois nodos que são filhos do mesmo pai são **irmãos**. Um nodo  $v$  é **externo** se  $v$  não tem filhos. Um nodo  $v$  é **interno** se tem um ou mais filhos. Nodos externos também são conhecidos como **folhas**.

**Exemplo 7.1** Na maioria dos sistemas operacionais, os arquivos são organizados hierarquicamente em diretórios aninhados (também chamados de pastas) que são apresentados ao usuário sob a forma de uma árvore (ver a Figura 7.3). Mais especificamente, os nodos internos de uma

árvore são associados a diretórios, e os nodos externos são associados a arquivos normais. Nos sistemas operacionais UNIX e Linux, a raiz da árvore é apropriadamente chamada de “diretório raiz”, e é representada pelo símbolo “/”.



**Figura 7.3** Árvore representando parte de um sistema de arquivos.

Um nodo  $u$  é **ancestral** de um nodo  $v$ , se  $u = v$ , ou  $u$  é ancestral do pai de  $v$ . Da mesma forma, diz-se que um nodo  $v$  é **descendente** de um nodo  $u$  se  $u$  é ancestral de  $v$ . Por exemplo, na figura 7.3, cs252/ é ancestral de papers/, e pr3 é descendente de cs016. A **subárvore** de  $T$  enraizada no nodo  $v$  é a árvore que consiste em todos os descendentes de  $v$  em  $T$  (incluindo o próprio  $v$ ). Na Figura 7.3, a subárvore enraizada em cs016/ consiste nos nodos cs016/, grades, homeworks/, programs/, hw1, hw2, hw3, pr1, pr2 e pr3.

### Arestas e caminhos em árvores

Uma aresta de uma árvore  $T$  é um par de nodos  $(u,v)$  tal que  $u$  é pai de  $v$  ou vice-versa. Um **caminho** de  $T$  é uma seqüência de nodos tais que quaisquer dois nodos consecutivos da seqüência formam uma aresta. Por exemplo, a árvore da Figura 7.3 contém o caminho (cs252/, projects/, demos/, market).

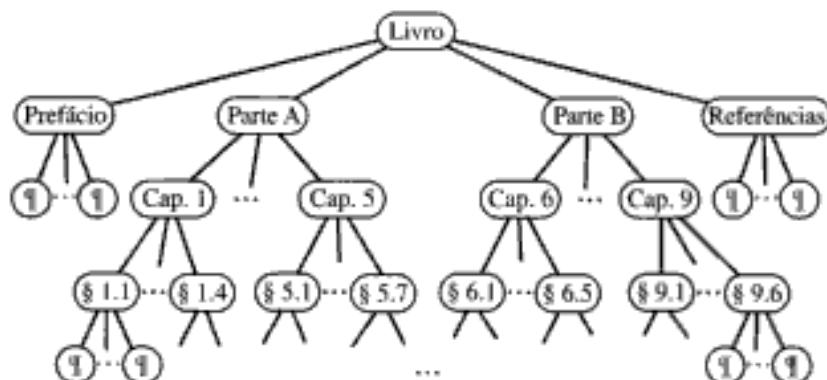
**Exemplo 7.2** O relacionamento de herança entre classes em programas Java forma uma árvore. A raiz, java.lang.Object, é o ancestral de todas as outras classes. Cada classe  $C$  é descendente desta raiz e é a raiz de uma subárvore de classes que estendem  $C$ . Logo, existe um caminho de  $C$  para a raiz, java.lang.Object, nesta árvore de herança.

### Árvores ordenadas

Uma árvore é **ordenada** se existe uma ordem linear definida para os filhos de cada nodo, ou seja, se é possível identificar os filhos de um nodo como sendo o primeiro, segundo, terceiro e assim por diante. Tal ordenação normalmente é desenhada organizando-se os irmãos da esquerda para direita, de acordo com a relação entre os mesmos. Árvores ordenadas normalmente indicam o relacionamento de ordem linear existente entre os irmãos, listando-os na ordem correta.

**Exemplo 7.3** Os componentes de um documento estruturado, tal como um livro, é organizado hierarquicamente como uma árvore cujos nodos internos são partes, capítulos e seções, e os nodos externos são os parágrafos, tabelas, figuras e assim por diante (ver Figura 7.4). A raiz da

árvore corresponde ao livro propriamente dito. Pode-se pensar ainda em expandir a árvore de maneira a mostrar parágrafos como conjuntos de frases, frases como conjuntos de palavras e palavras como conjuntos de letras. Esta árvore é um exemplo de uma árvore ordenada porque existe uma ordem bem-definida entre os filhos de cada nodo.



**Figura 7.4** Árvore ordenada associada a um livro.

### 7.1.2 O tipo abstrato de dados árvore

O TAD árvore armazena elementos em posições como as de uma lista, que são definidas em relação às posições de seus vizinhos. As *posições* de uma árvore são seus *nodos*, e o posicionamento pela vizinhança satisfaz as relações pai-filho, que definem uma árvore válida. Entretanto, os termos “posição” e “nodo” são usados com o mesmo sentido no caso de árvores. Como as posições de uma lista, um objeto posição para uma árvore suporta o método:

`element():` Retorna o objeto nesta posição.

O poder real de um nodo posição em uma árvore, entretanto, vem dos *métodos de acesso* do TAD árvore que retornam e aceitam posições, como os que seguem:

`root():` Retorna a raiz da árvore; um erro ocorre se a árvore está vazia.

`parent(v):` Retorna o nodo pai de *v*; ocorre um erro se *v* for a raiz.

`children(v):` Retorna uma coleção iterável contendo os filhos do nodo *v*.

Se uma árvore *T* é ordenada, então a coleção iterável `children(v)` permite o acesso aos filhos de *v* na ordem. Se *v* é um nodo externo, então `children(v)` está vazio. Além do método de acesso fundamental acima, também se incluem os seguintes *métodos de consulta*:

`isInternal(v):` Testa se um nodo *v* é interno.

`isExternal(v):` Testa se um nodo *v* é externo.

`isRoot(v):` Testa se um nodo *v* é a raiz.

Esses métodos tornam a programação com árvores mais fácil e mais legível, uma vez que pode-se usá-los nas condições de comandos `if` e de laços `while`, em vez de condições pouco intuitivas.

Existe também um conjunto de *métodos genéricos* que uma árvore deveria suportar que não estão necessariamente relacionados com sua estrutura, incluindo os seguintes:

`size():` Retorna o número de nodos na árvore.

`isEmpty():` Testa se a árvore tem ou não tem algum nodo.

`iterator():` Retorna um iterador de todos os elementos armazenados nos nodos da árvore.

`positions( )`: Retorna uma coleção iterável com todos os nodos da árvore.  
`replace(v,e)`: Retorna o elemento armazenado em *v* e o substitui por *e*.

Qualquer método que recebe uma posição por parâmetro deve gerar uma condição de erro se a posição for inválida. Não se definiu nenhum método especializado de atualização para árvores. Em vez disso, prefere-se descrever diferentes métodos de atualização juntamente com aplicações específicas para árvores nos capítulos que seguem. De fato, é possível imaginar diversos tipos de operações de atualização, além das fornecidas neste livro.

### 7.1.3 Implementando uma árvore

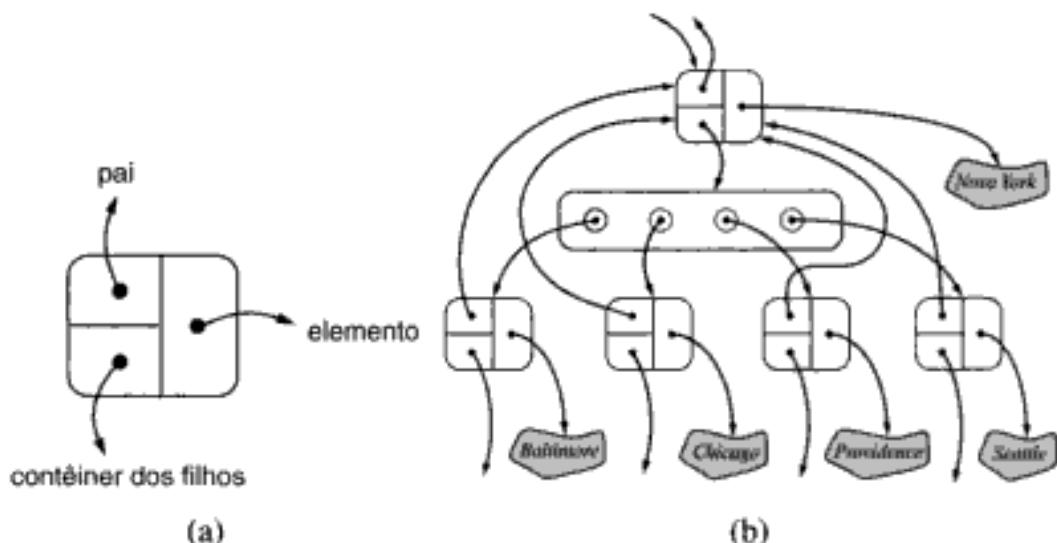
A interface Java apresentada no Trecho de código 7.1 representa o TAD árvore. Condições de erro são tratadas como segue: cada método que pode receber uma posição como argumento pode lançar uma `InvalidPositionException` para indicar que a posição é inválida. O método `parent` lança uma `BoundaryViolationException` se for chamado sobre uma árvore vazia.

```
/*
 * Interface para uma árvore onde os nodos podem ter uma quantidade arbitrária de filhos.
 */
public interface Tree<E> {
    /** Retorna a quantidade de nodos da árvore. */
    public int size();
    /** Retorna se a árvore está vazia. */
    public boolean isEmpty();
    /** Retorna um iterador sobre os elementos armazenados na árvore. */
    public Iterator<E> iterator();
    /** Retorna uma coleção iterável dos nodos. */
    public Iterable<Position<E>> positions();
    /** Substitui o elemento armazenado em um dado nodo. */
    public E replace(Position<E> v, E e)
        throws InvalidPositionException;
    /** Retorna a raiz da árvore. */
    public Position<E> root() throws EmptyTreeException;
    /** Retorna o pai de um dado nodo. */
    public Position<E> parent(Position<E> v)
        throws InvalidPositionException, BoundaryViolationException;
    /** Retorna uma coleção iterável dos filhos de um dado nodo. */
    public Iterable<Position<E>> children(Position<E> v)
        throws InvalidPositionException;
    /** Retorna se um dado nodo é interno. */
    public boolean isInternal(Position<E> v)
        throws InvalidPositionException;
    /** Retorna se um dado nodo é externo. */
    public boolean isExternal(Position<E> v)
        throws InvalidPositionException;
    /** Retorna se um dado nodo é a raiz da árvore. */
    public boolean isRoot(Position<E> v)
        throws InvalidPositionException;
}
```

**Trecho de código 7.1** Interface Java `Tree` representando o TAD árvore. Métodos adicionais de atualização podem ser acrescentados dependendo da aplicação. Entretanto, não se incluem tais métodos na interface.

### Uma estrutura encadeada para árvores genéricas

Uma forma natural de se implementar uma árvore  $T$  é usar uma *estrutura encadeada* em que se representa cada nodo  $v$  de  $T$  usando um objeto posição (ver Figura 7.5a) com os campos que seguem: uma referência para o elemento armazenado em  $v$ , uma conexão com o pai de  $v$  e algum tipo de coleção (por exemplo, uma lista ou artanjo) para armazenar as conexões com os filhos de  $v$ . Se  $v$  é a raiz de  $T$ , então o campo parent de  $v$  é nulo. Também se armazena uma referência para a raiz de  $T$  e o número de nodos de  $T$  em variáveis internas. Esta estrutura é apresentada de forma esquemática na Figura 7.5b.



**Figura 7.5** Estrutura encadeada de uma árvore genérica: (a) o objeto posição associado com um nodo; (b) a porção da estrutura de dados associada com o nodo e seus filhos.

A Tabela 7.1 resume a performance de implementação de uma árvore genérica usando uma estrutura encadeada. A análise é deixada como exercício (C-7.25), mas se observa que, usando uma coleção para armazenar cada um dos nodos de  $v$ , pode-se implementar  $\text{children}(v)$  simplesmente retornando uma referência para esta coleção.

Operação	Tempo
size, isEmpty	$O(1)$
iterator, positions	$O(n)$
replace	$O(1)$
root, parent	$O(1)$
children( $v$ )	$O(c_v)$
isInternal, isExternal, isRoot	$O(1)$

**Tabela 7.1** Tempos de execução dos métodos de uma árvore genérica com  $n$ -nodos, implementada usando-se uma estrutura encadeada. Usa-se  $C_v$  para denotar o número de filhos do nodo  $v$ . O espaço ocupado é  $O(n)$ .

## 7.2 Algoritmos de caminhamento em árvores

Nesta seção, serão apresentados algoritmos para executar computações de caminhamento sobre uma árvore, acessando-a através dos métodos do TAD árvore.

### 7.2.1 Altura e profundidade

Seja  $v$  um nodo de uma árvore  $T$ . A **profundidade** de  $v$  é o número de ancestrais de  $v$  excluindo o próprio  $v$ . Por exemplo, na árvore da Figura 7.2, o nodo que armazena *Internacional* tem profundidade 2. Observa-se que esta definição implica que a profundidade da raiz de  $T$  é 0.

A profundidade de um nodo  $v$  também pode ser definida recursivamente como segue:

- Se  $v$  é a raiz, então a profundidade de  $v$  é 0.
- Em qualquer outro caso, a profundidade de  $v$  é 1 mais a profundidade do pai de  $v$ .

Baseado nesta definição, no Trecho de código 7.2, é apresentado um algoritmo recursivo simples, `depth`, para calcular a profundidade de um nodo  $v$  de  $T$ . Este método chama a si próprio recursivamente sobre o pai de  $v$  e acrescenta 1 ao valor retornado. Uma implementação Java simples deste algoritmo é apresentada no Trecho de código 7.3.

**Algoritmo** `depth( $T, v$ )`:

```
se  $v$  é a raiz de  $T$  então
    retorne 0
senão
    retorne 1 + depth( $T, w$ ), onde  $w$  são os pais de  $v$  em  $T$ 
```

**Trecho de código 7.2** Algoritmo para computar a profundidade de um nodo  $v$  em uma árvore  $T$ .

```
public static <E> int depth(Tree<E> T, Position<E> v){
    if (T.isRoot(v))
        return 0;
    else
        return 1 + depth(T, T.parent(v));
}
```

**Trecho de código 7.3** Método `depth` escrito em Java.

O tempo de execução do algoritmo `depth( $T, v$ )` é  $O(d_v)$ , onde  $d_v$  denota a profundidade do nodo  $v$  na árvore  $T$ , porque o algoritmo executa um passo recursivo de tempo constante para cada ancestral de  $v$ . Logo, o algoritmo `depth( $T, v$ )` executa em  $O(n)$ , no pior caso, onde  $n$  é o número total de nodos de  $T$ , uma vez que um nodo de  $T$  pode ter profundidade  $n - 1$ , no pior caso. Apesar deste tempo de execução ser uma função do tamanho da entrada, é mais exato caracterizar o tempo de execução em termos do parâmetro  $d_v$ , uma vez que este parâmetro pode ser bem menor que  $n$ .

**Altura**

A **altura** de um nodo  $v$  em árvore  $T$  também é definida recursivamente:

- Se  $v$  é um nodo externo, então a altura de  $v$  é 0.
- Em qualquer outro caso, a altura de  $v$  é 1 mais a altura máxima dos filhos de  $v$ .

A **altura** de uma árvore não vazia  $T$  é a altura da raiz de  $T$ . Por exemplo, a árvore da Figura 7.2 tem altura 4. Além disso, a altura também pode ser entendida como segue.

**Proposição 7.4** A altura de uma árvore não-vazia  $T$  é igual à profundidade máxima dos nodos externos de  $T$ .

A justificativa deste fato é deixada como exercício (R-7.6). Apresenta-se o algoritmo, `height1`, mostrado no Trecho de código 7.4 e implementado em Java no Trecho de código 7.5, para cálculo

lo da altura de uma árvore não-vazia  $T$  baseado na proposição anterior e no algoritmo `depth` do Trecho de código 7.2.

**Algoritmo** `height1( $T$ )`:

```

 $h \leftarrow 0$ 
para cada vértice  $v$  em  $T$  faça
  se  $v$  é um nodo externo de  $T$  então
     $h \leftarrow \text{Max}(h, \text{depth}(T, v))$ 
  retorna  $h$ 
```

**Trecho de código 7.4** Algoritmo `height1` para computar a altura de uma árvore não-vazia  $T$ . Observa-se que este algoritmo chama o algoritmo `depth` (Trecho de código 7.2).

```

public static <E> int height1 (Tree<E> T) {
  int h = 0;
  for (Position<E> v : T.positions( )) {
    if (T.isExternal(v))
      h = Math.max(h, depth(T, v));
  }
  return h;
}
```

**Trecho de código 7.5** Método `height1` escrito em Java. Observa-se o uso do método `Max` da classe `java.lang.Math`.

Infelizmente, o algoritmo `height1` não é muito eficiente. Uma vez que `height1` chama o algoritmo `depth(v)` sobre cada nodo externo  $v$  de  $T$ , o tempo de execução de `height1` é dado por  $O(n + \sum_i (1 + d_i))$ , onde  $n$  é o número de nodos de  $T$ ,  $d_i$  é a profundidade do nodo  $v$  e  $E$  é o conjunto de nodos externos de  $T$ . No pior caso, o somatório  $\sum_i (1 + d_i)$  é proporcional a  $n^2$ . (Ver Exercício C-7.6.) Logo, o algoritmo `height1` executa em tempo  $O(n^2)$ .

O algoritmo `height2`, apresentado no Trecho de código 7.6 e implementado em Java no Trecho de código 7.7, computa a altura de uma árvore  $T$  de uma maneira mais eficiente, usando a definição recursiva de altura.

**Algoritmo** `height2( $T, v$ )`:

```

se  $v$  é um nodo externo  $T$  então
  retorna 0
senão
   $h \leftarrow 0$ 
  para cada filho  $w$  de  $v$  em  $T$  faça
     $h \leftarrow \max(h, \text{height2}(T, w))$ 
  retorna 1 +  $h$ 
```

**Trecho de código 7.6** Algoritmo `height2` para computar a altura da subárvore de  $T$  enraizada no nodo  $v$ .

```

public static <E> int height2 (Tree<E> T, Position<E> v) {
  if (T.isExternal(v)) return 0;
  int h = 0;
  for (Position<E> w : T.children(v))
    h = Math.max(h, height2(T, w));
  return 1 + h;
}
```

**Trecho de código 7.7** Método `height2` escrito em Java.

O algoritmo `height2` é mais eficiente que `height1` (do Trecho de código 7.4). O algoritmo é recursivo e, se for chamado inicialmente sobre  $T$ , será eventualmente chamado sobre cada um dos nodos de  $T$ . Logo, pode-se determinar o tempo de execução deste método somando, sobre todos os nodos, o tempo gasto em cada nodo (na parte não-recursiva). Processar cada nodo em `children(v)` consome tempo  $O(c_v)$ , onde  $c_v$  denota o número de filhos do nodo  $v$ . Assim, o laço **while** tem  $c_v$  iterações, e cada iteração do laço consome tempo  $O(1)$  mais o tempo das chamadas recursivas sobre os filhos de  $v$ . Logo, o algoritmo `height2` consome tempo  $O(1 + c_v)$  em cada nodo  $v$ , e seu tempo de execução é  $O(\sum_v (1 + c_v))$ . Para completar a análise, será usada a propriedade que segue.

**Proposição 7.5** Seja  $T$  uma árvore com  $n$  nodos e faça  $c_v$  denotar o número de filhos de um nodo  $v$  de  $T$ . Então o somatório dos vértices de  $T$ ,  $\sum c_v = n - 1$ .

**Justificativa** Cada nodo de  $T$ , com exceção da raiz, é filho de outro nodo, logo contribui com uma unidade na soma anterior. ■

Pela Proposição 7.5, o tempo de execução do algoritmo height2, quando chamado sobre a raiz de  $T$ , é  $O(n)$ , onde  $n$  é o número de nodos de  $T$ .

### 7.2.2 Caminhamento prefixado

O **caminhamento** de uma árvore  $T$  é uma forma sistemática de acessar ou “visitar” todos os nós de  $T$ . Nesta seção, apresenta-se um esquema básico de caminhamento para árvores chamado de caminhamento prefixado. Na seção seguinte, será estudado outro esquema de caminhamento denominado caminhamento pós-fixado.

Em um **caminhamento prefixado** de uma árvore  $T$ , a raiz de  $T$  é visitada primeiro e, então, as subárvore, cujas raízes são seus filhos, são percorridas recursivamente. Se a árvore está ordenada, então as subárvore são percorridas de acordo com a ordem dos filhos. A ação específica associada com a “visita” de um nodo  $v$  depende da aplicação do caminhamento, e pode envolver qualquer coisa, desde incremento de um contador até um cálculo complexo para  $v$ . O pseudocódigo para o caminhamento prefixado de uma subárvore cuja raiz é o nodo  $v$  é apresentado no Trecho de código 7.8. Inicialmente, ativa-se esta rotina chamando `preorder( $T.T.root()$ )`.

**Algoritmo preorder( $T, v$ ):**

executa a ação associada a “visita” do nodo  $v$

para cada filho  $w$  de  $v$  em  $T$  faça

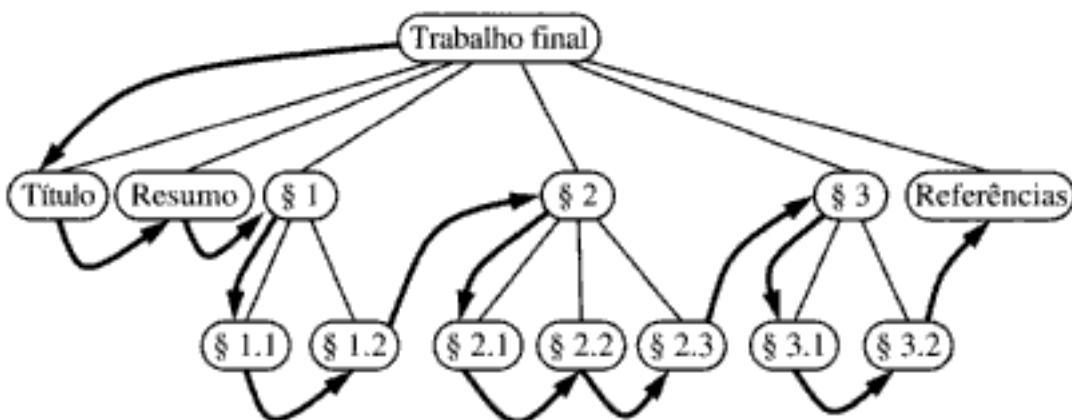
`preorder( $T, w$ )` {recursivamente percorre a subárvore enraizada em  $w$ }

**Trecho de código 7.8** Algoritmo preorder para executar um caminhamento prefixado sobre a subárvore  $T$  enraizada no nodo  $v$ .

O algoritmo de caminhamento prefixado é útil para produzir uma ordenação linear dos nodos de uma árvore, na qual os pais devem aparecer antes dos filhos na ordenação. Tais ordenações têm diferentes aplicações; uma dessas aplicações será explorada no próximo exemplo.

**Exemplo 7.6** O caminhamento prefixado de uma árvore associada a um documento, como no Exemplo 7.3, examina o documento inteiro, seqüencialmente, do início ao fim. Se os nodos externos são removidos antes do caminhamento, então o índice do documento é percorrido (ver a Figura 7.6).

O caminhamento prefixado é uma forma eficiente de se percorrer todos os nodos de uma árvore. Para justificar essa afirmação, considere-se o tempo de execução do caminhamento



**Figura 7.6** Caminhamento prefixado sobre uma árvore ordenada onde os filhos de cada nodo estão ordenados da esquerda para a direita.

prefixado de uma árvore  $T$  com  $n$  nodos, considerando que a “visita” aos nodos consome tempo  $O(1)$ . A análise do algoritmo de caminhamento prefixado é semelhante a do algoritmo `height2` (Trecho de código 7.7), fornecido na Seção 7.2.1. Para cada nodo  $v$ , a parte não-recursiva do algoritmo de caminhamento prefixado requer tempo  $O(1 + c_v)$ , onde  $c_v$  é o número de filhos de  $v$ . Desta forma, pela Proposição 7.5, o tempo total de execução do caminhamento prefixado de  $T$  é  $O(n)$ .

O algoritmo `toStringPreorder( $T, v$ )`, implementado em Java no Trecho de código 7.9, executa uma impressão prefixada da subárvore de um nodo  $v$  de  $T$ , isto é, executa o caminhamento prefixado da subárvore com raiz em  $v$  e imprime o elemento armazenado quando o nodo é visitado. Deve-se lembrar que, para uma árvore ordenada  $T$ , o método  $T.\text{children}(v)$  retorna uma coleção iterável que acessa os filhos de  $v$  em ordem.

```

public static <E> String toStringPreorder(Tree<E> T, Position<E> v) {
    String s = v.element().toString(); // principal ação de "visita"
    for (Position<E> w : T.children(v))
        s += ", " + toStringPreorder(T, w);
    return s;
}
  
```

**Trecho de código 7.9** Método `toStringPreorder( $T, v$ )` que executa uma impressão prefixada dos elementos na subárvore do nodo  $v$  de  $T$ .

Existe uma aplicação interessante do algoritmo de caminhamento prefixado que produz uma representação string de uma árvore inteira. Assume-se novamente que para cada elemento  $e$  armazenado na árvore  $T$ , a chamada  $e.\text{toString}()$  retorna a string associado com  $e$ . A **representação string usando parênteses**  $P(T)$  de uma árvore  $T$  é recursivamente definida como segue. Se  $T$  consiste em um único nodo  $v$ , então

$$P(T) = v.\text{element}().\text{toString}().$$

Se não,

$$P(T) = v.\text{element}().\text{toString}() + "( " + P(T_1) + ", " + \dots + ", " + P(T_k) + " )",$$

onde  $v$  é a raiz de  $T$  e  $T_1, T_2, \dots, T_k$  são as subárvores com raiz nos filhos de  $v$ , os quais são fornecidos em ordem se  $T$  for uma árvore ordenada.

Observa-se que a definição de  $P(T)$  é recursiva. Além disso, está-se usando “+” para denotar concatenação de strings. A representação usando parênteses da árvore da Figura 7.2 é apresentada na Figura 7.7.

```
Electronics R'Us ( P&D
    Vendas ( Nacional
        Internacional ( Canadá América do Sul
            Ultramar ( África Europa Ásia Austrália ) )
        Compras
        Manufatura ( TV CD Tuner ) )
```

**Figura 7.7** Representação usando parênteses da árvore da Figura 7.2. A indentação, as quebras de linha e os espaços foram adicionados por clareza.

Observa-se que, tecnicamente falando, existem alguns cálculos que ocorrem antes e depois das chamadas recursivas nos filhos do nodo no algoritmo anterior. Considera-se, entretanto, esse algoritmo como sendo de caminhamento prefixado, uma vez que a ação principal de impressão do conteúdo do nodo ocorre antes das chamadas recursivas.

O método Java `parentheticRepresentation`, apresentado no Trecho de código 7.10, é uma variação do método `toStringPreorder` (Trecho de código 7.9). Ele implementa a definição fornecida anteriormente para gerar uma representação string usando parênteses de uma árvore  $T$ . Da mesma forma que o método `toStringPreorder`, o método `parentheticRepresentation` faz uso do método `toString` definido para todo objeto Java. Na verdade, podemos entender este método como um tipo de método `toString()` para objetos árvore.

```
public static <E> String parentheticRepresentation(Tree<E> T, Position<E> v) {
    String s = v.element().toString(); // ação principal de visita
    if (T.isInternal(v)) {
        Boolean firstTime = true;
        for (Position<E> w : T.children(v))
            if (firstTime) {
                s += " (" + parentheticRepresentation(T, w); // primeiro filho
                firstTime = false;
            }
            else s += ", " + parentheticRepresentation(T, w); // filhos seguintes
        s += ") "; // fecha parênteses
    }
    return s;
}
```

**Trecho de código 7.10** Algoritmo `parentheticRepresentation`. Observa o uso do operador “+” para concatenar duas strings.

O Exercício R-7.9 explora uma modificação no Trecho de código 7.10 para exibir uma árvore de forma mais próxima à usada na Figura 7.7.

---

### 7.2.3 Caminhamento pós-fixado

Outro tipo importante de caminhamento em árvores é o **caminhamento pós-fixado**. Este algoritmo pode ser entendido como o oposto do caminhamento prefixado, porque primeiro percorre recursivamente as subárvores enraizadas nos filhos da raiz, e depois visita a raiz. É similar ao caminhamento prefixado, entretanto, na medida que usando o mesmo para resolver um determinado problema, especializa-se a ação associada com a “visitação” de um nodo  $v$ . Ainda, da mesma forma que o caminhamento prefixado, se a árvore for ordenada, as chamadas recursivas nos filhos de um nodo  $v$  são feitas de acordo com sua ordem específica. O pseudocódigo para o caminhamento pós-fixado é apresentado no Trecho de código 7.11.

**Algoritmo postorder( $T, v$ ):**

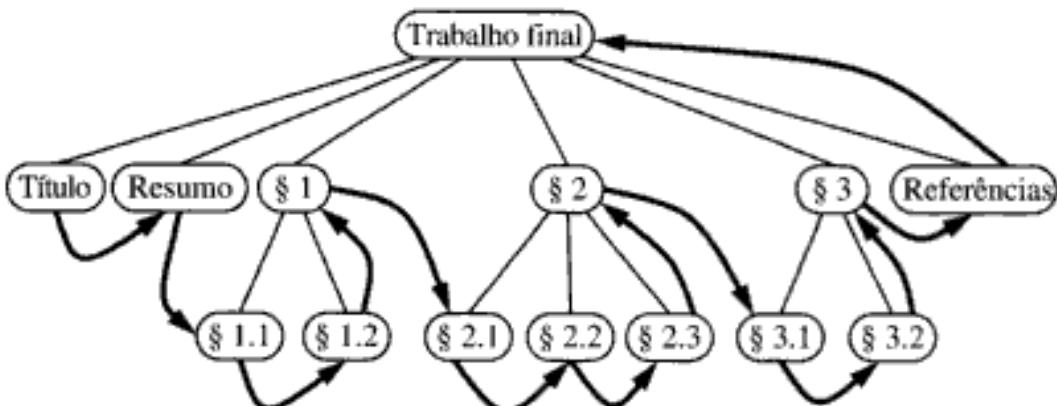
para cada filho  $w$  de  $v$  em  $T$  faça

postorder( $T, w$ ) {recursivamente percorre a subárvore enraizada em  $w$ }

executa a “ação de visita” para o nodo  $v$

**Trecho de código 7.11** Algoritmo postorder que executa um caminhamento pós-fixado sobre a subárvore da árvore  $T$  enraizada no nodo  $v$ .

O nome do caminhamento pós-fixado vem do fato de que o caminhamento visitará o nodo  $v$  depois de ter visitado todos os outros nodos da subárvore com raiz em  $v$  (ver a Figura 7.8).



**Figura 7.8** Caminhamento pós-fixado sobre a árvore ordenada da Figura 7.6.

A análise do tempo de execução de um caminhamento pós-fixado é análoga ao do caminhamento prefixado (ver a Seção 7.2.2). O tempo total gasto nas porções não recursivas do algoritmo é proporcional ao tempo gasto na visitação dos filhos de cada nodo da árvore. Desta forma, um caminhamento pós-fixado de uma árvore  $T$  com  $n$  nodos leva tempo  $O(n)$ , partindo do princípio que a visita a cada nodo leva tempo  $O(1)$ . Ou seja, o caminhamento pós-fixado executa em tempo linear.

Como exemplo de caminhamento pós-fixado, apresenta-se o método Java `toStringPostorder` no Trecho de código 7.12, que executa o caminhamento pós-fixado de uma árvore  $T$ . Este método imprime o elemento armazenado no nodo quando ele é visitado.

```
public static <E> String toStringPostorder(Tree<E> T, Position<E> v){
    String s = "";
    for (Position<E> w : T.children(v))
        s += toStringPostorder(T, w) + " ";
    s += v.element(); // ação principal de visitação
    return s;
}
```

**Trecho de código 7.12** Método `toStringPostorder( $T, v$ )` que executa uma impressão pós-fixada dos elementos da subárvore do nodo  $v$  de  $T$ . O método chama `toString` implicitamente para cada elemento quando os mesmos estão envolvidos em operações de concatenação.

O método de caminhamento pós-fixado é útil para resolver problemas em que se deseja calcular alguma propriedade para cada nodo  $v$  de uma árvore, mas o cálculo desta propriedade para  $v$  implica que se tenha calculado anteriormente a mesma propriedade para seus filhos. Um exemplo de tal aplicação é ilustrado a seguir.

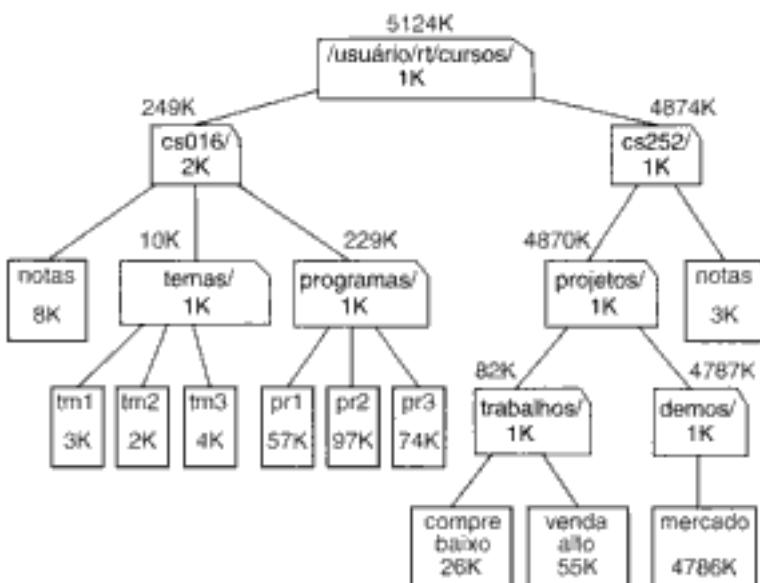
**Exemplo 7.7** Considere-se a árvore  $T$  de um sistema de arquivos, cujos nodos externos representam arquivos e os nodos internos representam diretórios (Exemplo 7.1). Supondo-se que se deseja calcular o espaço em disco usado por um diretório, o que é recursivamente definido pela soma do:

- tamanho do diretório propriamente dito;
- tamanhos dos arquivos armazenados no diretório;
- espaço usado pelos diretórios filhos.

(Ver Figura 7.9.) Este cálculo pode ser feito com um caminhamento pós-fixado sobre a árvore  $T$ . Depois que as subárvores de um nodo interno  $v$  forem percorridas, calcula-se o espaço usado por  $v$ , somando o tamanho do diretório  $v$  propriamente dito e o tamanho dos arquivos armazenados no próprio diretório  $v$  com o espaço usado por cada filho interno de  $v$ , que é calculado pelo caminhamento pós-fixado recursivo dos filhos de  $v$ .

Um método recursivo em Java para calcular o espaço em disco

Motivado pelo Exemplo 7.7, o algoritmo `diskSpace`, apresentado no Trecho de código 7.13, executa um caminhamento pós-fixado de uma árvore de um sistema de arquivos  $T$ , imprimindo o nome e o espaço em disco usado pelo diretório associado com cada nodo interno de  $T$ . Quando chamado a partir da raiz de  $T$ , `diskSpace` executa em tempo  $O(n)$ , onde  $n$  é o número de nodos de  $T$ , desde que os métodos auxiliares `name` e `size` executem em tempo  $O(1)$ .



**Figura 7.9** A árvore da Figura 7.3 representando um sistema de arquivos, mostrando o nome e o tamanho dos arquivos/diretórios associados a cada nodo e o espaço em disco usado para os diretórios associados a cada nodo interno.

```

public static <E> int diskSpace (Tree<E> T, Position<E> v) {
    int s = size(v);           // inicia com o tamanho do próprio nodo
    for (Position<E> w : T.children(v))
        // acrescenta o espaço ocupado pelos filhos de v calculado recursivamente
        s += diskSpace(T, w);
    if (T.isInternal(v)) {
        // imprime o nome e o espaço ocupado em disco
        System.out.print(name(v) + " : " + s);
    }
    return s;
}
  
```

**Trecho de código 7.13** O método `diskSpace` imprime o nome e o espaço em disco ocupado pelo diretório associado com cada nodo interno de uma árvore de um sistema de arquivos. Este método aciona os métodos auxiliares `name` e `size`, que são implementados de maneira a retornar o nome e o tamanho de um arquivo/diretório associado com um nodo.

## Outros tipos de caminhamentos

Apesar de caminhamentos prefixados e pós-fixados serem as formas mais comuns de se percorrer os nodos de uma árvore, é possível imaginar outros caminhamentos. Por exemplo, pode-se percorrer uma árvore de forma a visitar todos os nodos de profundidade  $d$  antes de visitar os nodos de profundidade  $d+1$ . Numerar os nodos de uma árvore  $T$  em seqüência, a medida em que são visitados neste caminhamento, resulta na chamada **numeração dos níveis** dos nodos de  $T$  (ver Seção 7.3.5).

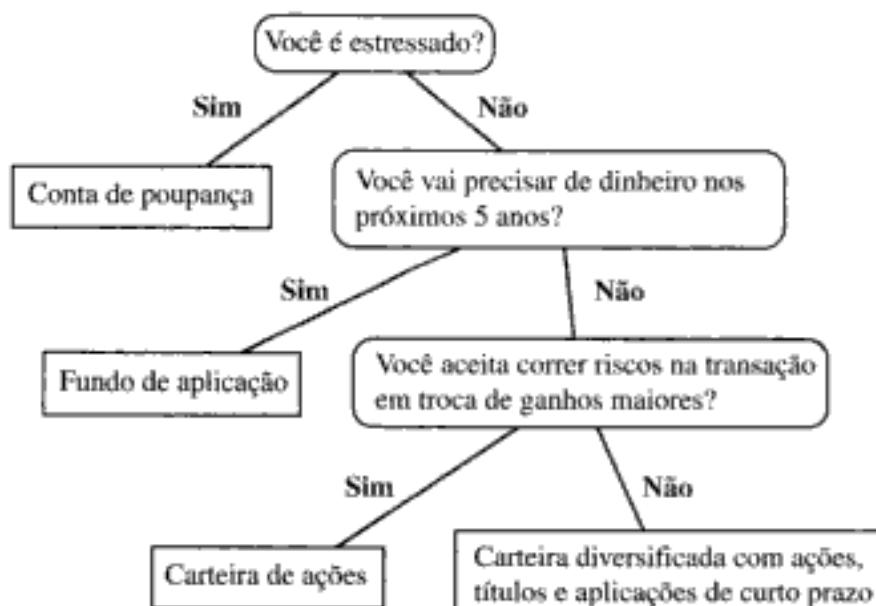
### 7.3 Árvores binárias

Uma **árvore binária** é uma árvore ordenada com as seguintes propriedades:

- Todos os nodos têm no máximo dois filhos.
- Cada nodo filho é rotulado como sendo um **filho da direita** ou um **filho da esquerda**.
- O filho da esquerda precede o filho da direita na ordenação dos filhos de um nodo.

A subárvore enraizada no filho da direita ou no filho da esquerda de um nodo interno  $v$  é chamada de **subárvore direita** ou **subárvore esquerda** de  $v$ , respectivamente. Uma árvore binária é **própria** se cada nodo tem zero ou dois filhos. Algumas pessoas também se referem a estas árvores, como árvores binárias **cheias**. Logo, em uma árvore binária própria todo nodo interno tem exatamente dois filhos. Uma árvore binária que não é própria é **imprópria**.

**Exemplo 7.8** *Uma importante classe de árvores binárias se aplica no contexto em que se pretende representar um conjunto de diferentes resultados a partir das respostas a uma série de questões do tipo sim ou não. Cada nodo interno é associado com uma questão. Começando pela raiz, avança-se pelo filho da direita ou pelo filho da esquerda do nodo corrente, dependendo se a resposta para a questão for "sim" ou "não". Em cada decisão, segue-se uma aresta de um pai para um filho, definindo um caminho sobre a árvore da raiz até um nodo externo. Tais árvores binárias são conhecidas como **árvores de decisão**, porque cada nodo externo  $v$  deste tipo de árvore representa uma decisão dependente das respostas que foram dadas às questões associadas com os ancestrais de  $v$ . A Figura 7.10 apresenta uma árvore de decisão que fornece recomendações para um investidor prospectivo.*

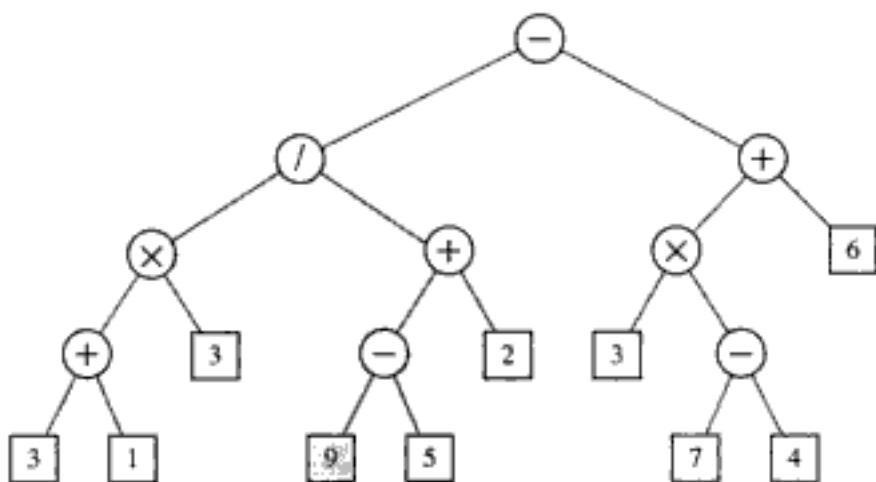


**Figura 7.10** Árvore de decisão que fornece dicas de investimento.

**Exemplo 7.9** Uma expressão aritmética pode ser representada por uma árvore binária cujos nodos externos são associados com variáveis ou constantes e cujos nodos internos são associados com um dos operadores  $+$ ,  $-$ ,  $\times$  e  $/$  (ver Figura 7.11). Cada nodo deste tipo de árvore tem um valor associado.

- Se o nodo é externo, seu valor é o de sua variável ou constante.
- Se o nodo é interno, então seu valor é definido aplicando-se sua operação sobre o valor de seus filhos.

Uma árvore de expressão aritmética é uma árvore binária própria, pois cada operador  $+$ ,  $-$ ,  $\times$  e  $/$  tem exatamente dois operandos. Naturalmente, se forem permitidos operadores unários, como negação ( $-$ ), como em " $-x$ ", então se pode ter uma árvore imprópria.



**Figura 7.11** Uma árvore binária representando uma expressão aritmética. Esta árvore representa a expressão  $((3 + 1) \times 3) / ((9 - 5) + 2)) - ((3 \times (7 - 4)) + 6)$ . O valor associado com o nodo interno rotulado com "/" é 2.

### Definição recursiva de árvore binária

Conseqüentemente, também se pode definir uma árvore binária de maneira recursiva, de maneira que uma árvore binária ou é vazia ou consiste em:

- Um nodo  $r$  chamado raiz de  $T$  e que armazena um elemento.
- Uma árvore binária chamada de subárvore esquerda de  $T$ .
- Uma árvore binária chamada de subárvore direita de  $T$ .

Na seqüência, serão discutidos alguns tópicos específicos de árvores binárias.

#### 7.3.1 O TAD árvore binária

Como tipo abstrato de dados, uma árvore binária é uma especialização da árvore que suporta quatro métodos de acesso adicionais:

`left(v)`: Retorna o filho da esquerda de  $v$ ; ocorre uma condição de erro se  $v$  não tiver filho da esquerda.

`right(v)`: Retorna o filho da direita de  $v$ ; ocorre uma condição de erro se  $v$  não tiver filho da direita.

`hasLeft(v)`: Testa se  $v$  tem um filho da esquerda.

`hasRight(v)`: Testa se  $v$  tem um filho da direita.

Neste caso, como na Seção 7.1.2 para o TAD árvore, não se define métodos especializados para a atualização de árvores binárias. Em vez disso, consideram-se alguns métodos de atualização quando se descrevem implementações e aplicações específicas de árvores binárias.

### 7.3.2 Uma interface de árvore binária em Java

Modela-se uma árvore binária como um tipo abstrato de dados que estende o TAD árvore e acrescenta três métodos especializados para árvores binárias. No Trecho de código 7.14, apresenta-se uma interface Java simples definida usando-se esta abordagem. A propósito, uma vez que árvores binárias são árvores ordenadas, a coleção iterável retornada pelo método `children(v)` (herdado da interface Tree) armazena o filho da esquerda de  $v$  antes do filho da direita.

```
/*
 * Uma interface para árvores binárias onde cada nodo tem zero, um ou dois filhos.
 */
public interface BinaryTree<E> extends Tree<E> {
    /** Retorna o filho da esquerda do nodo. */
    public Position<E> left(Position<E> v)
        throws InvalidPositionException, BoundaryViolationException;
    /** Retorna o filho da direita do nodo. */
    public Position<E> right(Position<E> v)
        throws InvalidPositionException, BoundaryViolationException;
    /** Retorna se o nodo tem filho da esquerda. */
    public boolean hasLeft(Position<E> v) throws InvalidPositionException;
    /** Retorna se o nodo tem filho da direita. */
    public boolean hasRight(Position<E> v) throws InvalidPositionException;
}
```

**Trecho de código 7.14** Interface Java `BinaryTree` para o TAD árvore binária. A interface `BinaryTree` estende a interface `Tree` (Trecho de código 7.1).

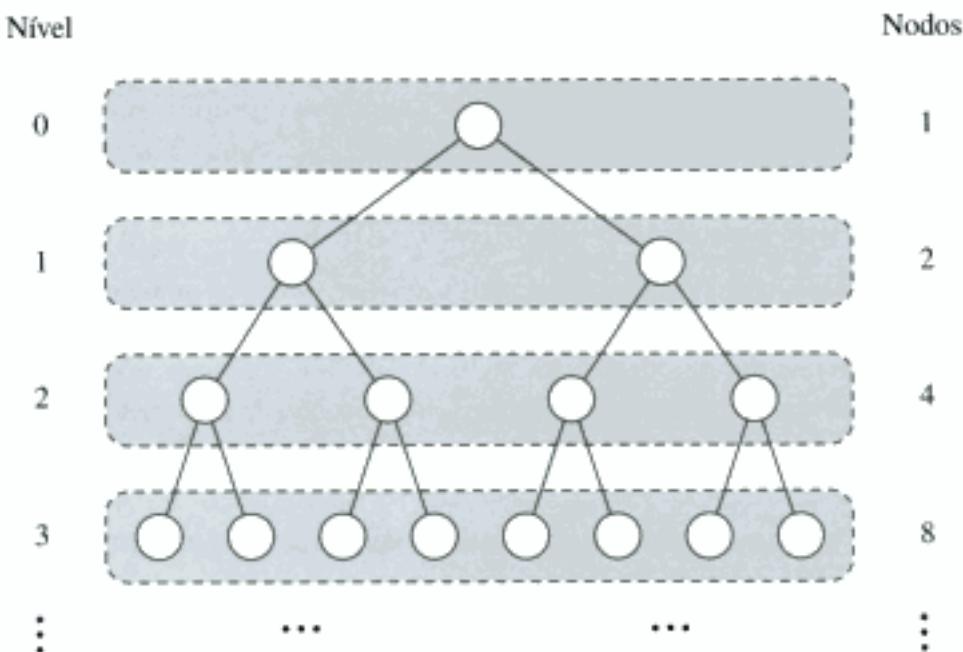
### 7.3.3 Propriedades de árvores binárias

As árvores binárias têm várias propriedades interessantes quanto às relações entre sua altura e número de nodos. Denota-se o conjunto de nodos de mesma profundidade  $d$  de uma árvore  $T$  como sendo o nível  $d$  de  $T$ . Em uma árvore binária, o nível 0 tem no máximo um nodo (a raiz), o nível 1 tem no máximo 2 (os filhos da raiz), o nível 2 tem no máximo 4, e assim por diante (ver a Figura 7.12). Generalizando, pode-se dizer que o nível  $d$  tem no máximo  $2^d$  nodos.

Pode-se observar que o número máximo de nodos nos níveis de uma árvore binária cresce de forma exponencial à medida que se desce na árvore. A partir desta observação, podem-se derivar as seguintes propriedades relacionando a altura de uma árvore binária  $T$  com o número de nodos. Uma explicação detalhada dessas propriedades fica como exercício (R-7.15).

**Proposição 7.10** Seja  $T$  uma árvore binária não-vazia que faça  $n$ ,  $n_E$ ,  $n_I$  e  $h$  denotarem o número de nodos, número de nodos externos, número de nodos internos e altura de  $T$ , respectivamente. Então  $T$  tem as seguintes propriedades:

1.  $h + 1 \leq n \leq 2^{h+1} - 1$
2.  $1 \leq n_E \leq 2^h$
3.  $h \leq n_I \leq 2^h - 1$
4.  $\log(n + 1) - 1 \leq h \leq n - 1$ .



**Figura 6.12** Número máximo de nodos nos níveis de uma árvore binária.

Além disso, se  $T$  é própria, aplicam-se as seguintes propriedades:

1.  $2h + 1 \leq n \leq 2^{h+1} - 1$
2.  $h + 1 \leq n_E \leq 2^h$
3.  $h \leq n_I \leq 2^h - 1$
4.  $\log(n + 1) - 1 \leq h \leq (n - 1)/2$ .

Relacionando nodos internos com nodos externos em uma árvore binária própria

Além das propriedades de árvore binária apresentadas, existem também as seguintes relações entre o número de nodos internos e o número de nodos externos em uma árvore binária própria.

**Proposição 7.11** Em uma árvore binária própria  $T$ , com  $n_E$  nodos externos e  $n_I$  nodos internos, tem-se que  $n_E = n_I + 1$ .

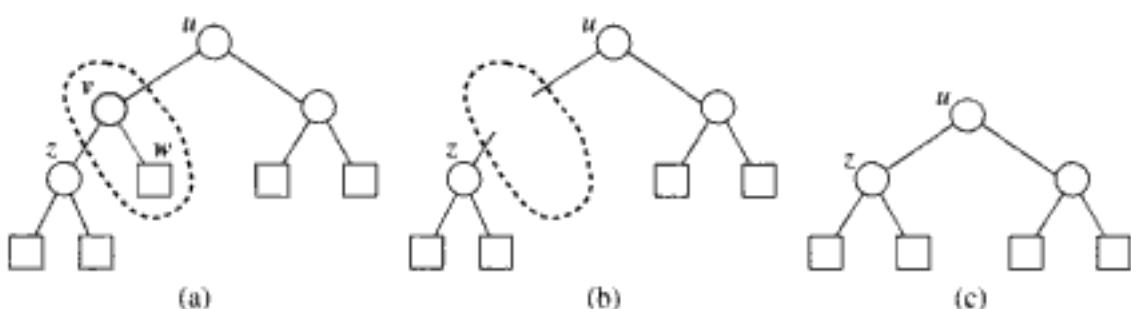
**Justificativa** Justifica-se essa proposição removendo os nodos de  $T$  e dividindo-se os mesmos em dois “montes”: o monte de nodos internos e o monte de nodos externos, até que  $T$  fique vazia. As pilhas estão inicialmente vazias. No final, o monte de nodos externos terá um nodo a mais que o monte de nodos internos. Consideram-se dois casos:

**Caso 1:** Se  $T$  tem apenas um nodo  $v$ , remove-se  $v$ , que é colocado no monte de nodos externos.

Assim, o monte de nodos externos terá um nodo e o monte de nodos internos estará vazio.

**Caso 2:** Por outro lado, ( $T$  tem mais de um nodo) remove-se de  $T$  um nodo externo (arbitrário)  $w$  e seu pai  $v$ , que é um nodo interno. Coloca-se  $w$  no monte de nodos externos e  $v$  no monte de nodos internos. Se  $v$  tem um pai  $u$ , então se reconecta  $u$  com o primeiro irmão  $z$  de  $w$ , como pode ser visto na Figura 7.13. Esta operação remove um nodo interno e um nodo externo e mantém a árvore como sendo uma árvore binária própria.

Repetindo esta operação, mais cedo ou mais tarde restará uma árvore com apenas um nodo. Observa-se que o mesmo número de nodos internos e externos foi removido e colocado em seus montes respectivos pela sequência de operações que resultou nesta árvore final. Agora, remove-se o nodo da árvore final e coloca-se o mesmo no monte de nodos externos. Assim, o monte de nodos externos terá um nodo a mais que o monte de nodos internos.



**Figura 7.13** Operação que remove um nodo externo e seu pai, usada na justificativa da Proposição 7.11.

Observa-se que a relação anterior não se aplica, normalmente, para árvores binárias impróprias e árvores não-binárias, apesar de que existem outras propriedades interessantes que se aplicam, como será investigado no Exercício C-7.7.

### 7.3.4 Estruturas encadeadas para árvores binárias

Da mesma forma que para uma árvore genérica, a forma mais natural de implementar uma árvore binária  $T$  é usar uma **estrutura encadeada**, em que se pode representar cada nodo  $v$  de  $T$  usando um objeto posição (ver Figura 7.14a) com campos provendo referências para os elementos armazenados em  $v$  e os objetos posição associados com os filhos e pais de  $v$ . Se  $v$  é a raiz de  $T$ , então o campo parent de  $v$  é nulo. Se  $v$  não tem o filho da esquerda, então o campo left de  $v$  é nulo. Se  $v$  não tem filho da direita, então o campo right de  $v$  é nulo. Armazena-se, também, a quantidade de nodos de  $T$  em uma variável chamada size. Apresenta-se uma representação da estrutura encadeada de uma árvore binária na Figura 7.14b.

#### Implementação Java de um nodo de árvore binária

Usa-se a interface Java BTPosition (não mostrada) para representar um nodo de árvore binária. Esta interface estende Position, logo herda o método element, e possui métodos adicionais para definir o elemento armazenado no nodo (setElement) e para definir e retornar o filho da esquerda (setLeft e getLeft), da direita (setRight e getRight) e o pai (setParent e getParent) do nodo. A classe BTNode (Trecho de código 7.15) implementa a interface BTPosition por meio de um objeto que tem os campos element, left, right e parent, que, para um nodo  $v$ , referenciam o elemento de  $v$ , o filho da esquerda de  $v$ , o filho da direita de  $v$  e o pai de  $v$ , respectivamente.

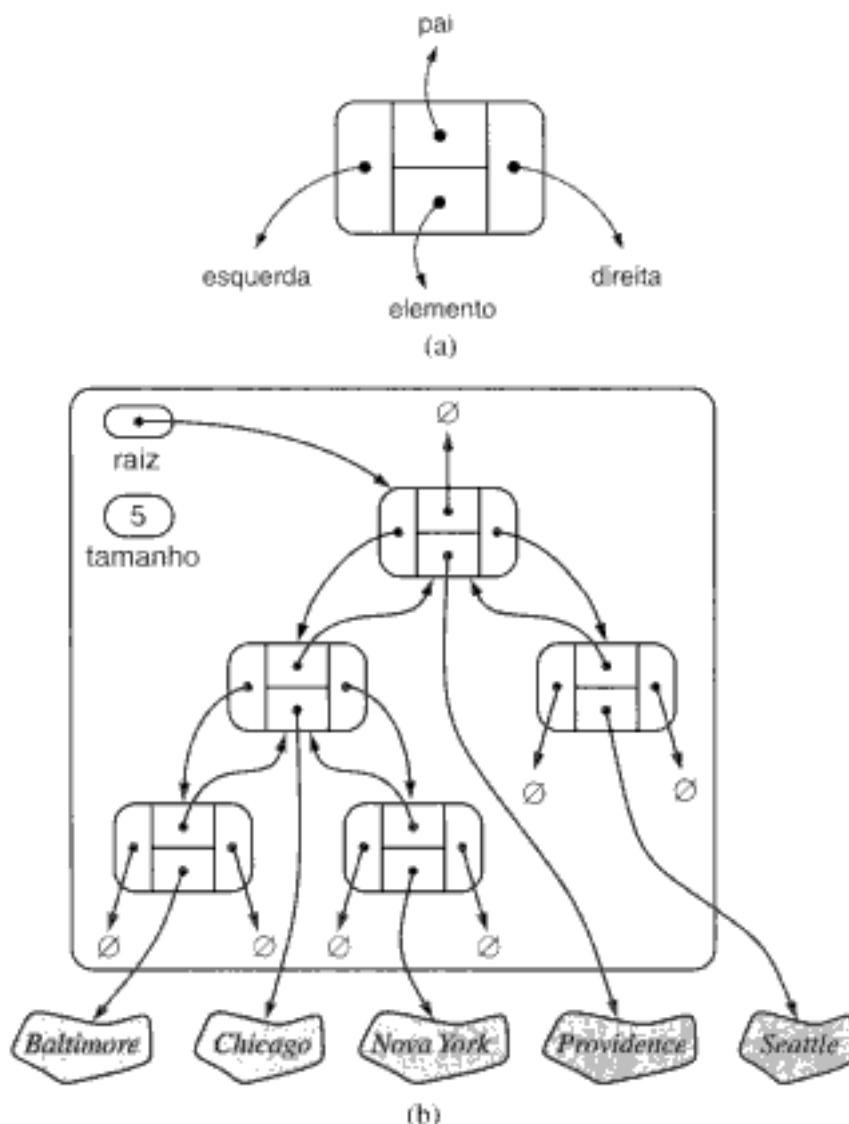
```
/*
 * Classe que implementa um nodo de árvore binária armazenando referencias para um
 * elemento, o nodo pai, o nodo da direita e o nodo da esquerda.
 */
public class BTNode<E> implements BTPosition<E> {
    private E element; // elemento armazenado neste nodo
    private BTPosition<E> left, right, parent; // nodos adjacentes
    /** Construtor principal */
    public BTNode(E element, BTPosition<E> parent,
                  BTPosition<E> left, BTPosition<E> right) {
        setElement(element);
        setParent(parent);
        setLeft(left);
        setRight(right);
    }
}
```

```

    /** Retorna o elemento armazenado nesta posição */
    public E element() { return element; }
    /** Define o elemento armazenado nesta posição */
    public void setElement(E o) { element=o; }
    /** Retorna o filho da esquerda desta posição */
    public BTPosition<E> getLeft() { return left; }
    /** Define o filho da esquerda desta posição */
    public void setLeft(BTPosition<E> v) { left=v; }
    /** Retorna o filho da direita desta posição */
    public BTPosition<E> getRight() { return right; }
    /** Define o filho da direita desta posição */
    public void setRight(BTPosition<E> v) { right=v; }
    /** Retorna o pai desta posição */
    public BTPosition<E> getParent() { return parent; }
    /** Define o pai desta posição */
    public void setParent(BTPosition<E> v) { parent=v; }
}

```

**Trecho de código 7.15** Classe auxiliar BTNode usada na implementação de nodos de árvores binárias.



**Figura 7.14** Um (a) nodo e (b) uma estrutura encadeada para representar uma árvore binária.

## Implementação Java de uma estrutura encadeada para árvore binária

Nos Trechos de código 7.16 – 7.18, são apresentadas partes da classe `LinkedBinaryTree`, que implementa a interface `BinaryTree` (Trecho de código 7.14) usando uma estrutura de dados encadeada. Esta classe armazena o tamanho da árvore e uma referência para o objeto `BTNode` associado com a raiz da árvore em variáveis internas. Além dos métodos da interface `BinaryTree`, `LinkedBinaryTree` tem vários outros métodos, incluindo o método de acesso `sibling(v)` que retorna o irmão de um nodo `v` além dos seguintes métodos de atualização:

- `addRoot(e)`: cria e retorna um nodo novo, `r`, que armazena o elemento `e` e torna `r` a raiz da árvore; um erro ocorre se a árvore não está vazia.
- `insertLeft(v,e)`: cria e retorna um nodo novo, `w`, que armazena o elemento `e`, acrescenta `w` como o filho da esquerda de `v` e retorna `w`; um erro ocorre se `v` já tem um filho da esquerda.
- `insertRight(v,e)`: cria e retorna um nodo novo, `z`, que armazena o elemento `e`, acrescenta `z` como o filho da direita de `v` e retorna `z`; um erro ocorre se `v` já tem um filho da direita.
- `remove(v)`: remove o nodo `v`, substituindo-o por seu filho, se houver algum, e retorna o elemento armazenado em `v`; um erro ocorre se `v` tem dois filhos.
- `attach(v,T1,T2)`: conecta `T1`, `T2`, respectivamente, como as subárvore da esquerda e da direita no nodo externo `v`; uma condição de erro se verifica se `v` não é externo.

A classe `LinkedBinaryTree` tem um construtor sem argumentos que retorna uma árvore binária vazia. A partir desta árvore vazia, pode-se construir qualquer árvore binária criando-se o primeiro nodo com o método `addRoot` e aplicando repetidamente os métodos `insertLeft` e `insertRight`, além do método `attach`. Da mesma forma, pode-se desmantelar qualquer árvore binária `T` usando a operação `remove`, resultando em uma árvore binária vazia.

Quando uma posição `v` é passada como argumento para um dos métodos da classe `LinkedBinaryTree`, sua validade é verificada chamando-se um método auxiliar, `checkPosition(v)`. Uma lista de nodos visitados em um caminhamento prefixado é construída usando-se o método recursivo `preorderPositions`. Condições de erro são indicadas lançando-se as exceções `InvalidPosition`, `BoundaryViolationException`, `EmptyTreeException` e `NonEmptyTreeException`.

```
/*
 * Implementação da interface BinaryTree usando uma estrutura encadeada.
 */
public class LinkedBinaryTree<E> implements BinaryTree<E> {
    protected BTPosition<E> root; // referencia para a raiz
    protected int size; // numero de nodos
    /** Cria uma árvore binária vazia. */
    public LinkedBinaryTree() {
        root = null; // inicia com uma árvore vazia
        size = 0;
    }
    /** Retorna o número de nodos da árvore. */
    public int size() {
        return size;
    }
    /** Retorna se um nodo é interno. */
    public boolean isInternal(Position<E> v) throws InvalidPositionException {
        checkPosition(v); // método auxiliar
    }
}
```

```

    return (hasLeft(v) || hasRight(v));
}
/** Retorna se um nodo é a raiz. */
public boolean isRoot(Position<E> v) throws InvalidPositionException {
    checkPosition(v);
    return (v == root());
}
/** Retorna se um nodo tem o filho da esquerda. */
public boolean hasLeft(Position<E> v) throws InvalidPositionException {
    BTPosition<E> vv = checkPosition(v);
    return (vv.getLeft() != null);
}
/** Retorna a raiz da árvore. */
public Position<E> root() throws EmptyTreeException {
    if (root == null)
        throw new EmptyTreeException("The tree is empty");
    return root;
}
/** Retorna o filho da esquerda de um nodo. */
public Position<E> left(Position<E> v)
    throws InvalidPositionException, BoundaryViolationException {
    BTPosition<E> vv = checkPosition(v);
    Position<E> leftPos = vv.getLeft();
    if (leftPos == null)
        throw new BoundaryViolationException("No left child");
    return leftPos;
}

```

**Trecho de código 7.16** Parte da classe `LinkedBinaryTree` que implementa a interface `BinaryTree` (continua no Trecho de código 7.17).

```

/** Retorna o pai de um nodo. */
public Position<E> parent(Position<E> v)
    throws InvalidPositionException, BoundaryViolationException {
    BTPosition<E> vv = checkPosition(v);
    Position<E> parentPos = vv.getParent();
    if (parentPos == null)
        throw new BoundaryViolationException("No parent");
    return parentPos;
}
/** Retorna uma coleção iterável contendo os filhos de um nodo. */
public Iterable<Position<E>> children(Position<E> v)
    throws InvalidPositionException {
    PositionList<Position<E>> children = new NodePositionList<Position<E>>();
    if (hasLeft(v))
        children.addLast(left(v));
    if (hasRight(v))
        children.addLast(right(v));
    return children;
}
/** Retorna uma coleção iterável contendo os nodos da árvore. */
public Iterable<Position<E>> positions() {
    PositionList<Position<E>> positions = new NodePositionList<Position<E>>();
    if (size != 0)

```

```

    preorderPositions(root(), positions); // atribui as posições usando caminhamento prefixado
    return positions;
}
/** Retorna um iterador sobre os elementos armazenados nos nodos */
public Iterator<E> iterator() {
    Iterable<Position<E>> positions = positions();
    PositionList<E> elements = new NodePositionList<E>();
    for (Position<E> pos: positions)
        elements.addLast(pos.element());
    return elements.iterator(); // Um iterador sobre os elementos
}
/** Substitui o elemento armazenado no nodo. */
public E replace(Position<E> v, E o)
    throws InvalidPositionException {
    BTPosition<E> vv = checkPosition(v);
    E temp = v.element();
    vv.setElement(o);
    return temp;
}

```

**Trecho de código 7.17** Parte da classe LinkedBinaryTree que implementa a interface BinaryTree (continua no Trecho de código 7.18).

```

// Método de acesso adicional
/** Retorna o irmão de um nodo */
public Position<E> sibling(Position<E> v)
    throws InvalidPositionException, BoundaryViolationException {
    BTPosition<E> vv = checkPosition(v);
    BTPosition<E> parentPos = vv.getParent();
    if (parentPos != null) {
        BTPosition<E> sibPos;
        BTPosition<E> leftPos = parentPos.getLeft();
        if (leftPos == vv)
            sibPos = parentPos.getRight();
        else
            sibPos = parentPos.getLeft();
        if (sibPos != null)
            return sibPos;
    }
    throw new BoundaryViolationException("No sibling");
}
// Métodos de acesso adicionais
/** Insere a raiz em uma árvore vazia */
public Position<E> addRoot(E e) throws NonEmptyTreeException {
    if(!isEmpty())
        throw new NonEmptyTreeException("Tree already has a root");
    size = 1;
    root = createNode(e,null,null,null);
    return root;
}
/** Insere o filho da esquerda em um nodo. */
public Position<E> insertLeft(Position<E> v, E e)
    throws InvalidPositionException {
    BTPosition<E> vv = checkPosition(v);

```

```

Position<E> leftPos = vv.getLeft();
if (leftPos != null)
    throw new InvalidPositionException("Node already has a left child");
BTPosition<E> ww = createNode(e, vv, null, null);
vv.setLeft(ww);
size++;
return ww;
}

```

**Trecho de código 7.18** Parte da classe LinkedBinaryTree que implementa a interface BinaryTree (continua no Trecho de código 7.19).

```

/** Remove um nodo com zero ou um filho. */
public E remove(Position<E> v)
throws InvalidPositionException {
    BTPosition<E> vv = checkPosition(v);
    BTPosition<E> leftPos = vv.getLeft();
    BTPosition<E> rightPos = vv.getRight();
    if (leftPos != null && rightPos != null)
        throw new InvalidPositionException("Cannot remove node with two children");
    BTPosition<E> ww; // o único filho de v, se houver
    if (leftPos != null)
        ww = leftPos;
    else if (rightPos != null)
        ww = rightPos;
    else // v é folha
        ww = null;
    if (vv == root) { // v é a raiz
        if (ww != null)
            ww.setParent(null);
        root = ww;
    }
    else { // v não é a raiz
        BTPosition<E> uu = vv.getParent();
        if (vv == uu.getLeft())
            uu.setLeft(ww);
        else
            uu.setRight(ww);
        if (ww != null)
            ww.setParent(uu);
    }
    size--;
    return v.element();
}

```

**Trecho de código 7.19** Parte da classe LinkedBinaryTree que implementa a interface BinaryTree (continua no Trecho de código 7.20).

```

/** Conecta duas árvores para serem subárvore de um nodo externo. */
public void attach(Position<E> v, BinaryTree<E> T1, BinaryTree<E> T2)
throws InvalidPositionException {
    BTPosition<E> vv = checkPosition(v);
    if (isInternal(v))
        throw new InvalidPositionException("Cannot attach from internal node");
    if (!T1.isEmpty( ))

```

```

BTPosition<E> r1 = checkPosition(T1.root());
vv.setLeft(r1);
r1.setParent(vv); // T1 deve ser invalidada
}
if (!T2.isEmpty()) {
    BTPosition<E> r2 = checkPosition(T2.root());
    vv.setRight(r2);
    r2.setParent(vv); // T2 deve ser invalidada
}
}
/** Se v é um nodo de árvore binária, converte para BTPosition, caso contrário lança exceção */
protected BTPosition<E> checkPosition(Position<E> v)
throws InvalidPositionException {
if (v == null || !(v instanceof BTPosition))
    throw new InvalidPositionException("The position is invalid");
return (BTPosition<E>) v;
}
/** Cria um novo nodo de árvore binária */
protected BTPosition<E> createNode(E element, BTPosition<E> parent,
                                BTPosition<E> left, BTPosition<E> right) {
    return new BTNode<E>(element, parent, left, right);
}
/** Cria uma lista que armazena os nodos da subárvore de um nodo ordenados de acordo
 * com o caminhamento prefixado da subárvore. */
protected void preorderPositions(Position<E> v, PositionList<Position<E>> pos)
throws InvalidPositionException {
pos.addLast(v);
if (hasLeft(v))
    preorderPositions(left(v), pos); // recursão sobre o filho da esquerda
if (hasRight(v))
    preorderPositions(right(v), pos); // recursão sobre o filho da direita
}

```

**Trecho de código 7.20** Parte da classe `LinkedBinaryTree` que implementa a interface `BinaryTree` (continuação do Trecho de código 7.19).

### Performance da implementação de `LinkedBinaryTree`

Serão analisados agora os tempos de execução dos métodos da classe `LinkedBinaryTree`, que usa uma representação através de lista encadeada:

- Os métodos `size()` e `isEmpty()` usam uma variável de instância para armazenar o número de nodos de  $T$ , e cada um consome tempo  $O(1)$ .
- Os métodos de acesso `root`, `left`, `right`, `sibling` e `parent` consomem tempo  $O(1)$ .
- O método `replace(v,e)` consome tempo  $O(1)$ .
- Os métodos `iterator()` e `positions()` são implementados usando-se caminhamento prefixado sobre a árvore (usando o método auxiliar `preorderPositions`). Os nodos visitados por este caminhamento são armazenados em uma lista de posições implementada usando-se a classe `NodePositionList` (Seção 6.2.4) e o iterador resultante é criado com o método `iterator()` da classe `NodePositionList`.
- Os métodos `iterator()` e `positions()` consomem tempo  $O(n)$  e os métodos `hasNext()` e `next()` do iterador retornado executam em tempo  $O(1)$ .
- O método `children` usa uma abordagem similar para construir e retornar uma coleção iterável, mas executa em tempo  $O(1)$ , uma vez que existem no máximo dois filhos por nodo em uma árvore binária.

- Os métodos de atualização `insertLeft`, `insertRight`, `attach` e `remove` todos executam em tempo  $O(1)$ , na medida em que envolvem manipulações de tempo constante de um número constante de nodos.

Considerando o espaço requerido por esta estrutura de dados para uma árvore de  $n$  nodos, observa-se que existe um objeto `BTNode` (Trecho de código 7.15) para cada nodo da árvore  $T$ . Logo, o espaço total necessário é  $O(n)$ . A Tabela 7.2 resume a performance da implementação usando estrutura encadeada de uma árvore binária.

Operação	Tempo
<code>size</code> , <code>isEmpty</code>	$O(1)$
<code>iterator</code> , <code>positions</code>	$O(n)$
<code>replace</code>	$O(1)$
<code>root</code> , <code>parent</code> , <code>children</code> , <code>left</code> , <code>right</code> , <code>sibling</code>	$O(1)$
<code>hasLeft</code> , <code>hasRight</code> , <code>isInternal</code> , <code>isExternal</code> , <code>isRoot</code>	$O(1)$
<code>insertLeft</code> , <code>insertRight</code> , <code>attach</code> , <code>remove</code>	$O(1)$

**Tabela 7.2** Tempos de execução para os métodos de uma árvore binária com  $n$  nodos implementada usando uma estrutura encadeada. Os métodos `hasNext()` e `next()` dos iteradores retornados por `iterator()`, `positions().iterator()` e `children(v).iterator()` executam em tempo  $O(1)$ . O espaço consumido é  $O(n)$ .

### 7.3.5 Uma estrutura baseada em lista arranjo para árvores binárias

Uma alternativa para representar uma árvore binária  $T$  é baseada em uma forma de numerar os nodos de  $T$ . Para cada nodo  $v$  de  $T$ , faça  $p(v)$  ser um inteiro definido como segue.

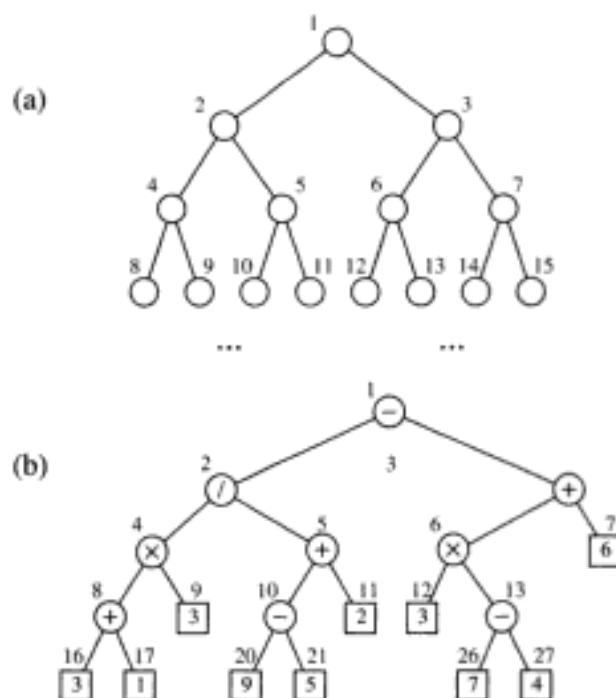
- Se  $v$  é a raiz de  $T$ , então  $p(v) = 1$ .
- Se  $v$  é o filho da esquerda do nodo  $u$ , então  $p(v) = 2p(u)$ .
- Se  $v$  é o filho da direita do nodo  $u$ , então  $p(v) = 2p(u) + 1$ .

A função de numeração  $p$  é conhecida como **numeradora por nível** dos nodos de uma árvore binária  $T$ , na medida em que numera os nodos de cada nível de  $T$  em ordem crescente, da esquerda para a direita, embora possa pular alguns números (ver a Figura 7.15).

A função numeradora por nível  $p$  sugere uma representação para uma árvore binária  $T$  através de um vetor  $S$ , em que cada nodo  $v$  de  $T$  é associado com um elemento de  $S$  em um índice  $p(v)$ . Como mencionado no capítulo anterior, implementa-se o vetor  $S$  usando-se um arranjo extensível (veja a Seção 6.1.4). Tal implementação é simples e eficiente, pois permite executar os métodos `root`, `parent`, `left`, `right`, `hasLeft`, `hasRight`, `isInternal`, `isExternal` e `isRoot` com facilidade, usando apenas operações aritméticas simples sobre os números  $p(v)$  associados com cada nodo  $v$  envolvido na operação. Os detalhes de tal implementação ficam como um exercício simples (R-7.26).

A Figura 7.16 apresenta um exemplo de representação usando lista arranjo de uma árvore binária.

Faz-se  $n$  ser o número de nodos de  $T$  e  $p_M$  ser o valor máximo de  $p(v)$  considerando todos os nodos de  $T$ . O vetor  $S$  tem tamanho  $N = p_M + 1$ , uma vez que o elemento de  $S$  na colocação 0 não está associado com nenhum nodo de  $T$ . Além disso, o vetor  $S$  terá, normalmente, uma certa quantidade de elementos vazios que não se referem a nenhum dos nodos existentes de  $T$ . Na verdade, no pior caso,  $N = 2^n$ , cuja justificativa fica como exercício (R-7.23). Na Seção 8.3, estuda-se uma classe de árvores binárias chamadas de heaps para as quais  $N = n + 1$ . Sendo assim, em vez do pior caso de consumo, existirão aplicações em que a representação por vetor de uma árvore binária será eficiente em termos de espaço. Porém, considerando árvores binárias genéricas, o custo exponencial do pior caso de necessidade de espaço desta representação será exorbitante.

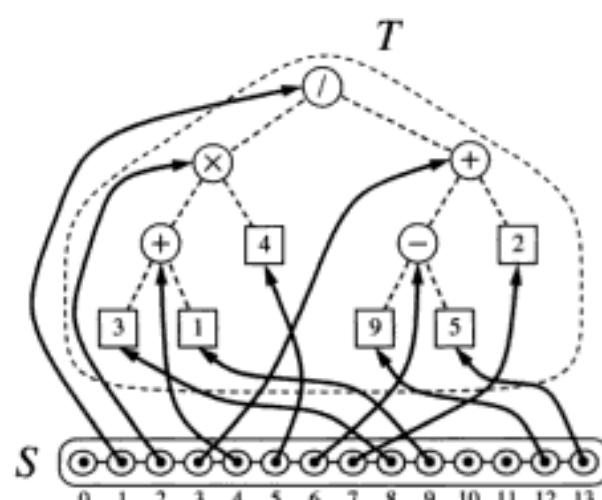


**Figura 7.15** Numeração por níveis de uma árvore binária: (a) esquema geral; (b) um exemplo.

A Tabela 7.3 resume os tempos de execução dos métodos de uma árvore binária implementada usando uma lista arranjo. Não são incluídos nesta tabela os métodos de atualização de uma árvore binária.

Operação	Tempo
size, isEmpty	$O(1)$
iterator, positions	$O(n)$
replace	$O(1)$
root, parent, children, left, right	$O(1)$
hasLeft, hasRight, isInternal, isExternal, isRoot	$O(1)$

**Tabela 7.3** Tempos de execução dos métodos de uma árvore binária  $T$  implementada usando uma lista arranjo  $S$ . Denota-se número de nodos de  $T$  com  $n$ , e  $N$  denota o tamanho de  $S$ . O consumo de espaço é  $O(N)$  e corresponde a  $O(2^n)$ , no pior caso.



**Figura 7.16** Representação de uma árvore binária  $T$  usando uma lista arranjo  $S$ .

### 7.3.6 Caminhamentos sobre árvores binárias

Da mesma forma que com árvores genéricas, os cálculos executados sobre árvores binárias com freqüência envolvem o caminhamento sobre árvores.

#### Construindo a árvore de uma expressão

Considere-se o problema de construir a árvore correspondente a uma expressão a partir de uma expressão aritmética totalmente entre parênteses de tamanho  $n$ . (Recorde-se do Exemplo 7.9 e do Trecho de código 7.24.) No Trecho de código 7.21, apresenta-se o algoritmo `buildExpression`, que cria este tipo de árvore, assumindo que todas as operações aritméticas são binárias e que as variáveis não estão entre parênteses. Logo, toda subexpressão entre parênteses contém um operador no meio. O algoritmo usa uma pilha  $S$  enquanto percorre a expressão de entrada  $E$  procurando por variáveis, operadores e “fecha parênteses”.

- Quando se encontra uma variável ou operador  $x$ , cria-se uma árvore binária de um nodo  $T$  cuja raiz armazena  $x$ , e insere-se  $T$  na pilha.
- Quando se encontra um “fecha parênteses”, “)”, retiram-se as três árvores do topo da pilha  $S$  que representam a subexpressão  $(E_1 \circ E_2)$ . Conectam-se, então, as árvores de  $E_1$  e  $E_2$  na árvore de  $\circ$ , e insere-se o resultado novamente na pilha  $S$ .

Repete-se este procedimento até que a expressão  $E$  tenha sido processada, quando o elemento do topo da pilha seja a árvore da expressão  $E$ . O tempo total de execução é  $O(n)$ .

#### Algoritmo `buildExpression(E)`:

**Entrada:** Uma expressão aritmética totalmente parentetizada  $E = e_0, e_1, \dots, e_{n-1}$ , com cada  $e_i$  sendo uma variável, operador ou símbolo de parênteses

**Saída:** Uma árvore binária  $T$  que representa a expressão aritmética  $E$

$S \leftarrow$  uma pilha nova vazia

**para**  $i \leftarrow 0$  até  $n-1$  **faça**

**se**  $e_i$  é uma variável ou um operador **então**

$T \leftarrow$  uma nova árvore binária vazia

$T.addRoot(e_i)$

$S.push(T)$

**senão se**  $e_i = '('$  **então**

Continua o laço

**senão**  $\{e_i = ')'\}$

$T_2 \leftarrow S.pop()$  {a árvore representando  $E_2$ }

$T \leftarrow S.pop()$  {a árvore representando  $\circ$ }

$T_1 \leftarrow S.pop()$  {a árvore representando  $E_1$ }

$T.attach(T.root(), T_1, T_2)$

$S.push(T)$

**retorna**  $S.pop()$

#### Trecho de código 7.21 Algoritmo `buidExpression`.

#### Caminhamento prefixado de uma árvore binária

Uma vez que qualquer árvore binária pode ser vista como uma árvore genérica, o caminhamento prefixado para árvores genéricas (Trecho de código 7.8) pode ser aplicado a qualquer árvore binária. Pode-se simplificar, entretanto, o algoritmo no caso de caminhamento sobre árvores binárias, como se mostrou no Trecho de código 7.22.

**Algoritmo** binaryPreorder( $T, v$ ):

```

executa a ação prevista para o nodo  $v$ 
se  $v$  tem um filho da esquerda  $u$  em  $T$  então
    binaryPreorder( $T, u$ )                                { recursivamente percorre a subárvore esquerda }
se  $v$  tem um filho da direita  $w$  em  $T$  então
    binaryPreorder( $T, w$ )                                { recursivamente percorre a subárvore direita }
```

**Trecho de código 7.22** Algoritmo binaryPreorder que executa caminhamento prefixado em uma subárvore de uma árvore binária  $T$  com raiz no nodo  $v$ .

Como no caso de árvores genéricas, existem muitas aplicações para o caminhamento prefixado sobre árvores binárias.

### Caminhamento pós-fixado sobre árvores binárias

De forma análoga, o caminhamento pós-fixado para árvores genéricas (Trecho de código 7.11) pode ser especializado para árvores binárias como mostrado no Trecho de código 7.23.

**Algoritmo** binaryPostorder( $T, v$ ):

```

se  $v$  tem um filho da esquerda  $u$  em  $T$  então
    binaryPostorder( $T, u$ )                                { recursivamente percorre a subárvore esquerda }
se  $v$  tem um filho da direita  $w$  em  $T$  então
    binaryPostorder( $T, w$ )                                { recursivamente percorre a subárvore direita }
executa a ação prevista para o nodo  $v$ 
```

**Trecho de código 7.23** Algoritmo binaryPostorder que executa um caminhamento pós-fixado sobre uma subárvore de uma árvore binária  $T$  com raiz no nodo  $v$ .

### Avaliação de uma árvore de expressão

O caminhamento pós-fixado de uma árvore binária pode ser usado para resolver o problema de avaliação de expressões. Neste problema, dada a árvore de uma expressão aritmética, ou seja, uma árvore binária na qual para cada nodo externo existe um valor associado e para cada nodo interno se associa um operador aritmético (ver Exemplo 7.9), deseja-se calcular o valor da expressão aritmética representada pela árvore.

O algoritmo evaluateExpression, indicado no Trecho de código 7.24, avalia a expressão associada com a subárvore com raiz no nodo  $v$  de uma árvore  $T$ , que representa uma expressão aritmética, executando um caminhamento pós-fixado  $T$  que se inicia em  $v$ . Neste caso, a ação “de visita” sobre cada nodo consiste na execução de uma operação aritmética simples. Observa-se que se explora o fato de que uma árvore de expressão aritmética é uma árvore binária própria.

**Algoritmo** evaluateExpression( $T, v$ ):

```

Se  $v$  é um nodo interno de  $T$  então
    seja  $\circ$  o operador armazenado em  $v$ 
     $x \leftarrow$  evaluateExpression( $T, T.\text{left}(v)$ )
     $y \leftarrow$  evaluateExpression( $T, T.\text{right}(v)$ )
    retorna  $x \circ y$ 
senão
    retorna o valor armazenado em  $v$ 
```

**Trecho de código 7.24** Algoritmo evaluateExpression para calcular a expressão representada pela subárvore de uma árvore  $T$  que representa uma expressão aritmética, enraizada no nodo  $v$ .

A aplicação de caminhamento pós-fixado na avaliação de expressões aritméticas resulta em um algoritmo que executa em tempo  $O(n)$  para avaliar uma expressão aritmética representada por uma árvore binária de  $n$  nodos. Na verdade, da mesma forma que o caminhamento pós-fixado genérico, o caminhamento pós-fixado para árvores binárias pode ser aplicado para outros problemas “bottom-up” (como, por exemplo, o problema de cálculo do tamanho apresentado no Exemplo 7.7).

### Caminhamento interfixado para árvores binárias

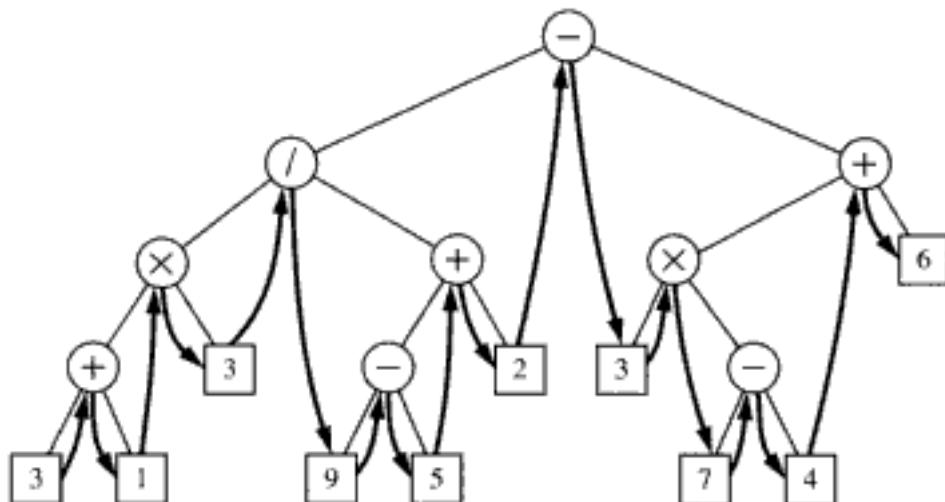
Um método de caminhamento adicional para uma árvore binária é o caminhamento **interfixado**\*. Neste método, visita-se o nodo entre os caminhamentos recursivos das subárvores direita e esquerda. O caminhamento interfixado da subárvore com raiz no nodo  $v$  da árvore binária  $T$  é fornecido no Trecho de código 7.25.

**Algoritmo** inorder( $T, v$ ):

```
se  $v$  tem um filho da esquerda  $u$  em  $T$  então
    inorder( $T, u$ )      {percorre recursivamente a subárvore esquerda}
    execute a ação “de visita” sobre o nodo  $v$ 
    se  $v$  tem um filho da direita  $w$  em  $T$  então
        inorder( $T, w$ )      {percorre recursivamente a subárvore direita}
```

**Trecho de código 7.25** Algoritmo inorder para executar o caminhamento interfixado da subárvore com raiz no nodo  $v$  da árvore binária  $T$ .

O caminhamento interfixado sobre uma árvore binária  $T$  pode ser informalmente considerado como a visita aos nodos de  $T$  “da esquerda para a direita”. De fato, para cada nodo  $v$ , o caminhamento interfixado visita  $v$  após todos os nodos da subárvore esquerda de  $v$  e antes de visitar todos os nodos da subárvore direita de  $v$  (ver Figura 7.17).



**Figura 7.17** Caminhamento interfixado sobre uma árvore binária.

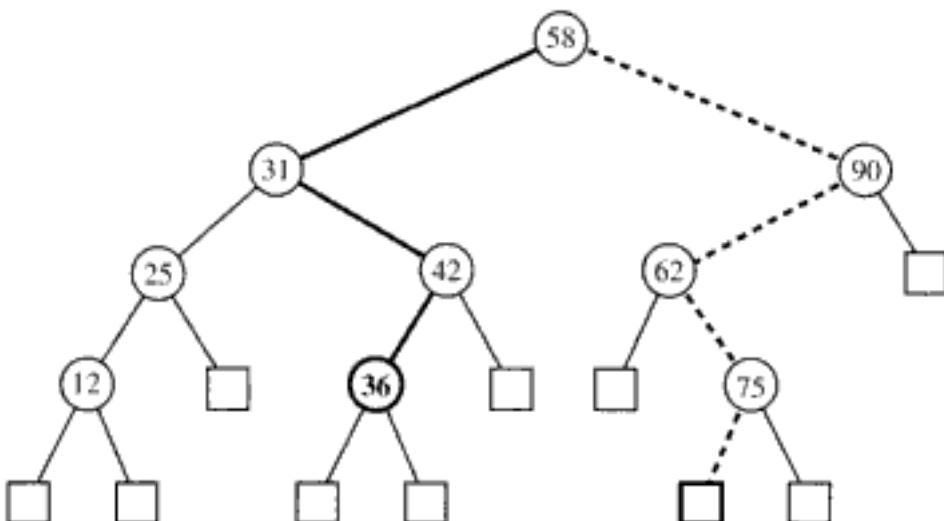
### Árvores binárias de pesquisa

Seja  $S$  um conjunto cujos elementos tem uma relação de ordem. Por exemplo,  $S$  pode ser um conjunto de inteiros. Uma **árvore binária de pesquisa** para  $S$  é uma árvore binária própria  $T$  tal que

\* N. de T. Em inglês, *in-order*. Em português também é conhecido como caminhamento “em-ordem” ou “in-fixado”.

- cada nodo interno  $v$  de  $T$  armazena um elemento de  $S$  denotado  $x(v)$ ;
- para cada nodo interno  $v$  de  $T$ , os elementos armazenados na subárvore esquerda de  $v$  são menores ou iguais a  $x(v)$ , e os elementos armazenados na subárvore direita de  $v$  sejam maiores ou iguais a  $x(v)$ ;
- os nodos externos de  $T$  não armazenam elementos.

Um caminhamento interfixado sobre uma árvore binária de pesquisa  $T$  visita os elementos armazenados em tal árvore, em uma seqüência não-decrescente (ver Figura 7.18).



**Figura 7.18** Uma árvore binária de pesquisa que armazena inteiros. O caminho indicado pela linha mais espessa corresponde ao caminhamento quando se busca (com sucesso) por 36. O caminho pontilhado corresponde ao caminhamento quando se busca (sem sucesso) por 70.

Pode-se usar uma árvore binária de pesquisa  $T$  do conjunto  $S$  para determinar se um certo valor  $y$  se encontra em  $S$  percorrendo para baixo a árvore  $T$  começando pela raiz (ver Figura 7.18). Em cada nodo interno  $v$ , compara-se o valor pesquisado  $y$  com o elemento  $x(v)$  armazenado em  $v$ . Se  $y \leq x(v)$  então a pesquisa continua na subárvore da esquerda de  $v$ . Se  $y = x(v)$ , então a pesquisa terminou com sucesso. Se  $y \geq x(v)$ , então a pesquisa continua na subárvore direita. Finalmente, se foi encontrado um nodo externo, então a pesquisa terminou sem sucesso. Em outras palavras, uma árvore de pesquisa binária pode ser entendida como uma árvore binária de decisão (deve-se relembrar do Exemplo 7.8), onde a questão formulada em cada nodo interno diz respeito ao fato do elemento armazenado naquele nodo ser menor, igual ou maior que o elemento sendo pesquisado. Na verdade, é exatamente esta correspondência com uma árvore de decisão binária que motiva a restrição de que árvores de pesquisa binária devem ser próprias (com nodos externos “de armazenamento”).

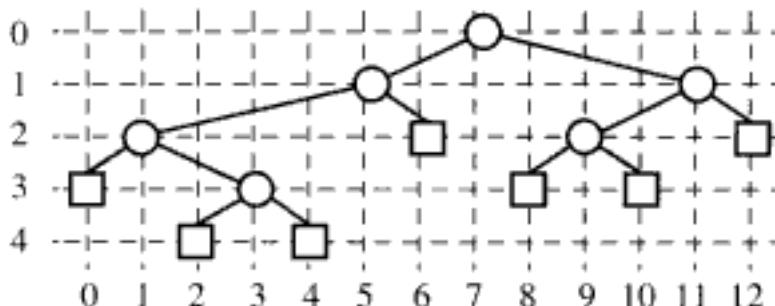
Observa-se que o tempo de execução de pesquisa em uma árvore binária de pesquisa  $T$  é proporcional à altura de  $T$ . Lembrando que a Proposição 7.10 diz que a altura da árvore com  $n$  nodos pode ser tão pequena quanto  $\log(n+1)$  ou tão grande quanto  $(n-1)/2$ . Assim, as árvores de pesquisa binária são mais eficientes quando têm altura pequena. Ilustra-se um exemplo de operação de pesquisa em uma árvore binária de pesquisa na Figura 7.18, e estas árvores serão estudadas com mais detalhes na Seção 10.1.

### Usando o caminhamento interfixado para desenhar uma árvore

O caminhamento interfixado pode também ser aplicado ao problema de computar o desenho de uma árvore binária. Pode-se desenhar uma árvore binária  $T$  com um algoritmo que atribui coordenadas  $x$  e  $y$  a um nodo  $v$  de  $T$ , usando as duas regras seguintes (ver a Figura 7.19):

- $x(v)$  é igual ao número de nodos visitados antes de  $v$  no caminhamento interfixado sobre  $T$ ;
- $y(v)$  é igual à profundidade de  $v$  em  $T$ .

Nesta aplicação, assume-se uma convenção comum em computação gráfica, que diz que as coordenadas  $x$  crescem da esquerda para a direita e as coordenadas  $y$  crescem de cima para baixo. Portanto, a origem localiza-se no canto superior esquerdo da tela do computador.



**Figura 7.19** Algoritmo de desenho interfixado para uma árvore binária.

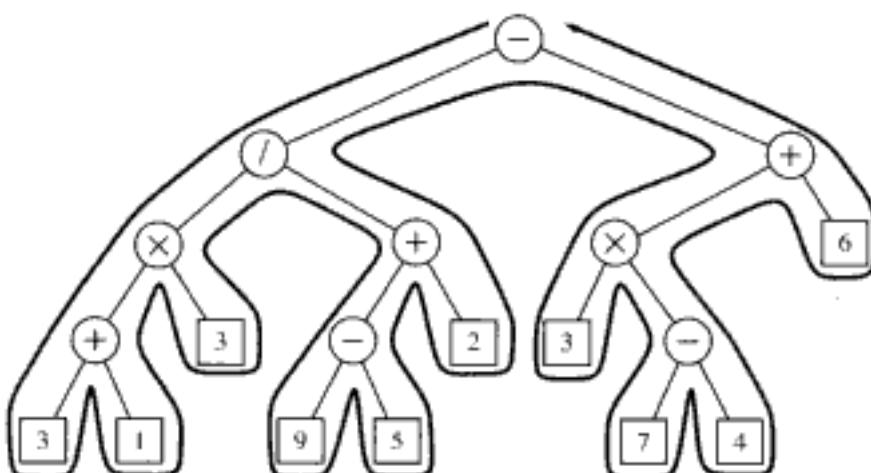
### Caminhamento de Euler sobre uma árvore binária

Os algoritmos de caminhamento em árvores discutidos até agora são formas de iteradores. Cada caminhamento visita os nodos de uma árvore em uma ordem determinada e assegura que cada nodo seja visitado apenas uma vez. Podem-se unificar os algoritmos de caminhamento fornecidos anteriormente em uma única estrutura; porém, será necessário relaxar o requisito que exige que cada nodo seja visitado exatamente uma vez. O método de caminhamento resultante é chamado de **caminhamento de Euler**, e será estudado em seguida. A vantagem deste caminhamento é que ele permite representar uma variedade de tipos de algoritmos com facilidade.

O caminhamento de Euler sobre uma árvore binária  $T$  pode ser informalmente definido como um “passeio” ao redor de  $T$ , no qual se inicia pela raiz em direção ao filho da esquerda, e se considera as arestas de  $T$  como sendo “paredes” que se devem sempre manter à esquerda (ver a Figura 7.20). Cada nodo  $v$  de  $T$  é visitado três vezes pelo caminhamento de Euler:

- “pela esquerda” (antes do caminhamento sobre a subárvore esquerda de  $v$ );
- “por baixo” (entre o caminhamento sobre as duas subárvores de  $v$ );
- “pela direita” (depois do caminhamento sobre a subárvore direita de  $v$ ).

Se  $v$  é externo, então estas três “visitas”, na verdade, ocorrem ao mesmo tempo. Descreve-se o caminhamento de Euler sobre a subárvore enraizada em  $v$  no Trecho de código 7.26.



**Figura 7.20** Caminhamento de Euler sobre uma árvore binária.

**Algoritmo eulerTour( $T, v$ ):**

```

executar a ação prevista para o nodo  $v$  quando encontrado pela esquerda
se  $v$  tem um filho da esquerda  $u$  em  $T$  então
    eulerTour( $T, u$ )          {percorre recursivamente a subárvore esquerda de  $v$ }
executa a ação de visita sobre  $v$  vindo de baixo
se  $v$  tem um filho da direita  $w$  em  $T$  então
    eulerTour( $T, w$ )          {percorre recursivamente a subárvore direita de  $v$ }
executa a ação prevista para o nodo  $v$  pela direita

```

**Trecho de código 7.26** Caminhamento de Euler de uma árvore binária  $T$  com raiz no nodo  $v$ .

Os tempos de execução do caminhamento de Euler são fáceis de analisar, assumindo que a visita a cada nodo consome tempo  $O(1)$ . Uma vez que se consome um tempo constante em cada nodo da árvore durante o percurso, o tempo total de execução é  $O(n)$ .

O caminhamento prefixado sobre uma árvore binária é equivalente ao caminhamento de Euler na medida em que a ação associada a cada nodo ocorre apenas quando o nodo é encontrado pela esquerda. Da mesma forma, os caminhamentos interfixados e pós-fixados de uma árvore binária são equivalentes ao caminhamento de Euler na medida em que as ações associadas aos nodos ocorrem quando os nodos são encontrados por baixo ou pela direita, respectivamente. O caminhamento de Euler estende os caminhamentos prefixado, interfixado e pós-fixado, mas também pode executar outros tipos. Por exemplo, suponha que se deseja calcular o número de descendentes de cada nodo  $v$  em uma árvore binária com  $n$  nodos. Inicia-se o caminhamento de Euler inicializando o contador em 0, e então se incrementa o contador cada vez que se visita um nodo pela esquerda. Para determinar o número de descendentes de um nodo  $v$ , calcula-se a diferença entre o valor do contador quando  $v$  é visitado pela esquerda e quando é visitado pela direita e soma-se 1. Esta regra simples fornece o número de descendentes de  $v$ , porque cada nodo na subárvore com raiz em  $v$  é contada entre a visita a  $v$  pela direita e a visita a  $v$  pela esquerda. Desta forma, tem-se um método que consome tempo  $O(n)$  para calcular o número de descendentes de cada nodo.

Outra aplicação do caminhamento de Euler é a impressão de uma expressão aritmética organizada entre parênteses a partir de sua árvore (Exemplo 7.9). O algoritmo printExpression, apresentado no Trecho de código 7.27, atinge este objetivo executando as seguintes ações durante o caminhamento de Euler:

- ação “pela esquerda”: se o nodo é interno, imprimir “(“;
- ação “por baixo”: imprimir o valor ou operador armazenado no nodo;
- ação “pela direita”: se o nodo é interno, imprimir “)“.

**Algoritmo printExpression( $T, v$ ):**

```

se  $T.isInternal(v)$  então
    imprimir "("
se  $T.hasLeft(v)$  então
    printExpression( $T, T.left(v)$ )
se  $T.isInternal(v)$  então
    imprimir o operador armazenado em  $v$ 
senão
    imprimir o valor armazenado em  $v$ 
se  $T.hasRight(v)$  então
    printExpression( $T, T.right(v)$ )
se  $T.isInternal(v)$  então
    imprimir ")"

```

**Trecho de código 7.27** Um algoritmo para imprimir a expressão aritmética associada com a subárvore com raiz no nodo  $v$  da árvore  $T$  de uma expressão aritmética.

### 7.3.7 O padrão do método modelo

Os métodos de caminhamento em árvores descritos até aqui são, na verdade, exemplos de um padrão de software orientado a objetos, o **padrão do método modelo**. O padrão do método modelo descreve um mecanismo de computação genérico que pode ser especializado para uma aplicação particular pela redefinição de certos passos. Segundo o padrão do método modelo, pode-se projetar um algoritmo que implementa o caminhamento genérico de Euler sobre uma árvore binária. Este algoritmo, chamado de `templateEulerTour`, é apresentado no Trecho de código 7.28.

**Algoritmo** `templateEulerTour(T,v)`:

```

 $r \leftarrow$  um objeto novo do tipo TourResult
visitLeft( $T, v, r$ )
se  $T.\text{hasLeft}(v)$  então
     $r.\text{left} \leftarrow \text{templateEulerTour}(T, T.\text{left}(v))$ 
visitBelow( $T, v, r$ )
se  $T.\text{hasRight}(v)$  então
     $r.\text{right} \leftarrow \text{templateEulerTour}(T, T.\text{right}(v))$ 
visitRight( $T, v, r$ )
retorna  $r.\text{out}$ 
```

**Trecho de código 7.28** Caminhamento de Euler sobre uma subárvore com raiz em um nodo  $v$  de uma árvore binária  $T$ , segundo o padrão do método modelo.

Quando chamado sobre um nodo  $v$ , o método `templateEulerTour` aciona diferentes métodos auxiliares, em diferentes fases do caminhamento. Na verdade ele

- cria uma variável local  $r$  do tipo `TourResult`, que é usada para armazenar os resultados intermediários da computação e tem os campos `left`, `right` e `out`;
- chama o método auxiliar `visitLeft( $T, v, r$ )`, que executa os cálculos associados com o encontro do nodo pela esquerda;
- se  $v$  tem um filho da esquerda, chama a si mesmo recursivamente sobre o filho da esquerda de  $v$  e armazena o valor retornado em  $r.\text{left}$ ;
- chama o método auxiliar `visitBelow( $T, v, r$ )`, que executa os cálculos associados com o encontro do nodo por baixo;
- se  $v$  tem um filho da direita, chama a si mesmo recursivamente sobre o filho da direita de  $v$  e armazena o valor retornado em  $r.\text{right}$ ;
- chama o método auxiliar `visitRight( $T, v, r$ )`, que executa os cálculos associados com o encontro do nodo pela direita;
- retorna  $r.\text{out}$ .

O método `templateEulerTour` pode ser visto como um **modelo** ou “esqueleto” do caminhamento de Euler. (Ver o Trecho de código 7.28.)

### Implementação Java

A classe Java `EulerTour`, apresentada no Trecho de código 7.29, implementa o caminhamento de Euler usando o padrão do método modelo. O caminhamento recursivo é executado pelo método `eulerTour`. Os métodos auxiliares chamados por `eulerTour` são vazios. Isto é, eles têm um corpo vazio ou apenas retornam `null`. A classe `EulerTour` é abstrata e não pode ser instanciada. Ela contém um método abstrato chamado `execute`, que necessita ser especificado em uma subclasse concreta de `EulerTour`. A classe `TourResult` com os campos `left`, `right` e `out` não é apresentada.

```

/**
 * Modelo para os algoritmos de caminhamento sobre uma árvore binária usando
 * caminhamento de Euler. As subclasses desta classe irão refinar alguns dos métodos desta
 * classe para criar um caminhamento específico.
 */
public abstract class EulerTour<E, R> {
    protected BinaryTree<E> tree;
    /** Execução do caminhamento. Este método abstrato deve ser especificado em uma
     * subclasse concreta. */
    public abstract R execute(BinaryTree<E> T);
    /** Inicialização do caminhamento */
    protected void init(BinaryTree<E> T) { tree = T; }
    /** Método modelo */
    protected R eulerTour(Position<E> v) {
        TourResult<R> r = new TourResult<R>();
        visitLeft(v, r);
        if (tree.hasLeft(v))
            r.left = eulerTour(tree.left(v));           // caminhamento recursivo
        visitBelow(v, r);
        if (tree.hasRight(v))
            r.right = eulerTour(tree.right(v));         // caminhamento recursivo
        visitRight(v, r);
        return r.out;
    }
    // Métodos auxiliares que podem ser redefinidos nas subclasses
    /** Método chamado na visita pela esquerda */
    protected void visitLeft(Position<E> v, TourResult<R> r) {}
    /** Método chamado na visita por baixo */
    protected void visitBelow(Position<E> v, TourResult<R> r) {}
    /** Método chamado na visita pela direita */
    protected void visitRight(Position<E> v, TourResult<R> r) {}
}

```

**Trecho de código 7.29** Classe EulerTour, que define um caminhamento genérico sobre uma árvore binária. Esta classe implementa o padrão de método modelo e deve ser especializada de maneira a obter um resultado interessante.

A classe EulerTour propriamente dita não faz nenhuma computação útil. Entretanto, pode-se estender a mesma sobrecarregando os métodos auxiliares para que executem tarefas úteis. Demonstra-se este conceito usando árvores de expressões aritméticas (ver o Exemplo 7.9). Assume-se que uma árvore de expressão aritmética tem objetos do tipo ExpressionTerm em cada nodo. A classe ExpressionTerm tem as subclasses ExpressionValue (para variáveis) e ExpressionOperator (para operadores). Por sua vez, a classe ExpressionOperator tem subclasses para os operadores aritméticos, tais como AdditionOperator e MultiplicationOperator. O método value de ExpressionTerm é sobrecarregado por suas subclasses. Para uma variável, ele retorna o valor da variável. Para um operador, ele retorna o resultado da aplicação do operador sobre seus operandos. Os operandos de um operador são definidos pelo método setOperands de ExpressionOperator. No Trecho de código 7.30, apresentam-se as classes ExpressionTerm, ExpressionVariable, ExpressionOperator e AdditionOperator.

```

/** Classe que representa um termo (operador ou variável) de uma expressão aritmética. */
public class ExpressionTerm {
    public Integer getValue() { return 0; }
    public String toString() { return new String(" "); }
}

```

```

/** Classe que representa uma variável de uma expressão aritmética. */
public class ExpressionVariable extends ExpressionTerm {
    protected Integer var;
    public ExpressionVariable(Integer x) { var = x; }
    public void setVariable(Integer x) { var = x; }
    public Integer getValue() { return var; }
    public String toString() { return var.toString(); }
}
/** Classe que representa um operador de uma expressão aritmética. */
public class ExpressionOperator extends ExpressionTerm {
    protected Integer firstOperand, secondOperand;
    public void setOperands(Integer x, Integer y) {
        firstOperand = x;
        secondOperand = y;
    }
}
/** Classe que representa o operador de soma de uma expressão aritmética. */
public class AdditionOperator extends ExpressionOperator {
    public Integer getValue() {
        return (firstOperand + secondOperand); //unboxing e então autoboxing
    }
    public String toString() { return new String("+"); }
}

```

**Trecho de código 7.30** Classes para uma variável, operador genérico e operador de adição de uma expressão aritmética.

Nos Trechos de código 7.31 e 7.32, são apresentadas as classes EvaluateExpressionTour e PrintExpressionTour, que especializam EulerTour avaliando e imprimindo a expressão aritmética armazenada em uma árvore binária, respectivamente. A classe EvaluateExpressionTour sobrecarrega o método visitRight( $T, v, r$ ) com as seguintes computações:

- se  $v$  é um nodo externo, atribua para  $r.out$  o mesmo valor de variável armazenado em  $v$ ;
- senão ( $v$  é um nodo interno) combine  $r.left$  e  $r.right$  com o operador armazenado em  $v$  faça  $r.out$  ser igual ao resultado da operação.

A classe printExpressionTour sobrecarrega os métodos visitLeft, visitBelow e visitRight seguindo a abordagem da versão em pseudocódigo apresentada no Trecho de código 7.27.

```

/** Calcula o valor de uma árvore de expressão aritmética. */
public class EvaluateExpressionTour extends EulerTour<ExpressionTerm, Integer> {
    public Integer execute(BinaryTree<ExpressionTerm> T) {
        init(T); // chama o método da superclasse
        return eulerTour(tree.root());
    }
    protected void visitRight(Position<ExpressionTerm> v, TourResult<Integer> r) {
        ExpressionTerm term = v.element();
        if (tree.isInternal(v)) {
            ExpressionOperator op = (ExpressionOperator) term;
            op.setOperands(r.left, r.right);
        }
        r.out = term.getValue();
    }
}

```

**Trecho de código 7.31** Classe EvaluateExpressionTour que especializa EulerTour para avaliar a expressão associada com uma árvore de expressão aritmética.

```

/** Imprime a expressão armazenada em uma árvore de expressão aritmética. */
public class PrintExpressionTour extends EulerTour<ExpressionTerm, String> {
    public String execute(BinaryTree<ExpressionTerm> T) {
        init(T);
        System.out.print("Expression: ");
        eulerTour(T.root());
        System.out.println();
        return null; // não retorna nada
    }
    protected void visitLeft(Position<ExpressionTerm> v, TourResult<String> r) {
        if (tree.isInternal(v)) System.out.print(" ( ");
    }
    protected void visitBelow(Position<ExpressionTerm> v, TourResult<String> r) {
        System.out.print(v.element());
    }
    protected void visitRight(Position<ExpressionTerm> v, TourResult<String> r) {
        if (tree.isInternal(v)) System.out.print(" ) ");
    }
}

```

**Trecho de código 7.32** Classe PrintExpressionTour que especializa EulerTour para imprimir a expressão associada com uma árvore de expressão aritmética.

## 7.4 Exercícios

Para obter ajuda e o código fonte dos exercícios, visite [java.datastructures.net](http://java.datastructures.net).

### Reforço

- R-7.1 As questões a seguir são relativas à árvore da Figura 7.3.
  - a. Qual nodo é a raiz?
  - b. Quais são os nodos internos?
  - c. Quantos descendentes tem o nodo cs016/?
  - d. Quantos ancestrais tem o nodo cs016/?
  - e. Quais são os irmãos do nodo temas/?
  - f. Que nodos pertencem à subárvore com raiz no nodo projetos/?
  - g. Qual é a profundidade do nodo trabalhos/?
  - h. Qual a altura da árvore?
- R-7.2 Encontre o valor da expressão aritmética associada com cada subárvore da árvore binária da Figura 7.11.
- R-7.3 Seja  $T$  uma árvore binária com  $n$  nodos que pode ser imprópria. Descreva como representar  $T$  como uma árvore binária **própria**  $T'$  com  $O(n)$  nodos.
- R-7.4 Quais são os números mínimo e máximo de nodos internos e externos em uma árvore binária imprópria com  $n$  nodos?
- R-7.5 Mostre uma árvore que resulta no pior caso para o tempo de execução do algoritmo depth.
- R-7.6 Apresente uma justificativa para a Proposição 7.4.
- R-7.7 Qual é o tempo de execução do algoritmo height2( $T, v$ ) (Trecho de código 7.6) quando ativado sobre um nodo  $v$  que não a raiz de  $T$ ?
- R-7.8 Seja  $T$  a árvore da Figura 7.3 e referindo-se aos Trechos de código 7.9 e 7.10,
  - a. forneça a saída do algoritmo `toStringPostorder( $T, T.root()$ )`;
  - b. forneça a saída do algoritmo `parentheticRepresentation( $T, T.root()$ )`.

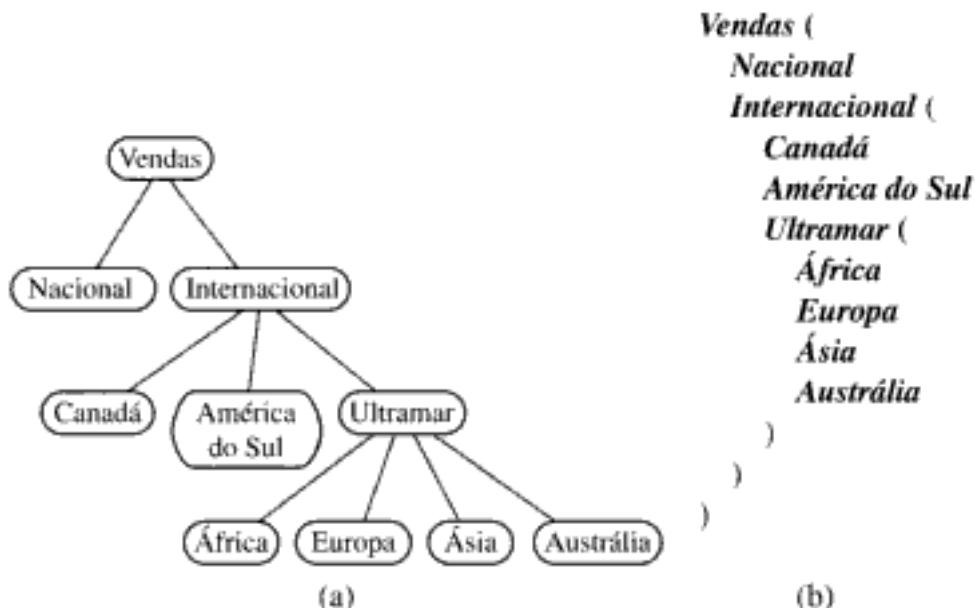
- R-7.9 Descreva as modificações no método `parentheticRepresentation`, apresentado no Trecho de código 7.10, de maneira que o mesmo use o método `length()` dos objetos `String` para exibir a representação entre parênteses de uma árvore, com a adição de quebras de linha e de espaços para exibir a árvore em uma janela de texto com 80 caracteres de largura.
- R-7.10 Desenhe uma árvore que represente uma expressão com quatro nodos externos armazenando os números 1, 5, 6 e 7 (com cada número armazenado em um nodo externo, ainda que não necessariamente nesta ordem) e três nodos internos – cada um armazenando uma operação do conjunto  $\{+, -, \times, /\}$  de operadores aritméticos, de maneira que o valor da raiz seja 21. Os operadores podem retornar e agir sobre frações e um operador pode ser usado mais de uma vez.
- R-7.11 Seja  $T$  uma árvore ordenada com mais de um nodo. É possível que o caminhamento prefixado de  $T$  visite os nodos na mesma ordem que o caminhamento pós-fixado de  $T$ ? Em caso afirmativo, forneça um exemplo; caso contrário, argumente por que isso não pode ocorrer. Da mesma forma, é possível que o caminhamento prefixado de  $T$  visite os nodos na ordem inversa do caminhamento pós-fixado? Em caso afirmativo, forneça um exemplo; caso contrário, argumente por que isso não pode ocorrer.
- R-7.12 Responda a questão anterior para o caso de  $T$  ser uma árvore binária própria com mais de um nodo.
- R-7.13 Qual é o tempo de execução de `parentheticRepresentation(T,T.root())` (Trecho de código 7.10) para uma árvore com  $n$  nodos?
- R-7.14 Desenhe uma (única) árvore binária tal que:
- cada nodo interno de  $T$  armazene um único caractere;
  - o caminhamento *prefixado* de  $T$  produza EXAMFUN;
  - o caminhamento *interfixado* de  $T$  produza MAFXUEN.
- R-7.15 Responda as seguintes questões de maneira a justificar a Proposição 7.10.
- a. Qual é o número mínimo de nodos externos de uma árvore binária própria com altura  $h$ ? Justifique sua resposta.
  - b. Qual é o número máximo de nodos externos de uma árvore binária própria com altura  $h$ ? Justifique sua resposta.
  - c. Seja  $T$  uma árvore binária com altura  $h$  e  $n$  nodos, mostre que
$$\log(n + 1) - 1 \leq h \leq (n - 1)/2.$$
  - d. Para quais valores de  $n$  e  $h$  acima, os limites superior e inferior de  $h$  podem ser atendidos com equilíbrio?
- R-7.16 Descreva uma generalização do caminhamento de Euler para árvores de maneira que cada nodo interno tenha três filhos. Descreva como você pode usar este caminhamento para computar a largura de cada nodo desta árvore.
- R-7.17 Compute a saída do algoritmo `toStringPostorder(T,T.root())`, a partir do Trecho de código 7.12 sobre a árvore  $T$  da Figura 7.3.
- R-7.18 Desenhe a execução do algoritmo `diskSpace(T,T.root())` (Trecho de código 7.13) sobre a árvore  $T$  da Figura 7.9.
- R-7.19 Seja  $T$  a árvore binária da Figura 7.11
- a. Apresente a saída de `toStringPostorder(T,T.root())` (Trecho de código 7.9).
  - b. Apresente a saída de `parenthicRepresentation(T,T.root())` (Trecho de código 7.10).

- R-7.20 Seja  $T$  a árvore binária da Figura 7.11.
- Apresente a saída do algoritmo `toStringPostorder( $T, T.root()$ )` (Trecho de código 7.12).
  - Apresente a saída do algoritmo `printExpression( $T, T.root()$ )` (Trecho de código 7.27).
- R-7.21 Descreva um algoritmo em pseudocódigo para computar a quantidade de descendentes de cada nodo de uma árvore binária. O algoritmo deve ser baseado no caminhamento de Euler.
- R-7.22 Seja  $T$  uma árvore binária (possivelmente imprópria) com  $n$  nodos e que denote por  $D$  a soma das profundidades de todos os nodos externos de  $T$ . Mostre que se  $T$  tem o número mínimo de nodos externos possíveis, então  $D$  é  $O(n)$ ; e que se  $T$  tem número máximo de nodos externos possível, então  $D$  é  $O(n \log n)$ .
- R-7.23 Seja  $T$  uma árvore binária com  $n$  nodos e seja  $p$  a numeração dos níveis de  $T$ , como visto na Seção 7.3.5,
- mostre que, para todo nodo  $v$  de  $T$ ,  $p(v) \leq 2^e - 1$ ;
  - mostre um exemplo de árvore binária com sete nodos que atinjam o limite superior acima no valor máximo de  $p(v)$  para algum nodo  $v$ .
- R-7.24 Mostre como usar o caminhamento de Euler para computar a numeração dos níveis definida na Seção 7.3.5 de cada nodo de uma árvore binária  $T$ .
- R-7.25 Desenhe uma árvore binária que represente a seguinte expressão aritmética: “ $((5 + 2) * (2 - 1)) / ((2 + 9) + (7 - 2) - 1) * 8$ ”.
- R-7.26 Seja  $T$  uma árvore binária com  $n$  nodos que é implementada sobre uma lista arranjo,  $S$ , e seja  $p$  a numeração dos níveis de  $T$  como mostrado na Seção 7.3.5, apresente descrições em pseudocódigo para os métodos `root`, `parent`, `left`, `right`, `hasLeft`, `hasRight`, `isInternal`, `isExternal` e `isRoot`.

## Criatividade

- C-7.1 Para cada nodo  $v$  de uma árvore  $T$ ,  $\text{pre}(v)$  é a colocação de  $v$  em um caminhamento prefixado sobre  $T$ ;  $\text{post}(v)$  é a colocação de  $v$  em um caminhamento pós-fixado sobre  $T$ ;  $\text{depth}(v)$  é a profundidade de  $v$ ; e  $\text{desc}(v)$  é o número de descendentes de  $v$  sem contar o próprio  $v$ . Derive a fórmula que define  $\text{post}(v)$  em termos de  $\text{desc}(v)$ ,  $\text{depth}(v)$  e  $\text{prev}(v)$  para cada nodo  $v$  de  $T$ .
- C-7.2 Seja  $T$  uma árvore cujos nodos armazenam strings. Forneça um algoritmo eficiente que calcule e imprima, para todo o nodo  $v$  de  $T$ , a string armazena da em  $v$  e a altura da subárvore com raiz em  $v$ .
- C-7.3 Projete algoritmos para as seguintes operações de uma árvore binária  $T$ :
- `preorderNext( $v$ )`: retorna o nodo visitado depois do nodo  $v$  em um caminhamento prefixado sobre  $T$ .
  - `inorderNext( $v$ )`: retorna o nodo visitado depois do nodo  $v$  em um caminhamento interfixado sobre  $T$ .
  - `postorderNext( $v$ )`: retorna o nodo visitado depois do nodo  $v$  em um caminhamento pós-fixado sobre  $T$ .
- Quais são os tempos de execução para o pior caso dos seus algoritmos?
- C-7.4 Apresente um algoritmo  $O(n)$  para calcular a profundidade de todos os nodos de uma árvore  $T$ , onde  $n$  é o número de nodos de  $T$ .

- C-7.5 A representação indentada entre parênteses de uma árvore  $T$  é uma variação da representação entre parênteses de  $T$  (ver a Figura 7.7), que usa indentação e quebras de linha como demonstrado na Figura 7.21. Apresente um algoritmo que imprime esta representação de uma árvore.



**Figura 7.21** (a) Árvore  $T$ ; (b) representação indentada entre parênteses de  $T$ .

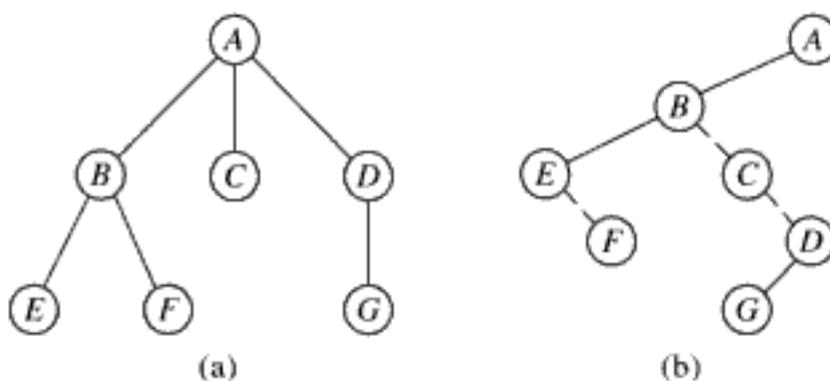
- C-7.6 Seja  $T$  uma árvore binária (possivelmente imprópria) com  $n$  nodos, e seja  $D$  a soma das profundidades de todos os nodos externos de  $T$ . Descreva uma configuração para  $T$  onde  $D$  seja  $\Omega(n^2)$ . Esta árvore deve corresponder ao pior caso para o tempo de execução assintótico do algoritmo `height1` (Trecho de código 7.5).
- C-7.7 Para uma árvore  $T$ , considere que  $n_i$  denota a quantidade de nodos internos e  $n_e$  denota a quantidade de nodos externos. Mostre que, se todo nodo interno de  $T$  tem exatamente três filhos, então  $n_e = 2n_i + 1$ .
- C-7.8 Descreva como clonar uma árvore binária própria usando o método `attach` em vez dos métodos `insertLeft` e `insertRight`.
- C-7.9 O **fator de balanceamento** de um nodo interno  $v$  de uma árvore binária própria é a diferença entre as alturas das subárvores direita e esquerda de  $v$ . Mostre como especializar o caminhamento de Euler da Seção 7.3.7 para imprimir os fatores de balanceamento de todos os nodos de uma árvore binária própria.
- C-7.10 Duas árvores ordenadas  $T'$  e  $T''$  são ditas *isomórficas* se uma das seguintes condições se aplicar:
- tanto  $T'$  como  $T''$  são vazias;
  - tanto  $T'$  como  $T''$  consistem em apenas um nodo;
  - tanto  $T'$  como  $T''$  têm o mesmo número  $k \geq 1$  de subárvores, e a  $i$ -ésima subárvore de  $T'$  é isomórfica à  $i$ -ésima subárvore de  $T''$  para  $i = 1, \dots, k$ .
- Projete um algoritmo que testa quando duas árvores ordenadas são isomórficas. Qual o tempo de execução de seu algoritmo?
- C-7.11 Estenda o conceito do caminhamento de Euler para uma árvore ordenada que não seja necessariamente binária.

- C-7.12 Pode-se definir a *representação de uma árvore binária*  $T'$  de uma árvore genérica  $T$  como segue (Figura 7.2.2):

- Para cada nodo  $u$  de  $T$ , existe um nodo  $u'$  de  $T'$  associado com  $u$ .
- Se  $u$  é um nodo externo de  $T$ , e não existe um irmão que o segue, então os filhos de  $u'$  em  $T'$  são nodos externos.
- Se  $u$  é um nodo interno de  $T$ , e  $v$  é o primeiro filho de  $u$  em  $T$ , então  $v'$  é o filho da esquerda de  $u'$  em  $T'$ .
- Se o nodo  $v$  tem um irmão  $w$  que o segue, então  $w'$  é o filho da direita de  $v'$  em  $T'$ .

Fornecida tal representação  $T'$  de uma árvore ordenada genérica  $T$ , responda cada uma das questões que seguem:

- O caminhamento prefixado sobre  $T'$  é equivalente ao caminhamento prefixado sobre  $T$ ?
- O caminhamento pós-fixado sobre  $T'$  é equivalente ao caminhamento pós-fixado sobre  $T$ ?
- O caminhamento interfixado de  $T'$  é equivalente a algum dos caminhamentos padrão sobre  $T$ ? Em caso positivo, qual deles?



**Figura 7.22** Representação de uma árvore binária: (a) árvore  $T$ ; (b) árvore binária  $T'$  correspondente a  $T$ . As arestas tracejadas conectam nodos de  $T'$  que correspondem a irmãos em  $T$ .

- C-7.13 Como foi mencionado no Exercício 5.8, a *notação pós-fixada* é uma forma não-ambígua de escrever expressões aritméticas sem usar parênteses. Se for definido que " $(exp_1)op(exp_2)$ " é uma expressão entre parênteses normal (interfixada) com operador **op**, então o pós-fixado equivalente é " $pexp_1 pexp_2 op$ ", onde  $pexp_1$  é a versão pós-fixada de  $exp_1$ , e  $pexp_2$  é a versão pós-fixada de  $exp_2$ . A versão pós-fixada de um único número ou variável é o próprio número ou variável. Assim, por exemplo, a versão pós-fixada da expressão interfixada " $((5 + 2) * (8 - 3)) / 4$ " é " $5\ 2 + 8\ 3 - * 4 /$ ". Forneça um algoritmo eficiente para converter uma expressão interfixada em sua equivalente em notação pós-fixada. (Dica: primeiro converta a expressão interfixada em sua árvore binária equivalente, usando o algoritmo do Trecho de código 7.21).

- C-7.14 Sendo dada uma árvore binária própria  $T$ , defina o *reflexo* de  $T$  como sendo uma árvore binária  $T'$  tal que cada nodo  $v$  de  $T$  esteja também em  $T'$ , mas de maneira que o filho da esquerda de  $v$  em  $T$  seja o filho da direita de  $v$  em  $T'$  e o filho da direita de  $v$  em  $T$  seja o filho da esquerda de  $v$  em  $T'$ . Mostre que um caminhamento prefixado sobre uma árvore binária  $T$  é o mesmo que o caminhamento pós-fixado sobre o reflexo de  $T$  mas na ordem inversa.

- C-7.15 O algoritmo `preorderDraw` desenha uma árvore binária  $T$  atribuindo coordenadas  $x$  e  $y$  para cada nodo  $v$ , de maneira que  $x(v)$  é igual ao número de nodos que precede  $v$  no caminhamento prefixado de  $T$ , e  $y(v)$  é igual à profundidade de  $v$  em  $T$ . O algoritmo `postOrderDraw` é similar a `preorderDraw`, mas atribui as coordenadas  $x$  usando um caminhamento pós-fixado.
- Mostre que o desenho de  $T$  produzido pelo algoritmo `preorderDraw` não apresenta arestas que se cruzem.
  - Redesenhe a árvore binária da Figura 7.19, usando o algoritmo `preorderDraw`.
  - Mostre que o desenho de  $T$  produzido pelo algoritmo `postorderDraw` não apresenta arestas que se cruzem.
  - Redesenhe a árvore binária da Figura 7.19, usando o algoritmo `postorderDraw`.
- C-7.16 Projete um algoritmo para desenhar árvores genéricas que generaliza a abordagem do caminhamento interfixado para o desenho de árvores binárias.
- C-7.17 Considere que a ação a ser aplicada durante o caminhamento de Euler seja denotada pelo par  $(v,a)$ , onde  $v$  é o nodo visitado e  $a$  é da **esquerda**, **abaixo** ou da **direita**. Projete e analise um algoritmo que execute a operação `tourNext(v,a)` que retorna a ação  $(w,b)$  que segue  $(v,a)$ .
- C-7.18 Considere uma variação da estrutura de dados encadeada para árvores binárias na qual cada objeto nodo tem referências para os objetos nodo filhos, mas não para o objeto nodo pai. Descreva a implementação dos métodos de uma árvore binária com esta estrutura e analise a complexidade temporal destes métodos.
- C-7.19 Projete uma implementação alternativa para a estrutura de dados encadeada para árvores binárias usando uma classe para os nodos que seja especializada em subclasses para nodo interno, nodo externo e raiz.
- C-7.20 Usando uma estrutura de dados encadeada para árvores binárias, explore um projeto alternativo para implementar os iteradores retornados pelos métodos `iterator()`, `positions().iterator()` e `children(v).iterator()` de maneira que cada um desses métodos execute em tempo  $O(1)$ . É possível obter implementações de tempo constante para os métodos de iteração `hasNext()` e `next()` dos iteradores retornados?
- C-7.21 Seja  $T$  uma árvore com  $n$  nodos. Defina o **ancestral comum mais baixo** (ACB) entre dois nodos  $v$  e  $w$  como o nodo mais baixo de  $T$  que tem ambos,  $v$  e  $w$  como descendentes (neste caso permitimos que um nodo seja descendente de si mesmo). Dados dois nodos  $v$  e  $w$ , descreva um algoritmo eficiente para encontrar o ACB de  $v$  e  $w$ . Qual é o tempo de execução do algoritmo?
- C-7.22 Seja  $T$  uma árvore com  $n$  nodos e para cada nodo  $v$  de  $T$  denotamos  $d_v$  a profundidade de  $v$  em  $T$ . A **distância** entre dois nodos  $v$  e  $w$  em  $T$  é  $d_v + d_w - 2d_u$ , onde  $u$  é o ACB  $u$  de  $v$  e  $w$  (como definido no exercício anterior). O **diâmetro** de  $T$  é a distância máxima entre dois nodos em  $T$ . Descreva um algoritmo eficiente para encontrar o diâmetro de  $T$ . Qual o tempo de execução de seu algoritmo?
- C-7.23 Suponha que cada nodo  $v$  de uma árvore binária  $T$  seja rotulado com seu valor  $p(v)$  em um dos níveis numerados de  $T$ . Projete um método rápido para determinar  $p(u)$  para o ancestral comum mais baixo (ACB),  $u$ , entre dois

nodos  $v$  e  $w$  de  $T$ , dados  $p(v)$  e  $p(w)$ . Não é necessário determinar  $u$ , apenas calcular o número que identifica seu nível.

- C-7.24 Justifique os limites da Tabela 7.3 com uma análise detalhada dos tempos de execução dos métodos de uma árvore binária  $T$ , implementada sobre uma lista arranjo  $S$ , onde  $S$  é definida sobre um arranjo.
- C-7.25 Justifique a Tabela 7.1 resumindo o tempo de execução dos métodos de uma árvore representada com uma estrutura encadeada apresentando, para cada método, uma descrição de sua implementação e uma análise do tempo de execução.
- C-7.26 Descreva um método não-recursivo para avaliar uma árvore binária que representa uma expressão aritmética.
- C-7.27 Seja  $T$  uma árvore binária com  $n$  nodos, defina um **nodo romano** como sendo um nodo  $v$  de  $T$ , de maneira que o número de descendentes na subárvore esquerda de  $v$  diferencie-se do número de descendentes da subárvore direita de  $v$  por no máximo 5. Descreva um método com tempo de execução linear para encontrar cada nodo  $v$  de  $T$ , tal que  $v$  não seja um nodo romano, mas que todos os seus descendentes o sejam.
- C-7.28 Descreva um método não-recursivo para executar o caminhamento de Euler sobre uma árvore binária que execute em tempo linear e não use uma pilha.
- C-7.29 Descreva em pseudocódigo um método não-recursivo para executar o caminhamento interfixado sobre uma árvore binária em tempo linear.
- C-7.30 Seja  $T$  uma árvore binária com  $n$  nodos ( $T$  pode ser implementada usando uma lista arranjo ou uma estrutura encadeada). Forneça um método de tempo linear que use métodos da interface `BinaryTree` para percorrer os nodos de  $T$  através do incremento dos valores da função de numeração por nível  $p$  apresentada na Seção 7.3.5. Esse caminhamento é conhecido como **caminhamento por nível**.
- C-7.31 O **tamanho do caminho** de uma árvore  $T$  é a soma das profundidades de todos os nodos de  $T$ . Descreva um método com tempo de execução linear para calcular o tamanho do caminho de uma árvore  $T$  (que não é necessariamente binária).
- C-7.32 Defina o **tamanho do caminho interno**,  $I(T)$ , de uma árvore  $T$  como sendo a soma das profundidades de todos os nodos internos de  $T$ . Da mesma forma, defina o **tamanho do caminho externo**,  $E(T)$ , de uma árvore  $T$  como sendo a soma das profundidades de todos os nodos externos de  $T$ . Mostre que se  $T$  é uma árvore binária com  $n$  nodos internos, então  $E(T) = I(T) + n - 1$ .

## Projetos

- P-7.1 Implemente o TAD árvore binária usando uma lista arranjo.
- P-7.2 Implemente o TAD árvore binária usando uma lista encadeada.
- P-7.3 Escreva um programa capaz de desenhar uma árvore binária.
- P-7.4 Escreva um programa capaz de desenhar uma árvore genérica.
- P-7.5 Escreva um programa que permita tanto entrar como exibir a árvore genealógica de alguém.

P-7.6 Implemente o TAD árvore usando a representação por árvores binárias descrita no Exercício C-7.12. Você pode reusar a implementação de `LinkedBinaryTree` para árvores binárias.

P-7.7 Uma *planta baixa fatiada* é a decomposição de um retângulo com lados horizontais e verticais usando *cortes* horizontais e verticais. (Veja a Figura 7.23a.) Uma planta baixa fatiada pode ser representada por uma árvore binária chamada de *árvore de fatias*, cujos nodos internos representam os cortes e os nodos externos representam os retângulos básicos em que o chão é dividido pelos cortes. (Veja a Figura 7.23b.) O *problema da compactação* é definido como segue. Pressuponha que para cada retângulo básico de uma planta baixa fatiada atribui-se uma largura mínima  $w$  e uma altura mínima  $h$ . O problema da compactação é encontrar a menor largura e altura para cada retângulo da planta baixa que seja compatível com as dimensões mínimas de cada retângulo básico. Em outras palavras, este problema requer a atribuição de valores  $h(v)$  e  $w(v)$  para cada nodo  $v$  da árvore de fatias de maneira que:

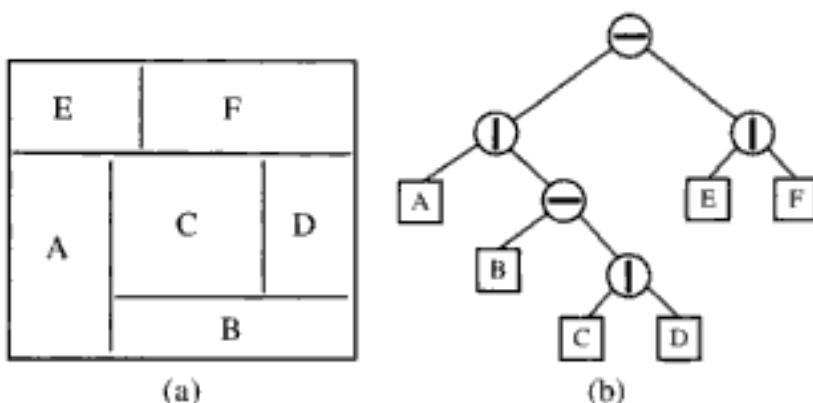
$$w(v) = \begin{cases} w & \text{se } v \text{ for um nodo externo cujo retângulo base tem a largura mínima } w \\ \max(w(w), w(z)) & \text{se } v \text{ for um nodo interno associado com um corte horizontal com o filho da esquerda } w \text{ e da direita } z \\ w(w) + w(z) & \text{se } v \text{ for um nodo interno associado com um corte vertical com o filho da esquerda } w \text{ e da direita } z \end{cases}$$
  

$$h(v) = \begin{cases} h & \text{se } v \text{ for um nodo externo cujo retângulo base tem a altura mínima } h \\ h(w) + h(z) & \text{se } v \text{ for um nodo interno associado com um corte horizontal com o filho da esquerda } w \text{ e da direita } z \\ \max(h(w), h(z)) & \text{se } v \text{ for um nodo interno associado com um corte vertical com o filho da esquerda } w \text{ e da direita } z \end{cases}$$

Projete uma estrutura de dados para plantas baixas fatiadas que suporte as operações:

- criar uma planta baixa composta de retângulos básicos individuais;
- decompor um retângulo básico através de um corte horizontal;
- decompor um retângulo básico através de um corte vertical;
- atribuir uma altura e largura mínima a um retângulo básico;
- desenhar a árvore de fatias associada à planta baixa;
- compactar e desenhar a planta baixa.

P-7.8 Escreva um programa que efetivamente possa jogar o jogo-da-velha (ver Seção 3.1.5). Para tanto, será necessário criar uma *árvore de jogadas*  $T$ , que é uma árvore na qual cada nodo representa uma *configuração de jogada*, o que, neste caso, corresponde a uma representação do tabuleiro do jogo-da-velha. O nodo raiz corresponde à configuração inicial. Para cada nodo interno  $v$  de  $T$ , os filhos de  $v$  correspondem aos estados do jogo possíveis de serem alcançados a partir do estado inicial em uma única



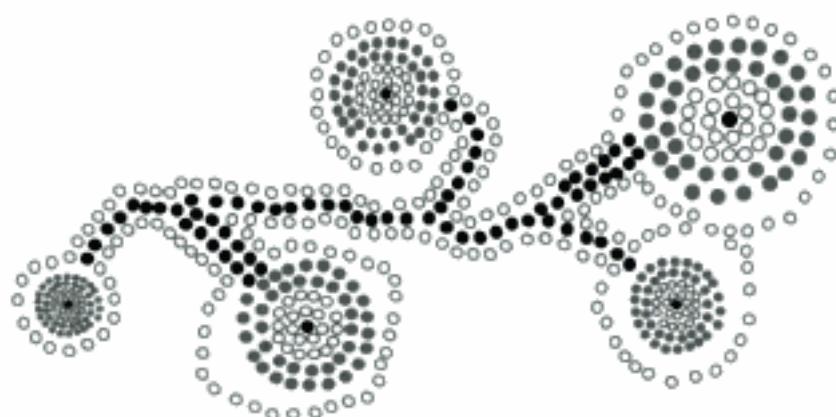
**Figura 7.23** (a) Planta baixa fatiada; (b) árvore de fatias associada com a planta baixa.

jogada do jogador da vez, *A* (o primeiro jogador) ou *B* (o segundo jogador). Nodos de profundidade par correspondem a jogadas de *A* e nodos de profundidade ímpar correspondem a jogadas de *B*. Nodos externos podem ser tanto estados finais do jogo ou estarem localizados em uma profundidade que não se deseja explorar. Para cada nodo externo atribui-se um valor que indica quão bom este estado é para o jogador *A*. Em jogos mais complexos, como o xadrez, é necessário adotar uma função heurística para atribuir este valor, mas para jogos simples, como jogo-da-velha, pode-se construir toda a árvore de jogadas e atribuir valor para os nodos com +1, 0, -1, indicando se o jogador *A* tem a ganhar ou a perder com está configuração. Um bom algoritmo de seleção de jogadas é o **minimax**. Neste algoritmo, atribui-se um valor para cada nodo interno *v* de *T*, de maneira que se *v* representa a vez de *A*, calcula-se o valor de *v* como o valor máximo dos filhos de *v* (que corresponde a melhor jogada para *A* a partir de *v*). Se um nodo interno *v* representa a vez de *B*, então calcula-se o valor de *v* como o menor valor dos filhos de *v* (o que corresponde a melhor jogada para *B* a partir de *v*).

- P-7.9 Escreva um programa que receba como entrada uma expressão aritmética toda entre parênteses e a converta em uma árvore binária que representa uma expressão. Seu programa deve exibir a árvore de alguma forma e também imprimir o valor associado com a raiz. Como desafio adicional, permita que se armazene nas folhas variáveis da forma  $x_1, x_2, x_3$  e, assim por diante que são inicializadas com 0 e que podem ser atualizadas interativamente por meio do programa, atualizando de forma coerente o valor impresso que corresponde ao valor da raiz da árvore que representa a expressão.
- P-7.10 Escreva um programa que visualiza o caminhamento de Euler sobre uma árvore binária própria, incluindo os movimentos nodo a nodo e as ações associadas com as visitas pela esquerda, por baixo e pela direita. Demonstre seu programa fazendo-o computar e mostrar os rótulos prefixados, interfixados e pós-fixados, bem como contadores de ancestrais e contadores de descendentes para cada nodo da árvore (não necessariamente todos ao mesmo tempo).
- P-7.11 A codificação de expressões aritmética apresentada nos Trechos de código 7.29-7.32 funciona apenas para expressões inteiras com operador de soma. Escreva um programa Java que possa calcular expressões arbitrárias com qualquer tipo numérico de objeto.

## Observações sobre o capítulo

Discussões sobre os caminhamentos clássicos, prefixado, interfixado e pós-fixado podem ser encontradas no livro do Knut, *Fundamental Algorithms* [62]. O caminhamento de Euler é originário da comunidade de processamento paralelo, e foi introduzido por Tarjan e Vishkin [89] e discutido por JáJá [53] e por Karp e Ramachandran [57]. O algoritmo de desenho de uma árvore é genericamente considerado como parte do “folclore” dos algoritmos de desenho de grafos. Para o leitor interessado no desenho de grafos, indicam -se os trabalhos de Tamassia [88] e Di Battista et al.[30]. O quebra-cabeça do Exercício R-7.10 foi apresentado por Micha Sharir.



## Conteúdo

<b>8.1 O tipo abstrato de dados fila de prioridade . . . . .</b>	<b>292</b>
8.1.1 Chaves, prioridades e relações de ordem total . . . . .	292
8.1.2 Entradas e comparadores . . . . .	293
8.1.3 O TAD fila de prioridade . . . . .	295
8.1.4 Ordenando com uma fila de prioridade. . . . .	297
<b>8.2 Implementando uma fila de prioridade com seqüências . . . . .</b>	<b>298</b>
8.2.1 Implementação com uma seqüência não-ordenada. . . . .	298
8.2.2 Implementação com uma seqüência ordenada . . . . .	299
8.2.3 Selection sort e insertion sort . . . . .	301
<b>8.3 Heaps . . . . .</b>	<b>303</b>
8.3.1 A estrutura de dados heap . . . . .	303
8.3.2 Árvores binárias completas e suas representações . . . . .	305
8.3.3 Implementando uma fila de prioridade com um heap. . . . .	309
8.3.4 Implementação em Java . . . . .	313
8.3.5 Heap-sort . . . . .	316
8.3.6 Construção bottom-up do heap ★ . . . . .	317
<b>8.4 Filas de prioridade adaptáveis . . . . .</b>	<b>320</b>
8.4.1 Métodos do TAD fila de prioridade adaptável. . . . .	320
8.4.2 Localizadores . . . . .	321
8.4.3 Implementando uma fila de prioridade adaptável. . . . .	322
<b>8.5 Exercícios . . . . .</b>	<b>324</b>

## 8.1 O tipo abstrato de dados fila de prioridade

Uma *fila de prioridade* é um tipo abstrato de dados para armazenar uma coleção de elementos priorizados que suporta a inserção de elementos arbitrários, mas suporta a remoção de elementos em ordem de prioridade, ou seja, o elemento com prioridade mais alta pode ser removido a qualquer momento. Este TAD é fundamentalmente diferente das estruturas de dados posicionais discutidas nos capítulos anteriores, tais como pilhas, filas, dequeus, seqüências e mesmo árvores. Essas estruturas de dados armazenam elementos em posições específicas, que são freqüentemente posições em uma estrutura linear de elementos, determinada pela seqüência efetuada de inserções e remoções. O TAD fila de prioridade armazena elementos de acordo com suas prioridades, e não tem noção de "posição".

---

### 8.1.1 Chaves, prioridades e relações de ordem total

As aplicações comumente requerem a comparação e classificação de objetos de acordo com parâmetros ou propriedades, chamadas "chaves", que são associadas a cada objeto em uma coleção. Formalmente, uma *chave* é definida como um objeto associado a um elemento como seu atributo específico, e que pode ser usada para identificar, classificar ou ponderar esse elemento. É importante observar que a chave é associada a um elemento, tipicamente por um usuário ou aplicação, e por isso pode representar uma propriedade que um objeto não possui originalmente.

No entanto, a chave que uma aplicação associa a um objeto não é necessariamente única, e uma aplicação pode alterar a chave de um elemento se for necessário. Por exemplo, podemos comparar companhias por seus lucros ou pelo número de funcionários. Portanto, qualquer um desses parâmetros pode ser usado como chave para uma companhia, dependendo da informação que se deseja buscar. De forma similar, pode-se comparar restaurantes pela qualificação dada por um gastrônomo ou pelo preço médio da refeição. Para obter maior generalidade, portanto, pode-se permitir que uma chave seja do tipo mais adequado a uma aplicação.

Como no exemplo anterior sobre o aeroporto, a chave usada para comparações é freqüentemente mais do que um simples valor numérico como preço, tamanho, peso ou velocidade. Ou seja, uma chave pode ser uma propriedade mais complexa que não pode ser quantificada com um simples número. Por exemplo, a prioridade de passageiros em espera é geralmente determinada levando-se em conta vários fatores diferentes, como condição de viajante freqüente, tarifa paga e hora de chegada. Em algumas aplicações, a chave para um objeto é parte do próprio objeto (por exemplo, pode ser o preço de um livro ou o peso de um automóvel). Em outras aplicações, a chave não faz parte do objeto, mas é associada a ele pela aplicação (por exemplo, o grau de retorno de uma ação dada por um analista de finanças, ou a prioridade dada a um passageiro pelo atendente de embarque).

#### Comparando chaves com ordens totais

Uma fila de prioridade precisa de uma regra de comparação que nunca se contradiga. Para que uma regra de comparação (denotada  $\leq$ ) seja robusta, ela deve definir uma relação de *ordem total*, o que significa dizer que a regra de comparação é definida para cada par de chaves e deve satisfazer as seguintes propriedades:

- **Propriedade reflexiva:**  $k \leq k$ .
- **Propriedade anti-simétrica:** se  $k_1 \leq k_2$  e  $k_2 \leq k_1$ , então  $k_1 = k_2$ .
- **Propriedade transitiva:** se  $k_1 \leq k_2$  e  $k_2 \leq k_3$ , então  $k_1 \leq k_3$ .

Qualquer regra de comparação  $\leq$  que satisfaça essas três propriedades nunca levará a uma contradição nas comparações. De fato, uma regra assim define uma relação de ordem linear sobre

um conjunto de chaves; assim, se uma coleção (finita) de elementos tem uma ordem total definida para si, então a noção de uma **menor** chave  $k_{\min}$  é bem definida, como uma chave para a qual  $k_{\min} \leq k$  para qualquer outra chave  $k$  na coleção.

Uma **fila de prioridade** é um contêiner de elementos, cada um tendo uma chave associada atribuída no instante em que o elemento é inserido. O nome “fila de prioridade” vem do fato de que as chaves fornecem a “prioridade” usada para escolher elementos a serem removidos. Os dois métodos fundamentais de uma fila de prioridade  $P$  são:

- `insertItem(k,e)`: insere o elemento  $e$  com chave  $k$  em  $P$ ;
- `removeMin()`: retorna e remove de  $P$  o elemento com a menor chave, ou seja, um elemento cuja chave é menor ou igual à chave de qualquer outro elemento em  $P$ .

Às vezes, as pessoas se referem ao método `removeMin` como o método “`extractMin`”, para enfatizar que este método simultaneamente remove e retorna o elemento com a menor chave em  $P$ . Há muitas aplicações em que as operações de `insertItem` e `removeMin` desempenham um papel importante. Uma aplicação assim é analisada no exemplo que se segue.

**Exemplo 8.1** Suponha que um vôo está totalmente reservado uma hora antes da decolagem. Por causa da possibilidade de cancelamentos, a companhia aérea mantém uma fila de prioridade de passageiros em espera por um assento. A prioridade de cada passageiro é determinada pela companhia levando em conta a tarifa paga, a condição (ou não) de cliente frequente do passageiro e desde quando o passageiro está à espera do lugar. Uma referência para o passageiro em espera é inserida na fila de prioridade com uma operação de `insertItem`. Pouco antes da saída do vôo, se houver lugares disponíveis (por exemplo, devido a ausências ou cancelamentos) a companhia remove da fila de prioridade o passageiro em espera com maior prioridade usando uma operação `removeMin`, e dá um lugar a ele. O processo é repetido até que todos os lugares livres tenham sido tomados ou que a fila de prioridade esteja vazia.

### 8.1.2 Entradas e comparadores

Existem dois tópicos importantes indeterminados até este ponto:

- Como mantêm-se o rastro de associações entre chaves e valores?
- Como comparam-se chaves e como determina-se a menor chave?

Responder estas questões envolve o uso de dois interessantes padrões de projeto.

A definição da fila de prioridade cria implicitamente o uso de dois tipos especiais de objetos que respondem as questões anteriores, **entrada** (*entry*) e **comparador** (*comparator*), os quais serão discutidos nesta subseção.

#### Entradas

Uma entrada é uma associação entre uma chave  $k$  e um valor  $x$ , isto é, uma **entrada** é simplesmente o par chave-valor. Usam-se entradas na fila de prioridade  $Q$  para se ter noção de como  $Q$  associa chaves com seus respectivos valores.

Uma entrada é realmente um exemplo mais abrangente de um padrão de projeto orientado a objetos, o **padrão composição**, o qual define um simples objeto que é composto de outros objetos. Usa-se este padrão na fila de prioridade quando definem-se as entradas sendo armazenadas na lista de prioridade, consistindo o par da chave  $k$  a no valor  $x$ .

Um par é a composição mais simples, para combinar dois objetos em um simples objeto (o par). Para implementar este conceito, define-se uma classe que armazena dois objetos na primeira e na segunda variável de instância, respectivamente, e provê métodos para acessar e alterar estas variáveis.

O Trecho de código 8.1, mostra uma implementação de entradas no padrão composição, armazenando pares chave-valor uma fila de prioridade. Implementa-se esta composição como uma interface chamada Entry (o pacote `java.util` inclui uma interface `Entry` similar). Outros tipos de composições incluem triplos, os quais armazenam três objetos, quádruplos, os quais armazenam quatro objetos, e assim por diante.

```
/** Interface para Entrada do par chave-valor */
public interface Entry<K,V> {
    /** Retorna uma chave armazenada nesta entrada. */
    public K getKey();
    /** Retorna o valor armazenado nesta entrada. */
    public V getValue();
}
```

**Trecho de código 8.1** Interface Java para uma entrada que armazena o par chave-valor em uma fila de prioridade

## Comparadores

Outro importante tópico no TAD fila de prioridade que é preciso definir é como especificar a relação a ser usada para comparar chaves. Existem algumas escolhas com respeito a este tópico que podem ser feitas neste ponto.

Uma possibilidade, que é a mais concreta, é implementar uma fila de prioridade diferente para cada tipo de chave que se deseja usar e para cada forma possível de comparar tais chaves. O problema com esta abordagem é que ela não é genérica, e requer que sejam criados muitos códigos similares.

Uma estratégia alternativa seria exigir que as chaves pudessem comparar a si mesmas. Essa solução permite a criação de uma classe de fila de prioridade genérica que armazena instâncias de uma classe de chaves que implementa algum tipo de interface `Comparable` e encapsula todos os métodos de comparação. Essa solução é um avanço sobre a abordagem especializada, pois ela permite que se escreva uma única classe para a fila de prioridade que pode tratar vários tipos diferentes de chave. Mesmo assim, existem contextos em que esta solução não é possível, pois muitas vezes as chaves não “sabem” como devem ser comparadas. A seguir, dois exemplos.

**Exemplo 8.2** Dadas as chaves 4 e 11, tem-se  $4 \leq 11$  se as chaves forem objetos inteiros (a ser comparados da forma usual), mas  $11 \leq 4$  se as chaves forem cadeias de caracteres (a ser comparadas lexicograficamente).

**Exemplo 8.3** Um algoritmo geométrico pode comparar os pontos  $p$  e  $q$  no plano por suas coordenadas  $x$  (ou seja,  $p \leq q$  se  $x(p) \leq x(q)$ ) e ordená-los da esquerda para a direita, enquanto outro algoritmo pode compará-los pela coordenada  $y$  (ou seja  $p \leq q$  se  $y(p) \leq y(q)$ ) e ordená-los de baixo para cima. Em princípio, não há nada no conceito de “ponto” que informe se os pontos devem ser comparados por coordenada  $x$  ou  $y$ . Muitas outras maneiras de comparar pontos podem ser definidas (por exemplo, comparar pela distância de  $p$  e  $q$  até a origem).

Assim, para obter a forma mais genérica e reutilizável de fila de prioridade, não se deve esperar que as chaves forneçam seu próprio mecanismo de comparação. Em vez disso, usam-se objetos **comparadores** especiais, que são externos às chaves e fornecem as regras de comparação. Um comparador é um objeto que compara duas chaves. Pressupõe-se que uma fila de prioridade  $P$  recebe um comparador quando é construída, e pode-se imaginar que uma fila de prioridade receba outro comparador se o antigo ficar “desatualizado”. Quando  $P$  precisa comparar duas chaves, ela usa o comparador para fazer as comparações. Assim, um programador pode escrever

uma implementação genérica para uma fila de prioridade que funcione corretamente em uma variedade de contextos.

Formalmente, o TAD comparador provê um eficiente mecanismo de comparação, baseado em um simples método que recebe duas chaves e comparando-as (ou relatando um erro se as chaves são incompatíveis):

**compare(*a,b*):** Retorna um valor inteiro *i* onde *i < 0* se *a < b*, *i = 0* se *a = b* e *i > 0* se *a > b*; um erro ocorre se *a* e *b* não podem ser comparados.

A interface padrão Java `java.util.Comparator` corresponde ao TAD comparador descrito acima, o qual oferece uma forma geral, dinâmica e reutilizável de comparar objetos. Isto também inclui um método `equals()` para comparar um objeto comparador com outro objeto comparador. O Trecho de código 8.2 apresenta um exemplo de um comparador, para pontos em 2D (Trecho de código 8.3), o qual é um exemplo do padrão composição.

```
/** Comparador para pontos 2D sobre o padrão de ordem lexicográfico. */
public class Lexicographic<E extends Point2D>
    implements java.util.Comparator<E> {
    /** Compare two points */
    public int compare(E a, E b) {
        if (a.getX() != b.getX())
            return b.getX() - a.getX();
        else
            return b.getY() - a.getY();
    }
} // Classe Lexicographic automaticamente herda o método equals() method da classe Object
```

**Trecho de código 8.2** Um comparador para pontos 2D baseado na ordem lexicográfica.

```
/** Classe que representa um ponto no plano com coordenadas inteiras */
public class Point2D {
    protected int xc, yc;      // coordenadas
    public Point2D(int x, int y) {
        xc = x;
        yc = y;
    }
    public int getX() { return xc; }
    public int getY() { return yc; }
}
```

**Trecho de código 8.3** Classe que representa pontos em um plano com coordenadas inteiras.

### 8.1.3 O TAD fila de prioridade

Tendo descrito os padrões composição e comparador, será definido agora o TAD fila de prioridade, que suporta os seguintes métodos para a fila de prioridade *P*:

- size():** retorna o número de entradas em *P*;
- isEmpty():** testa se *P* está vazia;
- min():** retorna (mas não remove) um entrada de *P* com a menor chave; uma condição de erro ocorre se *P* estiver vazia;
- insert(*k,x*):** insere em *P* a chave *k* com o valor *x* e retorna a entrada armazenada; uma condição de erro ocorre se *k* é inválido (isto é, *k* não pode ser comparado com outras chaves);

`removeMin()`: remove de  $P$  e retorna uma entrada com a menor chave; uma condição de erro ocorre se  $P$  estiver vazia.

Como mencionado acima, os métodos primários do TAD fila de prioridade são as operações de `insert` e `removeMin`. Os outros métodos são a operação de consulta `min` e operações gerais de coleções `size` e `isEmpty`. É permitido que uma fila de prioridade tenha múltiplas entradas com a mesma chave.

### Uma Interface Java para fila de prioridade

Uma interface Java, chamada `PriorityQueue`, para o TAD fila de prioridade é apresentado no Trecho de código 8.4.

```
/** Interface para o TAD fila de prioridade */
public interface PriorityQueue<K,V> {
    /** Retorna o número de itens em uma fila de prioridades. */
    public int size();
    /** Retorna se a fila de prioridade está vazia. */
    public boolean isEmpty();
    /** Retorna mas não remove um entrada com chave mínima. */
    public Entry<K,V> min() throws EmptyPriorityQueueException;
    /** Insere um para chave-valor e retorna a entrada criada. */
    public Entry<K,V> insert(K key, V value) throws InvalidKeyException;
    /** Remove e retorna uma entrada com chave mínima. */
    public Entry<K,V> removeMin() throws EmptyPriorityQueueException;
}
```

**Trecho de código 8.4** Interface Java para o TAD fila de prioridade.

Deve estar bastante óbvio agora que o TAD fila de prioridade é muito mais simples do que o TAD seqüência. Esta simplicidade se deve ao fato de que os elementos em uma fila de prioridade são inseridos e removidos baseando-se inteiramente em suas chaves, enquanto os elementos em seqüências são inseridos e removidos de acordo com suas posições e índices.

**Exemplo 8.4** A tabela a seguir mostra uma seqüência de operações e seus efeitos sobre uma fila de prioridade  $P$  inicialmente vazia. Marca-se com  $e_i$  um objeto entrada retornado do método `insert`. A coluna “Fila de prioridade” é um pouco enganosa, já que mostra as entradas ordenadas pelas chaves. Isto é mais do que é requerido pela fila de prioridades.

Operação	Saída	Fila de prioridade
<code>insert(5,A)</code>	$e_1 [= (5,A)]$	$\{(5,A)\}$
<code>insert(9,C)</code>	$e_2 [= (9,C)]$	$\{(5,A),(9,C)\}$
<code>insert(3,B)</code>	$e_3 [= (3,B)]$	$\{(3,B),(5,A),(9,C)\}$
<code>insert(7,D)</code>	$e_4 [= (7,D)]$	$\{(3,B),(5,A),(7,D),(9,C)\}$
<code>min()</code>	$e_3$	$\{(3,B),(5,A),(7,D),(9,C)\}$
<code>removeMin()</code>	$e_3$	$\{(5,A),(7,D),(9,C)\}$
<code>size()</code>	3	$\{(5,A),(7,D),(9,C)\}$
<code>removeMin()</code>	$e_1$	$\{(7,D),(9,C)\}$
<code>removeMin()</code>	$e_4$	$\{(9,C)\}$
<code>removeMin()</code>	$e_2$	$\{\}$

### A classe `java.util.PriorityQueue`

Não existe uma interface de fila de prioridades em Java, porém Java inclui uma classe, `java.util.PriorityQueue`, a qual implementa a interface `java.util.Queue`. Em vez de adicionar e remover elementos de acordo com a política FIFO, que é uma política padrão para filas, a classe `java.util.PriorityQueue` processa as entradas de acordo com a prioridade. Esta prioridade é definida por um objeto comparador, que é enviado para o construtor da fila, ou é definido pela ordenação natural dos elementos armazenados na fila. Embora a classe `java.util.PriorityQueue` seja baseada na interface `java.util.Queue`, pode-se definir uma simples correspondência entre os métodos desta classe e nosso TAD de fila de prioridade, como mostrado na Tabela 8.1, assumindo que se tem uma classe `PQEntry`, que implementa a interface `Entry`.

TAD de fila de prioridade	Classe <code>java.util.PriorityQueue</code>
<code>size()</code>	<code>size()</code>
<code>isEmpty()</code>	<code>isEmpty()</code>
<code>insert(<i>k, v</i>)</code>	<code>offer(new PQEntry(<i>k, v</i>)) or add(new PQEntry(<i>k, v</i>))</code>
<code>min()</code>	<code>peek(), or element()</code>
<code>removeMin()</code>	<code>poll(), or remove()</code>

**Tabela 8.1** Métodos do TAD fila de prioridades e métodos correspondentes da classe `java.util.PriorityQueue`. Assume-se que o comparador para objetos `PQEntry` é essencialmente o mesmo comparador para as chaves da fila de prioridades. O `java.util.PriorityQueue` tem um par de métodos para operações principais.

#### 8.1.4 Ordenando com uma fila de prioridade

Outra aplicação importante de uma fila de prioridade é a ordenação, na qual tem-se uma coleção  $S$  de  $n$  elementos que podem ser comparados com uma relação de ordem total e devem ser re-arranjados em ordem crescente de chave (ou em ordem não-decrescente, se houver empates). O algoritmo para ordenar  $S$  com uma fila de prioridade  $Q$ , chamado `PriorityQueueSort`, é bastante simples e consiste nas duas fases a seguir:

1. Na primeira fase, os elementos de  $S$  são colocados em uma fila de prioridade  $P$  inicialmente vazia através de uma série de operações `insert`, uma para cada elemento de  $S$ .
2. Na segunda fase, os elementos de  $P$  são retirados em ordem não-decrescente através de  $n$  operações `removeMin`, colocando-os novamente em  $S$  em ordem.

O pseudocódigo desse algoritmo é mostrado no Trecho de código 8.5, pressupondo que  $S$  é uma sequência (um pseudocódigo para um tipo de coleção diferente, como uma lista ou arranjo, seria semelhante). O algoritmo funciona corretamente para qualquer fila de prioridade  $P$ , não interessando como  $P$  é implementada. Entretanto, o tempo de execução do algoritmo é determinado pelos tempos de execução das operações `insert` e `removeMin`, que dependem de como  $P$  é implementada. Assim, `PriorityQueueSort` deve ser considerada mais um “esquema” de ordenação do que um algoritmo, porque ele não especifica como  $P$  deve ser implementada. O esquema `PriorityQueueSort` é o paradigma de vários algoritmos populares de ordenação, incluindo `selection sort`, `insertion sort` e `heapsort`, que serão discutidos neste capítulo.

**Algoritmo** PriorityQueueSort( $S, P$ ):

**Entrada:** uma seqüência  $S$  armazenando  $n$  elementos para os quais uma relação de ordem total está definida e uma fila de prioridade  $P$  que compara chaves usando a mesma relação de ordem total.

**Saída:** a seqüência  $S$  ordenada pela relação de ordem total.

**enquanto**  $\neg S.isEmpty()$  **faça**

$e \leftarrow S.removeFirst()$

$P.insert(e, \emptyset)$  {um valor nulo é utilizado}

**enquanto**  $\neg P.isEmpty()$  **faça**

$e \leftarrow P.removeMin().getKey()$

$S.addLast(e)$  {o menor elemento de  $P$  é adicionado no final de  $S$ }

**Trecho de código 8.5** Algoritmo PriorityQueueSort. Observe que os elementos da seqüência  $S$  servem tanto como chaves quanto como elementos da fila de prioridade  $P$ .

---

## 8.2 Implementando uma fila de prioridade com seqüências

Nesta seção será apresentado como implementar uma fila de prioridade por meio do armazenamento de entradas em uma lista  $S$ . (Veja Capítulo 6.2.) Duas alternativas são fornecidas, dependendo se forem mantidas as chaves em  $S$  ordenadas ou não. Quando é analisada a execução dos métodos da fila de prioridades implementados com uma seqüência, assume-se que a comparação das duas chaves ocorrem em um tempo  $O(1)$ .

---

### 8.2.1 Implementação com uma seqüência não-ordenada

Como na nossa primeira implementação de fila de prioridade  $P$ , considera-se o armazenamento das entradas de  $P$  em uma seqüência  $S$ , onde  $S$  é implementada com uma lista duplamente encadeada. Desta forma, os elementos de  $S$  são pares  $(k, x)$ , onde  $k$  é uma chave  $x$  é um valor.

#### Inserções rápidas e exclusões lentas

Uma forma simples de implementar o método  $insert(k, x)$  em  $P$  é criar um novo objeto  $e = (k, x)$  e adicioná-lo no final da lista  $S$ , executando o método  $addLast(e)$  de  $S$ . Esta implementação do método  $insert$  tem tempo  $O(1)$ .

Esta escolha implica que  $S$  não será ordenada, pois a inserção sempre no final de  $S$  não leva em conta a ordem das chaves. Como consequência, para realizar a operação  $min$  ou  $removeMin$  em  $P$ , deve-se inspecionar todos os elementos de  $S$  para encontrar o elemento  $p = (k, e)$  com o menor valor de  $k$ . Assim, independentemente de como a seqüência  $S$  é implementada, esses métodos de procura em  $P$  sempre custam tempo  $O(n)$  onde  $n$  é o número de elementos em  $P$  quando o método é executado. Adicionalmente, esses métodos são executados em tempo proporcional para  $n$ , mesmo no melhor caso, pois cada um deles requer que toda a seqüência seja pesquisada para se encontrar o menor elemento. Ou seja, usando a notação apresentada na Seção 4.2.3, pode-se dizer que estes métodos são executados em tempo  $\Theta(n)$ . Finalmente, implementam-se os métodos  $size$  e  $isEmpty$  que simplesmente retornam a saída das execuções dos métodos correspondentes da lista  $S$ .

Desta forma, usando uma seqüência não-ordenada para implementar uma fila de prioridade, obtemos inserção em tempo constante e remoção em tempo linear.

### 8.2.2 Implementação com uma seqüência ordenada

Uma implementação alternativa para uma fila de prioridade  $P$  também usa uma seqüência  $S$ , mas desta vez os elementos são armazenados em ordem de chave. Especificamente, a fila de prioridades  $P$  é representada usando a seqüência de entradas ordenadas de forma crescente pela chave, o que significa que o primeiro elemento de  $S$  é o elemento com a menor chave.

#### Rápida remoção e inserção lenta

Pode-se implementar o método `min`, neste caso, simplesmente acessando o primeiro elemento da seqüência usando o método `first` de  $S$ . Da mesma forma, implementa-se o método `removeMin` de  $P$  como sendo `S.remove(S.first())`. Assumindo que  $S$  seja implementada como uma lista duplamente encadeada, os métodos `min` e `removeMin` de  $P$  têm tempo  $O(1)$ . Desta forma, usando uma seqüência ordenada, permite uma simples e rápida implementação dos métodos de acesso e remoção de uma fila de prioridades.

Este benefício tem um custo, no entanto, pois agora o método `insert` de  $P$  requer que a seqüência  $S$  seja vasculhada para determinar a posição apropriada para inserir o novo elemento e sua chave. Assim, implementar o método `insert` de  $P$  agora exige tempo  $O(n)$  onde  $n$  é o número de elementos em  $P$  no momento em que o método é executado. Em suma, quando uma seqüência ordenada é usada para implementar uma fila de prioridade, a inserção é executada em tempo linear, enquanto a busca e a remoção de mínimos podem ser feitas em tempo constante.

#### Comparando as duas implementações

A Tabela 8.2 compara os tempos de execução dos métodos de uma fila de prioridade implementada com seqüências ordenadas e não-ordenadas, respectivamente. Uma seqüência não-ordenada permite inserções rápidas, mas consultas e remoções lentas, enquanto que uma seqüência ordenada permite consultas e remoções rápidas e inserções lentas.

Método	Seqüência não-ordenada	Seqüência ordenada
<code>size, isEmpty</code>	$O(1)$	$O(1)$
<code>insert</code>	$O(1)$	$O(n)$
<code>min, removeMin</code>	$O(n)$	$O(1)$

**Tabela 8.2** Piores casos na execução dos métodos de uma fila de prioridade de tamanho  $n$ , implementada com uma seqüência não-ordenada e seqüência ordenada, respectivamente. Assume-se que a seqüência é implementada com uma lista duplamente encadeada. O espaço requerido é  $O(n)$ .

#### Implementação Java

Os Trechos de código 8.6 e 8.8 mostram a implementação em Java da fila de prioridades baseada em uma seqüência ordenada de nodos. Esta implementação usa uma classe aninhada chamada `MyEntry`, que implementa a interface `Entry` (veja Seção 6.5.1). O método auxiliar `checkKey(k)`, o qual lança uma exceção chamada `InvalidKeyException` se a chave  $k$  não puder ser comparada com o comparador da fila de prioridade, não é mostrado. A classe `DefaultComparator`, que implementa o comparador usando ordenação natural, é apresentado no Trecho de código 8.7.

```

import java.util.Comparator;
/** implementação da fila de prioridade através de uma sequencia ordenada de nodos.*/
public class SortedListPriorityQueue<K,V> implements PriorityQueue<K,V> {
    protected PositionList<Entry<K,V>> entries;
    protected Comparator<K> c;
    protected Position<Entry<K,V>> actionPos;      // variável usada pela subclasse
    /** Classe interna para Entradas */
    protected static class MyEntry<K,V> implements Entry<K,V> {
        protected K k; // chave
        protected V v; // valor
        public MyEntry(K key, V value) {
            k = key;
            v = value;
        }
        // métodos da interface Entry
        public K getKey() { return k; }
        public V getValue() { return v; }
    }
    /** Cria uma fila de prioridades com o comparador padrão.*/
    public SortedListPriorityQueue () {
        entries = new NodePositionList<Entry<K,V>>();
        c = new DefaultComparator<K>();
    }
    /** Cria uma fila de prioridades com um comparador informado.*/
    public SortedListPriorityQueue (Comparator<K> comp) {
        entries = new NodePositionList<Entry<K,V>>();
        c = comp;
    }
}

```

**Trecho de código 8.6** Parte da classe `java SortedListPriorityQueue`, que implementa a interface `PriorityQueue`. A classe aninhada `MyEntry` implementa a interface `Entry` (continua no Trecho de código 8.8).

```

/** Comparador baseado na ordenação natural
 */
public class DefaultComparator<E> implements Comparator<E> {
    /** Compara dois elementos informados
     */
    public int compare(E a, E b) throws ClassCastException {
        return ((Comparable<E>) a).compareTo(b);
    }
}

```

**Trecho de código 8.7** Classe `java DefaultComparator` que implementa um comparador usando a ordenação natural e é o comparador padrão da classe `SortedListPriorityQueue`.

```

/** Retorna mas não remove uma entrada com a menor chave.*/
public Entry<K,V> min () throws EmptyPriorityQueueException {
    if (entries.isEmpty())
        throw new EmptyPriorityQueueException("Fila de prioridades está vazia ");
    else
        return entries.first().element();
}
/** Insere um par chave-valor e retorna o elemento criado.*/
public Entry<K,V> insert (K k, V v) throws InvalidKeyException {
    checkKey(k); // método auxiliary para verificação da chave (pode lança uma exceção)
}

```

Hidden page

$$O\left(n + (n-1) + \dots + 2 + 1\right) = O\left(\sum_{i=1}^n i\right).$$

Pela Proposição 4.3, tem-se  $\sum_{i=1}^n i = n(n+1)/2$ . Assim, a segunda fase custa tempo  $O(n^2)$ , como, o algoritmo completo.

		<i>Seqüência S</i>	<i>Fila de prioridade</i>
Entrada		(7, 4, 8, 2, 5, 3, 9)	0
Fase 1	(a)	(4, 8, 2, 5, 3, 9)	(7)
	(b)	(8, 2, 5, 3, 9)	(7, 4)
	:	:	:
	(g)	0	(7, 4, 8, 2, 5, 3, 9)
	(a)	(2)	(7, 4, 8, 5, 3, 9)
	(b)	(2, 3)	(7, 4, 8, 5, 9)
	(c)	(2, 3, 4)	(7, 8, 5, 9)
Fase 2	(d)	(2, 3, 4, 5)	(7, 8, 9)
	(e)	(2, 3, 4, 5, 7)	(8, 9)
	(f)	(2, 3, 4, 5, 7, 8)	(9)
	(g)	(2, 3, 4, 5, 7, 8, 9)	0

**Figura 8.1** Execução do algoritmo selection-sort na seqüência  $S = (7, 4, 8, 2, 5, 3, 9)$ .

### Insertion-Sort

Se a fila de prioridade  $P$  for implementada com uma seqüência ordenada, pode-se melhorar o tempo de execução da segunda fase para  $O(n)$ , pois cada operação `removeMin` em  $P$  custa tempo  $O(1)$ . Infelizmente, neste caso, a primeira fase se torna o gargalo para o tempo de execução, visto que, no pior caso, o tempo de execução de cada operação `insert` é proporcional ao tamanho de  $P$ . Este algoritmo de ordenação é, portanto mais conhecido como **insertion sort** (ver Figura 8.2), pois o seu gargalo envolve a repetida “inserção” de um novo elemento na posição apropriada da seqüência ordenada.

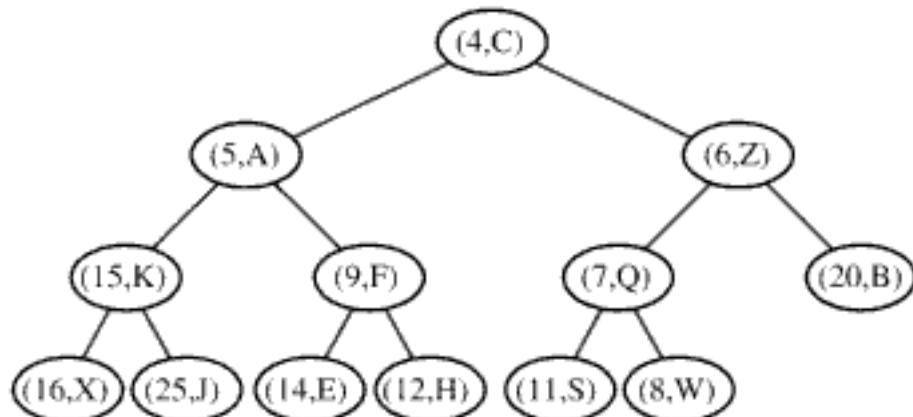
		<i>Seqüência S</i>	<i>Fila de prioridade</i>
Entrada		(7, 4, 8, 2, 5, 3, 9)	0
Fase 1	(a)	(4, 8, 2, 5, 3, 9)	(7)
	(b)	(8, 2, 5, 3, 9)	(4, 7)
	(c)	(2, 5, 3, 9)	(4, 7, 8)
	(d)	(5, 3, 9)	(2, 4, 7, 8)
	(e)	(3, 9)	(2, 4, 5, 7, 8)
	(f)	(9)	(2, 3, 4, 5, 7, 8)
	(g)	0	(2, 3, 4, 5, 7, 8, 9)
Fase 2	(a)	(2)	(3, 4, 5, 7, 8, 9)
	(b)	(2, 3)	(4, 5, 7, 8, 9)
	:	:	:
	(g)	(2, 3, 4, 5, 7, 8, 9)	0

**Figura 8.2** Execução do insertion sort na seqüência  $S = (7, 4, 8, 2, 5, 3, 9)$ .

Hidden page

a ver com o heap de memória (Seção 14.1.2) usado no ambiente de execução de uma linguagem de programação como Java.

Definindo-se o comparador para indicar o oposto da relação de ordem total entre as chaves (de forma que,  $\text{compare}(3,2) < 0$ , por exemplo), então a raiz do heap irá armazenar a chave maior. Essa versatilidade advém diretamente do uso do comparador padrão. Pela definição da chave mínima em termos de comparador, a chave “mínima” com o comparador “reverso” e ela será de fato a maior do heap.



**Figura 8.3** Exemplo de um heap armazenando 13 chaves inteiros. O último nodo é o que armazena a chave (8,W).

Assim, sem perda da generalidade, pode-se assumir que se está sempre interessado na chave mínima, que estará sempre na raiz do heap.

Para aumentar a eficiência, como será mostrado mais tarde, um heap  $T$  deve ter a menor altura possível. Efetiva-se esse requisito por meio de uma condição estrutural adicional: o heap deve ser **completo**. Antes de definir esta propriedade estrutural, algumas definições são necessárias. Foi visto na Seção 7.3.3 que o nível  $i$  de uma árvore binária  $T$  é o conjunto de nodos de  $T$  que tem profundidade  $i$ . Dados os nodos  $v$  e  $w$  no mesmo nível de  $T$ , diz-se que  $v$  está à **esquerda de**  $w$  se  $v$  é encontrado antes de  $w$  em um caminhamento interfixado de  $T$ . Isto é, há um nodo  $u$  em  $T$  em que  $v$  está na subárvore à esquerda de  $u$  e  $w$  está na subárvore à direita de  $u$ . Por exemplo, na árvore binária da Figura 8.3, o nodo armazenado na chave (15, K) está à esquerda do nodo armazenado na chave (7, Q). Em um padrão de projeto de árvores binárias, a relação “à esquerda” é visualizada pelo posicionamento horizontal relativo dos nodos.

**Árvore binária completa:** uma árvore binária  $T$  com altura  $h$  é **completa** se os níveis  $0, 1, 2, \dots, h - 1$  tiverem o maior número de nodos possível (ou seja, o nível  $i$  tem  $2^i$  nodos para  $0 \leq i \leq h - 1$ ) e no nível  $h - 1$  todos os nodos internos estão à esquerda dos nodos externos.

Insistindo que um heap  $T$  seja completo, identifica-se outro nodo importante do heap  $T$ , diferente da raiz: o **último nodo** de  $T$ , definida como sendo o nodo mais interno e mais à direita de  $T$  (ver Figura 8.3).

### Altura de um heap

Percebe-se que  $h$  indica a altura de  $T$ . Outra forma de definir o último nodo de  $T$  é verificar o nodo que está no nível  $h$  e em que todos os outros nodos do nível  $h$  estejam à esquerda deste. Insistir que  $T$  seja completo também tem uma importante consequência, como mostrado na Proposição 8.5.

**Proposição 8.5** Um heap  $T$  armazenando  $n$  chaves tem altura

$$h = \lfloor \log n \rfloor$$

**Justificativa** Já que  $T$  é completo, o número de nós internos é pelo menos

$$\begin{aligned} 1 + 2 + 4 + \dots + 2^{h-2} + 1 &= 2^{h-1} - 1 + 1 \\ &= 2^{h-1} \end{aligned}$$

Este limite inferior é obtido quando existe apenas um nó interno no nível  $h$ . Além disso, mas também vindo do fato de que  $T$  é completo, tem-se que o número de nós de  $T$  é no máximo

$$1 + 2 + 4 + \dots + 2^h = 2^{h+1} - 1.$$

Este limite superior é alcançado quando o nível  $h$  tem  $2^h$  nós. Desde que o número de nós é igual ao número de  $n$  chaves, obtém-se

$$2^h \leq n$$

e

$$n \leq 2^{h+1} - 1.$$

Assim, usando logaritmos de ambos os lados dessas desigualdades, vê-se que

$$h \leq \log n$$

e

$$\log(n+1) - 1 \leq h.$$

Desde que  $h$  é um número inteiro, as duas desigualdades acima implicam que

$$h = \lfloor \log n \rfloor.$$

A Proposição 8.5 tem uma consequência importante, pois ela indica que se operações de atualização no heap forem realizadas em um tempo proporcional à sua altura, então essas operações serão feitas em tempo logarítmico. O problema agora é, portanto, como efetuar eficientemente vários métodos de uma fila de prioridade usando um heap.

### 8.3.2 Árvores binárias completas e suas representações

A seguir, será discutido mais sobre árvores binárias completas e como representá-las.

#### O TAD árvore binária completa

Como um tipo de dado abstrato, uma árvore binária completa  $T$  implementa todos os métodos de um TAD árvore binária (Seção 7.3.1), adicionando os dois seguintes métodos:

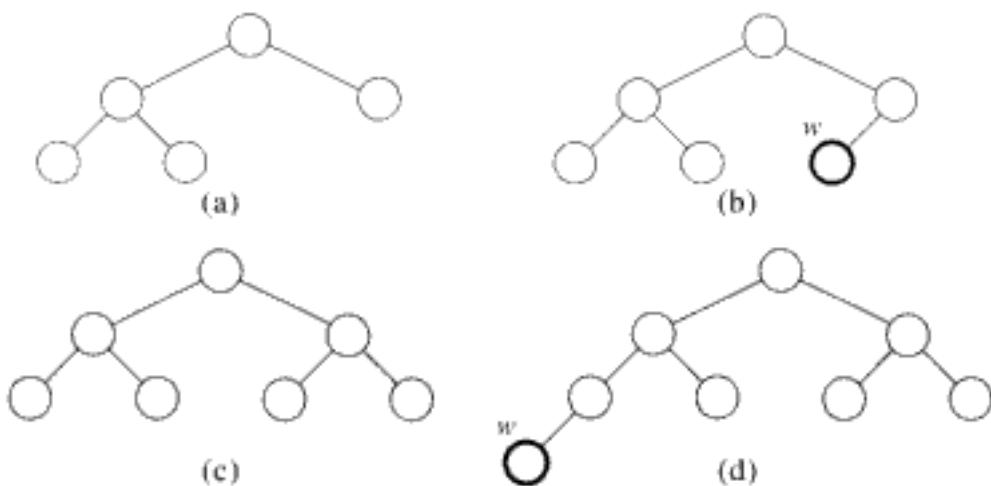
**add( $o$ )**: adiciona em  $T$  e retorna um novo nó externo  $v$  armazenando o elemento  $o$ , no qual a árvore resultante é uma árvore binária completa com o último nó sendo  $v$ .

**remove()**: Remove o último nó de  $T$  retornando-o.

Usando somente estas operações de atualização, sempre se terá uma árvore binária completa, como apresentado na Figura 8.4, na qual há dois casos para a realização de uma adição ou remoção. Especificamente, para uma adição, tem-se o seguinte (remoção é similar).

- Se o nível inferior de  $T$  não estiver completo, então **add** insere um novo nó no nível inferior de  $T$ , imediatamente após o nó mais à direita deste nível (isto é, o último nó); então, a altura de  $T$  continua a mesma.

- Se o nível inferior estiver cheio, então add insere um novo nodo como um filho à esquerda do nodo mais a esquerda do nível inferior de  $T$ ; então, a altura de  $T$  incrementa em um.



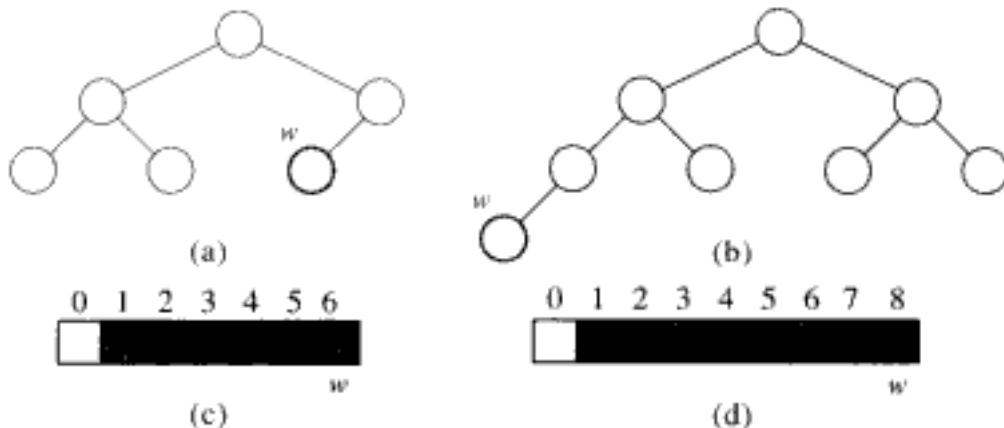
**Figura 8.4** Exemplos das operações `add` e `remove` em uma árvore binária completa, onde  $w$  indica o nodo inserido pelo método `add` ou removido pelo método `remove`. As árvores apresentadas em (b) e (d) são resultados da execução do método `add` nas árvores apresentadas em (a) e (c), respectivamente. Da mesma forma, as árvores apresentadas em (a) e (c) são resultados da execução do método `remove` nas árvores apresentadas em (b) e (d), respectivamente.

A representação de arranjo de uma árvore binária completa

A representação da árvore binária como arranjo (Seção 7.3.5) é especialmente apropriada para uma árvore binária completa  $T$ . Já foi visto que nesta implementação, os nodos de  $T$  são armazenados em um arranjo  $A$  no qual o nodo  $v$  de  $T$  é o elemento de  $A$  com índice igual ao nível  $p(v)$  de  $v$ , definido como segue:

- Se  $v$  é a raiz de  $T$ , então  $p(v) = 1$ .
- Se  $v$  é o nodo filho a esquerda do nodo  $u$ , então  $p(v) = 2p(u)$ .
- Se  $v$  é o nodo filho a direita do nodo  $u$ , então  $p(v) = 2p(u) + 1$ .

Com esta implementação, os nodos de  $T$  têm índices contínuos no intervalo  $[1, n]$  e o último nodo de  $T$  tem sempre o índice  $n$ , onde  $n$  é o número de nodos de  $T$ . A Figura 8.5 apresenta dois exemplos ilustrando esta propriedade do último nodo.



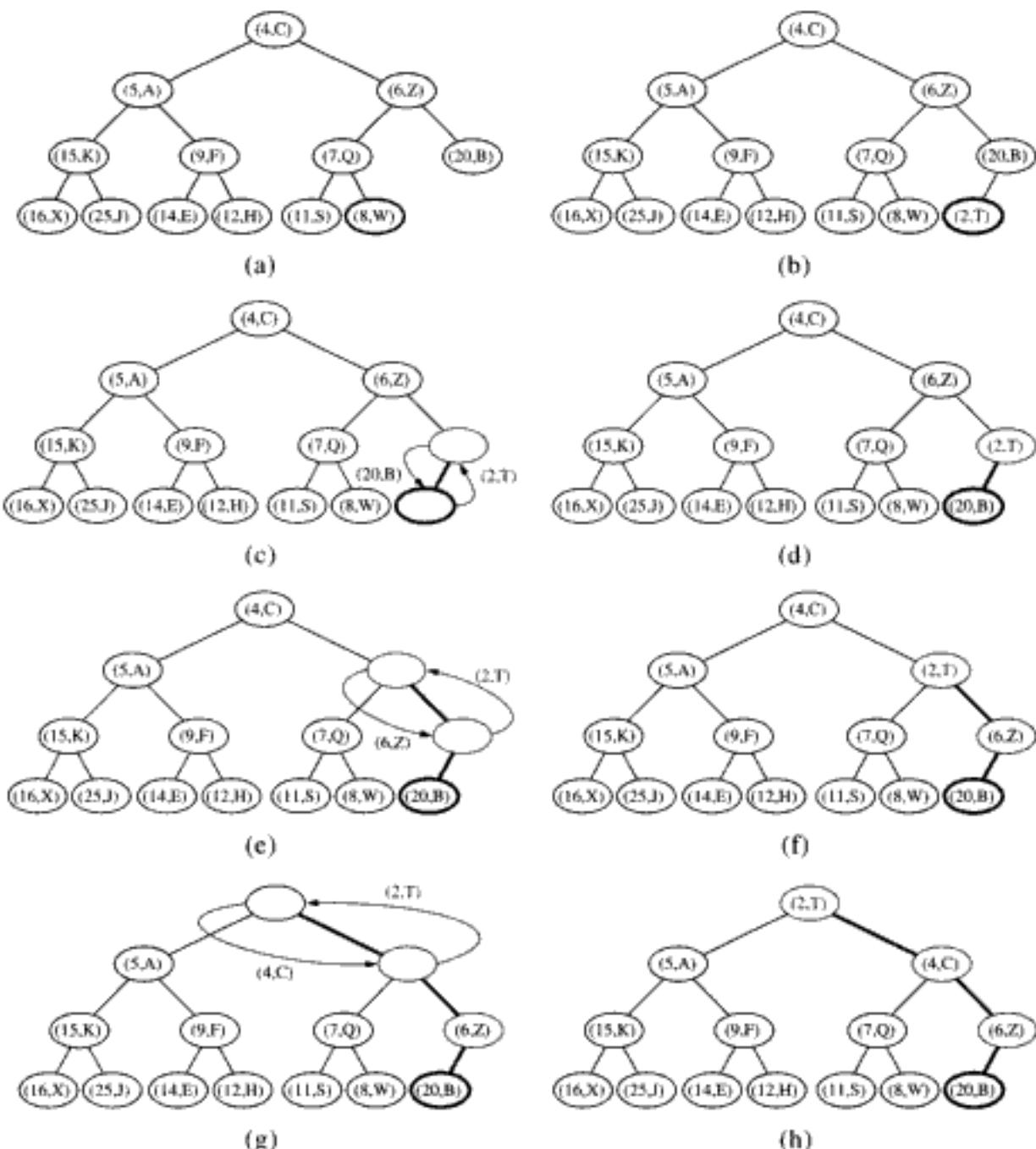
**Figura 8.5** Dois exemplos apresentando que o último nodo  $w$  do heap com  $n$  nodos tem  $n$  níveis: (a) heap  $T_1$  com mais de um nodo no nível inferior; (b) heap  $T_2$  com um nodo no nível inferior; (c) representação de arranjo de  $T_1$ ; (d) representação de arranjo de  $T_2$ .

Hidden page

Hidden page

Hidden page

Hidden page



**Figura 8.7** Inserção de um novo elemento com chave 2 no heap da Figura 8.6: (a) heap inicial; (b) após execução do método `add`; (c e d) troca local, restaurando parcialmente a propriedade de ordem; (e e f) outra troca; (g e h) troca final.

- Se  $r$  ao possuir filho a direita, então  $s$  será filho a esquerda de  $r$ .
- De outra forma ( $r$  tem ambos os filhos),  $s$  será um filho de  $r$  com a menor chave.

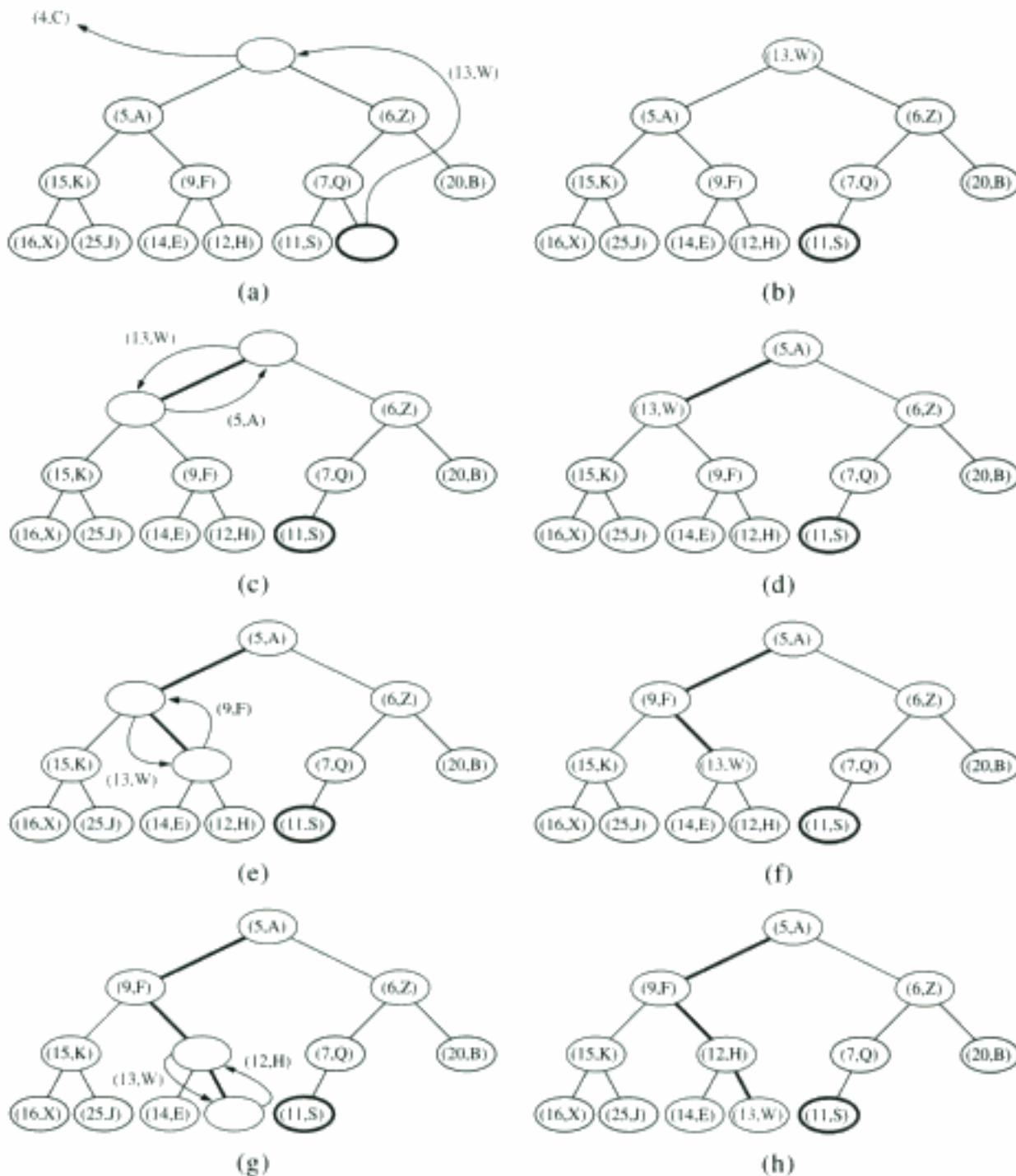
Se  $k(r) \leq k(s)$ , a propriedade de ordem do heap está satisfeita e o algoritmo finalizado. Caso contrário, se  $k(r) > k(s)$ , será preciso restaurar a propriedade de ordem do heap, o que pode ser feito localmente pela troca de elementos armazenados em  $r$  e  $s$ . (Ver Figura 8.8c e d.) (Não seria preciso trocar  $r$  com irmãos de  $s$ .) A troca restaura a propriedade ordem do heap para o nodo  $r$  e seus filhos, mas pode violar esta propriedade em  $s$ ; portanto, pode ser necessário continuar fazendo trocas em  $T$  até que não aconteça mais uma violação. (Ver Figura 8.8e e h.)

Essas trocas descendentes são chamadas de *down-heap bubbling*. Uma troca resolve a violação da propriedade ou a propaga um nível para baixo no heap. No pior caso, uma par cheve-

elemento move-se todo o caminho até o nível imediatamente acima do último. (Ver Figura 8.8.) Assim, o número de trocas executadas na execução do método `removeMin` é, no pior caso, igual a altura do heap  $T$ , isto é,  $\lfloor \log n \rfloor$  pela Proposição 8.5.

### Análise

A Tabela 8.3 mostra o tempo de execução dos métodos do TAD fila de prioridade para a implementação baseada em heap, assumindo que duas chaves podem ser comparadas no tempo  $O(1)$  e que o heap  $T$  está implementada ou como um arranjo ou como uma lista encadeada.



**Figura 8.8:** Remoção do elemento com a menor chave do heap: (a e b) remoção do último nodo, que possui o elemento armazenado na raiz; (c e d) troca localmente para restaurar a propriedade de ordem do heap; (e e f) outra troca; (g e h) troca final.

Hidden page

Hidden page

```

        comp.compare(key, key);
    }
    catch(Exception e) {
        throw new InvalidKeyException("Chave Inválida");
    }
}

```

**Trecho de código 8.14** Métodos min, insert e removeMin e alguns métodos auxiliares da classe HeapPriorityQueue. (Continua no Trecho de código 8.15.)

```

/** Executa up-heap bubbling */
protected void upHeap(Position<Entry<K,V>> v) {
    Position<Entry<K,V>> u;
    while (!heap.isRoot(v)) {
        u = heap.parent(v);
        if (comp.compare(u.element().getKey(), v.element().getKey()) <= 0) break;
        swap(u, v);
        v = u;
    }
}

/** Executa down-heap bubbling */
protected void downHeap(Position<Entry<K,V>> r) {
    while (heap.isInternal(r)) {
        Position<Entry<K,V>> s; // a posição do menor filho
        if (!heap.hasRight(r))
            s = heap.left(r);
        else if (comp.compare(heap.left(r).element().getKey(),
                             heap.right(r).element().getKey()) <= 0)
            s = heap.left(r);
        else
            s = heap.right(r);
        if (comp.compare(s.element().getKey(), r.element().getKey()) < 0) {
            swap(r, s);
            r = s;
        }
        else
            break;
    }
}

/** Troca as entradas das duas posições */
protected void swap(Position<Entry<K,V>> x, Position<Entry<K,V>> y) {
    Entry<K,V> temp = x.element();
    heap.replace(x, y.element());
    heap.replace(y, temp);
}

/** Texto de visualização para verificação */
public String toString() {
    return heap.toString();
}

```

**Trecho de código 8.15** Métodos auxiliares restantes da classe HeapPriorityQueue. (Continuação do Trecho de código 8.14.)

### 8.3.5 Heap-sort

Como já observado, construir uma fila de prioridade com um heap traz a vantagem de que todos os métodos do TAD fila de prioridade são executados em tempo logarítmico, ou melhor. Portanto, esta construção é adequada para aplicações em que tempos velozes são exigidos para todos os métodos da fila de prioridade. Desta forma, considera-se novamente o esquema de ordenação PriorityQueueSort, da Seção 8.1.4, que usa uma fila de prioridade  $P$  para ordenar uma seqüência  $S$  com  $n$  elementos.

Durante a primeira fase, a  $i$ -ésima operação `insert` ( $1 \leq i \leq n$ ) leva o tempo  $O(1 + \log i)$ , desde que o heap tenha  $i$  elementos após a operação ser executada. Da mesma forma, durante a segunda fase, a  $j$ -ésima operação `removeMin` ( $1 \leq j \leq n$ ) executa no tempo  $O(1 + \log(n - j + 1))$ , desde que o heap tenha  $n - j + 1$  elementos no momento da execução da operação. Assim, cada fase leva o tempo  $O(n \log n)$ , assim o algoritmo de ordenação da fila de prioridades executa no tempo  $O(n \log n)$  quando um heap é usado para implementar a fila de prioridades. Este algoritmo de ordenação é mais conhecido como **heap-sort**, e seu desempenho é resumido na seguinte proposição.

**Proposição 8.6** *O algoritmo heap-sort ordena uma seqüência  $S$  de  $n$  elementos no tempo  $O(n \log n)$ , assumindo que dois elementos de  $S$  podem ser comparados no tempo  $O(1)$ .*

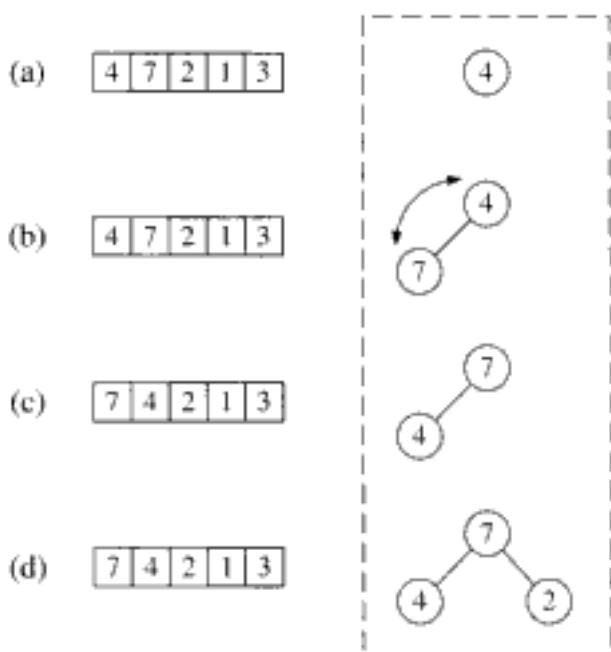
Enfatiza-se que o tempo de execução  $O(n \log n)$  do heap-sort é consideravelmente melhor do que o tempo de execução  $O(n^2)$  do selection-sort e do insertion-sort (Seção 8.2.3).

#### Implementando heap-sort

Se a seqüência  $S$  a ser ordenada é implementada através de um arranjo, pode-se acelerar o heap-sort e reduzir sua necessidade de memória em um fator constante usando uma parte da própria seqüência  $S$  para armazenar o heap, dessa forma evitando o uso de uma estrutura de dados heap externa. Isto é feito modificando-se o algoritmo da forma a seguir:

1. Usa-se um comparador reverso, o que corresponde a um heap com o maior elemento no topo. A qualquer momento durante a execução do algoritmo, utiliza-se a parte esquerda de  $S$  até uma certa posição  $i - 1$  para armazenar os elementos no heap, e a parte direita de  $S$  das posições  $i$  até  $n - 1$  para armazenar os elementos da seqüência. Assim, os primeiros  $i$  elementos de  $S$  (nas posições  $0, \dots, i - 1$ ) fornecem uma representação vetorial para o heap (com numeração iniciando em 0 e não mais em 1), ou seja, o elemento na posição  $k$  do heap é maior ou igual aos “filhos” nas posições  $2k + 1$  e  $2k + 2$ .
2. Na primeira fase do algoritmo, começa-se com um heap vazio e move-se a fronteira entre o heap e a seqüência da esquerda para a direita um passo de cada vez. No passo  $i$  ( $i = 1, \dots, n$ ) expande-se o heap adicionando o elemento na posição  $i - 1$ .
3. Na segunda fase do algoritmo, inicia-se com uma seqüência vazia e move-se a fronteira entre o heap e a seqüência da direita para a esquerda, um passo de cada vez. No passo  $i$  ( $i = 1, \dots, n$ ), o maior elemento do heap é removido e armazenado na posição  $n - i$ .

A variação do heap-sort acima é dita **in-place**, pois ela usa um espaço adicional constante além da própria seqüência. Em vez de retirar elementos da seqüência e depois recolocá-los, eles são simplesmente rearranjados. Esta versão do heap-sort é ilustrada na Figura 8.9. Em geral, um algoritmo de ordenação é in-place se ele usa uma quantidade constante de memória além da memória necessária para armazenar os elementos a serem ordenados.



**Figura 8.9** Os primeiros três passos da primeira fase do heap-sort in-place. A porção do heap da sequência está demarcada em cinza. Desenhamos ao lado do arranjo a árvore binária representando o heap, mesmo que esta árvore não seja realmente construída pelo algoritmo in-place.

### 8.3.6 Construção bottom-up do heap \*

A análise do algoritmo heap-sort mostra que é possível construir um heap armazenando  $n$  pares chave-elemento em tempo  $O(n \log n)$  através de  $n$  operações `insertItem`, e depois usar o heap para retirar os elementos em ordem decrescente. No entanto, se todas as chaves a serem armazenadas no heap forem dadas previamente, existe um método alternativo de construção que monta o heap de baixo para cima (*bottom-up*) em tempo  $O(n)$ . Esse método será descrito nesta seção, observando que ele poderia ser incluído como um dos construtores de uma classe que implementa uma fila de prioridades baseada em heap. Para manter a simplicidade, essa forma de construção será descrita pressupondo que o número  $n$  de chaves é um inteiro da forma  $n = 2^{h+1} - 1$ . Ou seja, o heap é uma árvore binária completa com cada nível completo, portanto tem altura  $h = \log(n + 1) - 1$ . Vista de forma não-recursiva, a construção bottom-up do heap consiste nos seguintes  $h + 1 = \log(n + 1)$  passos:

1. No primeiro passo (ver Figura 8.10a), constrói-se  $(n + 1)/2$  heaps elementares, armazenando uma chave cada um.
  2. No segundo passo (ver Figura 8.10b-c), forma-se  $(n + 1)/4$  heaps armazenando três chaves cada um, simplesmente unindo pares de heaps elementares e adicionando uma nova chave. A nova chave é colocada na raiz e pode precisar ser trocada com a chave colocada em um de seus filhos para preservar a propriedade de ordem do heap.
  3. No terceiro passo (ver Figura 8.10d-e), forma-se  $(n + 1)/8$  heaps armazenando sete chaves cada um, unindo pares de heaps com três chaves (construídas nos passos anteriores) e adicionando uma nova chave. A nova chave é colocada inicialmente na raiz, mas pode ter de passar pelo processo de down-bubbling para preservar a propriedade de ordem do heap.
- ⋮

- i. No  $i$ -ésimo passo genérico, com  $2 \leq i \leq h$ , forma-se  $(n+1)/2^i$  heaps armazenando  $2^i - 1$  chaves cada um, unindo pares de heaps com  $(2^{i-1} - 1)$  chaves (construídas nos passos anteriores) e adicionando uma nova chave. A nova chave é colocada inicialmente na raiz, mas pode ter de passar pelo processo de down-bubbling para preservar a propriedade de ordem do heap.
- ⋮
- $h+1$ . No último passo (ver Figura 8.10f-g), forma-se o heap final, armazenando todos os  $n$  elementos, pela união de dois heaps com  $(n-1)/2$  chaves (construídas nos passos anteriores) e adicionando uma nova chave. A nova chave é colocada inicialmente na raiz, mas pode ter de passar pelo processo de down-bubbling para preservar a propriedade de ordem do heap.

A construção bottom-up do heap é mostrada na Figura 8.10 com  $h = 3$ .

### Construção recursiva bottom-up do heap

Podemos descrever a construção bottom-up do heap de forma recursiva, como apresentado no Trecho de código 8.16, em que é passado por parâmetro uma lista de pares chave-valor que serão utilizados para a criação do heap.

#### Algoritmo BottomUpHeap( $S$ ):

**Entrada:** uma seqüência  $L$  armazenando  $n = 2^{h+1} - 1$  entradas

**Saída:** um heap  $T$  armazenando as entradas em  $L$

**se**  $S.isEmpty()$  **então**

**retorna** um heap vazio

$E \leftarrow L.remove(L.first())$

    Separa  $L$  em duas seqüências  $L_1$  e  $L_2$ , cada uma com o tamanho  $(n-1)/2$

$T_1 \leftarrow \text{BottomUpHeap}(L_1)$

$T_2 \leftarrow \text{BottomUpHeap}(L_2)$

    Cria uma árvore binária  $T$  com raiz  $r$  armazenando  $e$ , tendo a subárvore a esquerda  $T_1$  e a subárvore a direira  $T_2$ .

    Executa um down-heap bubbling a partir da raiz  $r$  de  $T$ , se necessário.

**retorna**  $T$

#### Trecho de código 8.16 Construção recursiva bottom-up do heap.

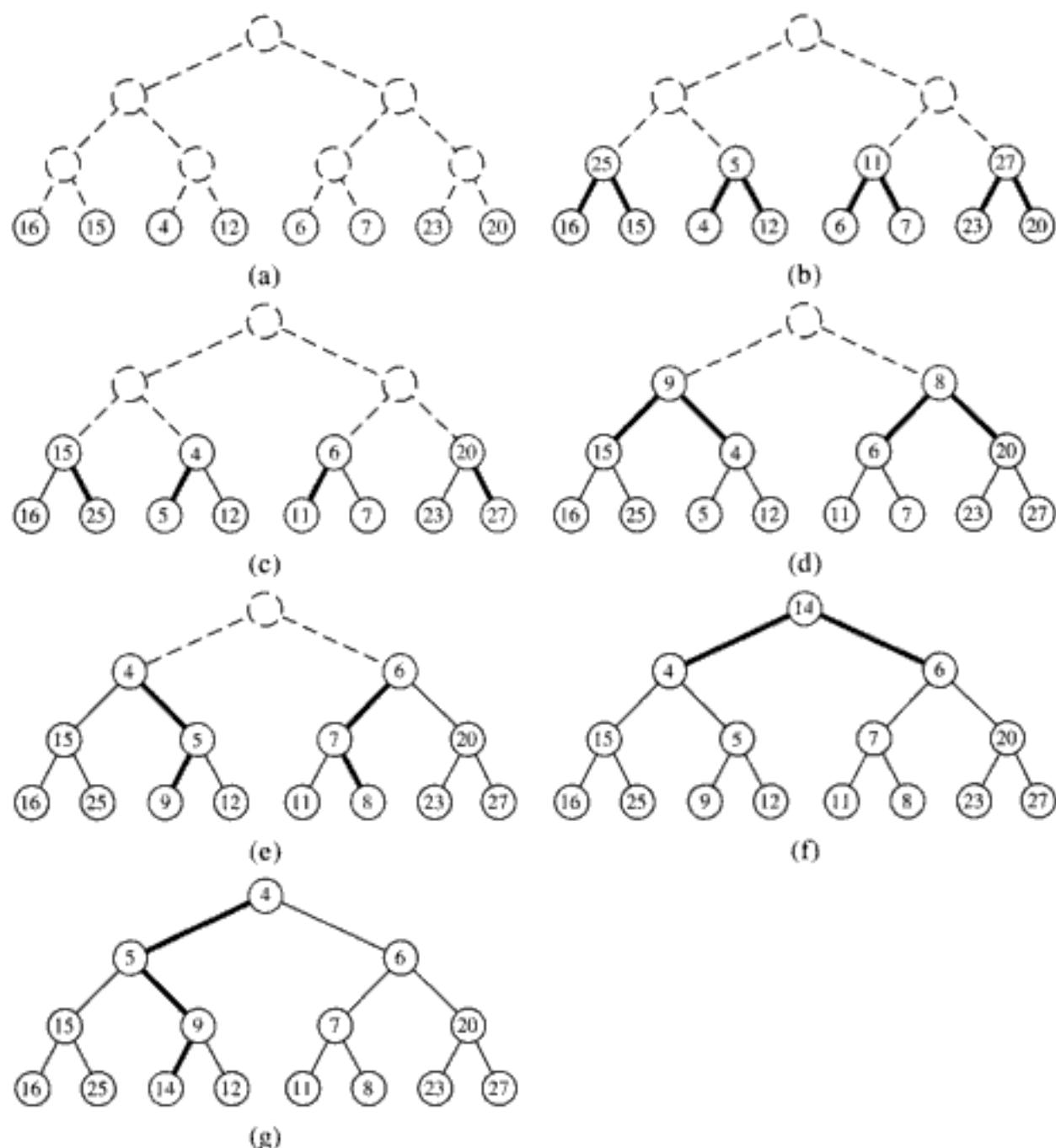
Construção bottom-up do heap é assintoticamente mais rápido que a inserção incremental de  $n$  chaves em um heap inicialmente vazio, como é apresentado na seguinte proposição.

**Proposição 8.7** *Construção bottom-up de um heap com  $n$  elementos custa o tempo  $O(n)$ , assumindo que duas chaves podem ser comparadas no tempo  $O(1)$ .*

**Justificativa** Analisa-se a construção bottom-up do heap usando uma abordagem “visual”, como é ilustrado na Figura 8.11.

Sendo  $T$  o heap final e  $v$  um nodo de  $T$ , denota-se como  $T(v)$  a subárvore de  $T$  com raiz  $v$ . No pior caso, o tempo para formar  $T(v)$  a partir das duas subárvores formadas recursivamente e tendo os filhos de  $v$  em suas raízes é proporcional à altura de  $T(v)$ . O pior caso acontece quando o down-heap bubbling a partir de  $v$  atravessa um caminho de  $v$  até um dos nodos mais externos de  $T(v)$ .

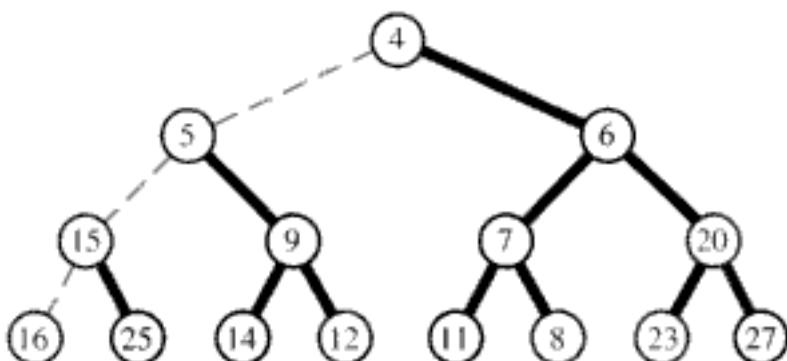
Considere-se agora o caminho  $p(v)$  em  $T$ , do nodo  $v$  até seu sucessor externo (em um caminhamento prefixado), ou seja, o caminho que inicia em  $v$ , visita o filho direito de  $v$  e desce sempre para a esquerda até chegar a um nodo externo. Diz-se que o caminho  $p(v)$  é **associado com** o nodo  $v$ . Observa-se que  $p(v)$  não é necessariamente o caminho seguido em um down-heap



**Figura 8.10** Construção bottom-up de um heap com 15 chaves: (a) incia-se pela construção da chave 1 no nível inferior; (b e c) combinam-se estes heaps em 3 chaves e então (d e e) 7 chaves, até (f e g) que se cria o heap final. Estes caminhos do down-heap bubbling estão demarcados em cinza. Para simplificação, está-se mostrando somente a chave de cada nodo ao invés de todo o elemento.

bubbling quando  $T(v)$  é formado. Claramente, o comprimento (número de arestas) de  $p(v)$  é igual à altura de  $T(v)$ . Portanto, formar  $T(v)$  toma no pior caso tempo proporcional ao comprimento de  $p(v)$ . Assim, o tempo de execução total da construção bottom-up é proporcional à soma dos comprimentos dos caminhos associados aos nodos internos de  $T$ .

Observa-se que cada nodo  $v$  de  $T$  distinto a partir da raiz pertence exatamente a dois caminhos: o caminho  $p(v)$  associado com  $v$  e o caminho  $p(u)$  associado com os pais de  $v$  de  $v$ . (Ver Figura 8.11.) Além disso, a raiz  $r$  de  $T$  pertence somente ao caminho  $p(r)$  associado com  $r$ . Portanto, a soma dos comprimentos dos caminhos associados aos nodos internos de  $T$  é  $2n - 1$ . Conclui-se que a construção bottom-up do heap  $T$  leva o tempo  $O(n)$ . ■



**Figura 8.11** Justificativa visual do tempo de execução linear do algoritmo de construção bottom-up do heap, onde os caminhos associados aos nodos internos são mostrados com cores alternadas. Por exemplo, o caminho associado com a raiz consiste nos nodos armazenando as chaves 4,6,7 e 11. Além disso, o caminho associado com a filho a direita da raiz consiste dos nodos internos armazenando as chaves 6, 20 e 23.

Resumido, a Proposição 8.7 garante que o tempo de execução da primeira fase do heap-sort pode ser reduzido até  $O(n)$ . Infelizmente, o tempo de execução da segunda fase do heap-sort não pode ser tornado assintoticamente melhor do que  $O(n \log n)$ , ou seja, ele sempre será  $\Omega(n \log n)$  no pior caso. Este limite inferior não será justificado até o Capítulo 11. Conclui-se este capítulo discutindo um padrão de projeto que permite estender o TAD fila de prioridade dando-lhe funcionalidade adicional.

## 8.4 Filas de prioridade adaptáveis

Os métodos do TAD fila de prioridades apresentados na Seção 8.1.3 são suficientes para as aplicações mais básicas de filas de prioridades, como um armazenamento. Entretanto, existem situações em que métodos adicionais seriam úteis, como apresentados nos cenários abaixo, o qual refere-se à aplicação de fila de espera de passageiros para uma empresa de vôos comerciais.

- Um passageiro pessimista sobre suas chances de ir a bordo pode decidir ir embora antes da entrada no avião, requisitando ser removido da lista de espera. Assim, se gostaria de remover da fila de prioridades a entrada associada a este passageiro. O método `removeMin` não é aplicado nesta situação desde que o passageiro a ser removido tenha a prioridade 1. Em vez disso, se gostaria de ter um novo método `remove(e)` que remove um elemento qualquer  $e$ .
- Outro passageiro procura seu cartão VIP e apresenta-o ao agente. Assim, sua prioridade tem que ser modificada conforme sua nova especificação. Para conseguir isso, se gostaria de ter um novo método chamado `replaceKey(e, k)`, que substitui com a chave  $k$  a entrada  $e$  na fila de prioridades.
- Finalmente, um terceiro passageiro notifica que seu nome indica que seu nome está escrito de forma errada no cartão de embarque solicitando a alteração. Para esta alteração, precisase alterar o registro do passageiro. Portanto, se gostaria de ter um novo método chamado `replaceValue(e, x)` que substitui com  $x$  o valor da entrada  $e$  na fila de prioridades.

### 8.4.1 Métodos do TAD fila de prioridade adaptável

Os cenários apresentados anteriormente motivam para a definição de um novo TAD que estende o TAD fila de prioridades com os métodos `remove`, `replaceKey` e `replaceValue`. Em outras pa-

Hidden page

(por exemplo, por causa das trocas em um down-heap ou up-heap bubbling). O método `replaceValue(e, x)` custa o tempo  $O(1)$  desde que se obtenha a posição  $p$  da entrada  $e$  no tempo  $O(1)$  seguindo a localização das referências armazenadas com a entrada. Métodos `remove(e)` e `replaceKey(e, k)` executam no tempo  $O(\log n)$  (detalhes são explorados no Exercício C-8.22). Usar localizador aumenta o tempo de execução dos métodos `insert` e `removeMin` em um fator constante elevado.

O uso do localizador para uma implementação de seqüência não ordenada é explorado no Exercício C-8.21.

### Desempenho das implementações de filas de prioridades adaptáveis

O desempenho de uma fila de prioridades adaptável implementada por várias estruturas de dados com localizador é resumido na Tabela 8.4.

Método	Seqüência não-ordenada	Seqüência ordenada	Heap
<code>size, isEmpty</code>	$O(1)$	$O(1)$	$O(1)$
<code>insert</code>	$O(1)$	$O(n)$	$O(\log n)$
<code>min</code>	$O(n)$	$O(1)$	$O(1)$
<code>removeMin</code>	$O(n)$	$O(1)$	$O(\log n)$
<code>remove</code>	$O(1)$	$O(1)$	$O(\log n)$
<code>replaceKey</code>	$O(1)$	$O(n)$	$O(\log n)$
<code>replaceValue</code>	$O(1)$	$O(1)$	$O(1)$

**Tabela 8.4** Tempos de execução dos métodos de uma fila de prioridades adaptável de tamanho  $n$ , implementada com uma seqüência não-ordenada, uma seqüência ordenada e um heap, respectivamente. O espaço requerido é  $O(n)$ .

#### 8.4.3 Implementando uma fila de prioridade adaptável

Os Trechos de código 8.17 e 8.18 apresentam a implementação Java de uma fila de prioridades adaptável baseada em uma seqüência ordenada. Esta implementação é obtida pela extensão da classe `SortedListPriorityQueue` apresentada no Trecho de código 8.6. Em particular, o Trecho de código 8.18 apresenta como implementar um localizador em Java, estendendo uma entrada regular.

```
/** Implementação de uma fila de prioridades adaptável com uma sequencia ordenada. */
public class SortedListAdaptablePriorityQueue<K,V>
    extends SortedListPriorityQueue<K,V>
    implements AdaptablePriorityQueue<K,V> {
    /** Cria uma fila de prioridades com o comparador padrão */
    public SortedListAdaptablePriorityQueue() {
        super();
    }
    /** Cria uma fila de prioridades com um dado comparador */
    public SortedListAdaptablePriorityQueue(Comparator<K> comp) {
        super(comp);
    }
    /** Insere um par chave-valor e retorna a entrada criada */
    public Entry<K,V> insert (K k, V v) throws InvalidKeyException {
```

```

checkKey(k);
LocationAwareEntry<K,V> entry = new LocationAwareEntry<K,V>(k,v);
insertEntry(entry);
entry.setLocation(actionPos);      // posição da nova entrada
return entry;
}
/** Remove e retorna uma dada entrada */
public Entry<K,V> remove(Entry<K,V> entry) {
    checkEntry(entry);
    LocationAwareEntry<K,V> e = (LocationAwareEntry<K,V>) entry;
    Position<Entry<K,V>> p = e.location();
    entries.remove(p);
    e.setLocation(null);
    return e;
}
/** Substitui a chave de uma dada entrada */
public K replaceKey(Entry<K,V> entry, K k) {
    checkKey(k);
    checkEntry(entry);
    LocationAwareEntry<K,V> e = (LocationAwareEntry<K,V>) remove(entry);
    K oldKey = e.setKey(k);
    insertEntry(e);
    e.setLocation(actionPos);      // posição da nova entrada
    return oldKey;
}

```

**Trecho de código 8.17** Implementação Java de uma fila de prioridades adaptável utilizando uma seqüência ordenada armazenando localizadores. A classe `SortedAdaptablePriorityQueue` estende a classe `SortedPriorityQueue` (Trecho de código 8.6) e implementa a interface `AdaptablePriorityQueue`. (Continua no Trecho de código 8.18.)

```

/** Substitui o valor de uma dada entrada */
public V replaceValue(Entry<K,V> e, V value) {
    checkEntry(e);
    V oldValue = ((LocationAwareEntry<K,V>) e).setValue(value);
    return oldValue;
}
/** Determina se uma dada entrada é válida */
protected void checkEntry(Entry ent) throws InvalidEntryException {
    if(ent == null || !(ent instanceof LocationAwareEntry))
        throw new InvalidEntryException("invalid entry");
}
/** Classe interna para um localizador */
protected static class LocationAwareEntry<K,V>
    extends MyEntry<K,V> implements Entry<K,V> {
    /** Posição onde a entrada será armazenada. */
    private Position<Entry<K,V>> loc;
    public LocationAwareEntry(K key, V value) {
        super(key, value);
    }
    public LocationAwareEntry(K key, V value, Position<Entry<K,V>> pos) {
        super(key, value);
        loc = pos;
    }
}

```

```
protected Position<Entry<K,V>> location() {
    return loc;
}
protected Position<Entry<K,V>> setLocation(Position<Entry<K,V>> pos) {
    Position<Entry<K,V>> oldPosition = location();
    loc = pos;
    return oldPosition;
}
protected K setKey(K key) {
    K oldKey = getKey();
    k = key;
    return oldKey;
}
protected V setValue(V value) {
    V oldValue = getValue();
    v = value;
    return oldValue;
}
```

**Trecho de código 8.18** Uma fila de prioridades adaptável implementada com uma seqüência ordenada armazenando localizadores. (Continuação do Trecho de código 8.17.) A classe aninhada LocationAwareEntry implementa um localizador e estende a classe aninhada MyEntry da SortedListPriorityQueue apresentada no Trecho de código 8.6.

---

## 8.5 Exercícios

Para obter o código fonte e auxflio com os exercícios, visite [java.datastructures.net](http://java.datastructures.net).

---

### Reforço

- R-8.1 Suponha que você chame cada nodo  $v$  da árvore binária  $T$  com uma chave igual ao nível anterior de  $v$ . Sobre que circunstância  $T$  é um heap?
- R-8.2 Qual é a saída da seguinte seqüência de métodos do TAD fila de prioridades: insert(5,A), insert(4,B), insert(7,I), insert(1,D), removeMin(), insert(3,J), insert(6,L), removeMin(), removeMin(), insert(8,G), removeMin(), insert(2,H), removeMin(), removeMin()?
- R-8.3 Um aeroporto está desenvolvendo uma simulação de controle de tráfego aéreo que trata eventos como decolagens e poucos. Cada evento tem um time-stamp que registra a hora em que o evento acontece. O programa de simulação deve realizar eficientemente as duas operações fundamentais a seguir:
  - inserir um evento com um dado time-stamp (ou seja, inserir um evento futuro);
  - extrair o evento com menor time-stamp (ou seja, determinar o próximo evento a processar);
  - que estrutura de dados você usaria para suportar estas operações? Justifique sua resposta.
- R-8.4 Embora seja correto usar um comparador “reverso” com o TAD fila de prioridade para recuperar e remover elementos com a maior chave a cada

operação, é um pouco confuso que um elemento com a maior chave seja retornado por um método chamado "removeMin". Escreva uma pequena classe adaptadora que recebe uma fila de prioridade  $P$  e um comparador associado  $C$ , e implementa uma fila de prioridade que opera com os elementos que têm a maior chave através de métodos com nomes como `removeMax`.

- R-8.5 Ilustre a execução do algoritmo selection-sort sobre os seguintes dados de entrada: {22, 15, 36, 44, 10, 3, 9, 13, 29, 25}.
- R-8.6 Ilustre a execução do algoritmo insertion-sort sobre os dados do exercício anterior.
- R-8.7 Forneça um exemplo de seqüência de pior caso com  $n$  elementos para o insertion-sort e mostre como ele é executado em tempo  $\Omega(n^2)$  nesta seqüência.
- R-8.8 Onde pode estar armazenado o elemento com a maior chave em um heap?
- R-8.9 Na definição da relação “a esquerda de” para dois nodos de uma árvore binária (Seção 8.3.1) pode-se usar um caminhamento prefixado em vez de um caminhamento interfixado? E com um caminhamento pós-fixado?
- R-8.10 Ilustre a execução do algoritmo heap-sort sobre os seguintes dados de entrada: {2, 5, 16, 4, 10, 23, 39, 18, 26, 15}.
- R-8.11 Sendo  $T$  uma árvore binária completa em que  $v$  armazena a entrada  $(p(v), 0)$ , onde  $p(v)$  é o número do nível de  $v$ . A árvore  $T$  é um heap? Justifique sua resposta.
- R-8.12 Explique por que não se considera o caso do filho direito de  $r$  ser interno e o filho esquerdo ser externo quando se descreve o processo do down-heap bubbling.
- R-8.13 Existe um heap  $T$  armazenando 7 elementos diferentes de forma que um caminhamento prefixado de  $T$  apresente os elementos de  $T$  em ordem crescente ou decrescente? E se for um caminhamento interfixado? E pós-fixado? Se sim, apresente um exemplo, caso contrário justifique.
- R-8.14 Considere  $H$  um heap que armazena 15 elementos usando uma representação de arranjo de uma árvore binária completa. Qual é a seqüência de índices do arranjo que é visitado no caminhamento prefixado de  $H$ ? E qual é a seqüência em um caminhamento interfixado? E em um caminhamento pós-fixado?
- R-8.15 Mostre que a soma

$$\sum_{i=1}^n \log i,$$

que aparece na análise do heap-sort, é  $\Omega(n \log n)$ .

- R-8.16 Bill afirma que um caminhamento prefixado em um heap não listará as chaves em ordem decrescente. Apresente um exemplo de um heap que prove que ele está errado.
- R-8.17 Hillary afirma que um caminhamento pós-fixado em um heap não listará as chaves em ordem crescente. Apresente um exemplo de um heap que prove que ela está errada.
- R-8.18 Apresente todos os passos do algoritmo para remover a chave 16 do heap da Figura 8.3.

Hidden page

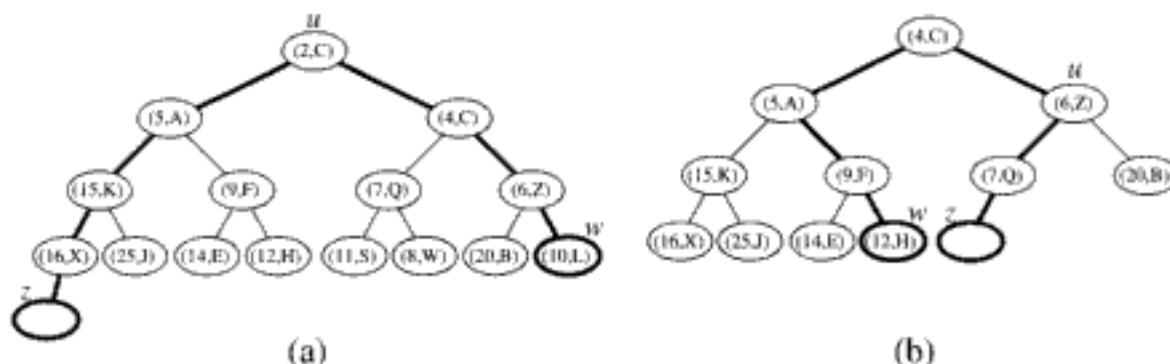
- C-8.8 Assumindo que a entrada para o problema de ordenação é dada em um arranjo  $A$ , apresente como implementar o algoritmo de insertion-sort usando somente o arranjo  $A$  e, no máximo, mais seis variáveis.

C-8.9 Descreva como implementar o algoritmo heap-sort usando, no máximo, seis variáveis inteiras em acréscimo a um arranjo de entrada.

C-8.10 Descreva a seqüência de  $n$  inserções em um heap que requer o tempo  $\Omega(n \log n)$  para processar.

C-8.11 Um método alternativo para encontrar o último nodo durante uma inserção em um heap  $T$  é armazenar no último nodo e em cada nodo externo de  $T$  uma referência para o nodo externo imediatamente à sua direita (“dando a volta” para o primeiro nodo no próximo nível no caso do nodo mais à direita). Mostre como manter essas referências em tempo  $O(1)$  por operação do TAD fila de prioridade assumindo que  $T$  é implementado como estrutura encadeada.

C-8.12 Descreva uma implementação completa de uma árvore binária completa  $T$  que utiliza uma estrutura encadeada e referencia o último nodo. Em particular, apresente como alterar a referência do último nodo através das operações `add` e `remove` no tempo  $O(\log n)$ , onde  $n$  é o número atual de nodos de  $T$ . Tenha certeza de tratar todos os casos possíveis, como ilustrado na Figura 8.12.



**Figura 8.12** Alteração do último nodo em uma árvore binária completa após a operação add ou remove. O nodo  $w$  é o último nodo antes da operação add ou após a operação remove. O nodo  $z$  é o último nodo após a operação add ou anterior a operação remove.

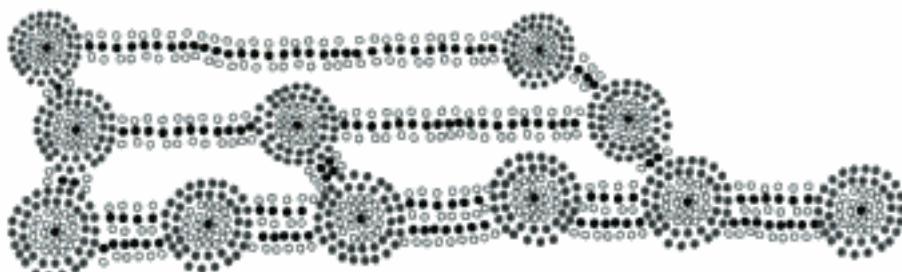
- C-8.13 Representa-se um caminho da raiz até um dado nodo de uma árvore binária através de uma string binária em que 0 significa “siga para o filho à esquerda” e 1 significa “siga para o filho à direita”. Por exemplo, o caminho da raiz até o nodo armazenando (8,W) no heap da Figura 8.12a é representado pela string 101. Proponha um algoritmo de tempo logarítmico para encontrar o último nodo de uma árvore binária completa com  $n$  nodos, baseado na representação apresentada. Mostre como este algoritmo pode ser usado em uma implementação de uma árvore binária completa utilizando uma estrutura encadeada que não mantém a referência para o último nodo.

C-8.14 Dado o heap  $T$  e a chave  $k$ , apresente um algoritmo para computar todas as entradas de  $T$  com chave menor ou igual a chave  $k$ . Por exemplo, dado o heap da Figura 8.12a e o filtro  $k = 7$ , o algoritmo deverá reportar as entradas com as chaves 2, 4, 5, 6 e 7 (mas não necessariamente nesta ordem).

Hidden page

Hidden page

Hidden page

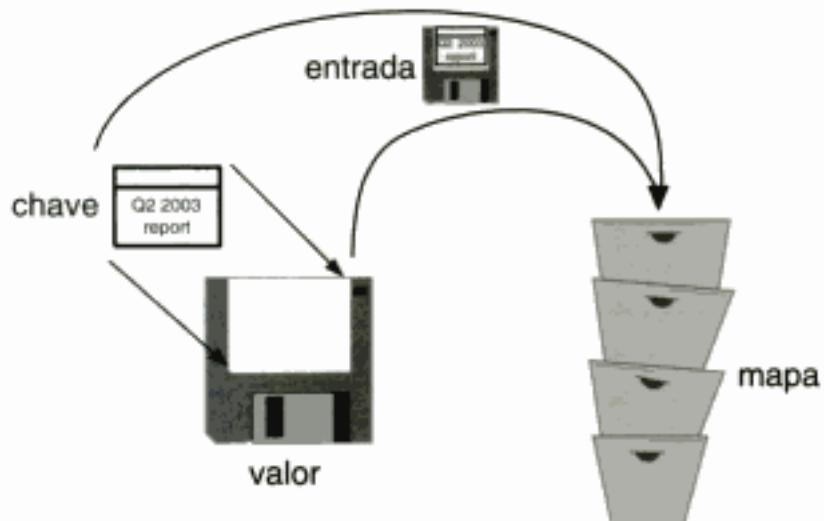


## Conteúdo

<b>9.1</b>	<b>O tipo abstrato de dados mapa . . . . .</b>	<b>332</b>
9.1.1	Uma implementação simples de mapa . . . . .	334
<b>9.2</b>	<b>Tabelas de hash . . . . .</b>	<b>335</b>
9.2.1	Arranjo de buckets . . . . .	335
9.2.2	Funções de hash . . . . .	336
9.2.3	Códigos de hash . . . . .	336
9.2.4	Funções de compressão . . . . .	339
9.2.5	Esquema para tratamento de colisões . . . . .	340
9.2.6	Uma implementação Java para tabelas de hash . . . . .	344
9.2.7	Fatores de carga e rehashing . . . . .	347
9.2.8	Aplicação: contador de freqüência de palavras . . . . .	348
<b>9.3</b>	<b>O TAD dicionário. . . . .</b>	<b>348</b>
9.3.1	Dicionários baseados em seqüências e auditorias . . . . .	350
9.3.2	Implementação de um dicionário com tabela de hash . . . . .	352
9.3.3	Tabelas de pesquisa ordenada e pesquisa binária . . . . .	352
<b>9.4</b>	<b>Skip list . . . . .</b>	<b>356</b>
9.4.1	Pesquisa e alteração em uma skip list . . . . .	357
9.4.2	Uma análise probabilística das skip lists ★ . . . . .	361
<b>9.5</b>	<b>Extensões e aplicações de dicionários . . . . .</b>	<b>363</b>
9.5.1	Suportando localizadores em um dicionário . . . . .	363
9.5.2	O TAD dicionário ordenado . . . . .	364
9.5.3	Banco de dados de vôos e conjuntos máximos . . . . .	364
<b>9.6</b>	<b>Exercícios . . . . .</b>	<b>367</b>

## 9.1 O tipo abstrato de dados mapa

Um *mapa* permite armazenar elementos que podem ser localizados rapidamente usando chaves. A motivação para cada pesquisa é que cada elemento armazena informações adicionais que são úteis junto com a chave, mas somente pode ser acessada através da chave. Especificamente, um mapa armazena um par chave-valor ( $k, v$ ), chamado de *entradas*, onde  $k$  é a chave e  $v$  é o valor correspondente. Além disso, o TAD mapa requer que cada chave seja única, e a associação da chave com o valor define um mapeamento. Para conseguir o maior nível de generalização, permite-se que as chaves e os valores possam armazenar qualquer tipo de objeto. (Ver Figura 9.1.) Em um mapa que armazena registro de estudantes (como o nome do estudante, endereço e suas notas), a chave pode ser o número do identificador (ID) do estudante. Em algumas aplicações, a chave e o valor podem ser o mesmo. Por exemplo, possuindo-se um mapa que armazena números primos cada número poderia ser usado como chave e como valor.



**Figura 9.1** Uma ilustração conceitual do TAD mapa. As chaves (rótulos) são definidas para valores (disquetes) por um usuário. As entradas resultantes (disquetes com rótulos) são inseridas em um mapa (fichário). As chaves podem ser usadas, mais tarde, para reaver ou remover os valores.

Em ambos os casos, usa-se a *chave* como um identificador único que é definido por uma aplicação ou usuário para um objeto valor associado. Assim, um mapa é mais apropriado em situações em que cada chave é para ser vista como um *índice* único para seu valor, ou seja, um objeto que serve como um tipo de localização para um determinado valor. Por exemplo, para armazenar informações de estudantes, provavelmente se precisaria usar o ID do estudante como chave (e não permitir que dois estudantes tenham o mesmo identificador). Em outras palavras, a chave associada com um objeto pode ser vista como um “endereço” para um objeto. Certamente, mapas são algumas vezes referidos como um *armazenamento associativo*, porque a chave associada com um determinado objeto determina sua “localização” na estrutura de dados.

### O TAD mapa

Visto que um mapa armazena uma coleção de objetos, ele deve ser visto como uma coleção de pares chave-valor. Como um TAD, um *mapa M* suporta os seguintes métodos:

- size( ):** Retorna o número de entradas de  $M$ ;
- isEmpty( ):** Testa se  $M$  está vazio;
- get( $k$ ):** Se  $M$  contém uma entrada  $e$  com chave igual a  $k$ , então retorna o valor de  $e$ , senão retorna **null**.

- `put(k,v):` se *M* não tem uma entrada com chave igual a *k*, então adiciona a entrada (*k,v*) em *M* retorna `null`; senão, substitui com *v* o valor existente na entrada com chave *k* e retorna o valor antigo;
- `remove(k):` remove a entrada de *M* com chave igual a *k*, e retorna seu valor; se *M* não possui a entrada com chave *k*, então retorna `null`;
- `keys( ):` retorna uma coleção contendo todas as chaves armazenadas em *M* (`keys( ).iterator( )` retorna um iterator das chaves);
- `values( ):` retorna uma coleção contendo todos os valores associados com as chaves armazenadas em *M* (`values( ).iterator( )` retorna um iterator dos valores);
- `entries( ):` retorna uma coleção contendo todas as entradas (chave-valor) de *M* (`entries( ).iterator( )` retorna um iterator das entradas).

Quando os métodos `get(k)`, `put(k,v)` e `remove(k)` são executados em um mapa *M* que não possui entrada com chave igual a *k*, usa-se a convenção de retornar `null`. Um valor especial como este é conhecido como **sentinela** (veja Seção 3.3). A desvantagem do uso do `null` como sentinelas é que esta escolha pode criar ambigüidade em que precisaria-se de uma entrada (*k, null*) com o valor `null` no mapa. Claro que outra escolha seria lançar uma exceção quando alguém solicita uma chave que não está no nosso mapa. Isso provavelmente não seria um uso apropriado de uma exceção. Entretanto, é normal questionar por algo que pode não estar no mapa. Além disso, lançar e capturar uma exceção é tipicamente mais lento que um teste de um sentinelas; portanto, o uso do sentinelas é mais eficiente (e, neste caso, conceitualmente mais apropriado). `null` é usado como um sentinelas para um valor associado com uma chave não existente.

**Exemplo 9.1** Na seguinte tabela, mostra-se o efeito de uma série de operações em um mapa inicialmente vazio que armazena chaves inteiros e valores com um único caractere.

Operação	Saída	Mapa
<code>isEmpty()</code>	<code>true</code>	$\emptyset$
<code>put(5, A)</code>	<code>null</code>	$\{(5, A)\}$
<code>put(7, B)</code>	<code>null</code>	$\{(5, A), (7, B)\}$
<code>put(2, C)</code>	<code>null</code>	$\{(5, A), (7, B), (2, C)\}$
<code>put(8, D)</code>	<code>null</code>	$\{(5, A), (7, B), (2, C), (8, D)\}$
<code>put(2, E)</code>	<code>C</code>	$\{(5, A), (7, B), (2, E), (8, D)\}$
<code>get(7)</code>	<code>B</code>	$\{(5, A), (7, B), (2, E), (8, D)\}$
<code>get(4)</code>	<code>null</code>	$\{(5, A), (7, B), (2, E), (8, D)\}$
<code>get(2)</code>	<code>E</code>	$\{(5, A), (7, B), (2, E), (8, D)\}$
<code>size()</code>	4	$\{(5, A), (7, B), (2, E), (8, D)\}$
<code>remove(5)</code>	<code>A</code>	$\{(7, B), (2, E), (8, D)\}$
<code>remove(2)</code>	<code>E</code>	$\{(7, B), (8, D)\}$
<code>get(2)</code>	<code>null</code>	$\{(7, B), (8, D)\}$
<code>isEmpty()</code>	<code>false</code>	$\{(7, B), (8, D)\}$

## Mapas no pacote java.util

O pacote Java `java.util` inclui uma interface para o TAD mapa, o qual é chamada `java.util.Map`. Esta interface é definida para que uma implementação de uma classe force chaves únicas e inclua todos os métodos de um TAD mapa apresentados abaixo, exceto para alguns casos com

o uso de nomes diferentes. A correspondência entre o TAD mapa e a interface `java.util.Map` é apresentada na Tabela 9.1.

Métodos do TAD mapa	Métodos da interface <code>java.util.Map</code>
<code>size()</code>	<code>size()</code>
<code>isEmpty()</code>	<code>isEmpty()</code>
<code>get(<i>k</i>)</code>	<code>get(<i>k</i>)</code>
<code>put(<i>k,v</i>)</code>	<code>put(<i>k,v</i>)</code>
<code>remove(<i>k</i>)</code>	<code>remove(<i>k</i>)</code>
<code>keys()</code>	<code>keySet()</code>
<code>values()</code>	<code>values()</code>
<code>entries()</code>	<code>entrySet()</code>

**Tabela 9.1** Correspondência entre os métodos do TAD mapa e os métodos da interface `java.util.Map`, o qual suporta também outros métodos.

### 9.1.1 Uma implementação simples de mapa

Uma simples forma de implementar um mapa é armazenar suas  $n$  entradas em uma seqüência  $S$ , implementada como uma lista duplamente encadeada. A execução dos métodos fundamentais, `get(k)`, `put(k,v)` e `remove(k)`, envolve busca simples sobre  $S$  procurando por uma entrada com chave  $k$ . Apresenta-se o pseudo-código da execução destes métodos em um mapa  $M$  no Trecho de código 9.1.

Esta implementação do mapa baseada em seqüência é simples, mas ela somente é eficiente para mapas realmente pequenos. Cada um dos métodos fundamentais leva o tempo  $O(1)$  em um mapa com  $n$  entradas, porque cada método pesquisará, no pior caso, em toda a seqüência. Assim, algo mais rápido seria preferido.

#### Algoritmo `get(k)`:

*Entrada:* uma chave  $k$

*Saída:* O valor para chave  $k$  em  $M$ , ou **nulo** se não existir uma chave  $k$  em  $M$

para cada posição  $p$  em  $S.positions()$  faça

  se  $p.element().getKey() = k$  então

    retorna  $p.element().getValue()$

  retorna **nulo** {Não existe elemento com chave igual a  $k$ }

#### Algoritmo `put(k,v)`:

*Entrada:* um par chave-valor  $(k,v)$

*Saída:* O antigo valor associado com a chave  $k$  em  $M$  ou **nulo** se  $k$  é uma nova chave.

para cada posição em  $S.positions()$  faça

  se  $p.element().getKey() = k$  então

$t \leftarrow p.element().getValue()$

$B.set(p,(k,v))$

    retorna  $t$  {retorna o valor antigo}

$S.addLast((k,v))$

$n \leftarrow n + 1$  {incrementa a variável que armazena o número de elementos}

  retorna **nulo** {Não existia elemento anterior com chave igual a  $k$ }

#### Algoritmo `remove(k)`:

*Entrada:* uma chave  $k$

*Saída:* O valor (removido) para a chave  $k$  em  $M$ , ou **nulo** se  $k$  não estiver em  $M$

**Para** cada posição  $p$  em  $S.positions()$  **faça**

**Se**  $p.element().getKey() = k$  **então**

$t \leftarrow p.element().getValue()$

$S.remove(p)$

$n \leftarrow n - 1$  {decrementa a variável que armazena o número de elementos}

**Retorna**  $t$  {retorna o valor removido}

**Retorna null** {não existe elemento com chave igual a  $k$ }

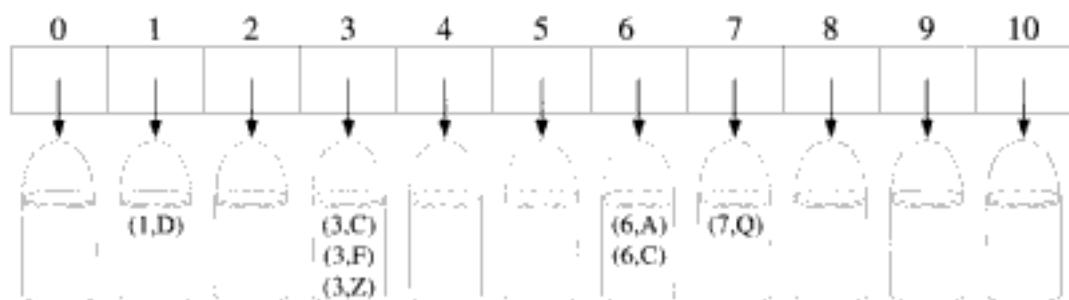
**Trecho de código 9.1** Algoritmos para os métodos fundamentais do mapa com uma seqüência  $S$ .

## 9.2 Tabelas de hash

As chaves associadas com elementos em um dicionário são freqüentemente consideradas como “endereços” dos elementos. Exemplos desse tipo de aplicações são as tabelas de símbolos de um compilador ou as listas de variáveis de ambiente em um sistema operacional. Em ambos os casos, essas estruturas consistem em uma coleção de nomes simbólicos na qual cada nome serve de “endereço” para propriedades sobre o tipo de uma variável ou seu valor. Uma das maneiras mais eficientes de implementar um dicionário em tais circunstâncias é usando uma **tabela de hash**. Embora, como será visto, o tempo de execução de pior caso das operações do TAD dicionário seja  $O(n)$  quando se usa uma tabela de hash, uma tabela dessas pode realizar essas operações em tempo esperado  $O(1)$ . Em geral, uma tabela de hash consiste em dois componentes principais, um **arranjo de buckets** e uma **função de hash**.

### 9.2.1 Arranjo de buckets

Um **arranjo de buckets** para uma tabela de hash é um arranjo  $A$  de tamanho  $N$ , em que cada célula de  $A$  é considerada como um “bucket” (ou seja, um contêiner para pares chave-elemento), e o inteiro  $N$  determina a **capacidade** do arranjo. Se as chaves forem inteiros bem distribuídos no intervalo  $[0, N - 1]$ , esse arranjo de buckets é tudo o que é necessário. Um elemento  $e$  com chave  $k$  é simplesmente inserido no bucket  $A[k]$ . (Ver Figura 9.2.) Para economizar espaço, um arranjo de buckets vazio pode ser substituído por um objeto **null**.



**Figura 9.2** Um arranjo de buckets de tamanho 11 para as entradas (1,D), (3,C), (3, F), (3,Z),(6,A),(6,C) e (7,Q).

Se nossas chaves são inteiros únicos no intervalo  $[0, N - 1]$ , então cada bucket armazenará no máximo uma entrada. Assim, pesquisas, inserções e remoções em um arranjo de buckets levará o tempo  $O(1)$ . Parece ser um grande resultado, mas tem duas desvantagens. Primeiro, o espaço utilizado é proporcional a  $N$ . Dessa forma, se  $N$  é muito maior que o número de entradas  $n$  realmente presentes no dicionário, será um desperdício de espaço. A segunda desvantagem é que é exigido que as chaves sejam inteiros no intervalo  $[0, N - 1]$ , o que freqüentemente não acontece.

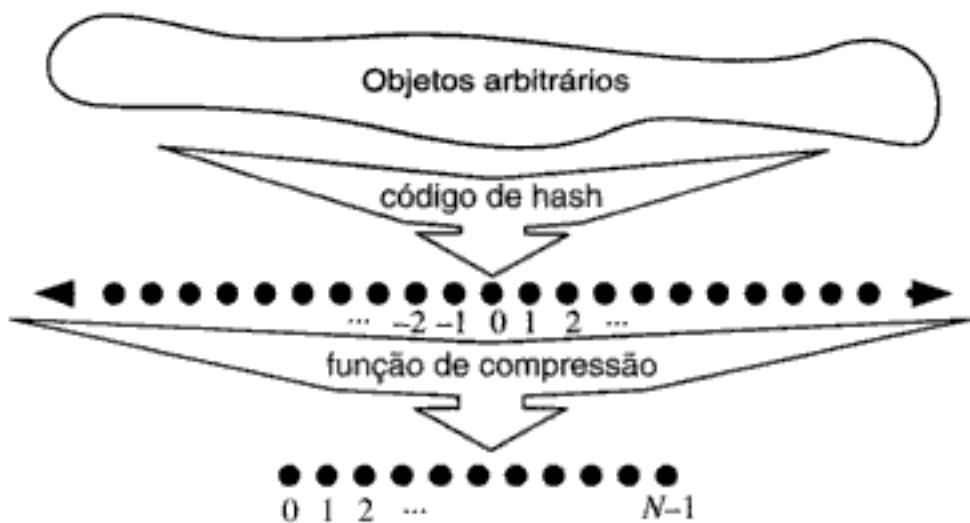
Pelo motivo destas duas desvantagens, usa-se o arranjo de buckets em conjunto com um “bom” mapeamento das chaves para inteiros no intervalo  $[0, N - 1]$ .

### 9.2.2 Funções de hash

A segunda parte de uma tabela de hash é uma função,  $h$ , chamada de *função de hash*, que mapeia cada chave  $k$  em um inteiro no intervalo  $[0, N - 1]$ , onde  $N$  é a capacidade do arranjo de buckets para essa tabela. Com uma função de hash  $h$  deste tipo, pode-se aplicar o método do arranjo de buckets para chaves arbitrárias. A idéia central desta abordagem é usar o valor da função de hash,  $h(k)$ , como um índice no arranjo de buckets  $A$ , em vez da chave  $k$  (que é provavelmente inadequada para uso como índice de um arranjo de buckets). Ou seja, o item  $(k, e)$  é armazenado no bucket  $A[h(k)]$ .

Claro, se existirem duas ou mais chaves com o mesmo valor de hash, então dois diferentes elementos serão mapeados para o mesmo bucket em  $A$ . Neste caso, diz-se que uma *colisão* ocorreu. Claramente, se cada bucket de  $A$  pode armazenar somente um elemento, então não se pode associar mais de um elemento com um simples bucket, o qual é um problema de casos de colisões. Para não se ter dúvidas, existem formas de tratar as colisões, as quais serão discutidas depois, mas a melhor estratégia é tentar evitá-las em um primeiro momento. Diz-se que uma função de hash é “boa” se o mapeamento das chaves no dicionário minimiza colisões o máximo possível. Por razões práticas, se gostaria que uma função de hash seja rápida e fácil de computar.

Seguindo a convenção do Java, visualiza-se a evolução de uma função de hash,  $h(k)$ , consistindo de duas ações – mapeamento da chave  $k$  para um inteiro, chamado de *código do hash*, e o mapeamento do código do hash para um inteiro em um intervalo de índices ( $[0, N - 1]$ ) de um arranjo de buckets chamado *função de compressão*. (Ver Figura 9.3.)



**Figura 9.3** As duas partes de uma função de hash: um código de hash e uma função de compressão.

### 9.2.3 Códigos de hash

A primeira ação que uma função de hash realiza é tomar uma chave arbitrária  $k$  no dicionário e atribuir a ela um valor inteiro. O inteiro associado a uma chave  $k$  é chamado de *código hash* ou *valor de hash* para  $k$ . Este inteiro não precisa estar no intervalo  $[0, N - 1]$  e pode mesmo ser negativo, mas se deseja que o conjunto de códigos hash associados às chaves reduza as colisões tanto quanto possível. Além disso, para ser consistente com todas as chaves, o código hash que se usa para uma chave  $k$  deve ser igual ao código hash de qualquer chave igual a  $k$ .

## Códigos hash em Java

A classe genérica `Object` definida em Java é equipada com um método padrão `hashCode()` para mapear as instâncias de um objeto em um inteiro que é a “representação” do objeto. Especificamente, o método `hashCode()` retorna um inteiro do tipo `int` de 32 bits. A não ser que seja especificamente sobreescrito, esse método é herdado por cada objeto usado em um programa Java. No entanto, deve-se ter cuidado ao usar a versão padrão de `hashCode()`, pois esta pode ser uma interpretação inteira da posição do objeto na memória (como é o caso em muitas implementações em Java). Este tipo de código não funciona bem com cadeias de caracteres, por exemplo, porque duas cadeias de caracteres em locais diferentes da memória poderiam ter o mesmo conteúdo e, neste caso, se desejaria que elas tivessem o mesmo código. De fato, a classe Java `String` fornece outro método `hashCode()`, mais apropriado para cadeias de caracteres. Da mesma forma, desejando-se usar determinados objetos como chaves de um dicionário, deve-se fornecer um método `hashCode()` para esses objetos, fornecendo um mapeamento que associa inteiros bem distribuídos aos objetos.

Serão analisados então vários tipos de dados comuns e alguns exemplos de métodos associando códigos hash a esses tipos de dados.

## Conversão para inteiros

Para iniciar, nota-se que para qualquer tipo de dado  $X$  representado com no máximo tantos bits quanto nosso código hash inteiro, pode-se simplesmente usar como código de hash para  $X$  uma interpretação inteira de seus bits. Assim, para os tipos Java `byte`, `short`, `int` e `char`, pode-se obter um bom código hash simplesmente convertendo este tipo para `int`. Da mesma forma, para uma variável  $x$  do tipo `float`, converte-se  $x$  para um inteiro com uma chamada para `floatToIntBits(x)`, e usa-se este inteiro como código hash para  $x$ .

## Somando componentes

Para tipos como `long` e `double`, cuja representação em bits é duas vezes maior do que um código de hash, a ideia acima não pode ser aplicada diretamente. Ainda assim, um código hash possível e usado por muitas implementações em Java é simplesmente converter a representação de um `long` para um inteiro do tamanho do código de hash. Este código de hash, naturalmente, ignora metade da informação presente no valor original, e se muitas das chaves em nosso dicionário diferem apenas na outra metade, então elas colidirão através deste algoritmo simples. Um código de hash alternativo, que leva todos os bits em consideração, é obtido somando-se a representação inteira dos bits de mais alta ordem e a representação inteira dos bits de mais baixa ordem. Esse código de hash pode ser descrito em Java como segue:

```
static int hashCode(long i) {return (int)(i >> 32) + (int)i;}
```

De fato, a alternativa baseada na soma de componentes pode ser estendida a qualquer objeto  $x$  cuja representação binária possa ser vista como uma  $k$ -tupla  $(x_0, x_1, \dots, x_{k-1})$  de inteiros, pois pode-se formar um código de hash para  $x$  como  $\sum_{i=0}^{k-1} x_i$ . Por exemplo, tendo-se um número de ponto flutuante, soma-se sua mantissa e seu expoente como inteiros longos e então aplicar um código hash para inteiros longos para o resultado.

## Códigos hash polinomiais

O código hash baseado em somas descrito acima não é uma boa escolha para cadeias de caracteres ou outros objetos longos que podem ser vistos como tuplas da forma  $(x_0, x_1, \dots, x_{k-1})$ , onde a ordem dos elementos  $x_i$  é relevante. Por exemplo, considere um código hash para uma cadeia

de caracteres  $s$  que soma os valores ASCII (ou Unicode) dos caracteres em  $s$ . Este código hash infelizmente produz muitas colisões indesejáveis para cadeias de caracteres bastante comuns. Em particular, "temp01" e "temp10" colidem com esta função, e também colidem as palavras "stop", "pots", "spot" e "tops". Um código de hash melhor deveria levar em conta a posição dos elementos  $x_i$ . Uma alternativa que faz exatamente isso é escolher uma constante  $a > 0$  e diferente de 1, e usar como código de hash o valor dado por

$$x_0a^{k-1} + x_1a^{k-2} + \dots + x_{k-2}a + x_{k-1}.$$

Matematicamente falando, é simplesmente um polinômio em  $a$  que usa os componentes  $(x_0, x_1, \dots, x_{k-1})$  como seus coeficientes. Este código hash é conhecido, portanto como **código hash polinomial**. Pela regra de Horner (veja o Exercício C-4.11), pode ser escrito como:

$$x_{k-1} + a(x_{k-2} + a(x_{k-3} + \dots + a(x_2 + a(x_1 + ax_0)) \dots)).$$

Intuitivamente, um código hash polinomial usa a multiplicação pela constante  $a$  como uma forma de "dar espaço" a cada componente em uma tupla de valores e ainda preserva a caracterização dos componentes anteriores.

Claro que, em um computador típico, a avaliação de um polinômio será feita com precisão finita e periodicamente o valor acumulado irá causar overflow no espaço usado para armazenar um inteiro. Já que se está interessado no espalhamento do código de hash em relação às chaves, pode-se simplesmente ignorar este overflow. Ainda assim, deve-se lembrar que esse tipo de overflow é possível e escolher uma constante  $a$  que tenha alguns bits de baixa ordem diferentes de zero, o que servirá para preservar um pouco da informação mesmo em caso de overflow.

Foram feitos alguns estudos experimentais que sugerem que 33, 37, 39 e 41 são valores particularmente bons para  $a$  quando as cadeias de caracteres a serem armazenadas são palavras da língua inglesa. De fato, em uma lista de mais de 50.000 palavras em inglês, formada por meio da união de listas de palavras fornecidas em duas versões de Unix, constatou-se que escolhendo  $a = 33, 37, 39$  ou  $41$  produz menos de 7 colisões em cada caso! Não deve ser uma surpresa, portanto, descobrir que várias versões de Java escolhem uma função de hash polinomial baseada em uma dessas constantes. Para obter maior velocidade, no entanto, algumas implementações em Java somente aplicam a função de hash polinomial em uma fração dos caracteres de cadeias de caracteres muito longas.

### Códigos hash com shift

Uma variação dos códigos hash polinomiais substitui a multiplicação por  $a$  por um shift do resultado parcial. Uma função assim, aplicada a cadeias de caracteres em Java, poderia ser a seguinte:

```
static int hashCode(String s) {
    int h=0;
    for (int i=0; i<s.length(); i++) {
        h = (h << 5) | (h >>> 27); // shift de cinco bits na soma atual
        h += (int) s.charAt(i); // somar novo caractere
    }
    return h;
}
```

Assim como o código hash polinomial, usar o código hash baseado em shift requer ajustes. Neste caso, deve-se escolher com cuidado a quantidade de bits a deslocar para cada caractere. Mostra-se na Tabela 9.2 os resultados de alguns experimentos em uma lista de pouco mais de 25 mil palavras em inglês, na qual se compara o número de colisões para deslocamentos diferentes. Esses experimentos, bem como os anteriores, mostram que se a constante  $a$  ou o deslocamento

Hidden page

a distribuição dos valores. De fato, se  $N$  não for primo, então existe uma maior probabilidade de que padrões na distribuição das chaves sejam repetidos na distribuição dos códigos de hash, causando colisões. Por exemplo, tendo-se as chaves  $\{200, 205, 210, 215, 220, \dots, 600\}$  em um arranjo de buckets de tamanho 100, então cada código hash irá colidir com três outros. Se esse mesmo conjunto de chaves for colocado em um arranjo de buckets de tamanho 101, no entanto, não haverá colisões. Se uma função de hash for bem escolhida, ela deve garantir que a probabilidade de duas chaves diferentes irem para a mesma posição no arranjo de buckets é de no máximo  $1/N$ . Escolher  $N$  como um número primo, no entanto, não é sempre suficiente, pois se há um padrão repetitivo de chaves com o formato  $pN + q$  para vários valores diferentes de  $p$ , então ainda ocorrerão colisões.

### O método MAD

Uma função de compressão mais sofisticada, que ajuda a eliminar padrões repetitivos em um conjunto de chaves inteiras, é o método de *multiplicação, adição e divisão* (ou “MAD”). Este método mapeia um inteiro  $i$  para

$$|ai + b| \bmod N,$$

onde  $N$  é um número primo, e  $a > 0$  (chamado de *fator de ativação*<sup>\*</sup>) e  $b \geq 0$  (chamado *shift*) são constantes inteiras escolhidas aleatoriamente quando a função de compressão foi determinada, de forma que  $a \bmod N \neq 0$ . Esta função de compressão é escolhida de forma a eliminar padrões repetidos no conjunto de códigos de hash e a conduzir mais perto de uma “boa” função de hash, ou seja, uma função em que a probabilidade de colisão de duas chaves seja no máximo  $1/N$ . Este comportamento seria o mesmo que se teria se as chaves fossem “jogadas” em  $A$  de forma aleatória e uniforme.

Com uma função de compressão como esta, que espalha  $n$  inteiros de forma bastante homogênea no intervalo  $[0, N - 1]$ , e com um mapeamento das chaves em nosso dicionário para os números inteiros, tem-se uma função de hash efetiva. Juntos, uma função assim e um arranjo de buckets definem os componentes essenciais de uma implementação baseada em tabela de hash para o TAD dicionário.

Antes de detalhar de como realizar operações como `put`, `get` e `remove`, deve-se primeiro resolver o problema do tratamento de colisões.

---

#### 9.2.5 Esquema para tratamento de colisões

A idéia principal de uma tabela de hash é tomar um arranjo de buckets  $A$  e uma função de hash  $h$  e usá-los para implementar um dicionário, armazenando cada item  $(k, v)$  em um “bucket”  $A[h(k)]$ . Esta idéia simples é tornada complicada, no entanto, quando se tem duas chaves distintas,  $k_1$  e  $k_2$ , tais que  $h(k_1) = h(k_2)$ . A existência desta *colisão* impede que se faça imediatamente a inserção do novo item  $(k, v)$  na posição  $A[h(k)]$ . Ela também complica as operações `get(k)`, `put(k)` e `remove(k)`.

##### Encadeamento separado

Uma maneira simples e eficiente de lidar com colisões é fazer com que cada posição  $A[i]$  armazene uma referência para um pequeno dicionário,  $M_i$ , implementado utilizando uma seqüência, como descrito na Seção 9.1.1, contendo itens  $(k, v)$  tais que  $h(k) = i$ . Isto é, cada encadeamento separado  $M_i$ , juntamente com os elementos que possuem índice  $i$  em uma lista encadeada. Esta

---

\* N. de R. T. O autor utilizou o termo *scaling factor*.

regra para a **resolução de colisões** é conhecida como **encadeamento separado**. Assumindo que inicializa-se cada “bucket”  $A[i]$  para ser um dicionário baseado em seqüência vazio, pode-se facilmente usar a regra do encadeamento separado para executar as operações fundamentais do mapa, como mostrado no Trecho de código 9.2.

**Algoritmo**  $\text{get}(k)$ :

**Saída:** O valor associado com a chave  $k$  em um mapa, ou **nulo** se não existir elemento com chave igual a  $k$  no mapa.

**retorna**  $A[h(k)].\text{get}(k)$  {delega a busca (get) para o mapa baseado em lista  $A[h(k)]$ }

**Algoritmo**  $\text{put}(k,v)$ :

**Saída:** Se existir um elemento no nosso mapa com chave igual a  $k$ , então se retorna seu valor (alterando ele com  $v$ ); caso contrário retorna-se **nulo**.

$t \leftarrow A[h(k)].\text{put}(k,v)$  {delega a inserção (put) no mapa baseado em lista com  $A[h(k)]$ }

**se**  $t = \text{nulo}$  **então** { $k$  é uma nova chave}

$n \leftarrow n + 1$

**retorna**  $t$

**Algoritmo**  $\text{remove}(k)$ :

**Saída:** O valor (removido) associado com a chave  $k$  no mapa, ou **nulo** se não existir elemento com chave igual a  $k$  no mapa.

$t \leftarrow A[h(k)].\text{remove}(k)$  {delega a remoção (remove) para mapa baseado em lista  $A[h(k)]$ }

**se**  $t \neq \text{nulo}$  **então** { $k$  foi encontrado}

$n \leftarrow n - 1$

**retorna**  $t$

**Trecho de código 9.2** Métodos fundamentais do TAD mapa, implementada com uma tabela de hash que usa encadeamento separado para resolver colisões entre  $n$  elementos.

Para cada uma das operações fundamentais de dicionários envolvendo uma chave  $k$ , delega-se o tratamento desta operação ao dicionário miniatura baseado em seqüência e armazenado em  $A[h(k)]$ . Assim,  $\text{put}(k,v)$  percorrerá esta seqüência procurando por um elemento com chave igual a  $k$ ; se encontrar, substitui o valor existente por  $v$ ; caso contrário, insere  $(k,v)$  no final desta seqüência. Da mesma forma,  $\text{get}(k)$  pesquisará nesta seqüência até chegar ao final da mesma ou encontrar um elemento com chave igual a  $k$ . E o  $\text{remove}(k)$  executará uma pesquisa similar, mas adicionando a remoção de um elemento após encontrá-lo. Pode-se “escapar” com esta simples abordagem baseada em seqüência, porque a propriedade de propagação de uma função de hash ajuda a manter cada pequena seqüência de “buckets”. De fato, uma boa função de hash tenta minimizar colisões tanto quanto possível, o que implica que a maior parte dos buckets estarão vazios ou contendo apenas um elemento. Essa observação permite fazer uma pequena mudança na implementação de forma que se um “bucket”  $A[i]$  está vazio, ele armazenará **null**, e se  $A[i]$  armazena um único elemento  $(k,v)$ , pode-se simplificar tendo  $A[i]$  apontando diretamente para o elemento  $(k,v)$  de preferência para um dicionário baseado em seqüência armazenando somente um elemento. Os detalhes desta otimização de espaço como um exercício (C-9.5). Na Figura 9.4, ilustra-se uma tabela de hash com encadeamento separado.

Assumindo que se está usando uma boa função de hash para colocar nossos  $n$  itens de nosso dicionário em um arranjo de buckets de tamanho  $N$ , espera-se que o número de elementos associados a cada posição seja  $n/N$ . Este valor, que é chamado de **fator de carga** da tabela de hash (e marcado com  $\lambda$ ), deveria portanto ser limitado por uma pequena constante, preferencialmente menor do que 1. Portanto, dada uma boa função de hash, o tempo de execução esperado das operações  $\text{get}$ ,  $\text{put}$  e  $\text{remove}$  em um dicionário implementado com uma tabela de hash que usa esta função é  $O(\lceil n/N \rceil)$ . Assim, implementam-se estas operações para executarem em um tempo esperado de  $O(1)$ , sabendo-se que  $n$  é  $O(N)$ .

Hidden page

Hidden page

ser implementado, desde que nossa estratégia de teste minimize a possibilidade de formação de agrupamentos decorrente do endereçamento aberto.

### 9.2.6 Uma implementação Java para tabelas de hash

Nos Trechos de códigos 9.3–9.5, mostra-se a classe `HashMap` que implementa um TAD dicionário usando uma tabela de hash com teste linear para resolver colisões. Estes trechos de códigos incluem toda a implementação do TAD dicionário, exceto para os métodos `values()` e `entries()`, os quais são deixados como exercício (R-9.10).

Os principais elementos de projeto da classe Java `HashMap` são apresentados a seguir:

- Mantém-se, em variáveis de instâncias, o tamanho,  $n$ , do dicionário, o arranjo de buckets,  $A$ , e a capacidade,  $N$ , de  $A$ .
- Usa-se o método `hashValue` para computar a função hash de uma chave através do método `hashCode` implementado e da função de compressão multiplicação-adição-e-divisão (MAD).
- Define-se uma sentinela, `AVAILABLE`, como um marcador de itens desativados.
- Provê-se um construtor opcional que permite especificar a capacidade inicial do arranjo de buckets.
- Se o arranjo de buckets atual estiver cheio e alguém tenta inserir um novo elemento, reprocessam-se todos os elementos em um novo arranjo que possui um tamanho duas vezes superior a versão antiga.
- Os seguintes métodos auxiliares (protegidos) são utilizados:
  - `checkKey(k)`, que verifica se a chave  $k$  é válida. Este método atualmente verifica que  $k$  não é `null`, mas a classe que estende `HashMap` pode sobrepor este método com um teste mais elaborado.
  - `rehash()`, que computa uma nova função hash MAD com parâmetros randômicos e reprocessa os elementos em um novo arranjo com o dobro de capacidade.
  - `findEntry(k)`, que procura por um elemento com a chave igual a  $k$ , iniciando pelo índice  $A[h(k)]$  e percorre o arranjo em uma forma circular. Se o método encontra uma posição com um elemento, então retorna o índice  $i$  desta posição. De outra forma, retorna  $-i-1$ , onde  $i$  é o índice da última posição vazia ou disponível encontrada.

```
/** Uma tabela de Hash com teste linear e a função de hash MAD */
import java.util.Iterator;
public class HashMap<K,V> implements Map<K,V> {
    public static class HashEntry<K,V> implements Entry<K,V> {
        protected K key;
        protected V value;
        public HashEntry(K k, V v) { key = k; value = v; }
        public V getValue() { return value; }
        public K getKey() { return key; }
        public V setValue(V val) {
            V oldValue = value;
            value = val;
            return oldValue;
        }
        public boolean equals(Object o) {
            HashEntry<K,V> ent;
            try { ent = (HashEntry<K,V>) o; }
            catch (ClassCastException ex) { return false; }
            return (ent.getKey() == key) && (ent.getValue() == value);
        }
    }
}
```

```

    }

protected Entry<K,V> AVAILABLE = new HashEntry<K,V>(null, null); // marcador
protected int n = 0;           // número de elementos no dicionário
protected int capacity;      // capacidade do arranjo de bucket
protected Entry<K,V>[] bucket; // arranjo de bucket
protected int scale, shift; // shift e fator de ativação
/** Cria uma tabela de hash com capacidade inicial 1023. */
public HashTableMap() { this(1023); }
/** Cria uma tabela de hash com uma capacidade informada. */
public HashTableMap(int cap) {
    capacity = cap;
    bucket = (Entry<K,V>[]) new Entry[capacity]; // Cast seguro
    java.util.Random rand = new java.util.Random();
    scale = rand.nextInt(capacity - 1) + 1;
    shift = rand.nextInt(capacity);
}
/** Determina se uma chave é válida. */
protected void checkKey(K k) {
    if (k == null) throw new InvalidKeyException("Chave inválida: null.");
}
/** Função de hash aplicando o método MAD para o hash code padrão. */
public int hashValue(K key) {
    return Math.abs(key.hashCode()) * scale + shift) % capacity;
}

```

**Trecho de código 9.3** Classe HashTableMap implementando um TAD dicionário, usando uma tabela de hash com teste linear. (Continua no Trecho de código 9.4.)

```

/** Retorna o número de elementos da tabela de hash. */
public int size() { return n; }
/** Verifica e retorna verdadeiro caso a tabela esteja vazia. */
public boolean isEmpty() { return (n == 0); }
/** RETorna um objeto contendo todas as chaves. */
public Iterable<K> keys() {
    PositionList<K> keys = new NodePositionList<K>();
    for (int i=0; i<capacity; i++)
        if ((bucket[i] != null) && (bucket[i] != AVAILABLE))
            keys.addLast(bucket[i].getKey());
    return keys;
}
/** Método de pesquisa auxiliar - retorna o índice da chave encontrada ou -(a + 1), onde a é
 * o índice da primeira posição vazia ou livre encontrada. */
protected int findEntry(K key) throws InvalidKeyException {
    int avail = -1;
    checkKey(key);
    int i = hashValue(key);
    int j = i;
    do {
        Entry<K,V> e = bucket[i];
        if (e == null) {
            if (avail < 0)
                avail = i; // chave não está na tabela
            break;
        }

```

```

if (key.equals(e.getKey( )))      // A chave é encontrada
    return i;          // chave encontrada
if (e == AVAILABLE) { // bucket está desativado
    if (avail < 0)
        avail = i;      // lembre que esta posição está livre
    }
    i = (i + 1) % capacity; // keep looking
} while (i != j);
return -(avail + 1); // Primeira posição vazia ou livre
}
/** Retorna o valor associado com a chave. */
public V get (K key) throws InvalidKeyException {
    int i = findEntry(key); // método auxiliar para encontrar aa chave
    if (i < 0) return null; // Não existe valor para esta chave
    return bucket[i].getValue(); // retorna o valor encontrado neste caso
}

```

**Trecho de código 9.4** Classe HashTableMap implementando um TAD dicionário, usando uma tabela de hash com teste linear. (Continua no Trecho de código 9.5.)

```

/** Insere um par chave-valor no mapa, substituindo o anterior, se existir. */
public V put (K key, V value) throws InvalidKeyException {
    int i = findEntry(key); // Encontra o espaço apropriado para este elemento
    if (i >= 0) // Esta chave tem um valor.
        return ((HashEntry<K,V>) bucket[i]).setValue(value); // define o novo valor
    if (n >= capacity/2) {
        rehash(); // rehash para manter o fator de carga <= 0.5
        i = findEntry(key); // Encontra novamente o local apropriado para este elemento
    }
    bucket[-i-1] = new HashEntry<K,V>(key, value); // converte para o índice próprio
    n++;
    return null; // Não existia valor antigo
}
/** Duplica o tamanho da tabela de hash e rehash todos os elementos. */
protected void rehash() {
    capacity = 2*capacity;
    Entry<K,V>[] old = bucket;
    bucket = (Entry<K,V>[ ]) new Entry[capacity]; // o novo bucket é duas vezes maior
    java.util.Random rand = new java.util.Random();
    scale = rand.nextInt(capacity-1) + 1; // novo fator de ativação para o hash
    shift = rand.nextInt(capacity); // novo fator de deslocamento para o hash
    for (int i=0; i<old.length; i++) {
        Entry<K,V> e = old[i];
        if ((e != null) && (e != AVAILABLE)) { // um elemento válido
            int j = - 1 - findEntry(e.getKey());
            bucket[j] = e;
        }
    }
}
/** Remove o par chave-valor com uma específica chave. */
public V remove (K key) throws InvalidKeyException {
    int i = findEntry(key); // encontra primeiro a chave
    if (i < 0) return null; // nada para remover
    V toReturn = bucket[i].getValue();

```

Hidden page

### 9.2.8 Aplicação: contador de freqüência de palavras

Como uma miniatura de estudo de caso usando tabela de hash, considere-se o problema de contagem do número de ocorrências de diferentes palavras em um documento, as quais aparecem, por exemplo, quando estudiosos de discursos políticos procuram por temas. Uma tabela de hash é uma estrutura de dados ideal para o uso neste problema, por podermos usar palavras como chaves, e contadores de palavras como valores. A aplicação será mostrada no Trecho de código 9.6.

```
import java.io.*;
import java.util.Scanner;
import net.datastructures.*;
/** Um programa que conta palavras em um documento, imprimindo a mais freqüente. */
public class WordCount {
    public static void main(String[ ] args) throws IOException {
        Scanner doc = new Scanner(System.in);
        doc.useDelimiter("[^a-zA-Z]"); // ignora caracteres que não são letras
        HashTableMap<String, Integer> h = new HashTableMap<String, Integer>();
        String word;
        Integer count;
        while (doc.hasNext( )) {
            word = doc.next( );
            if (word.equals(" ")) continue; // ignora strings nulas entre delimitadores
            word = word.toLowerCase( ); // ignora se maiúscula e minúscula
            count = h.get(word); // pega o contador anterior e conta com esta palavra
            if (count == null)
                h.put(word, 1); // autoboxing allows this
            else
                h.put(word, ++count); // autoboxing/unboxing allows this
        }
        int maxCount = 0;
        String maxWord = "sem palavras";
        for (Entry<String, Integer> ent : h.entries( )) { // procura o número máximo de palavras
            if (ent.getValue( ) > maxCount) {
                maxWord = ent.getKey( );
                maxCount = ent.getValue( );
            }
        }
        System.out.print("A palavra mais freqüente é \" " + maxWord);
        System.out.println("\" com um total de ocorrências = " + maxCount + ". ");
    }
}
```

**Trecho de código 9.6** Um programa para contar freqüências de palavras em um documento, apresentando as palavras mais freqüentes. O documento é analisado usando a classe `Scanner`, pelo qual se altera o delimitador de tokens de espaço em branco para qualquer símbolo que não seja letra. Convertem-se, também, as palavras para minúsculo.

---

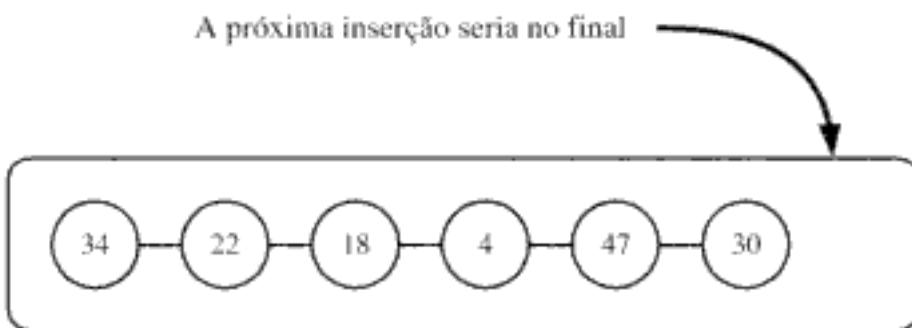
### 9.3 O TAD dicionário

Como um mapa, um dicionário armazena pares chave-valor ( $k, v$ ), os quais são chamados de **elementos**, onde  $k$  é a chave e  $v$  é o valor. Similarmente, um dicionário permite que chaves e valores sejam de qualquer tipo. Mas, apesar de que um mapa insiste que elementos devam ter chaves únicas, um dicionário permite que múltiplos elementos possam ter a mesma chave, bastante parecido com um dicionário de inglês, o qual oferece múltiplas definições para uma mesma palavra.

Hidden page

### 9.3.1 Dicionários baseados em seqüências e auditorias

Uma simples forma de realizar um dicionário é utilizar uma seqüência não-ordenada para armazenar os elementos chave-valor. Esta implementação é freqüentemente chamada *arquivo de log* ou *auditoria*\*. As primeiras aplicações de auditoria são situações em que se deseja armazenar na estrutura de dados. Por exemplo, muitas operações do sistema armazenam arquivos de log para requisições de autenticação processadas. O cenário típico é aquele que possui muitas inserções no dicionário e algumas pesquisas. Por exemplo, pesquisas por operações em um arquivo de log tipicamente ocorrem após algumas coisas de errado terem ocorrido. Assim, um dicionário baseado em seqüência suporta inserções simples e rápidas, possivelmente o gasto do tempo de pesquisa, pelo armazenamento dos elementos em um dicionário com ordem arbitrária. (Ver Figura 9.6.)



**Figura 9.6** Realização de um dicionário  $D$  utilizando um arquivo de log. Somente as chaves deste dicionário são mostradas para apresentar a implementação da seqüência não-ordenada.

#### Implementando um dicionário com uma seqüência não-ordenada

Assume-se que a seqüência  $S$ , usada para um dicionário baseado em seqüência, é implementada com uma lista duplamente encadeada. Apresentam-se as descrições dos principais métodos do dicionário para uma implementação baseada em seqüência no Trecho de código 9.7. Nesta simples implementação, não se assume que um elemento armazena uma referência para sua localização em  $S$ .

##### Algoritmo `findAll( $k$ )`:

**Entrada:** Uma chave  $k$

**Saída:** Uma coleção de elementos com chave igual a  $k$

Cria uma seqüência  $L$  inicialmente vazia

**para** cada elemento  $e$  em  $D.entries()$  **faça**

**se**  $e.getKey() = k$  **então**

$L.addLast(e)$

**retorna**  $L$      {os elementos em  $L$  são os elementos selecionados}

##### Algoritmo `insert( $k, v$ )`:

**Entrada:** uma chave  $k$  e valor  $v$ :

**Saída:** o elemento  $(k, v)$  adicionado em  $D$

Cria um novo elemento  $e = (k, v)$

Chama  $S.addLast(e)$      { $S$  está desordenado}

**retorna**  $e$

##### Algoritmo `remove( $e$ )`:

**Entrada:** Um elemento  $e$

**Saída:** O elemento removido  $e$  ou **nulo** se  $e$  não estiver em  $D$

\* N. de R. T. O autor utiliza a expressão *audit trail*.

{ Não se assume aqui que  $e$  armazena sua localização em  $S$  }

**para** cada posição  $p$  de  $S.positions()$  **faça**

**se**  $p.element() = e$  **então**

Chama  $S.remove(p)$

**retorna**  $e$

**retorna nulo** { não existe elemento  $e$  em  $D$  }

**Algoritmo** entries();

**Entrada:** Nenhuma

**Saída:** Uma coleção dos itens do dicionário  $D$

**retorna**  $S$  { os elementos de  $S$  são os itens de  $D$  }

**Trecho de código 9.7** Alguns dos principais métodos para um dicionário  $D$ , implementado com uma seqüência não-ordenada  $S$ .

### Análise de um dicionário baseado em seqüência

Será analisado, resumidamente, o desempenho de um dicionário implementado com uma seqüência não-ordenada. Iniciando com o uso da memória, nota-se que o espaço requerido por um dicionário baseado em seqüência com  $n$  elementos é  $O(n)$ , visto que a estrutura de dados de lista encadeada tem o uso de memória proporcional ao seu tamanho. Adicionalmente, com esta implementação de um TAD dicionário, pode-se realizar fácil e eficientemente a operação  $insert(k, v)$  com uma simples chamada ao método  $addLast$  de  $S$ , o qual simplesmente adiciona o novo elemento no final da seqüência. Assim, alcança-se o tempo  $O(1)$  para executar a operação  $insert(k, v)$  no dicionário  $D$ .

Infelizmente, essa implementação não permite uma execução eficiente do método  $find(k)$ . Uma operação  $find(k)$  requer, no pior caso, a varredura de toda a seqüência  $S$ , examinando cada um dos  $n$  elementos. Por exemplo, é possível usar um iterator nas posições de  $S$ , parando sempre que se encontra um elemento com chave igual a  $k$  (ou alcançar o final da seqüência). O pior caso para o tempo de execução deste método ocorre quando o elemento procurado não está na seqüência, e, portanto, percorre todos os  $n$  elementos da seqüência. Assim, o método  $find$  é executado em tempo  $O(n)$ .

De forma similar, o tempo proporcional para  $n$  é necessário no pior caso da execução da operação  $remove(e)$  em  $D$ , assumindo-se que os elementos não mantêm o endereço das suas posições em  $S$ . Assim, o tempo de execução para execução do método  $remove(e)$  é  $O(n)$ . De forma alternativa, usando-se elementos que conheçam os endereços em que armazenam suas posições em  $S$ , então se pode executar a operação  $remove(e)$  no tempo  $O(1)$ . (Ver Seção 9.5.1.)

A operação  $findAll$  sempre requer que se procure em toda a seqüência  $S$ , e, portanto, seu tempo de execução é  $O(n)$ . Mais precisamente, usando a notação “big-Theta” (Seção 4.2.3), diz-se que a operação  $findAll$  executa o tempo  $\Theta(n)$ , visto que tem-se o tempo proporcional para  $n$  no melhor e no pior caso. Ou seja, elas são executadas em tempo linear, tanto em seu melhor quanto em seu pior caso.

Concluindo, implementar um dicionário com uma seqüência não-ordenada provê inserções rápidas, mas ao custo de pesquisas e remoções lentas. Assim, só se deve usar esta implementação quando se espera que o dicionário sempre seja pequeno ou quando o número de inserções seja grande se comparado com pesquisas e remoções. É claro, arquivar transações de bases de dados ou de um sistema operacional são situações com essas características.

Apesar disso, existem muitos outros cenários em que o número de inserções em um dicionário é proporcional ao número de pesquisas e remoções, e, nesses casos, a implementação com uma seqüência é claramente inapropriada. A implementação de um dicionário não-ordenado, que se discutirá em seguida, pode frequentemente ser usada para que se tenham inserções, remoções e procura rápida em muitos casos.

### 9.3.2 Implementação de um dicionário com tabela de hash

Pode-se usar uma tabela de hash para implementar um TAD dicionário, quase da mesma forma que se fez para o TAD mapa. É claro que a principal diferença é que o dicionário permite elementos com chaves duplicadas. Assumindo que o fator de carga da nossa tabela de hash é mantido abaixo de 1, a função de hash espalha elementos uniformemente de forma correta, e se usa encadeamento separado para resolver colisões, de forma que se pode alcançar o tempo  $O(1)$  no desempenho dos métodos `find`, `remove` e `insert` e o tempo  $O(1 + m)$  no desempenho para o método `findAll`, onde  $m$  é o número de elementos retornados.

Adicionalmente, podem-se simplificar os algoritmos para a implementação deste dicionário, assumindo-se que se tem um dicionário baseado em seqüência, armazenando elementos em cada posição de um arranjo de buckets  $A$ . Tal suposição, estaria em conformidade com o nosso uso de encadeamento separado, desde que cada célula fosse uma seqüência. Esta abordagem permite, implementar os principais métodos do dicionário, como é mostrado no Trecho de código 9.8.

**Algoritmo** `insert( $k, v$ )`:

**Entrada:** uma chave  $k$  e valor  $v$

**Saída:** O elemento  $(k, v)$  adicionado em  $D$

**se**  $(n + 1) / N > \lambda$  **então**

    Dobre o tamanho de  $A$  e rehash todos os elementos existentes.

$e \leftarrow A[h(k)].insert(k, v)$

$n \leftarrow n + 1$

**retorna**  $e$

**Algoritmo** `findAll( $k$ )`

**Entrada:** uma chave  $k$

**Saída:** Uma coleção de elementos com chave igual a  $k$

**retorna**  $A[h(k)].findAll(k)$

**Algoritmo** `remove( $e$ )`:

**Entrada:** um elemento  $e$

**Saída:** O elemento removido  $e$  ou **nulo** se  $e$  não estiver em  $D$

$t \leftarrow A[h(k)].remove(e)$

**se**  $t \neq \text{nulo}$  **então**

$n \leftarrow n - 1$

**retorna**  $t$

**Trecho de código 9.8** Alguns dos principais métodos para um dicionário  $D$ , implementados com uma tabela de hash que usa um arranjo de buckets,  $A$ , e uma seqüência não-ordenada para cada posição em  $A$ . Usa-se  $n$  para denotar o número de elementos em  $D$ ,  $N$  para denotar a capacidade de  $A$ , e  $\lambda$  para denotar o maior fator de carga para a tabela de hash.

---

### 9.3.3 Tabelas de pesquisa ordenada e pesquisa binária

Se as chaves em um dicionário  $D$  estão ordenadas, podem-se armazenar os elementos de  $D$  em um arranjo  $S$  em ordem não-crescente de chave. (Ver Figura 9.7.) Especifica-se que  $S$  é um arranjo, mais particularmente, uma seqüência de elementos, pois a ordenação das chaves no vetor  $S$  permite uma pesquisa mais rápida do que seria possível se  $S$  fosse, por exemplo, uma lista encadeada. Reconhecidamente, uma tabela de hash possui uma boa expectativa do tempo de execução de pesquisas. Porém, seu pior tempo de pesquisa não é melhor que em uma lista encadeada, e em algumas aplicações, como em um processamento de tempo real, é necessário garantir um limite para o pior caso. O algoritmo rápido para pesquisa em um arranjo ordenado, o qual se discute

nesta subseção, tem garantidamente o melhor tempo de execução para o pior caso. Ele pode ser o preferido para uma tabela de hash em certas aplicações. Faz-se referência a esta implementação do arranjo ordenado de um dicionário  $D$  como uma **tabela de pesquisa ordenada**.

0	1	2	3	4	5	6	7	8	9	10
4	6	9	12	15	16	18	28	34		

**Figura 9.7** Realização de um dicionário  $D$  por uma tabela de pesquisa ordenada. Mostram-se somente as chaves deste dicionário para destacar sua ordenação.

O espaço requerido por uma tabela de pesquisa ordenada é  $O(n)$ , o qual é similar a implementação do dicionário baseado em seqüência (Seção 9.3.1), assumindo que se expande e retrai o arranjo suportando a seqüência  $S$  para manter o tamanho proporcional deste arranjo com o número de elementos de  $S$ . Entretanto, diferente de uma seqüência não-ordenada a execução de alterações em uma tabela de pesquisa leva uma quantidade de tempo considerável. Em particular, a execução da operação  $\text{insert}(k, v)$  em uma tabela de pesquisa requer o tempo de  $O(n)$ , desde que seja necessário deslocar todos os elementos do arranjo com chave maior que  $k$  para fazer espaço para o novo elemento  $(k, v)$ . Uma observação similar se aplica para a operação  $\text{remove}(k)$ , visto que ele leva o tempo  $O(n)$  para deslocar todos os elementos do arranjo com chave maior que  $k$  para fechar o “buraco” deixado pelo elemento removido (ou elementos). A implementação da tabela de pesquisa é, por esta razão, inferior ao arquivo de *log* em termos dos tempos de execução no pior caso das operações de atualização do dicionário. Apesar disso, podemos executar o método *find* mais rápido em uma tabela de pesquisa.

### Pesquisa binária

Uma vantagem significativa do uso de um arranjo ordenado  $S$  para implementar um dicionário  $D$  com  $n$  elementos é que o acesso a um elemento de  $S$  pelo seu **índice** custa o tempo  $O(1)$ . É preciso lembrar que o índice de um elemento em uma seqüência é o número do elemento anterior (Seção 6.1). Assim, o primeiro elemento de  $S$  tem o índice 0, e o último elemento tem o índice  $n - 1$ .

Os elementos em  $S$  são os itens do dicionário  $D$ , e já que  $S$  está ordenado, o item com colocação  $i$  tem uma chave que não é menor do que as chaves dos itens com colocações  $0, 1, \dots, i - 1$ , e não maior do que as chaves dos itens com colocações  $i + 1, \dots, n - 1$ . Esta observação permite que se ache um item usando um método de procura muito rápido. Chama-se de **candidato** um item de  $D$  se, no estágio atual de procura, não se pode garantir que a chave seja igual a  $k$ . O algoritmo mantém dois parâmetros,  $\text{low}$  e  $\text{high}$ , tais que todos os itens candidatos tenham colocação maior ou igual a  $\text{low}$  e menor ou igual a  $\text{high}$ . Inicialmente, tem-se  $\text{low} = 0$  e  $\text{high} = n - 1$ . Então, se compara  $k$  à chave do candidato mediano, ou seja, o item com colocação

$$\text{mid} = \lfloor (\text{low} + \text{high})/2 \rfloor.$$

Analisa-se três casos:

- Se  $k = e.\text{getKey}()$ , então se acha o item que se estava procurando, e a pesquisa termina com sucesso, retornando  $e$ .
- Se  $k < e.\text{getKey}()$ , então se reexamina a primeira metade do vetor, ou seja, a metade com colocações entre  $\text{low}$  e  $\text{mid} - 1$ .
- Se  $k > e.\text{getKey}()$ , então se reexamina a segunda metade do vetor, ou seja, a metade com colocações entre  $\text{mid} + 1$  e  $\text{high}$ .

Hidden page

$$(mid - 1) - low + 1 = \left\lfloor \frac{low + high}{2} \right\rfloor - low \leq \frac{high - low + 1}{2}$$

ou

$$high - (mid + 1) + 1 = high - \left\lfloor \frac{low + high}{2} \right\rfloor \leq \frac{high - low + 1}{2}.$$

Inicialmente, o número de itens candidatos é  $n$ ; após a primeira chamada a `BinarySearch`, ele é de no máximo  $n/2$ ; após a segunda, ele é de no máximo  $n/4$ , e assim sucessivamente. Em geral, após a  $i$ -ésima chamada a `BinarySearch`, o número de candidatos restantes é de no máximo  $n/2^i$ . No pior caso (o elemento não existe), as chamadas recursivas param quando não há mais itens candidatos. Portanto, o número máximo de chamadas recursivas feitas é o menor inteiro  $m$  tal que

$$n/2^m < 1.$$

Em outras palavras (e lembrando que a base de um logaritmo é omitida quando é 2),  $m > \log n$ . Assim, tem-se

$$m = \lfloor \log n \rfloor + 1,$$

o que implica que a pesquisa binária é executada no tempo  $O(\log n)$ .

Existe uma variação simples da pesquisa binária que realiza `findAll(k)` em tempo  $O(\log n + s)$ , onde  $s$  é o número de elementos retornados. Os detalhes são deixados como um exercício (C-9.4).

Portanto, pode-se usar uma tabela de pesquisa ordenada para pesquisas rápidas em um dicionário, mas usar uma tabela de pesquisa ordenada para muitas atualizações do dicionário tomaria um tempo considerável. Por essa razão, a aplicação primária para uma tabela de pesquisa deve ser executada em uma situação em que se esperam poucas atualizações no dicionário, mas muitas pesquisas. Uma situação assim surgiria, por exemplo, em uma lista ordenada das palavras usadas para ordenar uma enciclopédia ou arquivo de ajuda.

### Comparando implementações de dicionário

A Tabela 9.3 compara os tempos de execução dos métodos de um dicionário realizado por uma seqüência não-ordenada, uma tabela de hash ou uma tabela de pesquisa ordenada. Nota-se que uma seqüência não-ordenada permite inserções rápidas, mas pesquisas e remoções lentas, enquanto a tabela de pesquisa permite pesquisas rápidas, mas inserções e remoções lentas. Embora não se tenha discutido o assunto explicitamente, nota-se que uma seqüência ordenada implementada com uma lista duplamente encadeada seria lenta em quase todas as operações de um dicionário. (Ver Exercício R-9.3.)

Método	Seqüência	Tabela de hash	Tabela de pesquisa
<code>size, isEmpty</code>	$O(1)$	$O(1)$	$O(1)$
<code>entries</code>	$O(n)$	$O(n)$	$O(n)$
<code>find</code>	$O(n)$	$O(1)$ exp., $O(n)$ pior caso	$O(\log n)$
<code>findAll</code>	$O(n)$	$O(1 + s)$ exp., $O(n)$ pior caso	$O(\log n + s)$
<code>insert</code>	$O(1)$	$O(1)$	$O(n)$
<code>remove</code>	$O(n)$	$O(1)$ exp., $O(n)$ pior caso	$O(n)$

**Tabela 9.3** Comparação dos tempos de execução dos métodos de um dicionário realizado através de uma seqüência não-ordenada, uma tabela de hash ou uma tabela de pesquisa ordenada. Indica-se  $n$  como o número de itens no dicionário,  $N$  como sendo a capacidade do arranjo de

Hidden page

Hidden page

Hidden page

finalmente uma moeda dê cara. Em seguida, ligam-se todas as referências ao novo item  $(k, v)$ , criadas neste processo, para criar a torre para o novo elemento. Uma “jogada de moeda” pode ser simulada com a classe `java.util.Random`, que é um gerador randômico de números criado em Java, chamando o método `nextInt(2)`, o qual retorna 0 ou 1, cada um com probabilidade de 50%.

Fornece-se o algoritmo de inserção para uma skip list  $S$  no Trecho de código 9.11, e ilustra-se este algoritmo na Figura 9.11. O algoritmo de inserção usa uma operação `insertAfterAbove( $p, q, (k, v)$ )` que insere uma posição armazenando o item  $(k, v)$  após a posição  $p$  (no mesmo nível de  $p$ ) e acima da posição  $q$ , retornando a posição  $r$  do novo item (e acertando as referências internas para que os métodos `next`, `prev`, `above` e `below` funcionem corretamente para  $p$ ,  $q$  e  $r$ ). O tempo de execução esperado do algoritmo de inserção em uma skip list com  $n$  elementos é  $O(\log n)$ , o qual se mostra na Seção 9.4.2.

#### Algoritmo SkipInsert( $k, v$ )

**Entrada:** Chave  $k$  e valor  $v$

**Saída:** Elemento inserido na skip list

$p \leftarrow \text{SkipSearch}(k)$

$q \leftarrow \text{insertAfterAbove}(p, \text{nulo}, (k, v))$  {está-se no nível inferior}

$e \leftarrow q.\text{element}()$

$i \leftarrow 0$

**enquanto** `coinFlip() = heads` **faça**

$i \leftarrow i + 1$

**se**  $i \geq h$  **então**

$h \leftarrow h + 1$  {adicionar um novo elemento na skip list}

$t \leftarrow \text{next}(s)$

$s \leftarrow \text{insertAfterAbove}(\text{nulo}, s, (-\infty, \text{nulo}))$

$\text{insertAfterAbove}(s, t, (+\infty, \text{nulo}))$

**enquanto** `above(p) = nulo` **faça**

$p \leftarrow \text{prev}(p)$  {varredura}

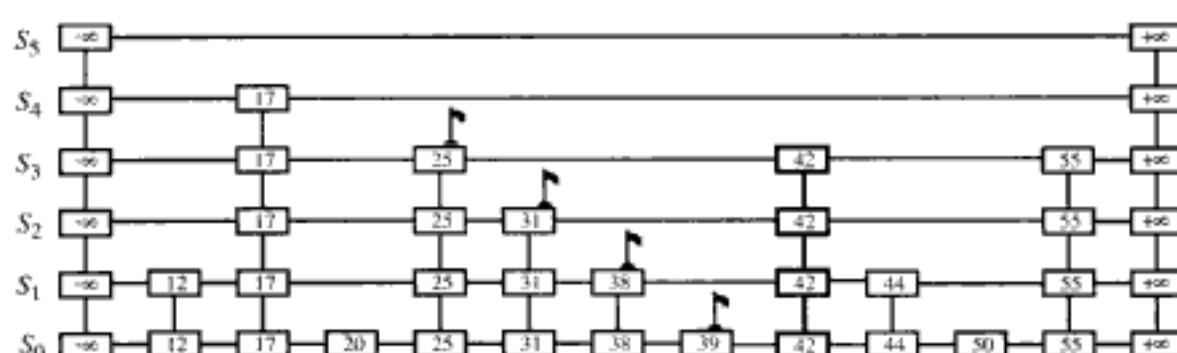
$p \leftarrow \text{above}(p)$  {ir para o nível mais alto}

$q \leftarrow \text{insertAfterAbove}(p, q, e)$  {adicionar uma posição na torre do novo elemento}

$n \leftarrow n + 1$

**retorna**  $e$

**Trecho de código 9.11** Inserção em uma skip list. O método `coinFlip()` retorna “cara” ou “coroa”, cada uma com probabilidade de 50%. As variáveis  $n$ ,  $h$  e  $s$  armazenam o número dos elementos, a altura e o nodo inicial da skip list.



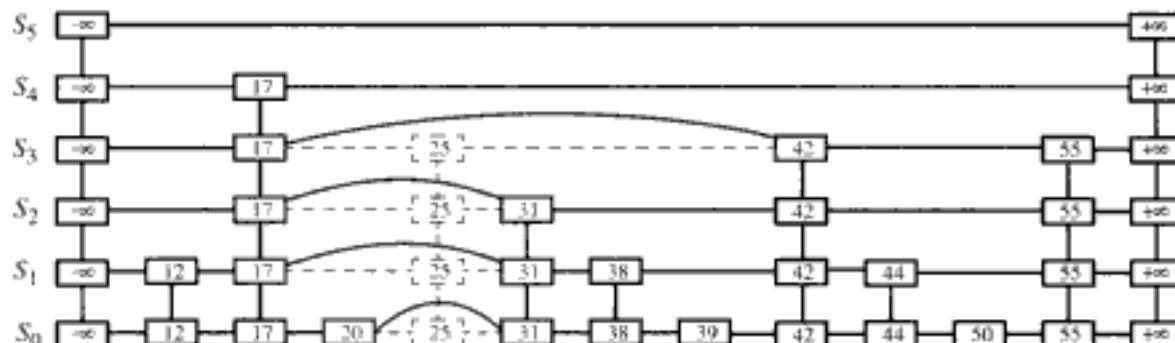
**Figura 9.11** Inserção de um elemento com chave 42 na skip list da Figura 9.9. Assume-se que a “jogada da moeda” randômica para o novo elemento retornará 3 coroas seguidas em uma linha, seguido pela cara. As posições visitadas estão marcadas em cinza. A posição inserida para armazenar o novo elemento está desenhada com linhas grossas, e as posições precedentes estão marcadas com bandeiras.

### Remoção em uma skip list

Como os algoritmos de pesquisa e inserção, o algoritmo de remoção para uma skip list  $S$  é bastante simples. De fato, ele é ainda mais simples do que o algoritmo de inserção. Isto é, para realizar uma operação  $\text{remove}(k)$ , começa-se pela execução do método  $\text{SkipSearch}(k)$ . Se a posição  $p$  armazena um elemento com chave diferente de  $k$ , então retorna  $\text{null}$ . Caso contrário, remove-se  $p$  e todas as posições acima de  $p$ , as quais são facilmente acessadas usando a operação `above` para subir na torre deste elemento em  $S$ , iniciando na posição  $p$ . O algoritmo de remoção é ilustrado na Figura 9.12, e sua descrição detalhada é deixada como um exercício (R-9.16). Como será mostrado na próxima subseção, o tempo de execução esperado para remoção em uma skip list é  $O(\log n)$ .

Antes de descrever a análise, no entanto, existem alguns melhoramentos para a estrutura de dados skip list que devem ser discutidos. Primeiro, não se precisa realmente armazenar referências para os itens da skip list acima do nível base, pois o que é necessário nestes níveis são as referências para as chaves. Segundo, não se precisa realmente do método `above`. Na verdade, nem se necessita do método `prev`. Pode-se realizar a inserção e remoção de itens de cima para baixo baseando-as em varreduras e economizando referências para os elementos anteriores e acima de um elemento. Os detalhes dessa otimização serão explorados no Exercício C-9.10. Nenhuma dessas otimizações melhora o desempenho assintótico das skip lists por mais do que um fator constante, mas ainda assim esses melhoramentos podem ser importantes na prática. De fato, experimentos sugerem que as skip lists otimizadas são mais rápidas, na prática, do que árvores AVL e outras árvores平衡adas de pesquisa, que são discutidas no Capítulo 10.

O tempo de execução esperado para o algoritmo de remoção é  $O(\log n)$ , o que será mostrado na Seção 9.4.2.



**Figura 9.12** Remoção do elemento com chave 25 da skip list da Figura 9.11. As posições visitadas após a pesquisa pela posição  $S_0$  guardando o item são demarcados em cinza. As posições removidas são desenhadas com linhas tracejadas.

### Mantendo o nível superior

Uma skip list  $S$  deve manter uma referência para a posição inicial (o elemento mais acima e à esquerda em  $S$ ) como uma variável instanciada, e deve-se ter uma política para tratar qualquer inserção que queira continuar inserindo um item acima do nível superior de  $S$ . Existem duas possíveis opções neste caso, e as duas têm seus méritos.

Uma possibilidade é restringir o nível superior,  $h$ , para mantê-lo em algum valor fixo que seja função de  $n$ , o número de itens atuais no dicionário (veremos pela análise que  $h = \max\{10, 2 \lceil \log n \rceil\}$  é uma boa escolha e que  $h = 3 \lceil \log n \rceil$  é ainda mais seguro.) Implementar esta opção significa que se deve modificar o algoritmo de inserção para que ele termine quando se atingir o nível mais alto (a não ser que  $\lceil \log n \rceil < \lceil \log(n + 1) \rceil$ ), pois, neste caso, pode-se subir ainda mais um nível, uma vez que o limite de altura estará crescendo).

A outra possibilidade é deixar a inserção continuar inserindo o elemento enquanto a moeda lançada pelo gerador de números aleatórios der coroa. Esta abordagem usa o Algoritmo `skipInsert`

Hidden page

Por exemplo, se  $n = 1000$ , esta probabilidade é uma em um milhão. Generalizando, dada uma constante  $c > 1$ ,  $h$  é maior do que  $c \log n$  com probabilidade no máximo  $1/n^{c-1}$ . Ou seja, a probabilidade de que  $h$  seja menor ou igual a  $c \log n$  é no mínimo  $1 - 1/n^{c-1}$ . Assim, com uma alta probabilidade, a altura  $h$  de  $S$  é  $O(\log n)$ .

### Analizando o tempo de pesquisa em uma skip list

Considere-se o tempo de execução de uma pesquisa em uma skip list  $S$ , lembrando que uma pesquisa envolve dois laços **while** aninhados. O laço interno realiza a varredura em um nível de  $S$  enquanto a próxima chave não for maior do que a chave  $k$  sendo procurada, e o laço externo desce para o nível inferior e repete a varredura. Como a altura  $h$  de  $S$  é  $O(\log n)$ , com grande probabilidade, o número de passos de descida nos níveis é  $O(\log n)$ , também com grande probabilidade.

Ainda temos de limitar o número de passos de varredura que se fez. Seja  $n_i$  o número de chaves examinadas quando se está fazendo uma varredura no nível  $i$ . Observe-se que, depois da chave na posição inicial, cada chave adicional examinada em uma varredura no nível  $i$  não pode pertencer ao nível  $i + 1$ , pois a teríamos encontrado na varredura anterior. Assim, a probabilidade de que uma chave seja contada em  $n_i$  é  $1/2$ . Portanto, o valor esperado de  $n_i$  é igual ao número esperado de vezes em que se deve jogar uma moeda antes que ela dê cara. Esse valor esperado é 2. Assim, o total de tempo esperado que será gasto em varreduras em qualquer nível é  $O(1)$ . Como  $S$  tem  $O(\log n)$  níveis com grande probabilidade, uma pesquisa em  $S$  toma um tempo esperado  $O(\log n)$ . Com uma análise similar, pode-se mostrar que o tempo de execução esperado para inserção e remoção é  $O(\log n)$ .

### Espaço utilizado em uma skip list

Finalmente, examinaremos a exigência de espaço de uma skip list  $S$  com  $n$  elementos. Como se observa acima, o número de itens esperado no nível  $i$  é  $n/2^i$ , significando que o número total de elementos esperados em  $S$  é

$$\sum_{i=0}^h \frac{n}{2^i} = n \sum_{i=0}^h \frac{1}{2^i}.$$

Usando a Proposição 4.5 de soma geométrica, tem-se:

$$\sum_{i=0}^h \frac{1}{2^i} = \frac{\left(\frac{1}{2}\right)^{h+1} - 1}{\frac{1}{2} - 1} = 2 \cdot \left(-1 \frac{1}{2^{h+1}}\right) < 2 \quad \text{para todo } h \geq 0.$$

Portanto, a exigência de memória esperada para  $S$  é  $O(n)$ .

A Tabela 9.4 resume o desempenho de um dicionário realizado com uma skip list.

Operação	Tempo
size, isEmpty	$O(1)$
entries	$O(n)$
find, insert, remove	$O(\log n)$ (esperado)
findAll	$O(\log n + s)$ (esperado)

**Tabela 9.4** Desempenho de um dicionário realizado com uma skip list. Denota-se o número de itens no dicionário no momento da operação com  $n$ , e o tamanho do iterador retornado pelas operações findAll com  $s$ . A exigência de memória esperada é  $O(n)$ .

## 9.5 Extensões e aplicações de dicionários

Nesta seção, exploram-se várias extensões e aplicações de dicionários.

### 9.5.1 Suportando localizadores em um dicionário

Como foi feito com as filas de prioridades (Seção 8.4.2), pode-se também usar localizadores para acelerar o tempo de execução de algumas operações em um dicionário. Em particular, um localizador pode acelerar muito a remoção de um elemento em um dicionário. Para remoção de um localizador  $e$ , pode-se simplesmente ir diretamente ao local na estrutura de dados em que  $e$  está armazenado e removê-lo. Seria possível implementar um localizador, por exemplo, pela adição a nossa classe de  $i$ , a variável privada `location` e métodos protegidos `location()` e `setLocation(p)`, o qual retorna e define esta variável respectivamente. Então, se requer que a variável `location` para um elemento  $e$  sempre referenciará a posição de  $e$  ou o índice na estrutura de dados da implementação do dicionário. Seria necessário alterar esta variável toda vez que o elemento fosse movido, assim, provavelmente faria mais sentido que esta classe fosse mais relacionada com a classe que implementa o dicionário (a classe localizador poderia ser aninhada na classe dicionário). Abaixo, se mostrará como definir localizadores para diversas estruturas de dados apresentadas neste capítulo.

- **Seqüência não-ordenada:** em uma seqüência não-ordenada  $L$ , implementando um dicionário, é possível manter a variável `location` de cada elemento  $e$  para apontar para a posição de  $e$  em uma lista encadeada suportada por  $L$ . Esta escolha permite executar o método `remove(e)` como  $L.remove(e.location())$ , o qual executaria no tempo  $O(1)$ .
- **Tabela de hash com encadeamento separado:** considere uma tabela de hash, com um arranjo de buckets  $A$  e uma função de hash  $h$ , que usa encadeamento separado para tratar colisões. Usa-se a variável `location` de cada elemento  $e$  para apontar para a posição de  $e$  na seqüência  $L$  implementando um mini-mapa  $A[h(k)]$ . Esta escolha permite executar o principal trabalho de um método `remove(e)` como  $L.remove(e.location())$ , o qual executaria em um esperado tempo constante.
- **Tabela de pesquisa ordenada:** em uma tabela ordenada  $T$ , implementando um dicionário, se manteria a variável `location` de cada elemento  $e$  para ser o índice de  $e$  em  $T$ . Esta escolha permitiria executar o método `remove(e)` como  $T.remove(e.location())$ . (Relembrando que `location()` agora retorna um inteiro.) Esta abordagem executaria de forma rápida se o elemento  $e$  estiver armazenado próximo do fim de  $T$ .
- **Skip list:** Em uma skip list  $S$  implementando um dicionário, se manteria a variável `location` de cada elemento  $e$  para apontar a posição de  $e$  no nível base de  $S$ . Esta escolha permitia saltar o passo da pesquisa em nosso algoritmo para executar o método `remove(e)` em uma skip list.

Apresenta-se o resumo do desempenho da remoção de um elemento em um dicionário com localizadores na Tabela 9.5.

Seqüência	Tabela de hash	Tabela de pesquisa	Skip list
$O(1)$	$O(1)$ (esperado)	$O(n)$	$O(\log n)$ (esperado)

**Tabela 9.5** Desempenho do método `remove` em dicionário implementados com localizadores. Usa-se  $n$  para denotar o número de elementos em um dicionário.

### 9.5.2 O TAD dicionário ordenado

Em um dicionário ordenado, precisa-se executar as operações comuns dos dicionários, mas também manter uma relação de ordem para as chaves do nosso dicionário. Pode-se usar um comparador para prover a relação de ordem entre as chaves, como se fez para as implementações de dicionários, tabela de pesquisa ordenada e skip list descritas anteriormente. De fato, todas as implementações de dicionários discutidas no Capítulo 10 usam um comparador para armazenar o dicionário em ordem não-decrescente de chave.

Quando os elementos do dicionário são armazenados em ordem, pode-se prover implementações eficientes de métodos adicionais em um TAD dicionário. Por exemplo, poderia-se considerar a adição dos seguintes métodos no TAD dicionário, bem como definir no **TAD dicionário ordenado**.

`first()`: Retorna um elemento com a menor chave.

`last()`: Retorna um elemento com a maior chave.

`successors(k)`: Retorna um iterator dos elementos com chaves maiores ou iguais a *k*, em ordem não-decrescente.

`predecessors(k)`: Retorna um iterator dos elementos com chaves menores ou iguais a *k*, em ordem não-crescente.

### Implementando um dicionário ordenado

A ordem natural das operações acima torna o uso de uma seqüência não-ordenada ou uma tabela de hash inapropriada para implementar o dicionário, pois nenhuma destas estruturas de dados mantém qualquer informação de ordenação para as chaves do dicionário. De fato, tabelas de hash alcançam seus melhores desempenhos de pesquisa quando suas chaves estão distribuídas quase que randomicamente. Assim, se deveria considerar uma tabela de pesquisa ordenada ou skip list (ou uma estrutura de dados do Capítulo 10) quando da distribuição com dicionários ordenados.

Por exemplo, usando uma skip list para implementar um dicionário ordenado, pode-se implementar os métodos `first()` e `last()` no tempo  $O(1)$  acessando a segunda e a penúltima posição da seqüência de base. Também os métodos `successors(k)` e `predecessors(k)` podem ser implementados para executar no tempo  $O(\log n)$ . Além disso, o iterator retornado pelos métodos `successors(k)` e `predecessors(k)` poderiam ser implementados usando uma referência para a posição atual do nível base da skip list. Assim, os métodos `hasNext` e `next` destes iteradores executariam cada um em um tempo constante usando esta abordagem.

### A interface `java.util.SortedMap`

Java provê uma versão ordenada para a interface `java.util.Map` chamada `java.util.SortedMap`. Esta interface estende a interface `java.util.Map` com métodos que recebem ordem na contas. Como a interface pai, uma `SortedMap` não permite chaves duplicadas.

Ignorando o fato de que dicionários permitem múltiplos elementos com a mesma chave, possíveis correspondências entre métodos do nosso TAD dicionário ordenado e os métodos da interface `java.util.SortedMap` são apresentados na Tabela 9.6.

### 9.5.3 Banco de dados de vôos e conjuntos máximos

Como mencionado nas seções anteriores, dicionários não-ordenados e ordenados têm muitas aplicações.

Nesta seção, serão exploradas algumas aplicações específicas de dicionários ordenados.

Métodos do dicionário ordenado	Métodos da interface java.util.SortedMap
first().getKey()	firstKey()
first().getValue()	get(firstKey())
last().getKey()	lastKey()
last().getValue()	get(lastKey())
successors( <i>k</i> )	tailMap( <i>k</i> ).entrySet().iterator()
predecessors( <i>k</i> )	headMap( <i>k</i> ).entrySet().iterator()

**Tabela 9.6** Livre correspondência entre métodos do TAD dicionário ordenado e métodos da interface java.util.SortedMap, o qual suporta outros métodos. A expressão java.util.SortedMap para predecessors(*k*) não é, entretanto, exatamente uma correspondência como o iterador retornado seria pela ordenação crescente das chaves, e não incluiria o elemento com chave igual a *k*. Parece não ser uma eficiente forma de pegar a correspondência verdadeira para predecessors(*k*) usando os métodos da java.util.SortedMap.

### Banco de dados de vôos

Existem vários sites da Web que permitem aos usuários executar pesquisas em banco de dados de vôos para encontrar vôos entre várias cidades, tipicamente com a intenção de vender uma passagem. Para criar a pesquisa, um usuário especifica a cidade de origem e de destino, uma data da partida e hora da partida. Para suportar essas pesquisas, pode-se modelar o banco de dados de vôos como um dicionário, onde as chaves são objetos Flight que contêm campos correspondentes aos seus quatro parâmetros. Isto é, a chave é uma *tupla*

$$k = (\text{origem}, \text{destino}, \text{data}, \text{hora}).$$

Informações adicionais, como o número do vôo, o número de assentos vagos na primeira classe (F) e na classe econômica (Y), duração do vôo e valor da passagem, podem ser armazenadas em um objeto valor.

Entretanto, encontrar o vôo requisitado não é simplesmente encontrar uma chave em um dicionário com a pesquisa correspondente. A principal dificuldade é que, embora um usuário precise encontrar exatamente as cidades de origem e destino, bem como a data de partida, ele provavelmente estará satisfeito com qualquer hora de partida que estiver próxima da hora de partida informada. É claro que se pode gerenciar cada pesquisa por meio da ordenação das chaves usando o léxico. Assim, dada uma chave de pesquisa do usuário *k*, pode-se invocar o método successors(*k*) para retornar um conjunto de todos os vôos entre as cidades desejadas na data de partida informada, com hora de partida ordenada de forma crescente a partir da hora de partida informada. Um uso similar do método predecessors(*k*) daria vôos com horas de partidas anteriores à hora de partida informada. Então, uma implementação eficiente para um dicionário ordenado, como um que usa uma skip list, seria uma boa forma para satisfazer cada pesquisa. Por exemplo, chamando o método successors(*k*) com uma chave de pesquisa *k* = (ORD, PVD, 05Maio, 09:30), resultaria em um conjunto com os seguintes elementos:

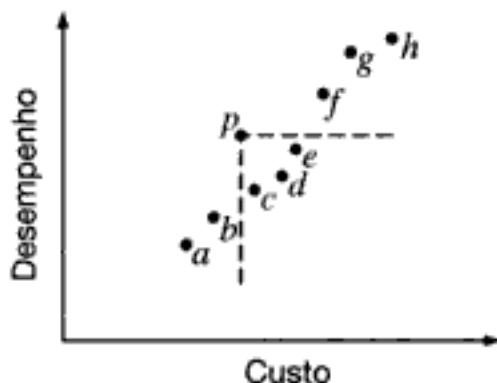
- ( (ORD, PVD, 05Maio, 09:53), (AA 1840, F5, Y15, 02:05, \$251) )
- ( (ORD, PVD, 05Maio, 13:29), (AA 600, F2, Y0, 02:16, \$713) )
- ( (ORD, PVD, 05Maio, 17:39), (AA 416, F3, Y9, 02:09, \$365) )
- ( (ORD, PVD, 05Maio, 19:50), (AA 1828, F9, Y25, 02:13, \$186) )

### Conjunto máximo

A vida é cheia de trocas. Frequentemente, é necessário trocar um desejo de avaliar o desempenho em oposição a um custo correspondente. Supõe-se para o propósito de um exemplo, que estamos

interessados na manutenção de um banco de dados de classificação de automóveis pela velocidade máxima e pelo custo. Seria desejável permitir a alguém, com uma certa quantia para gastar, pesquisar na nossa base de dados a fim de procurar pelo carro mais rápido que esteja dentro das suas possibilidades.

Pode-se modelar um problema de comercialização como este pelo uso de um par chave-valor para modelar os dois parâmetros que se está comercializando, os quais, neste caso, seriam o par (custo, velocidade) para cada carro. Note-se que usando esta medida alguns carros são estritamente melhores que outros. Por exemplo, um carro com o par custo-velocidade (20.000-100) é estritamente melhor que o carro com o par custo-velocidade (30.000-90). Ao mesmo tempo, existem carros que não são sobrepujados por outro carro. Por exemplo, um carro com o par custo-velocidade (20.000-100) pode ser melhor ou pior que um carro com o par custo-velocidade (30.000-120), dependendo de quanto dinheiro se possui para gastar (Ver Figura 9.13.)



**Figura 9.13** Ilustração do custo-desempenho das trocas com pares chave-valor representados por pontos no plano. Note-se que o ponto *p* é estritamente melhor que os pontos *c*, *d* e *e*, mas pode ser melhor ou pior que os pontos *a*, *b*, *f*, *g* e *h*, dependendo do preço que se deseja pagar. Assim, se adicionando o ponto *p* no nosso conjunto, poderia-se remover os pontos *c*, *d* e *e*, mas não os outros.

Formalmente, diz-se que um par preço-desempenho  $(a,b)$  **domina** um par  $(c,d)$  se  $a < c$  e  $b > d$ . Um par  $(a,b)$  é chamado par **máximo** se este não é dominado por nenhum outro par. Estão interessados na manutenção do conjunto de máximos de uma coleção  $C$  de pares preço-desempenho. Isto é, se gostaria de adicionar novos pares nesta coleção (por exemplo, quando um novo carro é introduzido), e se gostaria de pesquisar esta coleção para um dado valor em dólar –  $d$  – para procurar o carro mais rápido que não custe mais que  $d$ .

Pode-se armazenar o conjunto de pares máximos em um dicionário ordenado  $D$ , ordenado pelo custo, mas que o custo seja o campo chave, e desempenho (velocidade) seja o campo valor. Pode-se então implementar as seguintes operações:  $\text{add}(c,p)$ , que adiciona um novo par custo-desempenho  $(c,p)$ ; e  $\text{best}(c)$ , que retorna o melhor par com o valor custo no máximo  $c$ , como mostrado no Trecho de código 9.12.

#### Algoritmo $\text{best}(c)$ :

**Entrada:** um custo  $c$

**Saída:** o par custo-desempenho de  $D$  com o maior custo menor ou igual a  $c$  ou **nulo** se não existir

$B \leftarrow D.\text{predecessors}(c)$

se  $B.\text{hashNext}()$  então

retorna  $B.\text{next}()$  {o primeiro elemento no iterator dos predecessores}

senão

retorna **nulo**

**Algoritmo** add( $c, p$ ):

**Entrada:** um para custo-desempenho ( $c, p$ )

**Saída:** Nenhuma

$B \leftarrow D.\text{predecessors}(c)$  [iterador dos pares com custo de no máximo  $c$ ]

**se**  $B.\text{hasNext}()$  **então**

$e \leftarrow B.\text{next}()$  {predecessor de  $c$ }

**se**  $e.\text{getValue}() > p$  **então**

**retorna** { $(c, p)$  está controlado, desta forma não insira em  $D$ }

$C \leftarrow D.\text{successor}(c)$  [iterador do pares com custo de no mínimo  $c$ ]

**enquanto**  $C.\text{hasNext}()$  **faça**

$e \leftarrow C.\text{next}()$  {sucessor de  $c$ }

**se**  $e.\text{getValue}() < p$  **então**

$D.\text{remove}(e)$  {este par está controlado por  $(c, p)$ }

**senão**

pare com o laço “**enquanto**” {não existem mais pares controlados por  $(c, p)$ }

$D.\text{insert}(c, p)$  {adiciona o par  $(c, p)$ , que não está controlado}

**Trecho de código 9.12** Os métodos para manutenção do conjunto máximo, implementado com um dicionário ordenado  $D$ .

Ao implementar  $D$  usando skip list, pode-se executar a pesquisa best( $c$ ) no tempo esperado de  $O(\log n)$ , e add( $c, p$ ) no tempo esperado  $O((1 + r)\log n)$ , onde  $r$  é o número de pontos removidos. Assim, se estará habilitado para conseguir um bom tempo de execução para os métodos que mantêm um conjunto de máximos.

## 9.6 Exercícios

Para obter auxílio e o código fonte dos exercícios, visite [java.datastructures.net](http://java.datastructures.net).

### Reforço

- R-9.1 Qual é o pior caso de tempo de execução para inserções de  $n$  elementos chave-valor em um mapa  $M$  inicialmente vazio que é implementado com uma seqüência?
- R-9.2 Descreva como usar um mapa para implementar o TAD dicionário, assumindo que o usuário não tente inserir elementos com a mesma chave.
- R-9.3 Descreva como uma seqüência ordenada implementada como uma lista duplamente encadeada poderia ser usada para implementar o TAD mapa.
- R-9.4 Qual seria um bom código de hash para um número de identificação de veículo que é uma cadeia de caracteres representando números e letras no formato “9X9XX99X9XX999999,” onde um “9” representa um dígito e um “X” representa uma letra?
- R-9.5 Desenhe a tabela de hash com 11 elementos, que resulta a partir do uso da função de hash,  $h(i) = (21 + 5) \bmod 11$ , para colocar as chaves 12, 44, 13, 88, 23, 94, 11, 39, 20, 16 e 5, assumindo que as colisões serão tratadas por encadeamento.
- R-9.6 Qual será o resultado do exercício anterior se assumirmos que as colisões serão tratadas por teste linear?

- R-9.7 Mostre o resultado do Exercício R-9.5, assumindo que as colisões são tratadas por teste quadrático, até o ponto em que o método falha.
- R-9.8 Qual o resultado do Exercício R-9.5, assumindo que as colisões são tratadas por *hashing duplo* usando uma função de hash secundária  $h'(k) = 7 - (k \bmod 7)$ ?
- R-9.9 Forneça uma descrição em pseudocódigo da inserção em uma tabela de hash que usa teste quadrático para resolver colisões, assumindo que se usa o truque de substituir elementos deletados com um objeto indicando “item desativado”.
- R-9.10 Forneça uma descrição de Java dos métodos `values()` e `entries()` que poderiam ser incluídas na implementação da tabela de hash apresentada nos Trechos de código 9.3 – 9.5
- R-9.11 Explique como modificar a classe `HashMap`, apresentada nos Trechos de códigos 9.3 – 9.5, para implementar o TAD dicionário em vez do TAD mapa.
- R-9.12 Mostre o resultado de fazer *rehash* na tabela de hash, mostrada na Figura 9.4, para uma tabela de tamanho 19, usando a nova função de hash  $h(k) = 2k \bmod 19$ .
- R-9.13 Discuta porque uma tabela de hash não é adaptada para implementar um dicionário ordenado.
- R-9.14 Qual é o pior tempo para inserir  $n$  elementos em uma tabela de hash inicialmente vazia, com colisões sendo resolvidas por encadeamento? Qual seria o melhor caso?
- R-9.15 Desenhe a skip list resultante da execução da seguinte sequência de operações sobre a skip list da Figura 9.12: `remove(38)`, `insert(48,x)`, `insert(24,y)`, `remove(55)`. Registre as jogadas de cara e coroa.
- R-9.16 Apresente a descrição de um pseudocódigo da operação de remoção em uma skip list.
- R-9.17 Qual é o tempo de execução esperado dos métodos para manutenção de um conjunto de máximos inserindo-se  $n$  pares tal que cada par tenha o menor custo e desempenho que um anterior a ele? O que estará contido em um dicionário ordenado ao final desta série de operações? Se um par tem o menor custo e maior desempenho qual será o anterior a ele?
- R-9.18 Argumente por que os localizadores não são realmente necessários para um dicionário implementado com uma boa tabela de hash.

---

### Criatividade

- C-9.1 Descreva como usar um mapa para implementar o TAD dicionário, assumindo que o usuário pode tentar inserir elementos com a mesma chave.
- C-9.2 Suponha que são dadas duas tabelas de pesquisa ordenada  $S$  e  $T$ , cada qual com  $n$  elementos (com  $S$  e  $T$  sendo implementadas com arranjos). Descreva um algoritmo  $O(\log^2 n)$  para encontrar a  $k$ -ésima menor chave na união das chaves de  $S$  e  $T$  (assumindo que não há chaves duplicadas).
- C-9.3 Apresente uma solução  $O(\log n)$  para o problema anterior.

Hidden page

- C-9.13 Suponha que cada linha de um arranjo  $A$  de tamanho  $n \times n$  consiste em 1 e 0, tal que em qualquer linha de  $A$  todos os valores 1 venham antes de todos os valores 0. Assumindo que  $A$  esteja em memória, descreva um método com tempo  $O(n \log n)$  (e não tempo  $O(n^2)!$ ) para contar o número de valores 1 em  $A$ .
- C-9.14 Descreva uma estrutura ordenada eficiente para um dicionário que armazena  $n$  elementos e tem um conjunto de ordem total associado de  $k < n$  chaves. Ou seja, o conjunto de chaves é menor que o elemento. Sua estrutura deverá executar a operação `findAll` no tempo esperado de  $O(\log r + s)$ , onde  $s$  é o número de elementos retornados, a operação `entries()` no tempo  $O(n)$  e operações restantes do TAD dicionário no tempo esperado de  $O(\log r)$ .
- C-9.15 Descreva uma estrutura eficiente de dicionário para armazenar  $n$  elementos com  $r < n$  e chaves que contenham distintos códigos de hash. Sua estrutura deverá executar a operação `findAll` no tempo esperado de  $O(1 + s)$ , onde  $s$  é o número de elementos retornados, e a operação `entries()` no tempo  $O(n)$ , e operações restantes do TAD dicionário no tempo esperado de  $O(1)$ .
- C-9.16 Descreva uma eficiente estrutura de dados para implementar o TAD *sacola*, o qual suporta um método `add(e)`, para adicionar um elemento arbitrário  $e$  na sacola, e um método `remove()`, o qual remove um elemento arbitrário da sacola. Mostre que ambos os métodos podem ser feitos no tempo  $O(1)$ .
- C-9.17 Descreva como modificar uma skip list para suportar o método `atIndex(i)`, que retorna a posição do elemento “base” na lista  $S_0$  com colocação  $i$ , pois  $i \in [0, n - 1]$ . Mostre que sua implementação deste método tem tempo esperado  $O(\log n)$ .

## Projetos

- P-9.1 Implemente uma classe que implemente o TAD dicionário pela adaptação da classe `java.util.HashMap`
- P-9.2 Implemente o TAD dicionário com uma tabela de hash que trata as colisões com encadeamento separado (não adapte qualquer classe do pacote `java.util`).
- P-9.3 Implemente o TAD dicionário ordenado usando uma seqüência ordenada.
- P-9.4 Implemente os métodos de um TAD dicionário ordenado usando uma skip list.
- P-9.5 Estenda o projeto anterior fornecendo uma animação gráfica da operação da skip list. Visualize como os itens se movem para cima na skip list durante a operação de inserção e como são retirados durante a remoção. Na operação de procura, visualize as varreduras e descidas para o nível inferior.
- P-9.6 Implemente um dicionário que suporta métodos baseados em localizadores através de uma seqüência ordenada
- P-9.7 Faça uma análise comparativa que estuda as taxas de colisão para vários códigos de hash para cadeias de caracteres, tais como códigos hash polinomiais para diferentes valores do parâmetro  $a$ . Use uma tabela de hash para determinar colisões, mas apenas conte colisões em que cadeias diferentes levam ao mesmo código de hash (e não se elas levam à mesma posição da tabela de hash). Teste os códigos hash em arquivos encontrados na Internet.

- P-9.8 Faça uma análise comparativa, como no exercício anterior, para números de telefone de dez dígitos, em vez de cadeias de caracteres.
- P-9.9 Projete uma classe Java que implemente a estrutura de dados skip list. Use esta classe para criar implementações de TAD mapa e TAD dicionário, incluindo métodos localizadores para o dicionário.

---

### Observações sobre o capítulo

É interessante notar que o algoritmo de pesquisa binária foi publicado pela primeira vez em 1946, mas só foi publicado em uma versão completamente correta em 1962. Para mais discussões sobre as lições a serem aprendidas desta história, veja o livro de Knuth [60] e os artigos de Bentley [12] e Levisse [65]. As skip lists foram introduzidas por Pugh [84]. A presente análise das skip lists é uma simplificação da apresentação feita no livro de Motwani e Raghavan [79]. Além disso, a discussão de hashing também é uma simplificação da apresentação daquele livro. O leitor interessado em outras construções probabilísticas para suportar dicionários (incluindo mais informação sobre hashing) pode verificar o livro de Motwani e Raghavan [79]. Para uma análise mais profunda sobre skip lists, o leitor pode consultar os artigos na literatura de estruturas de dados [57,81,82]. O Exercício C-9.9 foi uma contribuição de James Lee.

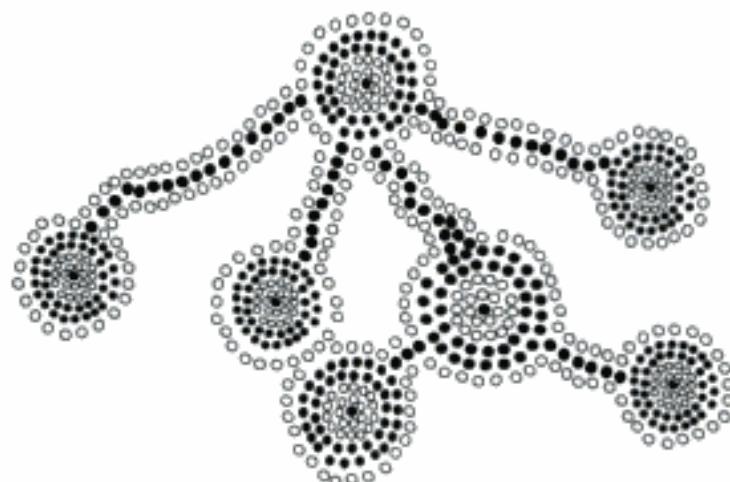
Hidden page

# Capítulo —

# 10

# Árvores de Pesquisa

---



## Conteúdo

---

<b>10.1 Árvores binárias de pesquisa .....</b>	<b>374</b>
10.1.1 Pesquisa .....	374
10.1.2 Operações de atualização.....	376
10.1.3 Implementação Java .....	379
<b>10.2 Árvores AVL .....</b>	<b>383</b>
10.2.1 Operações de atualização.....	385
10.2.2 Implementação Java .....	388
<b>10.3 Árvores splay .....</b>	<b>392</b>
10.3.1 Espalhamento .....	392
10.3.2 Quando espalhar .....	393
10.3.3 Análise amortizada do espalhamento ★ .....	396
<b>10.4 Árvores (2,4) .....</b>	<b>401</b>
10.4.1 Árvore genérica de pesquisa .....	401
10.4.2 Operações de atualização em árvores (2,4) .....	405
<b>10.5 Árvores vermelho-pretas.....</b>	<b>410</b>
10.5.1 Operações de atualização .....	412
10.5.2 Implementação Java .....	420
<b>10.6 Exercícios .....</b>	<b>425</b>

## 10.1 Árvores binárias de pesquisa

Todas as estruturas que serão discutidas neste capítulo são **árvores de pesquisa**, isto é, estruturas de dados árvore que podem ser usadas para implementar um dicionário. Segue uma breve revisão dos métodos fundamentais do TAD dicionário:

- `find(k)`: Retorna um elemento com chave *k*, se ele existir.
- `findAll(k)`: Retorna uma coleção de todos os elementos com chave igual a *k*.
- `insert(k, x)`: Insere um elemento com chave *k* e valor *x*.
- `remove(e)`: Remove um elemento *e*, retornando-o.
- `removeAll(k)`: Remove todos os elementos com chave *k*, retornando um iterador dos seus valores.

O método `find` retorna `null` se *k* não for encontrada. O TAD dicionário ordenado inclui alguns métodos adicionais para pesquisar usando predecessores e sucessores de uma chave ou elemento, mas seus desempenhos são similares ao método `find`. Assim, este capítulo será focado no método `find` como operação de pesquisa básica.

Árvores binárias são estruturas de dados excelentes para armazenar elementos de um dicionário, assumindo que se tem uma relação de ordem definida entre as chaves. Como mencionado anteriormente (Seção 7.3.6), uma **árvore binária de pesquisa** é uma árvore binária *T* em que cada nodo interno *v* de *T* armazena um elemento (*k*, *x*) que:

- As chaves armazenadas na subárvore esquerda de *v* são menores ou iguais a *k*.
- As chaves armazenadas na subárvore direita de *v* são maiores ou iguais a *k*.

Como mostrado abaixo, as chaves armazenadas nos nodos de *T* provêem uma forma de execução de uma pesquisa pela comparação de cada nodo interno *v*, o qual pode parar em *v* ou continuar com os filhos à esquerda ou à direita de *v*. Assim, armazenam-se elementos *x* somente na árvore interna da árvore binária de pesquisa, e os nodos externos servem somente como placeholders. Esta abordagem simplifica vários de nossos algoritmos de pesquisa ou alteração. Casualmente, poderiam ser permitidas árvores binárias de pesquisa impróprias, as quais usam melhor o espaço, mas às custas de métodos de pesquisa e atualizações mais complicados.

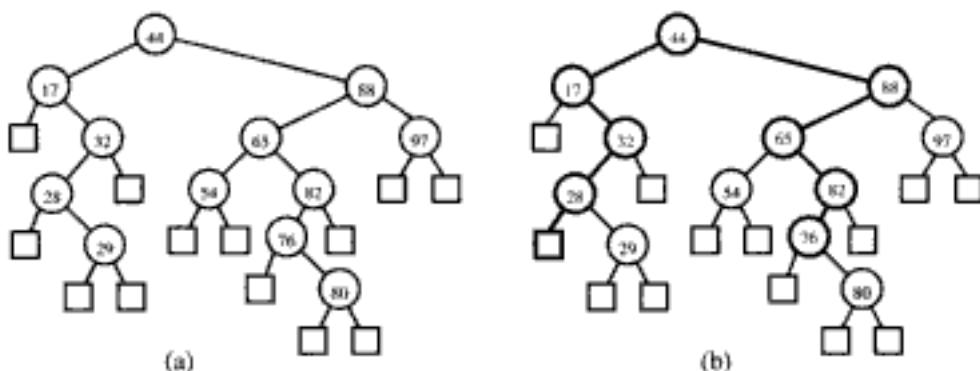
Independentemente de as árvores binárias de pesquisas serem ou não próprias, a propriedade importante de uma árvore binária de pesquisa é a concretização de um dicionário ordenado (ou mapa). Isto é, uma árvore binária de pesquisa deve representar hierarquicamente a ordenação de suas chaves, usando relacionamentos entre pais e filhos. Especificamente, um caminhamento interfixado (Seção 7.3.6) dos nodos de uma árvore binária de pesquisa *T* deverá visitar as chaves em ordem não-decrescente.

---

### 10.1.1 Pesquisa

Para executar a operação `find(k)` em um dicionário *D* que é representado por uma árvore binária de pesquisa *T*, enxerga-se a árvore *T* como uma árvore de decisão (lembre-se da Figura 7.10). Neste caso, a questão feita para cada nodo interno *v* de *T* é se a chave de pesquisa *k* é maior, menor ou igual que a chave armazenada no nodo *v*, denotado `key(v)`. Se a resposta for “menor”, então a pesquisa continua na subárvore esquerda. Se a resposta for “igual”, então a pesquisa terminou com sucesso. Se a resposta for “maior”, então a pesquisa continua na subárvore direita. Finalmente, encontrando-se um nodo externo, então a pesquisa se encerra com falha. (Ver Figura 10.1.)

Esta abordagem é descrita em detalhes no Trecho de código 10.1. Dada uma chave de pesquisa *k* e um nodo *v* de *T*, o método `TreeSearch` retorna um nodo (posição) *w* da subárvore *T(v)* enraizada em *v*, de maneira que um dos dois casos ocorre:



**Figura 10.1** (a) Uma árvore binária de pesquisa  $T$  representando um dicionário  $D$  com chaves inteiros; (b) nodos de  $T$  visitados quando da execução das operações  $\text{find}(76)$  (com sucesso) e  $\text{find}(25)$  (sem sucesso) em  $D$ . Para simplicidade, são mostradas as chaves mas não os valores dos elementos.

- $w$  é um nodo interno que armazena a chave  $k$ ;
- $w$  é um nodo externo representando a posição de  $k$  em um caminhamento interfixado de  $T(v)$ , mas  $k$  não é uma chave contida em  $T(v)$ .

Deste modo, o método  $\text{find}(k)$  pode ser executado chamando-se o método  $\text{TreeSearch}(k, T, \text{root}())$ . Seja  $w$  o nodo de  $T$  retornado por esta chamada ao método  $\text{TreeSearch}$ . Se o nodo  $w$  for interno, retorna-se o elemento armazenado em  $w$ ; por outro lado, se  $w$  for externo, então se retorna **null**.

**Algoritmo**  $\text{TreeSearch}(k, v)$ :

```

se  $T.\text{isExternal}(v)$  então
    retorna  $v$ 
se  $k < \text{key}(v)$  então
    retorna  $\text{TreeSearch}(k, T.\text{left}(v))$ 
senão se  $k > \text{key}(v)$  então
    retorna  $\text{TreeSearch}(k, T.\text{right}(v))$ 
retorna  $v$       {conhece-se  $k = \text{key}(v)$ }
```

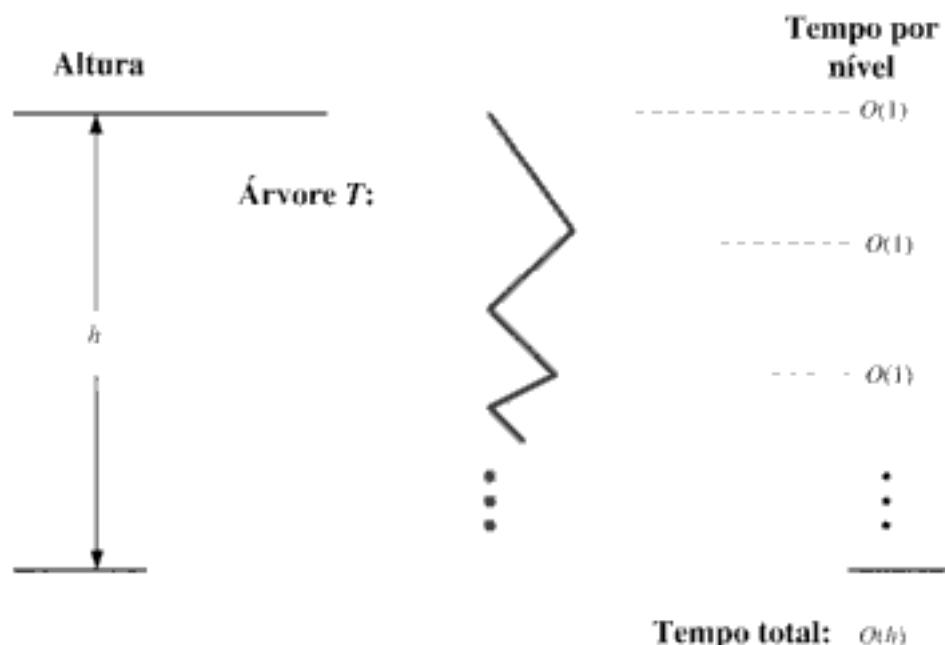
**Trecho de código 10.1** Pesquisa recursiva em uma árvore binária de pesquisa.

### Análise da árvore de pesquisa binária

A análise do pior caso para o tempo de execução de uma pesquisa em uma árvore de pesquisa binária  $T$  é simples. O algoritmo  $\text{TreeSearch}$  é recursivo, e executa um número constante de operações primitivas em cada chamada recursiva. Cada chamada recursiva de  $\text{TreeSearch}$  é feita sobre um filho do nodo anterior. Isto é,  $\text{TreeSearch}$  é chamada nos nodos de um caminho de  $T$  que inicia na raiz e desce um nível por vez. Assim, o número de nodos é limitado por  $h + 1$ , onde  $h$  é a altura de  $T$ . Em outras palavras, uma vez que se gasta um tempo  $O(1)$  em cada nodo encontrado na pesquisa, o método  $\text{find}$  sobre um dicionário  $D$  executa em tempo  $O(h)$ , onde  $h$  é a altura da árvore de pesquisa binária  $T$  usada para implementar  $D$ . (Ver Figura 10.2.)

Pode-se demonstrar também que uma variação do algoritmo acima executa a operação  $\text{findAll}(k)$  em tempo  $O(h + s)$ , onde  $s$  é o número de elementos no iterador retornado. Entretanto, este método é um pouco mais complicado e seus detalhes ficam como um exercício (C-10.1).

Na verdade, a altura  $h$  de  $T$  pode ser grande como  $n$ , mas espera-se que normalmente seja menor. Além disso, será mostrado como manter o limite superior de  $O(\log n)$  usando a altura da árvore de pesquisa  $T$  da Seção 10.2. Antes de se apresentar tal esquema, entretanto, serão descritas implementações de método de atualização de dicionários.



**Figura 10.2** Demonstra o tempo de execução de uma pesquisa sobre uma árvore binária de pesquisa. A figura usa as formas de representação padrão, visualizando uma árvore binária de pesquisa como um grande triângulo, e o caminho a partir da raiz como uma linha em zigue-zague.

### 10.1.2 Operações de atualização

Árvores binárias de pesquisa permitem implementações de operações de inserção e remoção usando algoritmos que são mais diretos, mas não triviais.

#### Inserção

Uma árvore binária de pesquisa  $T$  suporta as seguintes operações de atualização:

**inserAtExternal( $v, e$ ):** insere o elemento  $e$  no nodo externo  $v$ , expande  $v$  para ser interno, tendo um novo (e vazio) filho do nodo externo; um erro ocorre se  $v$  é um nodo interno.

Dado este método, pode-se executar o método `insert( $k, x$ )` para um dicionário implementado com uma árvore binária de pesquisa  $T$  chamando `TreeInsert( $k, x, T.root()$ )`, o qual é apresentado no Trecho de código 10.2.

#### Algoritmo TreeInsert( $k, x, v$ ):

**Entrada:** uma chave de pesquisa  $k$ , um valor associado  $x$  e um nodo  $v$  de  $T$

**Saída:** Um novo nodo  $w$  na subárvore  $T(v)$  que armazena o elemento  $(k, v)$

$W \leftarrow \text{TreeSearch}(k, v)$

**se**  $k = \text{key}(w)$  **então** {a chave  $w$  é igual a  $k$ , desta forma faça uma chamada recursiva para um filho}

**retorna** `TreeInsert( $k, x, T.\text{left}(v)$ )` {poderia ser utilizado o filho da direira}

$T.\text{insertAtExternal}(w, (k, x))$  {este é o local apropriado para inserir  $(k, x)$ }

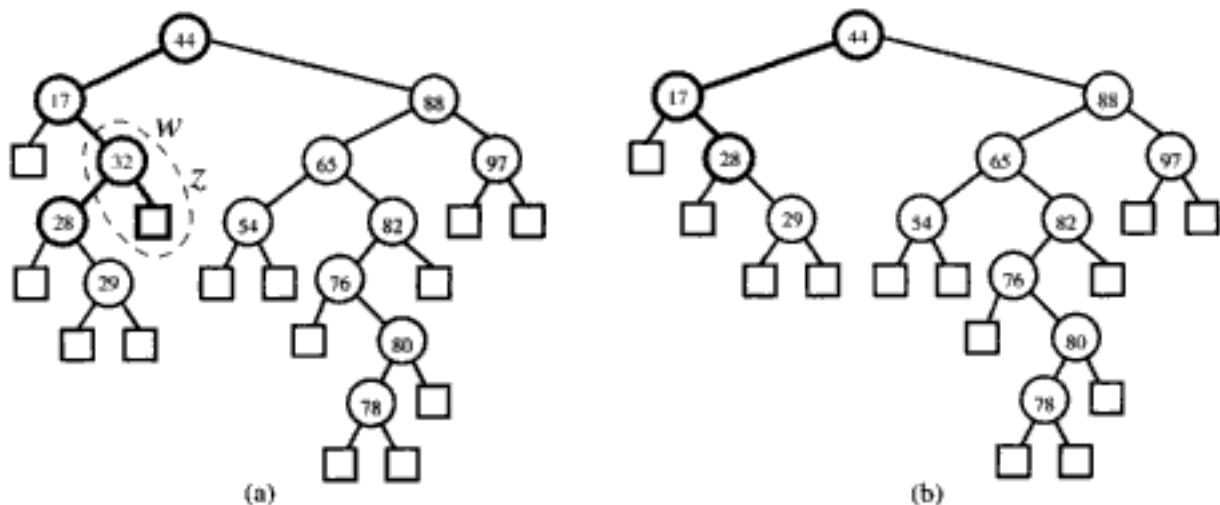
**retorna**  $w$

**Trecho de código 10.2** Algoritmo recursivo para inserção em uma árvore binária de pesquisa.

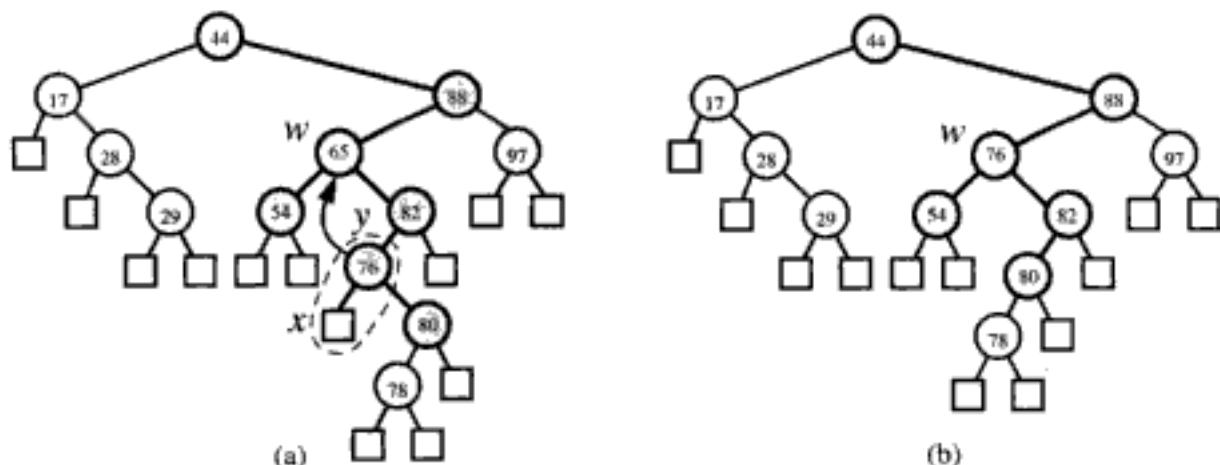
Hidden page

- Retorna-se o elemento previamente armazenado em  $w$ , que se salva na variável temporária  $t$ .

Como com pesquisa e inserção, este algoritmo de remoção cria um caminho a partir da raiz para um modo externo, possivelmente movendo um elemento entre dois nodos deste caminho, e então executa a operação `removeExternal` para o nodo externo.



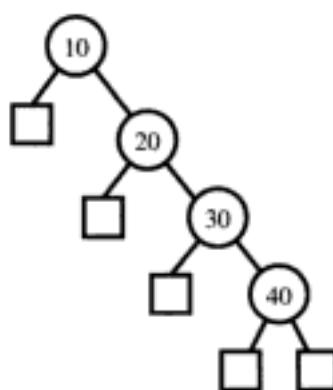
**Figura 10.4** Remoção da árvore de pesquisa binária da Figura 10.3b, na qual a chave a ser removida (32) está armazenada no nodo ( $w$ ) com um filho externo: (a) antes da remoção; (b) após a remoção.



**Figura 10.5** Remoção da árvore binária de pesquisa da Figura 10.3b, na qual o elemento a ser removido (com chave 65) está armazenado no nodo ( $w$ ) cujos filhos são internos: (a) antes da remoção; (b) após a remoção.

### Desempenho da árvore binária de pesquisa

A análise do algoritmo de remoção é análoga à dos algoritmos de inserção e pesquisa. Gasta-se tempo  $O(1)$  em cada nodo visitado e, no pior caso, o número de nodos visitados é proporcional à altura  $h$  de  $T$ . Portanto, em um dicionário  $D$  implementado usando uma árvore de pesquisa binária  $T$ , os métodos `find`, `insert` e `remove` executam no tempo  $O(h)$ , onde  $h$  é a altura da árvore  $T$ . Assim, uma árvore binária de pesquisa  $T$  é pequena. No melhor caso,  $T$  tem altura  $h$ , tal que  $h = \lceil \log(n + 1) \rceil$ , o que resulta em uma performance logarítmica para todas as operações do dicionário. No pior caso, entretanto,  $T$  tem altura  $n$ ; consequentemente, sua aparência é a de uma seqüência ordenada de um dicionário. Esta configuração de pior caso ocorre, por exemplo, inserindo-se uma série de chaves em ordem crescente ou decrescente (ver Figura 10.6).



**Figura 10.6** Exemplo de uma árvore binária de pesquisa com altura linear, obtida pela inserção de elementos com chaves em ordem crescente.

O desempenho de um dicionário implementado com uma árvore binária de pesquisa é resumida na seguinte proposição e na Tabela 10.1.

**Proposição 10.1** *Uma árvore binária de pesquisa  $T$  com altura  $h$  para  $n$  elementos chave-valor usa o espaço  $O(n)$  e executa as operações do TAD dicionário com os seguintes tempos de execução. Operações size e isEmpty custam o tempo  $O(1)$  cada uma. Operações find, insert e remove levam o tempo  $O(h)$  cada uma. A operação findAll custa o tempo  $O(h + s)$ , onde  $s$  é o tamanho da coleção retornada.*

Método	Tempo
size, isEmpty	$O(1)$
find, insert, remove	$O(h)$
findAll	$O(h + s)$

**Tabela 10.1** Tempos de execução dos principais métodos de um dicionário implementado com uma árvore binária de pesquisa. Denota-se a altura da árvore como sendo  $h$  e o tamanho da coleção retornada pelo método findAll como  $s$ . O espaço usado é  $O(n)$ , onde  $n$  é o número de elementos armazenados no dicionário.

O tempo de execução das operações de pesquisa e atualização em uma árvore binária de pesquisa varia dramaticamente dependendo da altura da árvore. No entanto, na média, uma árvore binária de pesquisa com  $n$  chaves geradas a partir de uma série randômica de inserções e remoções de chaves tem a altura esperada de  $O(\log n)$ . Como um comando requer cuidado de uma linguagem matemática para precisar o que se quer prover, esta justificativa vai além do escopo deste livro. Apesar disso, é preciso manter em mente o pior caso de desempenho e tomar cuidado para o uso padrão de árvores binárias de pesquisa em aplicações nas quais as alterações não são randômicas. Após tudo isso, existem aplicações em que é essencial ter um dicionário com um rápido pior caso nos tempos de pesquisas e atualizações. A estrutura de dados apresentada na próxima seção endereça esta necessidade.

### 10.1.3 Implementação Java

Nos Trechos de códigos 10.3 a 10.5, é descrita uma classe de árvore binária de pesquisa, `BinarySearchTree`, a qual armazena objetos da classe `BSTEntry` (implementação da interface `Entry`) nos seus nodos. A classe `BinarySearchTree` estende a classe `LinkedBinaryTree` dos Trechos de código 7.16 a 7.18 e assim usando da vantagem do reuso de código.

Esta classe usou vários métodos auxiliares para fazer o trabalho pesado. O método auxiliar `treeSearch`, baseado no algoritmo TreeSearch (Trecho de código 10.1), é invocado pelos métodos `find`, `findAll` e `insert`. Usa-se um método `addAll` recursivo como o principal mecanismo para o método `findAll`, no qual executa um caminhamento interfixado de todos os elementos com chave igual a  $k$  (não através de um algoritmo rápido, visto que ele executa uma pesquisa falha para cada elemento que encontra). Usam-se dois métodos de atualização adicionais, `insertAtExternal`, o qual insere um nodo elemento em um nodo externo, e `removeExternal`, o qual remove um nodo externo e seus pais.

A classe `BinarySearchTree` usa localizadores (ver Seção 8.4.2). Desta forma, seus métodos de atualização informam a qualquer objeto `BSTEntry` alterado sua nova posição. Também usam-se vários métodos auxiliares simples para acessar e testar os dados, como `checkKey`, o qual verifica se a chave é válida (apesar de usar uma simples regra neste caso). Usa-se, também, uma variável de instância, `actionPos`, a qual armazena a posição onde a mais recente pesquisa, inserção ou remoção foi finalizada. Esta variável de instância não é necessária para a implementação de uma árvore binária de pesquisa, mas é útil para classes que estenderão a classe `BinarySearchTree` (ver Trechos de código 10.7, 10.8, 10.10 e 10.11) para identificar a posição onde a pesquisa, inserção ou remoção anterior ocorreu. A posição `actionPos` tem a intenção de provar o uso correto após a execução dos métodos `find`, `insert` e `remove`.

```
// Implementação de um dicionário com uma árvore binária de pesquisa
public class BinarySearchTree<K,V>
    extends LinkedBinaryTree<Entry<K,V>> implements Dictionary<K,V> {
    protected Comparator<K> C; // comparador
    protected Position<Entry<K,V>>
        actionPos; // pai do nodo inserido ou removido
    protected int numEntries = 0; // número de elementos
    /** Cria uma BinarySearchTree com um comparador padrão. */
    public BinarySearchTree() {
        C = new DefaultComparator<K>();
        addRoot(null);
    }
    public BinarySearchTree(Comparator<K> c) {
        C = c;
        addRoot(null);
    }
    /** Classe aninhada para o localizador dos elementos da árvore binária de pesquisa */
    protected static class BSTEntry<K,V> implements Entry<K,V> {
        protected K key;
        protected V value;
        protected Position<Entry<K,V>> pos;
        BSTEntry() { /* construtor padrão */ }
        BSTEntry(K k, V v, Position<Entry<K,V>> p) {
            key = k; value = v; pos = p;
        }
        public K getKey() { return key; }
        public V getValue() { return value; }
        public Position<Entry<K,V>> position() { return pos; }
    }
    /** Retorna a chave do elemento de um dado nodo da árvore. */
    protected K key(Position<Entry<K,V>> position) {
        return position.element().getKey();
    }
    /** Retorna o valor do elemento de um dado nodo da árvore. */
}
```

```

protected V value(Position<Entry<K,V>> position) {
    return position.element().getValue();
}
 $\ast\ast$  Retorna o elemento de um dado nodo da árvore.
protected Entry<K,V> entry(Position<Entry<K,V>> position) {
    return position.element();
}
 $\ast\ast$  Substitui um elemento por um novo elemento (e inicializa a localização do elementos)
protected void replaceEntry(Position<Entry<K,V>> pos, Entry<K,V> ent) {
    ((BSTEntry<K,V>) ent).pos = pos;
    replace(pos, ent);
}

```

**Trecho de código 10.3** Classe BinarySearchTree (continua no Trecho de código 10.4).

```

 $\ast\ast$  Verifica se uma determinada chave é válida.
protected void checkKey(K key) throws InvalidKeyException {
    if(key == null) // um simples teste
        throw new InvalidKeyException("chave nula");
}
 $\ast\ast$  Verifica se um determinado elemento é válido.
protected void checkEntry(Entry<K,V> ent) throws InvalidEntryException {
    if(ent == null || !ent instanceof BSTEntry)
        throw new InvalidEntryException("elemento inválido");
}
 $\ast\ast$  Método auxiliary para inserir um elemento em um nodo externo
protected Entry<K,V> insertAtExternal(Position<Entry<K,V>> v, Entry<K,V> e) {
    expandExternal(v,null,null);
    replace(v, e);
    numEntries++;
    return e;
}
 $\ast\ast$  Método auxiliary para remover um nodo externo e seu pai
protected void removeExternal(Position<Entry<K,V>> v) {
    removeAboveExternal(v);
    numEntries--;
}
 $\ast\ast$  Método auxiliary usado para pesquisar, inserir e remover.
protected Position<Entry<K,V>> treeSearch(K key, Position<Entry<K,V>> pos) {
    if (isExternal(pos)) return pos; // chave não encontrada; retorna o nodo externo
    else {
        K curKey = key(pos);
        int comp = C.compare(key, curKey);
        if (comp < 0)
            return treeSearch(key, left(pos)); // Pesquisa na subárvore à esquerda
        else if (comp > 0)
            return treeSearch(key, right(pos)); // Pesquisa na subárvore à direita
        return pos; // retorna o nodo interno onde a chave foi encontrada
    }
}
// Adiciona a L todos os elementos da subárvore enraizada em v, tendo as chaves iguais a k
protected void addAll(PositionList<Entry<K,V>> L,
                      Position<Entry<K,V>> v, K k) {
    if (isExternal(v)) return;

```

```

Position<Entry<K,V>> pos = treeSearch(k, v);
if (!isExternal(pos)) { // encontrou-se um elemento com chave igual a k
    addAll(L, left(pos), k);
    L.addLast(pos.element());           // adiciona elementos
    addAll(L, right(pos), k);
} // este algoritmo recursivo é simples, mas não é o mais rápido
}

```

**Trecho de código 10.4** Classe BinarySearchTree (continua no Trecho de código 10.5).

```

// métodos do TAD dicionário
public int size() { return numEntries; }
public boolean isEmpty() { return size() == 0; }
public Entry<K,V> find(K key) throws InvalidKeyException {
    checkKey(key); // pode lançar uma exceção InvalidKeyException
    Position<Entry<K,V>> curPos = treeSearch(key, root());
    actionPos = curPos;           // nodo onde a pesquisa finalizou
    if (isInternal(curPos)) return entry(curPos);
    return null;
}
public Iterable<Entry<K,V>> findAll(K key) throws InvalidKeyException {
    checkKey(key); // pode lançar uma exceção InvalidKeyException
    PositionList<Entry<K,V>> L = new NodePositionList<Entry<K,V>>();
    addAll(L, root(), key);
    return L;
}
public Entry<K,V> insert(K k, V x) throws InvalidKeyException {
    checkKey(k); // pode lançar uma exceção InvalidKeyException
    Position<Entry<K,V>> insPos = treeSearch(k, root());
    while (!isExternal(insPos)) // pesquisa iterativa para encontrar a posição de inserção
        insPos = treeSearch(k, left(insPos));
    actionPos = insPos;           // nodo onde o novo elemento está sendo inserido
    return insertAtExternal(insPos, new BSTEntry<K,V>(k, x, insPos));
}
public Entry<K,V> remove(Entry<K,V> ent) throws InvalidEntryException {
    checkEntry(ent); // pode lançar uma exceção InvalidEntryException
    Position<Entry<K,V>> remPos = ((BSTEntry<K,V>) ent).position();
    Entry<K,V> toReturn = entry(remPos); // elemento a ser retornado
    if (isExternal(left(remPos))) remPos = left(remPos); // left easy case
    else if (isExternal(right(remPos))) remPos = right(remPos); // right easy case
    else { // elemento está no nodo com filhos internos
        Position<Entry<K,V>> swapPos = remPos; // encontra o nodo movendo o elemento
        remPos = right(swapPos);
        do
            remPos = left(remPos);
        while (isInternal(remPos));
        replaceEntry(swapPos, (Entry<K,V>) parent(remPos).element());
    }
    actionPos = sibling(remPos); // irmãos da folha a ser removida
    removeExternal(remPos);
    return toReturn;
}
// método entries() é omitido aqui

```

**Trecho de código 10.5** Classe BinarySearchTree (continuação do Trecho de código 10.4).

Hidden page

árvores AVL com o número mínimo de nodos: um com altura  $h - 1$  e o outro com altura  $h - 2$ . Levando a raiz em consideração, a seguinte fórmula relaciona  $n(h)$  com  $n(h - 1)$  e  $n(h - 2)$ , para  $h \geq 3$ :

$$n(h) = 1 + n(h - 1) + n(h - 2). \quad (10.1)$$

Neste ponto, o leitor familiarizado com as propriedades de uma progressão Fibonacci (ver Seção 2.2.3 e Exercício C-4.12) verá que  $n(h)$  é uma função exponencial em  $h$ . Para os demais leitores, se prosseguirá com este raciocínio.

A Fórmula 10.1 implica que  $n(h)$  é uma função crescente de  $h$ . Desta forma, sabe-se que  $n(h - 1) > n(h - 2)$ . Substituindo  $n(h - 1)$  por  $n(h - 2)$  na Fórmula 10.1 e descartando o 1, obtém-se, para  $h \geq 3$ ,

$$n(h) > 2 \cdot n(h - 2). \quad (10.2)$$

A Fórmula 10.2 indica que  $n(h)$  no mínimo dobra cada vez que  $h$  cresce em 2, o que intuitivamente significa que  $n(h)$  cresce exponencialmente. Para mostrar este fato de uma maneira formal, aplica-se a Fórmula 10.2 repetidamente, revelando a seguinte série de desigualdades:

$$\begin{aligned} n(h) &> 2 \cdot n(h - 2) \\ &> 4 \cdot n(h - 4) \\ &> 8 \cdot n(h - 6) \\ &\vdots \\ &> 2^i \cdot n(h - 2i). \end{aligned} \quad (10.3)$$

Ou seja,  $n(h) > 2^i \cdot n(h - 2i)$ , para qualquer inteiro  $i$ , tal que  $h - 2i \geq 1$ . Uma vez que os valores de  $n(1)$  e  $n(2)$  já são conhecidos, pega-se  $i$  de maneira que  $h - 2i$  seja igual a 1 ou 2. Ou seja, usa-se

$$i = \left\lceil \frac{h}{2} \right\rceil - 1.$$

Substituindo o valor de  $i$  na Fórmula 10.3 obtém-se, para  $h \geq 3$ ,

$$\begin{aligned} n(h) &> 2^{\left\lceil \frac{h}{2} \right\rceil - 1} \cdot n\left(h - 2 \left\lceil \frac{h}{2} \right\rceil + 2\right) \\ &\geq 2^{\left\lceil \frac{h}{2} \right\rceil - 1} n(1) \\ &\geq 2^{\frac{h}{2} - 1}. \end{aligned} \quad (10.4)$$

Pegando os logaritmos de ambos os lados da Fórmula 10.4, resulta

$$\log n(h) > \frac{h}{2} - 1,$$

a partir do qual obtém-se

$$h < 2 \log n(h) + 2, \quad (10.5)$$

que implica que uma árvore AVL armazenando  $n$  chaves tem altura no mínimo  $2 \log n + 2$ . ■

Pela Proposição 10.2 e pela análise das árvores de pesquisa binária vista na Seção 10.1, as operações `find` e `findAll`, em um dicionário implementado usando-se uma árvore AVL, executam em tempo  $O(\log n)$  e  $O(\log n + s)$ , respectivamente, onde  $n$  é o número de itens no dicionário e  $s$  é o tamanho do iterador retornado por `findAll`. O aspecto importante que resta é mostrar como manter a propriedade da altura/balanceamento de uma árvore AVL depois de uma inserção ou remoção.

### 10.2.1 Operações de atualização

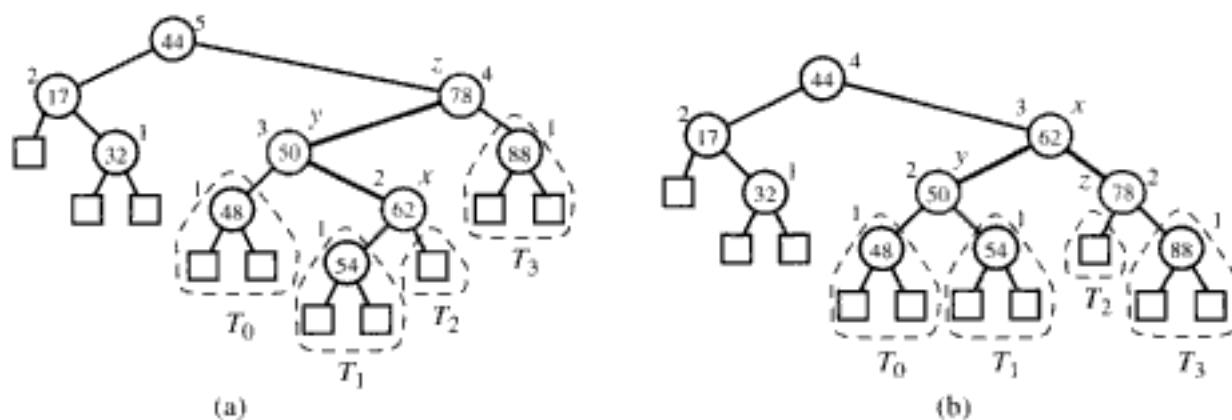
As operações de inserção e remoção para árvores AVL são similares àquelas para árvores binárias, mas com árvores AVL é preciso executar cálculos adicionais.

#### Inserção

Uma inserção em uma árvore AVL  $T$  inicia como em uma operação `insert`, descrita na Seção 10.1.2, para uma árvore binária de pesquisa (simples). Deve-se lembrar que essa operação sempre insere o novo item no nodo  $w$  de  $T$ , que foi previamente um nodo externo, e transforma  $w$  em nodo interno com a operação `insertAtExternal`. Isto é, adiciona dois nodos externos em  $w$ . Entretanto, essa ação pode violar a propriedade de balanceamento da altura, pois alguns nodos incrementam sua altura em um. Em particular, o nodo  $w$ , e possivelmente alguns de seus ancestrais, terão sua altura acrescida de um. Conseqüentemente, será descrito como reestruturar  $T$  para restaurar sua altura balanceada.

Dada uma árvore de pesquisa binária  $T$ , diz-se que um nodo  $v$  de  $T$  está **balanceado** se o valor absoluto da diferença entre as alturas dos filhos de  $v$  for no máximo 1, e diz-se que está **desbalanceado** no caso contrário. Então, caracterizar uma árvore AVL pela propriedade da altura/balanceamento equivale a dizer que todos os seus nodos internos estão平衡ados.

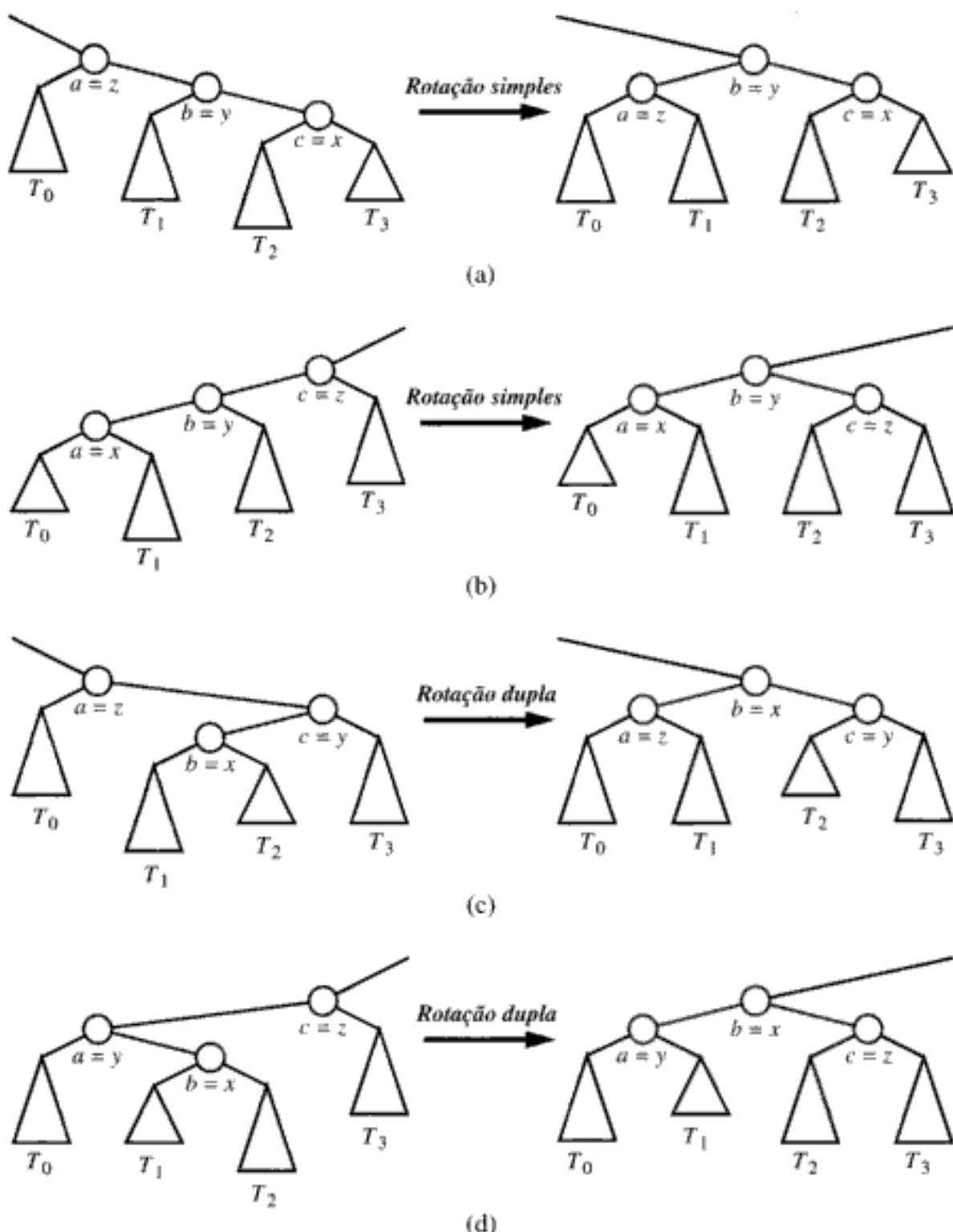
Suponha que  $T$  satisfaça a propriedade da altura/balanceamento e por isso é uma árvore AVL, antes de se inserir um novo item. Como mencionamos, depois de executar a operação `insertAtExternal` em  $T$ , as alturas de alguns nodos de  $T$ , incluindo  $w$ , crescem. Todos esses nodos estão no caminho de  $T$ , que parte de  $w$  e vai até a raiz de  $T$ , e são os únicos nodos de  $T$  que podem ter se desbalanceado. (Ver Figura 10.8a.) Naturalmente, se isso ocorrer, então  $T$  não será mais uma árvore AVL; conseqüentemente, necessita-se de um mecanismo para consertar o “desbalanceamento” recém-causado.



**Figura 10.8** Um exemplo de inserção de um elemento com chave 54 na árvore AVL da figura 10.7: (a) depois da inserção de um novo nodo para a chave 54, os nodos que armazenam as chaves 78 e 44 se tornam desbalanceados.; (b) uma reestruturação trinodo restaura a propriedade da altura/balanceamento. Mostram-se as alturas dos nodos próximos aos mesmos e identificam-se os nodos  $x$ ,  $y$  e  $z$  como participantes da reestruturação do trinodo.

O balanceamento dos nodos em uma árvore binária  $T$  é restaurado por meio de uma estratégia simples de “pesquise e conserte”. Em especial, faz-se  $z$  ser o primeiro nodo que se encontra indo para cima a partir de  $w$  em direção à raiz de  $T$ , de maneira que  $z$  fique desbalanceado (ver Figura 10.8a). Além disso, faz-se  $y$  denotar os filhos de  $z$  com uma altura maior (e observa-se que  $y$  tem de ser um ancestral de  $w$ ). Finalmente, faz-se  $x$  ser o filho de  $y$  com uma altura maior (e se houver um nó, escolhe-se  $x$  para ser o ancestral de  $w$ ). É importante observar que o nodo  $x$  por ser

Hidden page

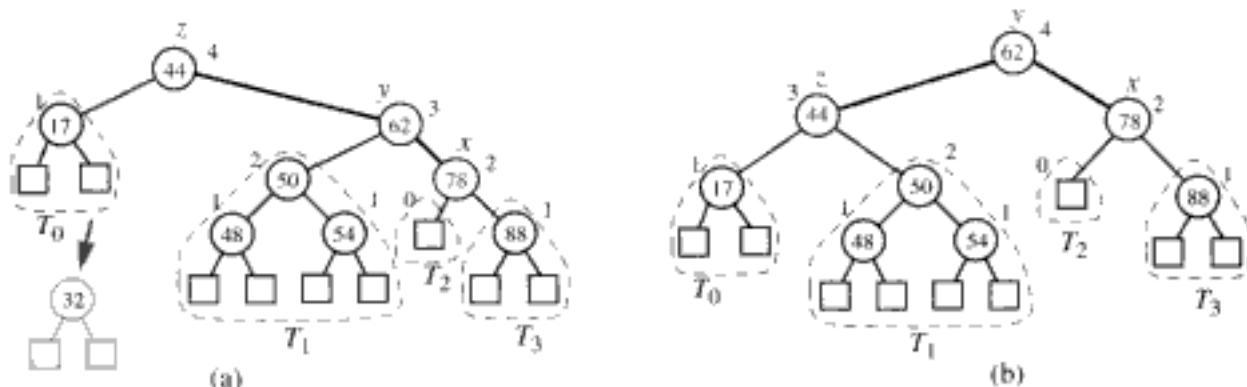


**Figura 10.9** Ilustração esquemática de uma operação de reestruturação trinodo (Trecho de código 10.6). As partes (a) e (b) mostram uma rotação simples, enquanto as partes (c) e (d) mostram uma rotação dupla.

### Remoção

Como foi no caso da operação de inserção no dicionário, inicia-se a implementação da operação de remoção em uma árvore AVL  $T$  pelo uso do algoritmo para executar esta operação em uma árvore binária de pesquisa. A dificuldade adicionada nesta abordagem com árvores AVL é que pode ser violada a propriedade da altura/balanceamento. Em particular, após remover um nodo interno com a operação `removeExternal` e elevar um de seus filhos para o seu lugar, pode ficar um nodo

não-balanceado em  $T$  no caminho a partir do pai  $w$  do nodo removido anteriormente para a raiz de  $T$ . (Ver Figura 10.10a.) De fato, pode existir somente um nodo não-balanceado no máximo. A justificativa deste fato é deixada como exercício (C-10.10).



**Figura 10.10** Remoção do elemento com chave 32 da árvore AVL da Figura 10.7: (a) após a remoção do nodo que armazena a chave 32, a raiz fica não-balanceada; (b) uma (simples) rotação restaura a propriedade da altura/balanceamento.

Usa-se a reestruturação trinodo para restaurar o balanceamento na árvore  $T$ , como na inserção. Em particular, sendo  $z$  o primeiro nodo não-balanceado encontrado a partir de  $w$  através da raiz de  $T$ . Além disso, sendo  $y$  o filho de  $z$  com uma grande altura (vide que  $y$  é o filho de  $z$ , que não é um ancestral de  $w$ ), e  $x$  sendo o filho de  $y$  definido como segue: se um dos filhos de  $y$  é mais alto que o outro, toma-se  $x$  como o filho mais alto de  $y$ ; senão (ambos os filhos de  $y$  tem a mesma altura), toma-se  $x$  como o filho de  $y$  no mesmo lado de  $y$  (isto é, se  $y$  é um filho à esquerda,  $x$  será um filho à esquerda de  $y$ , senão  $x$  será o filho à direita de  $y$ ). Neste caso, executa-se uma operação `restructure(x)`, que restaura a propriedade altura-balanceamento *localmente*, na subárvore que foi previamente enraizada em  $z$  e é agora enraizada no nodo que temporariamente se chama de  $b$ . (Ver Figura 10.10b.)

Desafortunadamente, esta reestruturação trinodo pode reduzir em 1 a altura da subárvore enraizada em  $b$ , o que pode causar que algum ancestral de  $b$  fique desbalanceado. Assim, depois do rebalanceamento de  $z$ , continua-se caminhando em  $T$  procurando por nodos desbalanceados. Encontrando-se algum, executa-se uma operação `restructure` para restaurar seu balanceamento. Ainda, desde que a altura de  $T$  seja  $O(\log n)$ , onde  $n$  é o número de elementos, pela Proposição 10.2,  $O(\log n)$  reestruturações trinodo são suficientes para restaurar a propriedade altura-balanceamento.

### Desempenho das árvores AVL

Segue-se um resumo da análise de desempenho de uma árvore AVL  $T$ . As operações `find`, `insert` e `remove` visitam o nodo junto com um caminho raiz-para-folha de  $T$ , mais, possivelmente, seus irmãos, e gastam o tempo  $O(1)$  por nodo. Assim, desde que a altura de  $T$  seja  $O(\log n)$  dada pela Proposição 10.2, cada uma das operações citadas gasta o tempo  $O(\log n)$ . Deixa-se a implementação e análise de uma versão eficiente da operação `findAll` como um interessante exercício. Na Tabela 10.2, resume-se o desempenho de um dicionário implementado com uma árvore AVL. Este desempenho é ilustrado na Figura 10.11.

### 10.2.2 Implementação Java

Volta-se a atenção agora para detalhes de implementação e analisa-se o uso de uma árvore AVL  $T$  com  $n$  nodos internos para implementar um dicionário ordenado de  $n$  itens. Os algoritmos

Hidden page

10.3–10.5) e inclui uma classe aninhada, `AVLNode`, que estende a classe `BTNode` usada para representar os nodos de uma árvore binária. A classe `AVLNode` define uma variável de instância adicional `height`, que representa a altura do nodo. Pega-se nossa árvore binária para usar esta classe de nodo em vez da classe `BTNode` simplesmente sobrecarregando o método `createNode`, o qual é usado exclusivamente para criar um novo nodo da árvore binária. A classe `AVLTree` herda os métodos `size`, `isEmpty`, `find` e `findAll` da superclasse, `BinarySearchTree`, mas sobrecarrega os métodos `insert` e `remove` para manter a árvore de pesquisa balanceada.

O método `insert` (Trecho de código 10.8) inicia chamando o método `insert` da superclasse, que insere o novo item e atribui a posição de inserção (nodo armazenando a chave 54, na Figura 10.8) para a variável de instância `actionPos`. O método auxiliar `rebalance` é então usado para percorrer o caminho desde a posição de inserção até a raiz. Este caminhamento atualiza a altura de todos os nodos visitados e executa uma reestruturação trinodo se necessário. Da mesma forma, o método `remove` (Trecho de código 10.8) se inicia chamando o método da superclasse `remove`, que executa a remoção do item e atribui a posição que substitui o item eliminado para a variável de instância `actionPos`. O método auxiliar `rebalance` é então usado para percorrer o caminho desde a posição removida até a raiz, executando qualquer necessidade de reestruturação.

```
/** Implementação de uma árvore AVL. */
public class AVLTree<K,V>
    extends BinarySearchTree<K,V> implements Dictionary<K,V> {
    public AVLTree(Comparator<K> c) { super(c); }
    public AVLTree() { super(); }
    /** classe aninhada para os nodos de uma árvore AVL. */
    protected static class AVLNode<K,V> extends BTNode<Entry<K,V>> {
        protected int height; // adiciona-se um campo height para um nodo BTNode
        AVLNode() {/* construtor padrão */}
        /* construtor preferido */
        AVLNode(Entry<K,V> element, BTPosition<Entry<K,V>> parent,
            BTPosition<Entry<K,V>> left, BTPosition<Entry<K,V>> right) {
            super(element, parent, left, right);
            height = 0;
            if (left != null)
                height = Math.max(height, 1 + ((AVLNode<K,V>) left).getHeight());
            if (right != null)
                height = Math.max(height, 1 + ((AVLNode<K,V>) right).getHeight());
        } // Assume-se que o pai revisará sua altura se necessário
        public void setHeight(int h) { height = h; }
        public int getHeight() { return height; }
    }
    /** Cria uma nova árvore binária de pesquisa (versão de sobrecarga). */
    protected BTPosition<Entry<K,V>> createNode(Entry<K,V> element,
        BTPosition<Entry<K,V>> parent, BTPosition<Entry<K,V>> left,
        BTPosition<Entry<K,V>> right) {
        return new AVLNode<K,V>(element, parent, left, right); // uso de nodos AVL
    }
    /** Retorna a altura de um nodo (retornando para um AVLNode). */
    protected int height(Position<Entry<K,V>> p) {
        return ((AVLNode<K,V>) p).getHeight();
    }
    /** Define a altura de um nodo interno (retornando para um AVLNode). */
    protected void setHeight(Position<Entry<K,V>> p) {
        ((AVLNode<K,V>) p).setHeight(1+Math.max(height(left(p)), height(right(p))));
    }
}
```

Hidden page

### 10.3 Árvores splay

Outra forma que se pode implementar as operações fundamentais de um dicionário é por meio de uma estrutura de dados para árvore de pesquisa balanceada conhecida como **árvore splay**. Esta estrutura é, sob o ponto de vista conceitual, totalmente diferente das outras árvores de pesquisa balanceadas discutidas neste capítulo: para uma árvore splay não se usa regras explícitas para forçar o seu balanceamento. Ao contrário, aplica-se uma certa operação mover-para-raiz, chamada *splaying*, após cada acesso, para manter a árvore de pesquisa balanceada em um senso amortizado. A operação *splaying* é executada no nodo  $x$  mais abaixo durante uma inserção, remoção ou uma pesquisa. A surpresa sobre o *splaying* é que este permite garantir uma amortizada no tempo de execução, para inserções, remoções e pesquisas, que é logarítmica. A estrutura da **árvore splay** é simplesmente uma árvore binária de pesquisa  $T$ . De fato, não existe altura, balanceamento ou cor do rótulo adicional que se associa com os nodos desta árvore.

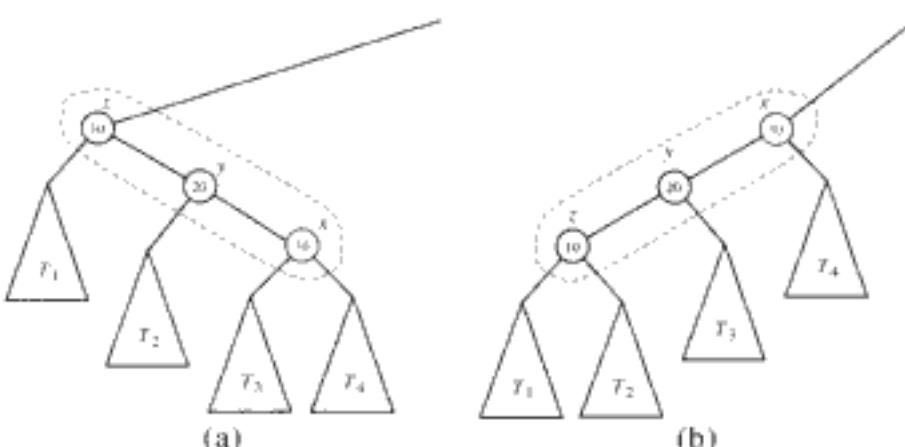
#### 10.3.1 Espalhamento

Dado um nodo interno  $x$  de uma árvore binária de pesquisa  $T$ , **aumenta-se**  $x$  pelo movimento de  $x$  para a raiz de  $T$  através de uma seqüência de reestruturações. As reestruturações particulares são executadas como importantes, e isso não é suficiente para mover  $x$  para a raiz de  $T$  justamente com qualquer reestruturação de seqüências. A operação específica é executada para mover  $x$  para cima, dependendo da posição relativa de  $x$ , seus pais  $y$  e  $e$  (caso exista) os seus avós  $z$ . Três casos são considerados:

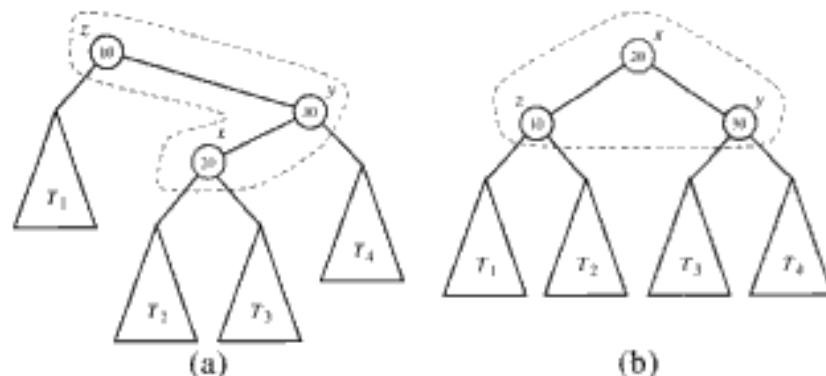
**zig-zig:** O nodo  $x$  e seus pais  $y$  são filho à esquerda ou filho à direita. (Ver Figura 10.12.) Troca-se  $z$  por  $x$ , fazendo com que  $y$  seja um filho de  $x$  e  $z$  seja um filho de  $y$ , enquanto mantém-se o relacionamento interfixado dos nodos de  $T$ .

**zig-zag:** Um de  $x$  e  $y$  é um filho à esquerda, e o outro é um filho à direita. (Ver Figura 10.13.) Neste caso, troca-se  $z$  por  $x$  e faz-se com que  $x$  tenha  $y$  e  $z$  como filhos, enquanto se reestruturam os relacionamentos interfixado dos nodos de  $T$ .

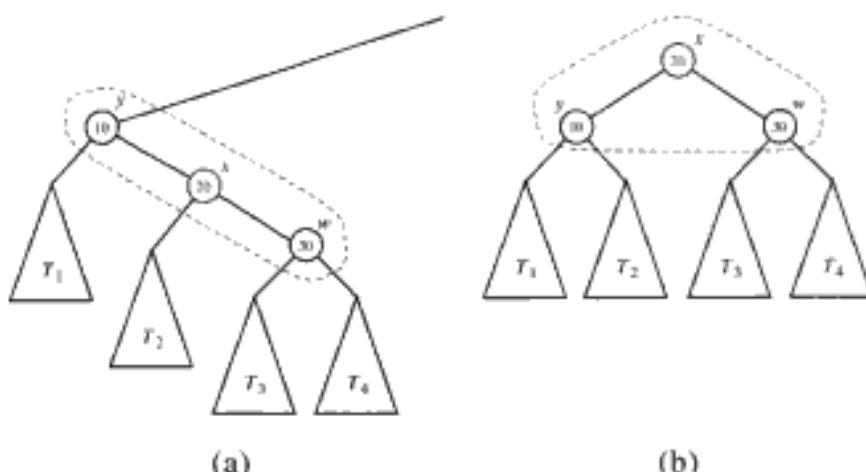
**zig:**  $x$  não tem avós (ou não se está considerando avós de  $x$  para algumas razões). (Ver Figura 10.14.) Neste caso, rotaciona-se  $x$  sobre  $y$ , fazendo com que os filhos de  $x$  sejam o nodo  $y$  e filho  $w$  de  $x$ , assim para manter o relacionamento interfixado relativo dos nodos de  $T$ .



**Figura 10.12** Zig-zig: (a) antes; (b) depois. Existe outra configuração simétrica onde  $x$  e  $y$  são filhos a esquerda.



**Figura 10.13** Zig-zag: (a) antes; (b) depois. Existe outra configuração simétrica onde  $x$  é um filho à direita e  $y$  é um filho à esquerda.



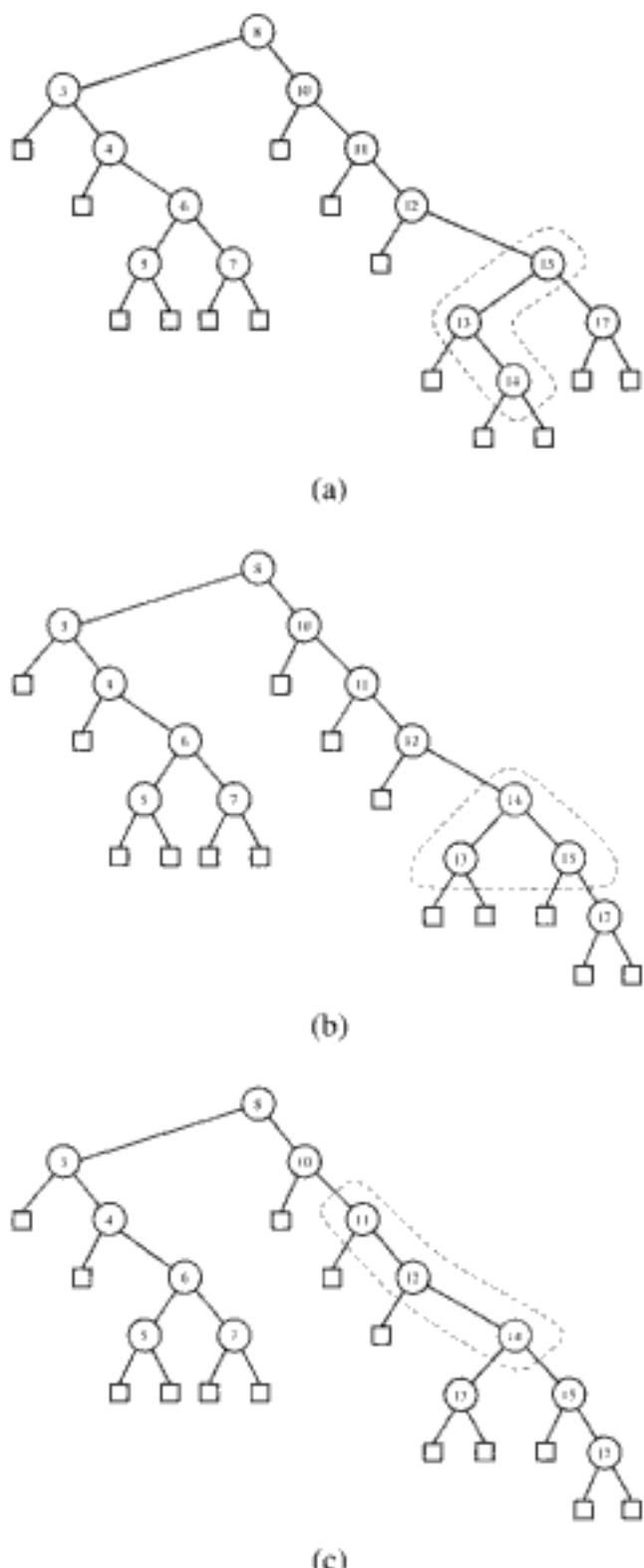
**Figura 10.14** Zig: (a) antes; (b) depois. Existe uma outra configuração simétrica onde  $x$  e  $w$  são filhos à esquerda.

Executa-se um zig-zig ou um zig-zag quando  $x$  tem um avô, e executa-se um zig quando  $x$  possui um pai, mas não possui avô. Um **espalhamento** consiste em repetir estas reestruturações de  $x$  até que  $x$  se torne a raiz de  $T$ . Deve-se observar que isso não é o mesmo que a seqüência de rotações simples que levam  $x$  para a raiz. Um exemplo de espalhamento de um nodo é mostrado nas Figuras 10.15 e 10.16.

### 10.3.2 Quando espalhar

As regras que ditam quando espalhar são executadas como segue:

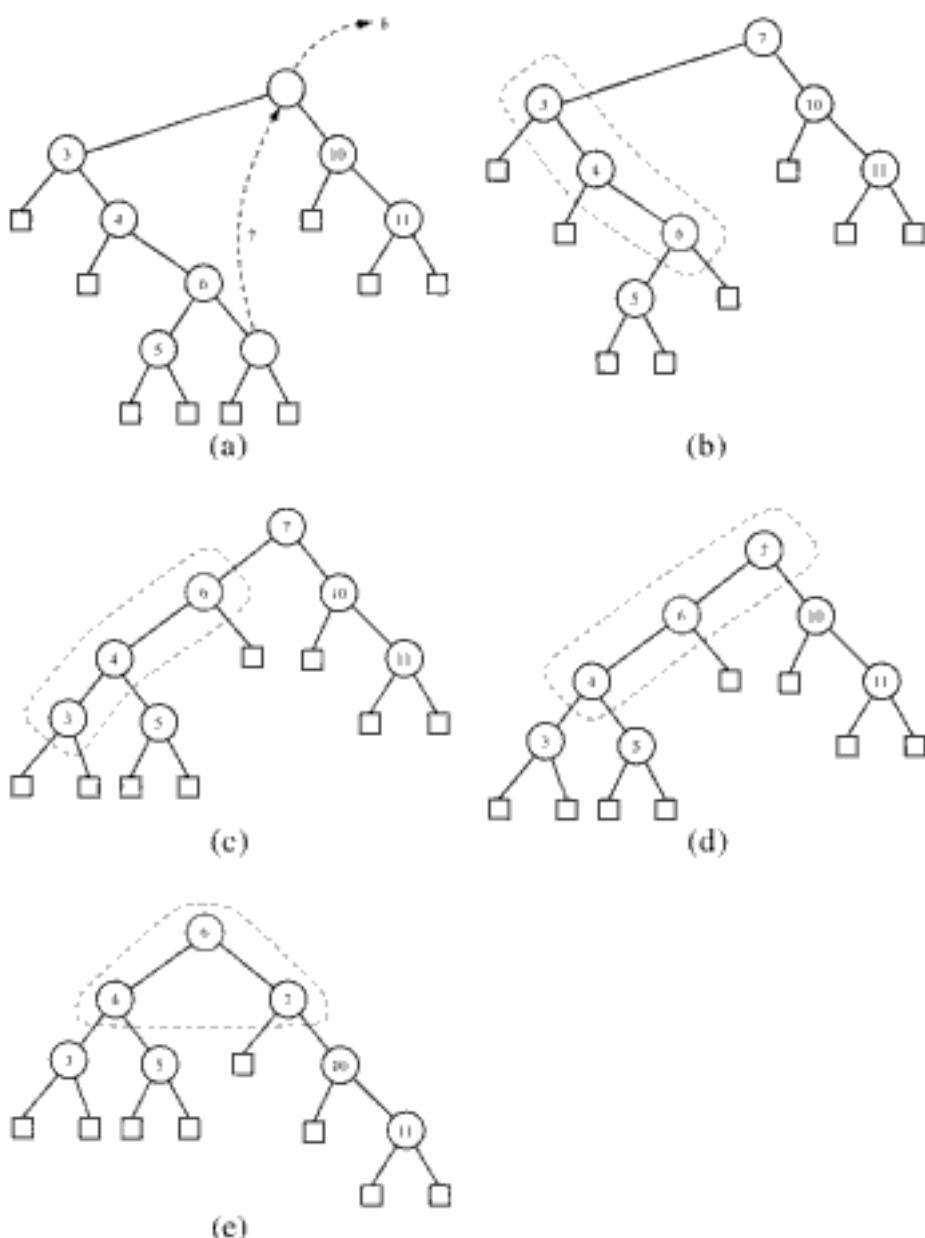
- Quando pesquisamos pela chave  $k$ , se  $k$  é encontrada em um nodo  $x$ , espalhamos  $x$ , senão espalhamos o pai de um nodo externo no qual termina-se a pesquisa sem sucesso. Por exemplo, os espalhamentos nas Figuras 10.15 e 10.16 seriam executados após a pesquisa ocorrer com sucesso para a chave 14, ou sem sucesso para a chave 14.5.
- Quando se insere a chave  $k$ , espalha-se o novo nodo interno criado onde  $k$  é inserido. Por exemplo, os espalhamentos das Figuras 10.15 e 10.16 seriam executados se 14 for a nova chave inserida. Mostra-se uma seqüência de inserções em uma árvore splay na Figura 10.17.
- Quando se remove uma chave  $k$ , espalha-se o pai do nodo  $w$  que é removido, isto é,  $w$  é o nodo que armazena a chave  $k$  ou é um de seus descendentes. (Deve-se lembrar o algoritmo de remoção das árvores binárias de pesquisa.) Um exemplo de espalhamento seguido de uma remoção é apresentado na Figura 10.18.



**Figura 10.15** Exemplo de espalhamento de um nodo: (a) espalhamento o nodo que armazena 14 iniciando com um zig-zag; (b) após o zig-zag; (c) próximo passo é um zig-zig. (Continua na Figura 10.16.)

Hidden page

Hidden page



**Figura 10.18** Remoção em uma árvore splay: (a) um remoção da chave 8 do nodo  $r$  é executada pela mudança de  $r$  para o nodo interno mais à direita  $v$ , na subárvore a esquerda de  $r$ , removendo  $v$ , e espalhando o pai  $u$  de  $v$ ; (b) expansão de  $u$  inicia com um zig-zig; (c) após o zig-zig; (d) o próximo passo é um zig; (e) após o zig.

### Desempenho amortizado das árvores splay

Para esta análise, deve-se observar que o tempo para execução de uma pesquisa, inserção ou remoção é proporcional ao tempo para ao espalhamento associado. Assim, considera-se somente o tempo de espalhamento.

Considere-se  $T$  uma árvore splay com  $n$  chaves, e  $v$  um nodo de  $T$ . Define-se o **tamanho**  $n(v)$  de  $v$  como o número de nodos em uma subárvore enraizada com  $v$ . Nota-se que esta definição implica que o tamanho de um nodo interno é maior que a soma dos tamanhos de seus dois filhos. Define-se a **classificação**  $r(v)$  de um nodo  $v$  como o logaritmo na base 2 do tamanho de  $v$ , isto é,  $r(v) = \log(n(v))$ . Claramente, a raiz de  $T$  tem o tamanho máximo ( $2n + 1$ ) e a classificação máxima,  $\log(2n + 1)$ , enquanto que cada nodo externo tem o tamanho 1 e classificação 0.

Usam-se ciberdólares para pagar pelo trabalho que se executa no espalhamento de um nodo  $x$  em  $T$  e assume-se que um ciberdólar paga um zig, enquanto que dois ciberdólares pagam para

um zig-zig ou um zig-zag. Então, o custo do espalhamento de um nodo com profundidade  $d$  é  $d$  ciberdólares. Mantém-se uma conta virtual que armazena os cyber-dollars de cada nodo interno de  $T$ . Deve-se observar que esta conta existe somente como proposta da análise amortizada, e não necessita ser incluída em uma estrutura de dados que implementa a árvore splay  $T$ .

### Uma análise contadora do espalhamento

Quando se executa um espalhamento, paga-se um certo número de ciberdólares (o valor exato do pagamento será determinado no final da análise). Distinguem-se três casos:

- Se o pagamento for igual ao trabalho de espalhar, então tudo é usado tudo para pagar a expansão.
- Se o pagamento for maior que o trabalho de espalhar, o excesso é depositado nas contas de diversos nodos.
- Se o pagamento for menor que o trabalho de espalhar, são feitas retiradas das contas de vários nodos para cobrir a deficiência.

Será mostrado, no resto desta seção, que um pagamento de  $O(\log n)$  ciberdólares por operação é suficiente para manter o sistema trabalhando, isto é, para assegurar que cada nodo mantenha na conta um balanço não-negativo.

### Uma invariante de ciberdólar para o espalhamento

Usa-se um esquema em que transferências são criadas entre as contas dos nodos para garantir que sempre existirão ciberdólares para retiradas para o pagamento do trabalho de espalhar quando necessário.

Para fazer o uso do método contador para executar nossa análise de espalhar, mantém-se a seguinte invariante:

*Antes e depois de um espalhamento, cada nodo  $v$  de  $T$  tem  $r(v)$  ciberdólares na sua conta.*

Deve-se observar que a invariante é "financeiramente sólida", visto que não requer que se crie um depósito preliminar para favorecer uma árvore sem chaves.

Seja  $r(T)$  a soma da classificação de todos os nodos de  $T$ . Para preservar a invariante após um espalhamento, deve-se fazer um pagamento igual ao trabalho de espalhar mais a alteração total de  $r(T)$ . Refere-se a uma simples operação zig, zig-zig ou zig-zag em um espalhamento como um *subpasso* de um espalhamento. Além disso, denota-se a classificação de um nodo  $v$  de  $T$  antes e depois de um subpasso do espalhamento com  $r(v)$  e  $r'(v)$ , respectivamente. A seguinte proposição apresenta um limite superior das alterações de  $r(T)$  causado por um simples subpasso do espalhamento. Será usado repetidamente este lema em nossa análise de um espalhamento completo de um nodo para a raiz.

**Proposição 10.3** *Seja  $\delta$  a variação de  $r(T)$  causada por um simples subpasso do espalhamento (um zig, zig-zig ou zig-zag) para um nodo  $x$  em  $T$ . Tem-se o seguinte:*

- $\delta \leq 3(r'(x) - r(x)) - 2$  se o subpasso for um zig-zig ou um zig-zag.
- $\delta \leq 3(r'(x) - r(x))$  se o subpasso for um zig.

**Justificativa** Usa-se o fato (ver Proposição A.1, Apêndice A) que, se  $a > 0$ ,  $b > 0$  e  $c > a + b$ ,

$$\log a + \log b \leq 2 \log c - 2. \quad (10.6)$$

Considere-se a alteração em  $r(T)$  causada por cada tipo de subpasso do espalhamento.

**zig-zig:** (Deve-se relembrar a Figura 10.12.) Visto que o tamanho de cada nodo é um a mais que o tamanho de seus dois filhos, nota-se que somente as classificações de  $x$ ,  $y$  e  $z$  alteram em uma operação zig-zig, onde  $y$  é o pai de  $x$  e  $z$  é pai de  $y$ . Além disso,  $r'(x) = r(z)$ ,  $r'(y) \leq r'(x)$  e  $r'(y) \geq r(x)$ . Assim

$$\begin{aligned}\delta &= r'(x) + r'(y) + r'(z) - r(x) - r(y) - r(z) \\ &\leq r'(y) + r'(z) - r(x) - r(y) \\ &\leq r'(x) + r'(z) - 2r(x).\end{aligned}\quad (10.7)$$

Vide que  $n(x) + n'(z) \leq n'(x)$ . Assim, em 10.6,  $r(x) + r'(z) \leq 2r'(x) - 2$ , que é,

$$r'(y) \leq 2r'(x) - r(x) - 2.$$

Esta desigualdade e 10.7 implicam

$$\begin{aligned}\delta &\leq r'(x) + (2r'(x) - r(x) - 2) - 2r(x) \\ &\leq 3(r'(x) - r(x)) - 2.\end{aligned}$$

**zig-zag:** (Deve-se relembrar a Figura 10.13.) Novamente, pela definição do tamanho e classificação, somente a classificação de  $x$ ,  $y$  e  $z$  mudam, onde  $y$  denota o pai de  $x$  e  $z$  denota o pai de  $y$ . Além disso,  $r'(x) = r(z)$  e  $r(x) \leq r(y)$ . Assim

$$\begin{aligned}\delta &= r'(x) + r'(y) + r'(z) - r(x) - r(y) - r(z) \\ &\leq r'(y) + r'(z) - r(x) - r(y) \\ &\leq r'(y) + r'(z) - 2r(x)\end{aligned}\quad (10.8)$$

Vide que  $n'(y) + n'(z) \leq n'(x)$ ; Então, pela 10.6,  $r'(y) + r'(z) \leq 2r'(x) - 2$ . Assim,

$$\begin{aligned}\delta &\leq 2r'(x) - 2 - 2r(x) \\ &\leq 3(r'(x) - r(x)) - 2.\end{aligned}$$

**zig:** (Deve-se relembrar a Figura 10.14.) Neste caso, somente a classificação de  $x$  e  $y$  alteram, onde  $y$  denota o pai de  $x$ . Além disso,  $r'(y) \leq r(y)$  e  $r'(x) \geq r(x)$ . Assim

$$\begin{aligned}\delta &= r'(y) + r'(x) - r(y) - r(x) \\ &\leq r'(x) - r(x) \\ &\leq 3(r'(x) - r(x)).\end{aligned}$$

**Proposição 10.4** Seja  $T$  uma árvore splay com raiz  $t$  e  $\Delta$ , a variação total de  $r(T)$  causada pelo espalhamento de um nodo  $x$  com profundidade  $d$ . Tem-se

$$\Delta \leq 3(r(t) - r(x)) - d + 2$$

**Justificativa** O espalhamento do nodo  $x$  consiste de  $p = \lceil d/2 \rceil$  subpassos do espalhamento, cada qual é um zig-zig ou zig-zag, exceto pelo último que é um zig se  $d$  for ímpar. Seja  $r_0(x) = r(x)$  a classificação inicial de  $x$ , e para  $i = 1, \dots, p$ , seja  $r_i(x)$  a classificação de  $x$  após o enésimo subpasso e  $\delta_i$  seja a variação de  $r(T)$  causado pelo enésimo subpasso. Pelo Lema 10.3, a variação total  $\Delta$  de  $r(T)$  causada pelo espalhamento de  $x$  é

$$\begin{aligned}\Delta &= \sum_{i=1}^p \delta_i \\ &\leq \sum_{i=1}^p (3(r_i(x) - r_{i-1}(x)) - 2) + 2 \\ &= 3(r_p(x) - r_0(x)) - 2p + 2 \\ &\leq 3(r(t) - r(x)) - d + 2.\end{aligned}$$

Pela Proposição 10.4, fazendo-se um pagamento de  $3(r(t) - r(x)) + 2$  ciberdólares através do espalhamento do nodo  $x$ , haverá ciberdólar suficientes para manter a invariante, mantendo  $r(v)$  ciberdólares em cada nodo  $v$  de  $T$ , pagando para todo o trabalho de espalhar, com custo de  $d$  dólares. Posto que o tamanho da raiz  $t$  é  $2n + 1$ , sua classificação será  $r(t) = \log(2n + 1)$ . Além disso, tem-se  $r(x) < R(t)$ . Assim, o pagamento a ser feito para o espalhamento será  $O(\log n)$  ciberdólares. Para completar nossa análise, tem-se computado o custo para manutenção da invariante quando um nodo é inserido ou removido.

Quando se insere um novo nodo  $v$  em uma árvore splay com  $n$  chaves, a classificação de todos os ancestrais de  $v$  são incrementados. Em outras palavras, seja  $v_0, v_1, \dots, v_d$  os ancestrais de  $v$ , onde  $v_0 = v$ ,  $v_i$  é o pai de  $v_{i+1}$ , e  $v_d$  é a raiz. Para  $i = 1, \dots, d$ , seja  $n'(v_i)$  e  $n(v_i)$  o tamanho de  $v_i$  antes e depois da inserção, respectivamente, e seja  $r'(v_i)$  e  $r(v_i)$  a classificação de  $v_i$  antes e depois da inserção, respectivamente. Tem-se

$$n'(v_i) = n(v_i) + 1.$$

Além disso, desde que  $n(v_i) + 1 \leq n(v_{i+1})$ , para  $i = 0, 1, \dots, d-1$ , tem-se o seguinte para cada  $i$  deste domínio:

$$r'(v_i) = \log(n'(v_i)) = \log(n(v_i) + 1) \leq \log(n(v_{i+1})) = r(v_{i+1}).$$

Assim, a variação total de  $r(T)$  causada pela inserção é

$$\begin{aligned} \sum_{i=1}^d (r'(v_i) - r'(v_i)) &\leq r'(v_d) + \sum_{i=1}^{d-1} (r(v_{i+1}) - r(v_i)) \\ &= r'(v_d) - r(v_0) \\ &\leq \log(2n + 1). \end{aligned}$$

Por esta razão, um pagamento de  $O(\log n)$  ciberdólares é suficiente para manter a invariante quando um novo nodo é inserido.

Quando se remove um nodo  $v$  de uma árvore splay com  $n$  chaves, a classificação de todos os ancestrais de  $v$  são decrementadas. Assim, a variação total de  $r(T)$  causada pela remoção é negativa, e não se precisa fazer nenhum pagamento para manter a invariante quando o nodo for removido. Então, pode-se resumir a análise amortizada na seguinte proposição (que algumas vezes é chamada de "Proposição de balanceamento" para árvores splay):

**Proposição 10.5** Consider-se uma seqüência de  $m$  operações em uma árvore splay, cada uma sendo uma pesquisa, inserção ou remoção, iniciando em uma árvore splay com nenhuma chave. Sendo  $n_i$  o número de chaves na árvore após a operação  $i$ , e  $n$  sendo o número total de inserções. O tempo de execução total para a execução da seqüência de operações é

$$O\left(m + \sum_{i=1}^m \log n_i\right),$$

o qual é  $O(m \log n)$ .

Em outras palavras, o tempo de execução amortizado da execução de uma pesquisa, inserção ou remoção em uma árvore splay é  $O(\log n)$ , onde  $n$  é o tamanho da árvore splay neste momento. Assim uma árvore splay pode conseguir um tempo logarítmico, com desempenho amortizado para a implementação de um TAD dicionário ordenado. Este desempenho amortizado é compatível com o desempenho do pior caso de árvores AVL, árvores (2.4) e árvores vermelho-pretas, mas ela usa uma simples árvore binária que não precisa de qualquer balanceamento extra para informações armazenadas em cada um dos seus nodos. Além disso, árvores splay têm um número de outras propriedades interessantes que não são compartilhadas com estas outras árvores平衡adas. Explora-se mais uma propriedade adicional na seguinte proposição (que algumas vezes é chamada de "Proposição Static Optimally" para árvores splay):

**Proposição 10.6** Considere-se uma sequência de  $m$  operações em uma árvore splay, cada uma sendo uma pesquisa, inserção ou remoção, iniciando em uma árvore splay  $T$  com nenhuma chave. Sendo  $f(i)$  o número de vezes que o elemento  $i$  é acessado na árvore splay, isto é, sua frequência, e sendo  $n$  o número total de elementos. Assume-se que cada elemento é acessado pelo menos uma vez, então o tempo de execução total para a execução da sequência de operações é:

$$O\left(m + \sum_{i=1}^n f(i) \log(m/f(i))\right).$$

Omite-se a prova desta proposição, mas isso não é tão difícil justificar, como se pode imaginar. O excelente é que esta proposição expressa que o tempo de execução amortizado do acesso ao elemento  $i$  é  $O(\log(m/f(i)))$ .

## 10.4 Árvores (2,4)

Algumas estruturas de dados que se discutem neste capítulo, incluindo a árvore (2, 4), são árvores genéricas de pesquisa, isto é, árvores com nodos internos que tem dois ou mais filhos. Assim, antes de se definir árvores (2,4), serão discutidas árvores genéricas de pesquisa.

### 10.4.1 Árvore genérica de pesquisa

É preciso lembrar que árvores genéricas são definidas de forma que cada nodo interno pode ter vários filhos. Nesta seção, se discutirá como árvores genéricas podem ser usadas como árvores de pesquisa. Lembrando sempre que o **elemento** que se armazena em uma árvore de pesquisa é um par no formato  $(k, x)$ , onde  $k$  é a **chave** e  $x$  é o **valor** associado com a chave. Entretanto, não se discutirá agora como executar atualizações em árvores genéricas de pesquisa, visto que os detalhes dos métodos de atualizações dependem de propriedades adicionais que se desejam para manter árvores genéricas, que se analisarão na Seção 14.3.1.

#### Definição de uma árvore genérica de pesquisa

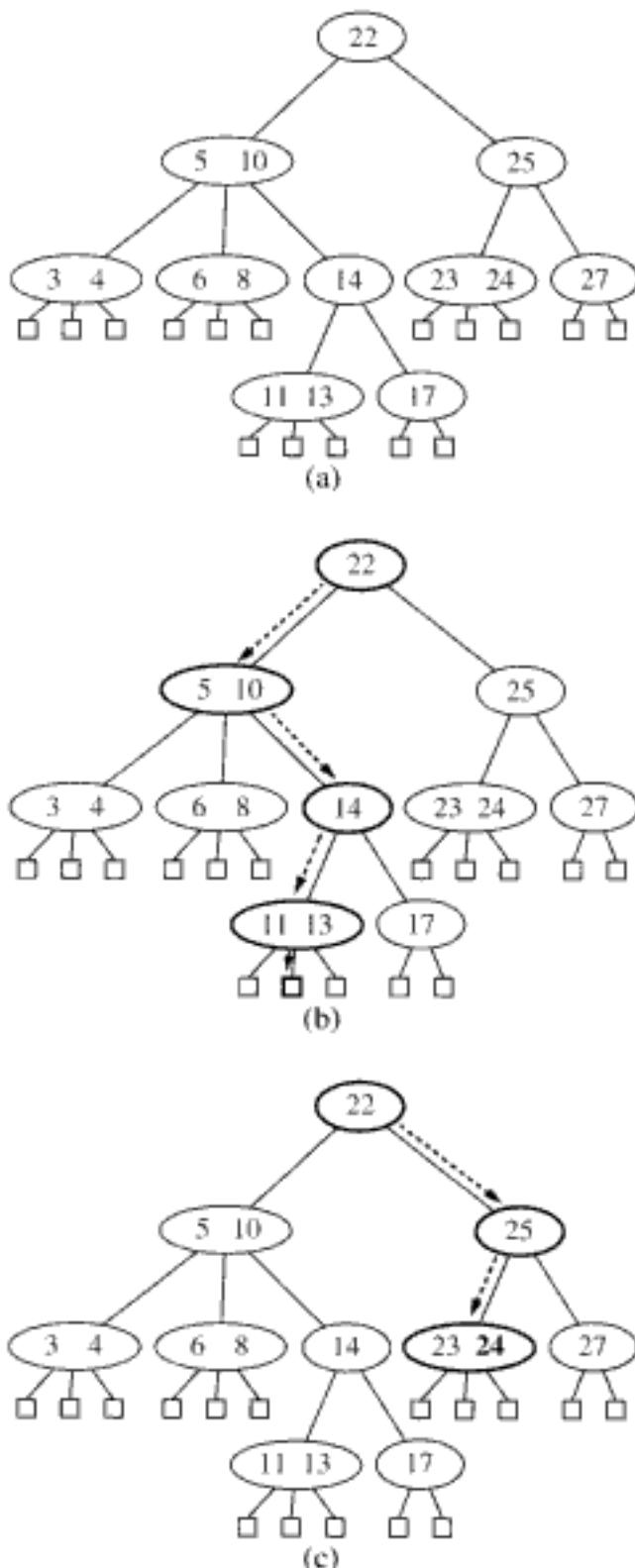
Seja  $v$  um nodo de uma árvore ordenada. Diz-se que  $v$  é um **nodo- $d$**  se  $v$  tiver  $d$  filhos. Define-se uma **árvore genérica de pesquisa** como sendo uma árvore ordenada  $T$  que tem as seguintes propriedades, que são ilustradas na Figura 10.19a:

- Cada nodo interno de  $T$  tem ao menos dois filhos. Isto é, cada nodo interno é um nodo- $d$ , onde  $d \geq 2$ .
- Cada nodo- $d$   $v$  de  $T$ , com filhos  $v_1, \dots, v_d$ , armazena  $d - 1$  itens  $(k_1, x_1), \dots, (k_{d-1}, x_{d-1})$ , onde  $k_1 \leq \dots \leq k_{d-1}$ .
- Define-se, por convenção,  $k_0 = -\infty$  e  $k_d = +\infty$ . Para cada item  $(k, x)$  armazenado em um nodo da subárvore de  $v$  enraizada em  $v_i$ ,  $i = 1, \dots, d$ , tem-se que  $k_{i-1} \leq k \leq k_i$ .

Ou seja, considerando-se o conjunto de chaves armazenadas em  $v$ , incluindo as chaves fictícias especiais  $k_0 = -\infty$  e  $k_d = +\infty$ , então uma chave  $k$  armazenada na subárvore de  $T$  enraizada no nodo filho  $v_i$  deve estar “entre” duas chaves armazenadas em  $v$ . Este ponto de vista simples origina a regra que diz que um nodo com  $d$  filhos armazena  $d - 1$  chaves regulares e forma a base do algoritmo de pesquisa em uma árvore genérica de pesquisa.

Pela definição acima, os nodos externos de uma árvore genérica de pesquisa não armazenam nenhum item, servindo apenas como “guardadores de locais”. Desta forma, vê-se uma árvore binária de pesquisa (Seção 10.1) como um caso especial de árvore genérica de pesquisa onde cada nodo

interno armazena um item e tem dois filhos. No extremo oposto, uma árvore genérica de pesquisa pode ter apenas um único nodo interno que armazena todos os itens. Além disso, apesar dos nodos externos poderem ser `null`, assume-se por definição que são nodos que não armazenam nada.



**Figura 10.19** (a) Uma árvore genérica de pesquisa  $T$ ; (b) Caminho de pesquisa em  $T$  para a chave 12 (pesquisa sem sucesso); (c) caminho de pesquisa em  $T$  para a chave 24 (pesquisa com sucesso).

Tendo os nodos internos de uma árvore genérica dois ou mais filhos, entretanto, existe uma relação interessante entre o número de itens e o número de nodos externos.

**Proposição 10.7** Uma árvore de pesquisa genérica que armazena  $n$  itens tem  $n + 1$  nodos externos.

Deixa-se a justificativa desta proposição como um exercício (C-10.14).

### Pesquisando em uma árvore genérica

Dada uma árvore genérica  $T$ , pesquisar por um elemento com chave  $k$  é simples. Executa-se tal pesquisa seguindo um caminho em  $T$  que se inicia na raiz (ver Figura 10.19b e c). Quando se estiver em um nodo- $d$   $v$  durante esta pesquisa, se comparará a chave  $k$  com as chaves  $k_1, \dots, k_{d-1}$  armazenadas em  $v$ . Se  $k = k_i$  para algum  $i$ , a pesquisa é encerrada com sucesso. Caso contrário, continua-se a pesquisa no filho  $v_i$  de  $v$  de maneira que  $k_{i-1} < k < k_i$ . (É preciso lembrar que se considera  $k_0 = -\infty$  e  $k_d = +\infty$ .) Atingindo-se um nodo externo, então sabe-se que não há nenhum item com a chave  $k$  em  $T$ , e a pesquisa termina sem sucesso.

### Estruturas de dados para árvores de pesquisa genéricas

Na Seção 7.1.3, discutem-se diferentes maneiras de representar árvores genéricas. Cada uma dessas representações também pode ser reutilizada para árvores de pesquisa genérica. Na verdade, ao se usar uma árvore genérica para implementar uma árvore de pesquisa genérica, a única informação adicional que se precisa armazenar em cada nodo é o conjunto de itens (incluindo as chaves) associados com os mesmos. Ou seja, precisa-se armazenar em  $v$  uma referência para um contêiner ou objeto coleção que armazene os itens de  $v$ .

É preciso lembrar que quando se usa uma árvore binária para representar um dicionário ordenado  $D$ , simplesmente se armazena uma referência para um único item em cada nodo interno. Usando uma árvore de pesquisa genérica  $T$  para representar  $D$ , deve-se armazenar uma referência para um conjunto ordenado de itens associados com  $v$  em cada nodo interno  $v$  de  $T$ . Esta argumentação pode parecer recursiva em um primeiro momento, uma vez que se necessita de uma representação de um dicionário ordenado para representar um dicionário ordenado. Pode-se evitar esta recursividade, entretanto, usando a técnica **bootstrapping**, em que a solução anterior (menos desenvolvida) de um problema é usada para uma criar uma solução nova (mais avançada). Neste caso, o bootstrapping consiste em representar o conjunto ordenado associado com cada nodo interno usando a estrutura de dados para dicionário que se construiu anteriormente (por exemplo, uma tabela de pesquisa baseada em um vetor ordenado, como mostrado na Seção 9.3.3). Em particular, assumindo que se dispõe de uma maneira de implementar dicionários ordenados, pode-se implementar uma árvore de pesquisa genérica usando uma árvore  $T$  e armazenando tal dicionário em cada nodo- $d$   $v$  de  $T$ .

O dicionário que se armazena em cada nodo  $v$  é conhecido como uma estrutura de dados **secundária**, na medida em que é usado para suportar a estrutura de dados maior, a **primária**. Denota-se o dicionário armazenado no nodo  $v$  de  $T$  como  $D(v)$ . Os itens que armazenamos em  $D(v)$  nos permitem determinar para qual nodo filho se deve ir durante uma operação de pesquisa. Especificamente, para cada nodo  $v$  de  $T$ , com filhos  $v_1, \dots, v_d$  e itens  $(k_1, x_1), \dots, (k_{d-1}, x_{d-1})$ , armazena-se no dicionário  $D(v)$  os itens

$$(k_1, (x_1, v_1)), (k_2, (x_2, v_2)), \dots, (k_{d-1}, (x_{d-1}, v_{d-1})), (+\infty, (\emptyset, v_d)).$$

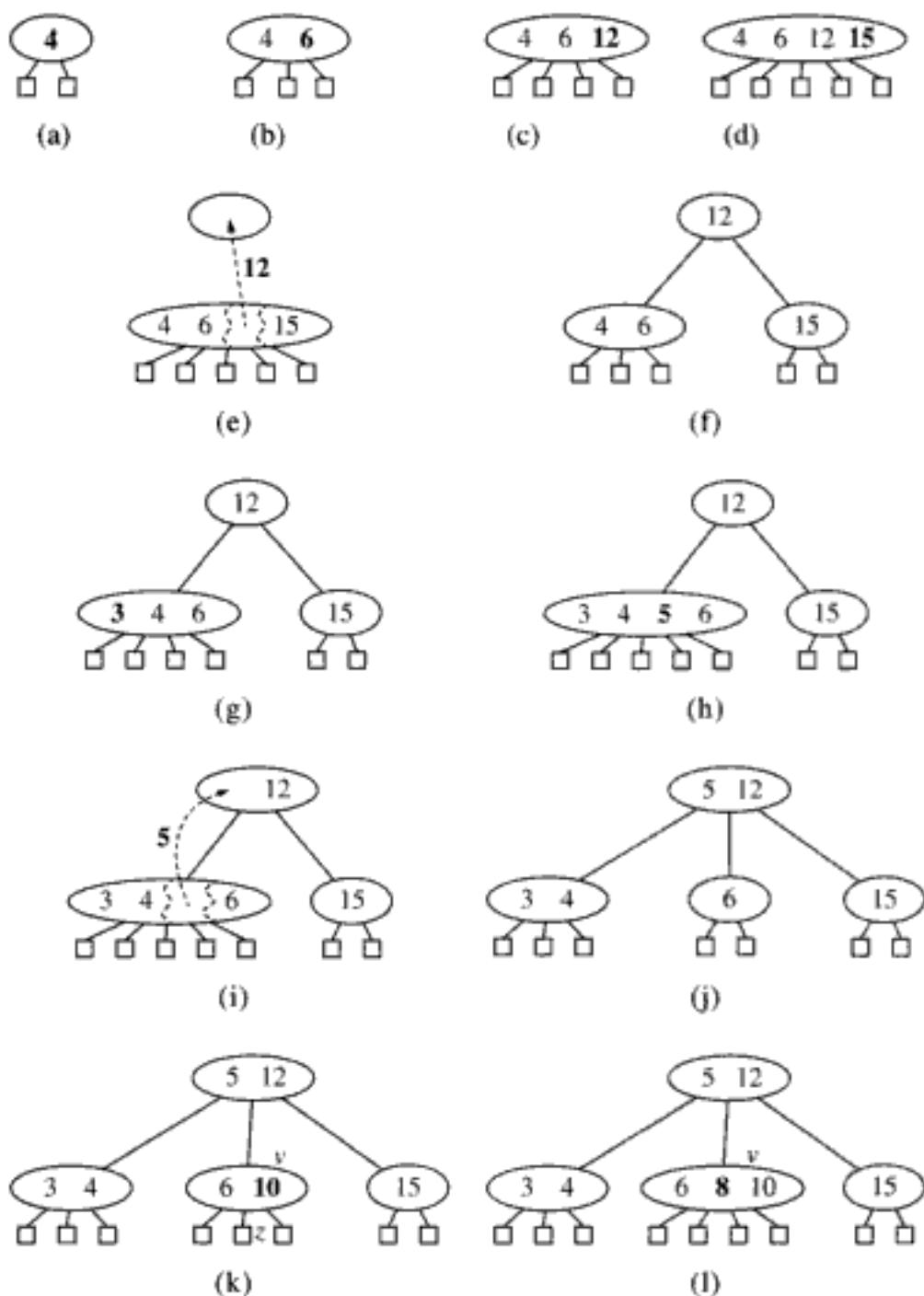
Ou seja, um item  $(k_i, (x_i, v_i))$  de um dicionário  $D(v)$  tem chave  $k_i$  e elemento  $(x_i, v_i)$ . Observe que o último item armazena a chave especial  $+\infty$ .

Com esta implementação de uma árvore de pesquisa genérica  $T$ , o processamento de um nodo- $d$   $v$  durante uma pesquisa por um elemento de  $T$  com chave  $k$  pode ser feito executando-se uma operação de pesquisa para encontrar o item  $(k_i, (x_i, v_i))$  em  $D(v)$  com a menor chave maior ou igual a  $k$ . Distinguem-se dois casos:

Hidden page

Hidden page

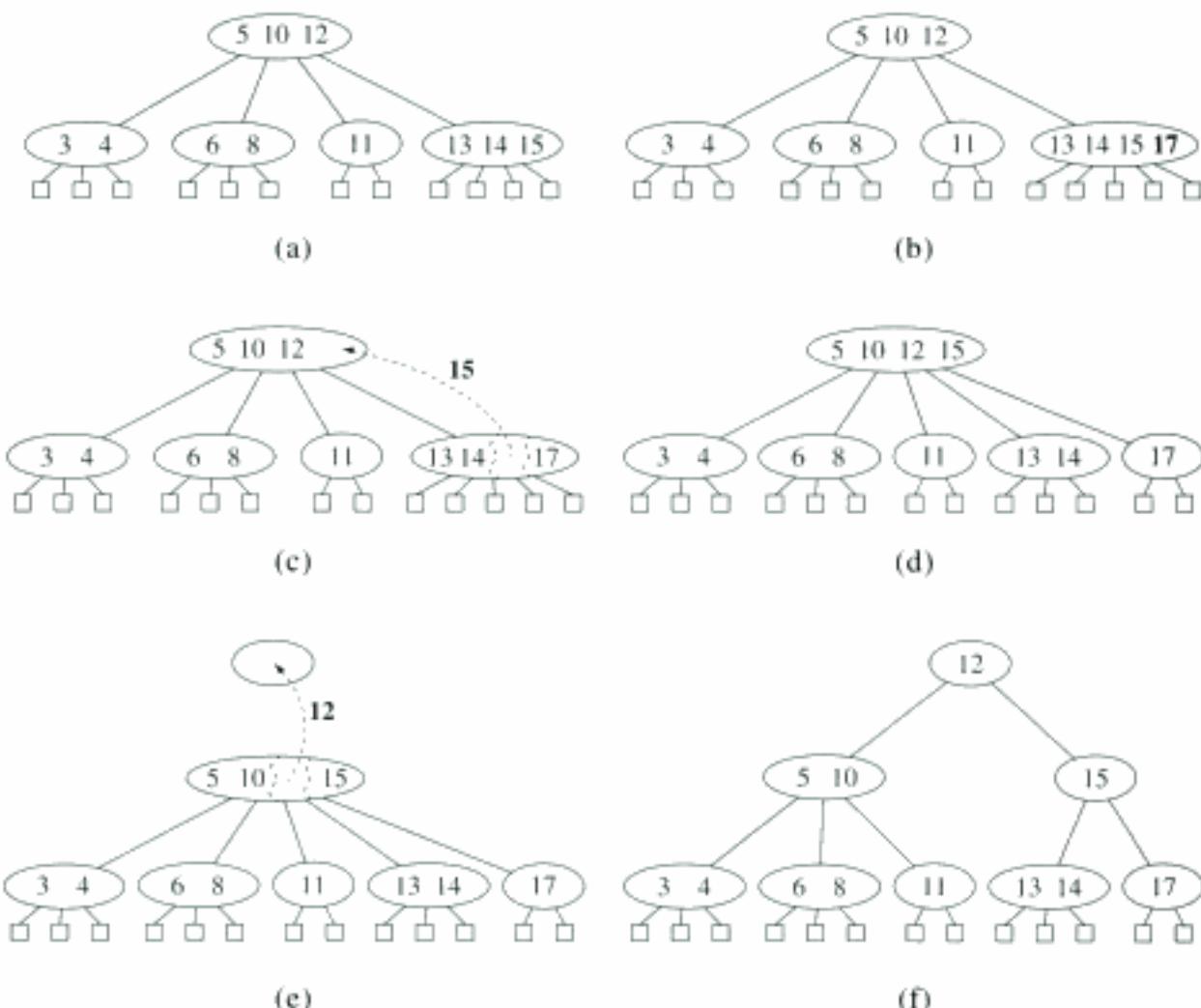
Hidden page



**Figura 10.22** Seqüência de inserções em uma árvore (2,4): (a) árvore inicial com um item; (b) inserção de 6; (c) inserção de 12; (d) inserção de 15, causando um overflow; (e) divisão, implica na criação de um novo nodo raiz; (f) após a divisão; (g) inserção de 3; (h) inserção de 5, causando um overflow; (i) divisão; (j) após a divisão; (k) inserção de 10; (l) inserção de 8.

árvore (2,4) sempre pode cair no caso em que o item a ser removido esteja armazenado em um nodo  $v$  cujos filhos são nodos externos. Supondo-se, por exemplo, que o item com chave  $k$  que se deseja remover esteja armazenado no  $i$ -ésimo item ( $k_i, x_i$ ) no nodo  $z$ , que tem apenas nodos internos como filhos. Neste caso, troca-se o item ( $k_i, x_i$ ) por um item apropriado que esteja armazenado no nodo  $v$  com nodos externos como filhos, como segue (Figura 10.24d):

1. Encontra-se o nodo interno  $v$  mais à direita da subárvore enraizada no  $i$ -ésimo filho de  $z$ , notando que os filhos do nodo  $v$  são todos nodos externos.
2. Troca-se o item ( $k_i, x_i$ ) de  $z$  pelo último item de  $v$ .

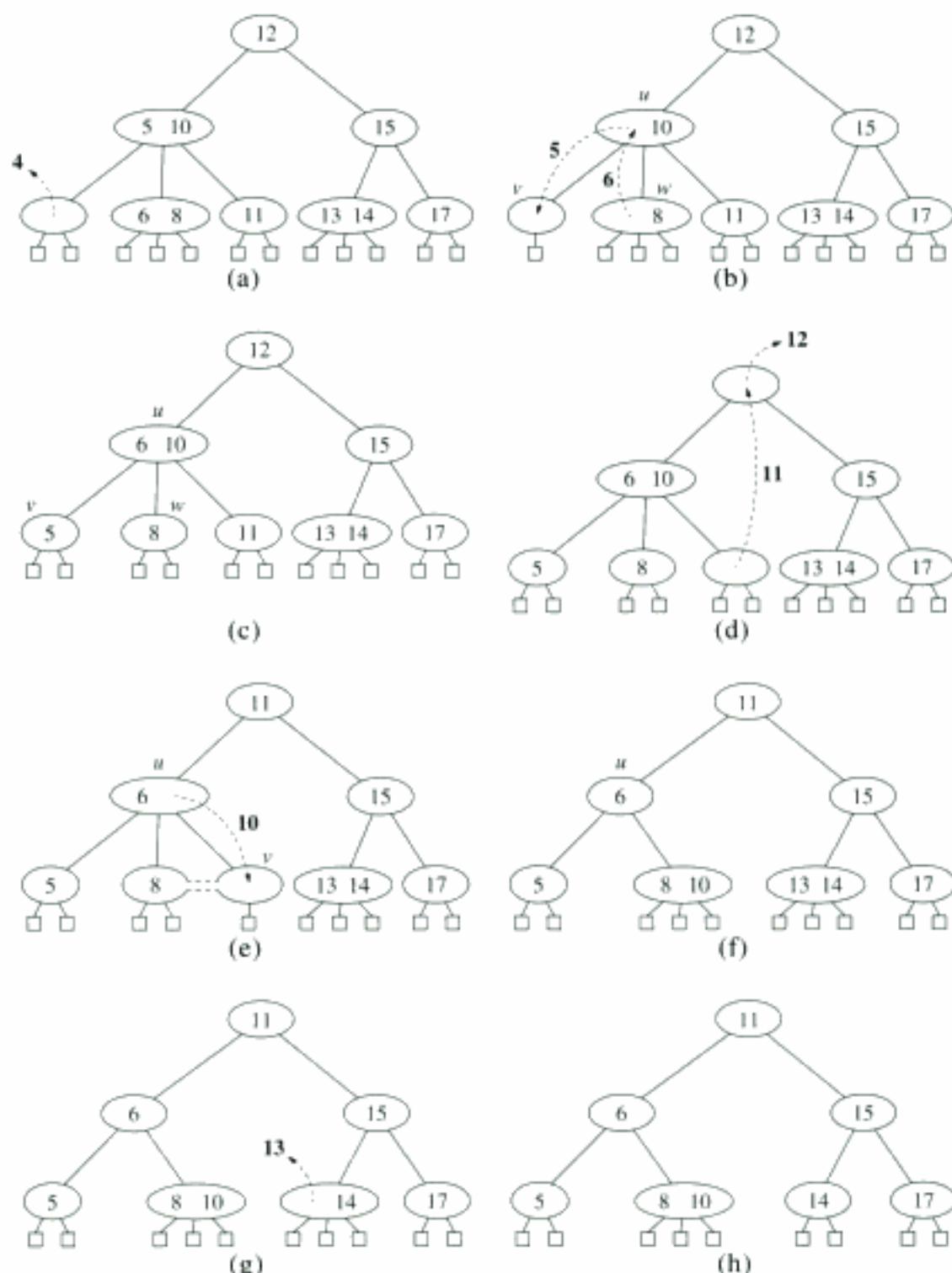


**Figura 10.23** Uma inserção em uma árvore (2,4) causa divisões em cascata: (a) antes da inserção; (b) inserção de 17 causando overflow; (c) uma divisão; (d) após a divisão um novo overflow ocorre; (e) outra divisão criando um novo nodo raiz; (f) árvore final.

Uma vez que se garante que o item a ser removido esteja armazenado em um nodo  $v$  que tem apenas nodos externos como filhos (porque já estava em  $v$  ou porque foi tirado de  $v$ ), simplesmente se remove o item de  $v$  (isto é, do dicionário  $D(v)$ ) e o  $i$ -ésimo nodo externo de  $v$ .

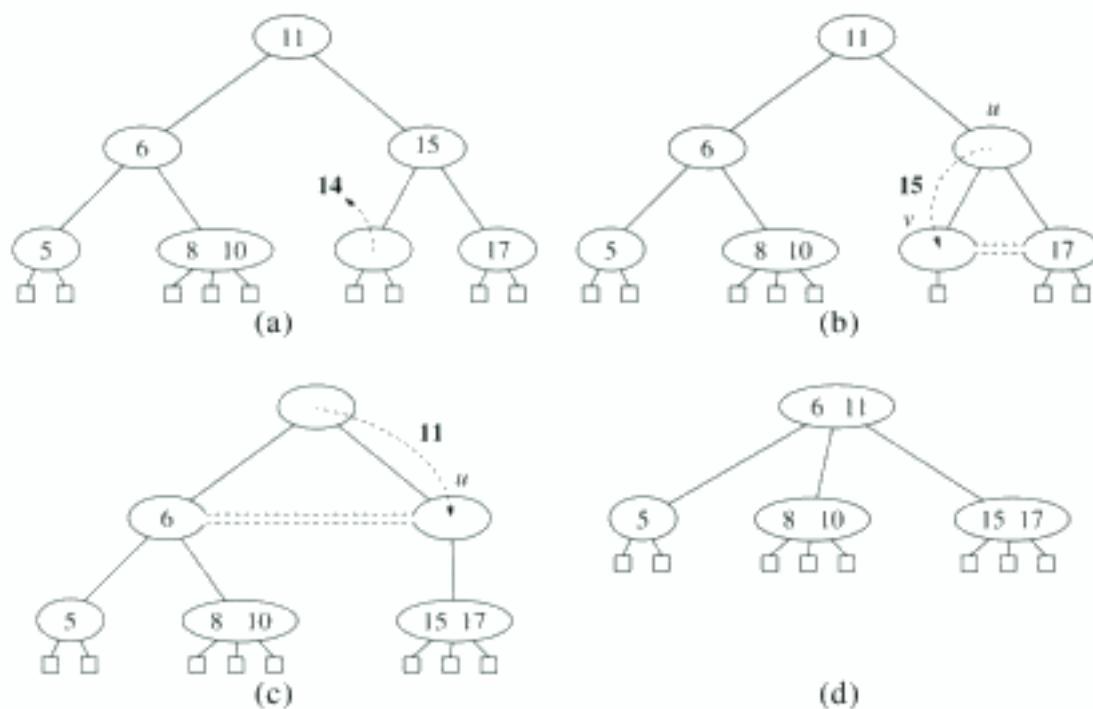
A remoção de um item (e um filho) de um nodo  $v$ , como descrito anteriormente, preserva a propriedade da profundidade, porque sempre se remove um nodo externo filho de um nodo  $v$  que tem apenas nodos externos como filhos. Entretanto, retirando nodos externos, pode-se violar a propriedade do tamanho em  $v$ . Na verdade, se  $v$  era um nodo-2, então ele se torna um nodo-1 sem itens após a remoção (Figuras 10.24d e e), o que não é permitido em uma árvore (2,4). Este tipo de violação da propriedade do tamanho é chamado de underflow do nodo  $v$ . Para remediar um underflow, verifica-se quando um irmão de  $v$  é um nodo-3 ou um nodo-4. Encontrando-se tal irmão  $w$ , então se executa uma operação de **transferência**, na qual se move um filho de  $w$  para  $v$ , uma chave de  $w$  para o pai  $u$  de  $v$ , e uma chave de  $u$  para  $v$  (ver Figura 10.24b e c). Se  $v$  tiver apenas um irmão ou se os dois irmãos, imediatos de  $v$  são nodos-2, então executa-se uma operação de **fusão**, na qual se une  $v$  com um irmão, criando um novo nodo  $v'$ , e movendo uma chave do pai  $u$  de  $v$  para  $v'$ . (Ver Figura 10.25e e f.)

Uma operação de fusão no nodo  $v$  pode causar um novo underflow, que irá ocorrer no pai  $u$  de  $v$ , que por sua vez dispara uma transferência ou fusão em  $u$  (ver Figura 10.25). Então, o



**Figura 10.24** Seqüência de remoções de uma árvore (2,4): (a) remoção de 4, causando underflow; (b) operação de transferência; (c) após a operação de transferência; (d) remoção de 12, causando underflow; (e) operação de fusão; (f) após a operação de fusão; (g) remoção de 13; (h) após a remoção de 13.

número de operações de fusão é limitado pela altura da árvore que é  $O(\log n)$  pela Proposição 10.8. Se um underflow se propaga até a raiz, então esta é simplesmente removida. (Ver Figura 10.25c e d.) Apresenta-se uma seqüência de remoções de uma árvore (2,4) nas Figuras 10.24 e 10.25.



**Figura 10.25** Propagação de uma sequencia de fusões em uma árvore (2,4): (a) remoção de 14, causando um underflow; (b) fusão, causando outro underflow; (c) segunda operação de fusão, causando a remoção da raiz; (d) árvore final.

### Desempenho de árvores (2,4)

A Tabela 10.3 resume os tempos de execução das principais operações de um dicionário implementado usando uma árvore (2,4). A análise de complexidade do tempo é baseada no seguinte:

- A altura de uma árvore (2,4) que armazena  $n$  itens é  $O(\log n)$ , pela Proposição 10.8.
- Uma operação de divisão, transferência ou fusão leva tempo  $O(1)$ .
- Uma pesquisa, inserção ou remoção de um item visita  $O(\log n)$  nodos.

Operação	Tempo
size, isEmpty	$O(1)$
find, insert, remove	$O(\log n)$
findAll	$O(\log n + s)$

**Tabela 10.3** Performance de um dicionário com  $n$  elementos implementados usando uma árvore (2,4), onde  $s$  denota o tamanho dos iteradores retornados por findAll. O espaço utilizado é  $O(n)$ .

Desta forma, árvores (2,4) oferecem operações rápidas de pesquisa e alteração em dicionários. As árvores (2,4) também têm um relacionamento interessante com a estrutura de dados que será discutido a seguir.

---

## 10.5 Árvores vermelho-pretas

Apesar de árvores AVL e (2,4) terem várias propriedades interessantes, existem algumas aplicações de dicionário para as quais elas não são muito adequadas. Por exemplo, árvores AVL podem requerer a execução de muitas operações de reestruturação (rotações) após a remoção de um elemento, e as árvores (2,4) podem exigir a execução de muitas operações de fusão ou divisão tanto

após inserções como remoções. A estrutura de dados que será discutida nesta seção, a árvore vermelho-preta, não apresenta esses problemas, pois exige que podem ser feitas somente alterações estruturais  $O(1)$  após uma atualização, visando manter o balanceamento.

Uma **árvore vermelho-preta** é uma árvore de pesquisa binária (ver a Seção 10.1) com nodos coloridos de vermelho e preto, de forma a satisfazer as seguintes propriedades:

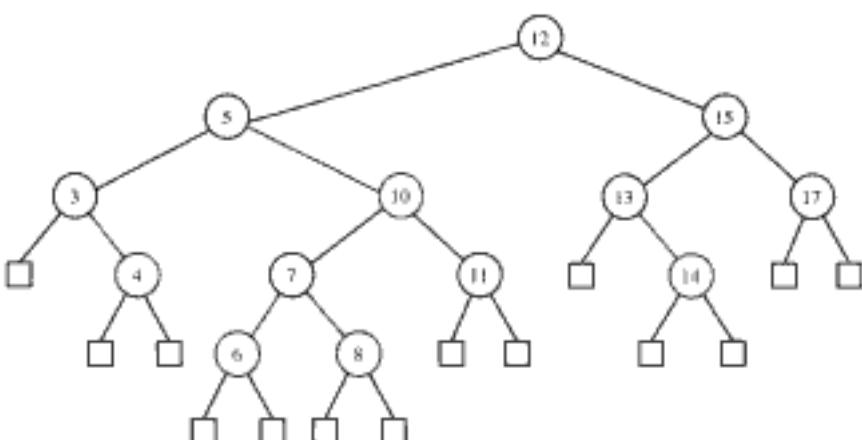
**Propriedade da raiz:** a raiz é preta.

**Propriedade externa:** todo nodo externo é preto.

**Propriedade interna:** os filhos de um nodo vermelho são pretos.

**Propriedade da profundidade:** todos os nodos externos têm a mesma **profundidade preta** que é definida como o número de ancestrais pretos menos um. (Deve-se lembrar que um nodo é um ancestral dele mesmo.)

Um exemplo de árvore vermelho-preta é apresentado na Figura 10.26.



**Figura 10.26** Árvore vermelho-preta relacionada com a árvore (2,4) da Figura 10.20. Cada nodo externo desta árvore vermelho-preta tem 4 ancestrais pretos (incluindo ele mesmo); portanto, tem profundidade preta 3. Foi utilizada a cor cinza em vez de vermelho. Além disso, usa-se a convenção de dar para as arestas a mesma cor do nodo filho.

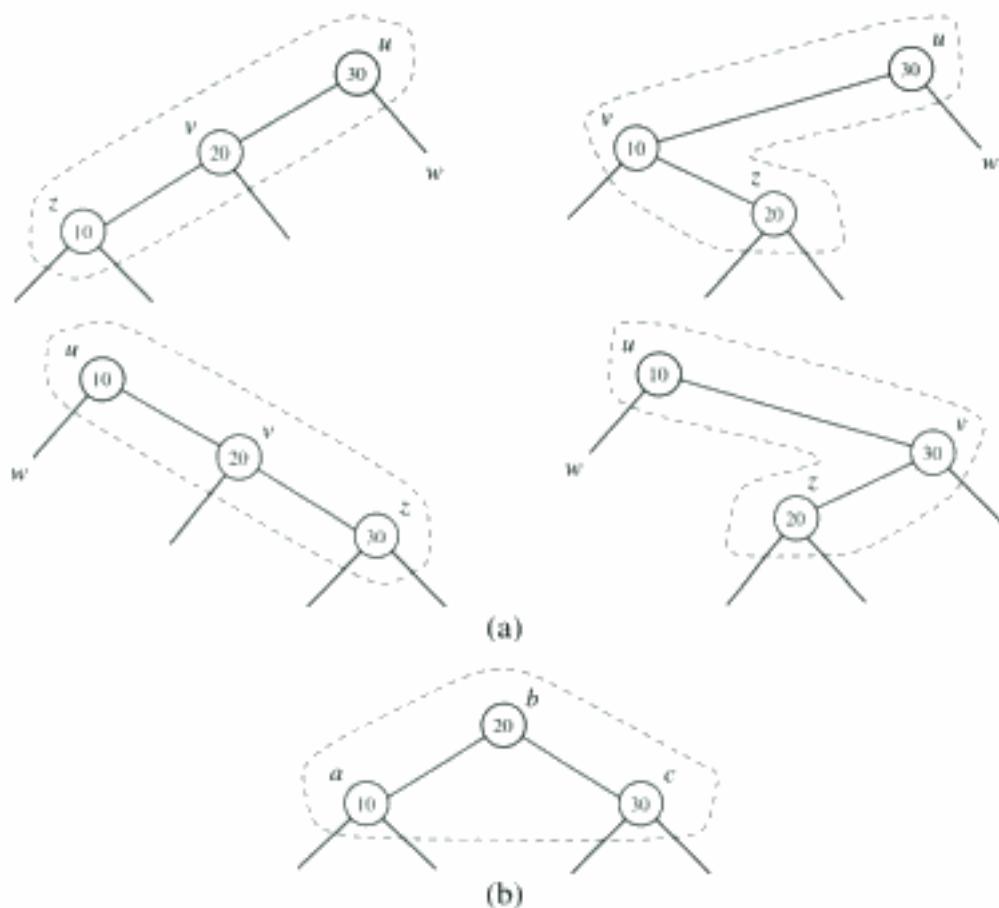
Como tem sido convencionado neste capítulo, pressupõe-se que os itens são armazenados nos nodos internos da árvore vermelho-preta, com os nodos externos sendo lugares vagos. Além disso, descrevem-se nossos algoritmos pressupondo que são nodos reais, mas se nota que ao custo de algoritmos de pesquisa e atualização um pouco mais complicados, nodos externos podem ser **null**.

Pode-se tornar a definição de uma árvore vermelho-preta mais intuitiva, observando uma correspondência interessante entre árvores vermelho-pretas e árvores (2,4), como demonstrado na Figura 10.27. Isto é, dada uma árvore vermelho-preta, pode-se construir a árvore (2,4) correspondente combinando todo nodo vermelho  $v$  com seu pai, e armazenando o item de  $v$  no seu pai. Da mesma forma, pode-se transformar qualquer árvore (2,4) em sua árvore vermelho-preta correspondente, colorindo cada nodo de preto e executando as seguintes transformações sobre cada nodo interno  $v$ :

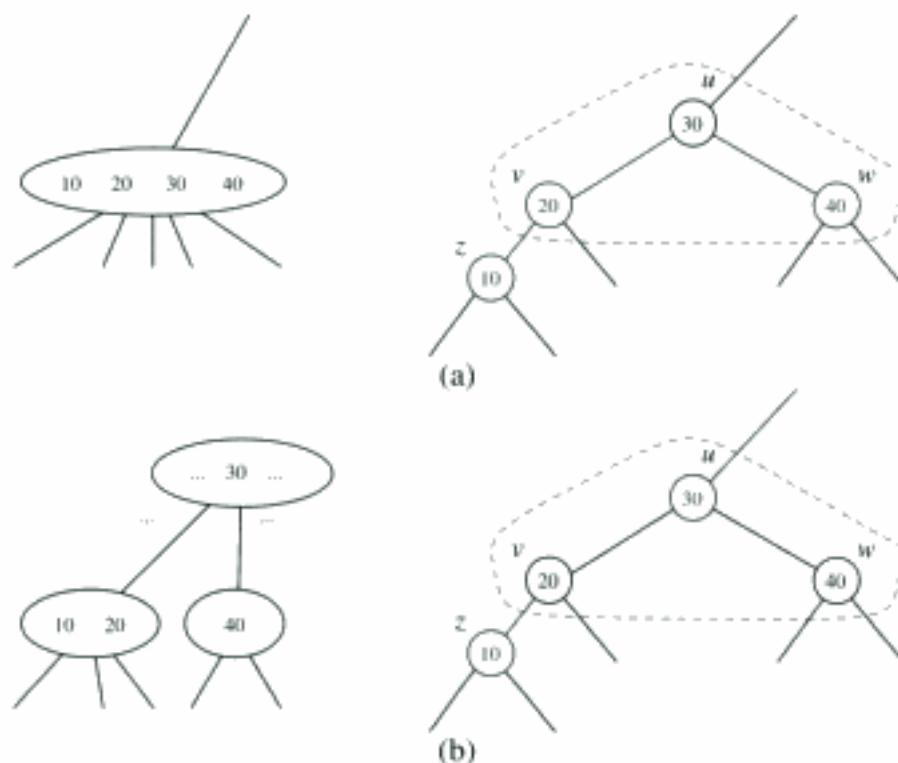
- Se  $v$  é um nodo-2, então mantenha os filhos (pretos) de  $v$  como estão.
- Se  $v$  é um nodo-3, então crie um novo nodo vermelho  $w$ , passe os primeiros dois filhos (pretos) de  $v$  para  $w$ , e faça  $w$  e o terceiro filho de  $v$  serem os filhos de  $v$ .
- Se  $v$  é um nodo-4, então crie dois novos nodos vermelhos  $w$  e  $z$ , passe os dois primeiros filhos (pretos) de  $v$  para  $w$ , passe os dois últimos filhos (pretos) de  $v$  para  $z$ , e faça  $w$  e  $z$  serem os dois filhos de  $v$ .

Hidden page

Hidden page



**Figura 10.28** Reestruturação de uma árvore vermelho-preta para remediar um vermelho duplo:  
(a) as quatro configurações para  $u,v$  e  $z$  antes da reestruturação; (b) após a reestruturação.

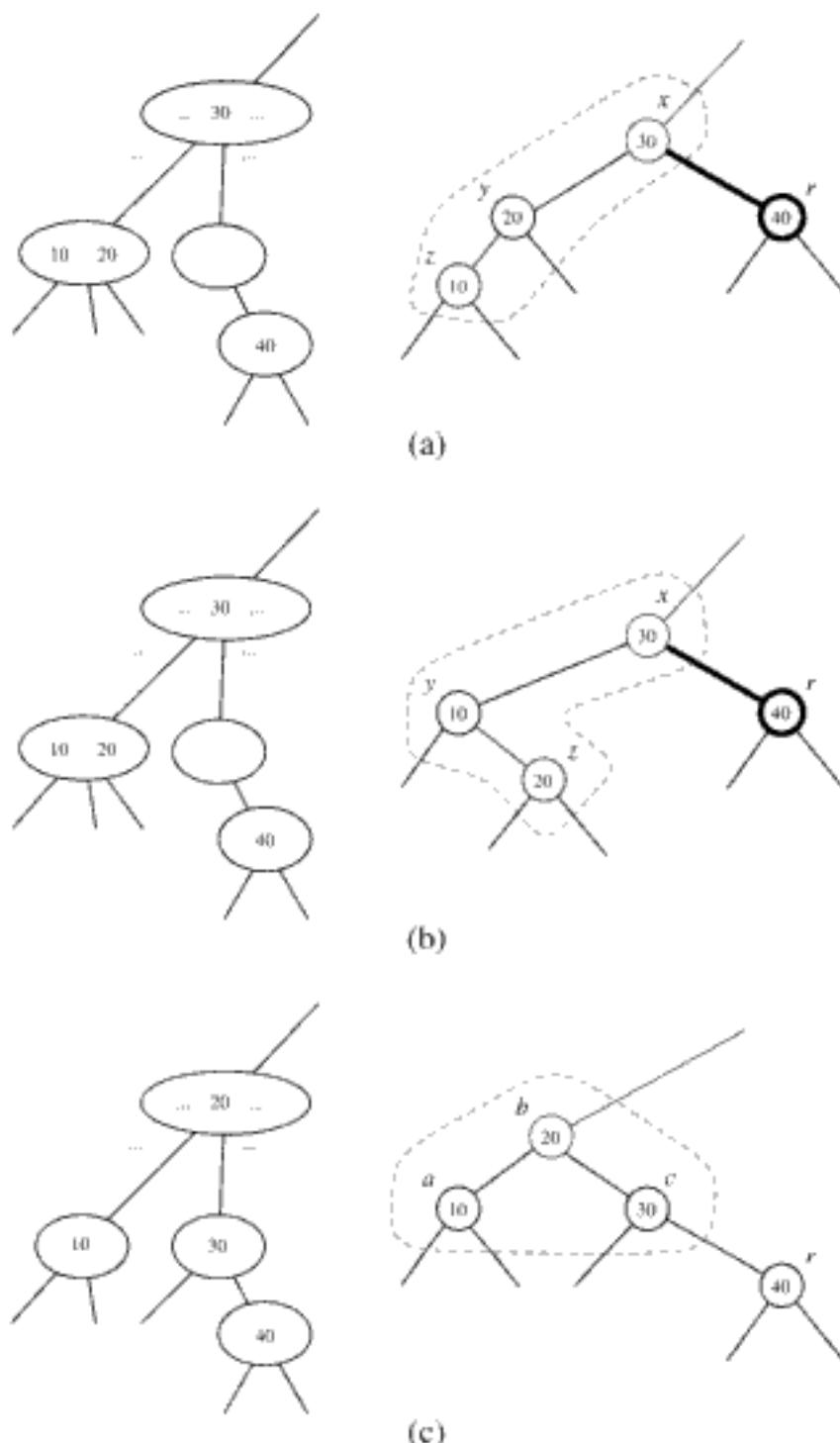


**Figura 10.29** Trocando cores para remediar o problema do vermelho duplo: (a) antes da troca de cores e o nodo-5 correspondente na árvore (2,4) associada antes da divisão; (b) depois da troca de cores (e os nodos correspondentes na árvore (2,4) associada após a divisão).

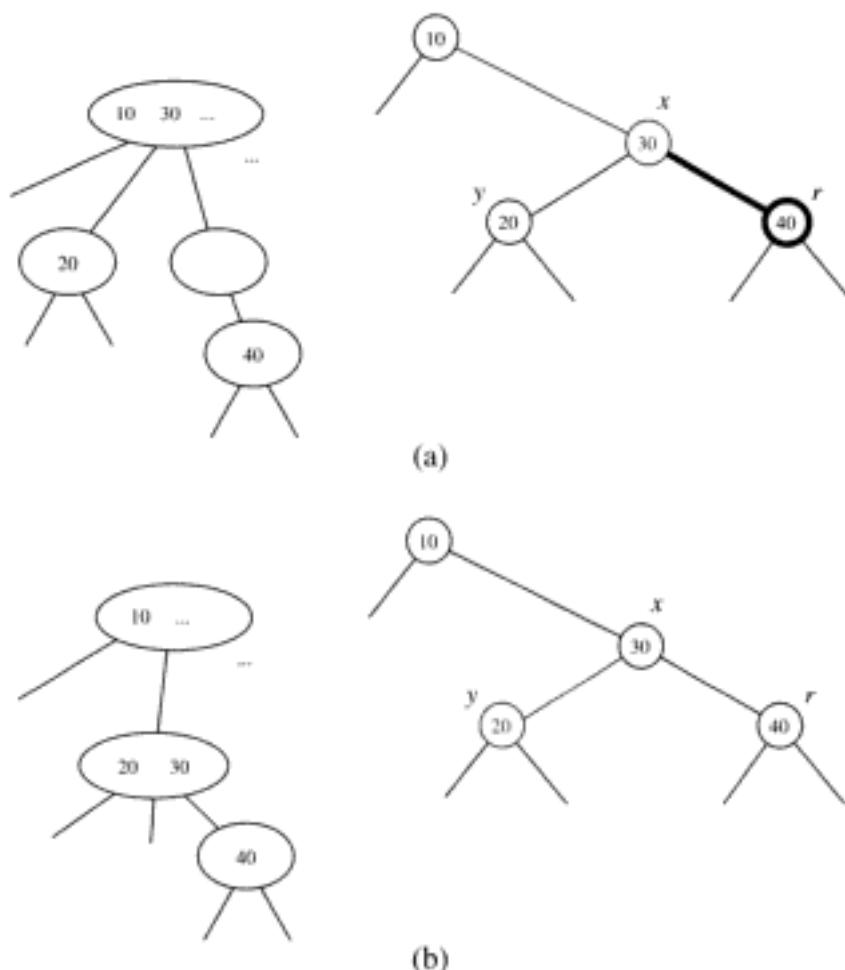
Hidden page

Hidden page

Hidden page



**Figura 10.32** Reestruturação de uma árvore vermelho-preta para remediar o problema do duplo preto: configurações (a) e (b) antes da reestruturação, com  $r$  sendo um filho da direita, juntamente com os nodos associados na árvore (2,4) correspondente antes da transferência (duas outras configurações simétricas são possíveis com  $r$  sendo o filho da esquerda; configuração (c) após a reestruturação, e os nodos associados na árvore (2,4) correspondente após a transferência. A cor cinza do nodo  $x$  nas partes (a) e (b) e para o nodo  $b$  na parte (c) denotam o fato de que este nodo pode ser colorido, tanto de vermelho como de preto.



**Figura 10.33** Alterando as cores de uma árvore vermelho-preta para consertar o problema do duplo preto: (a) antes da alteração de cores e os nodos correspondentes na árvore (2,4) associada antes da fusão (outras configurações semelhantes são possíveis); (b) após a troca de cores e nodos correspondentes na árvore (2,4) associada após a fusão.

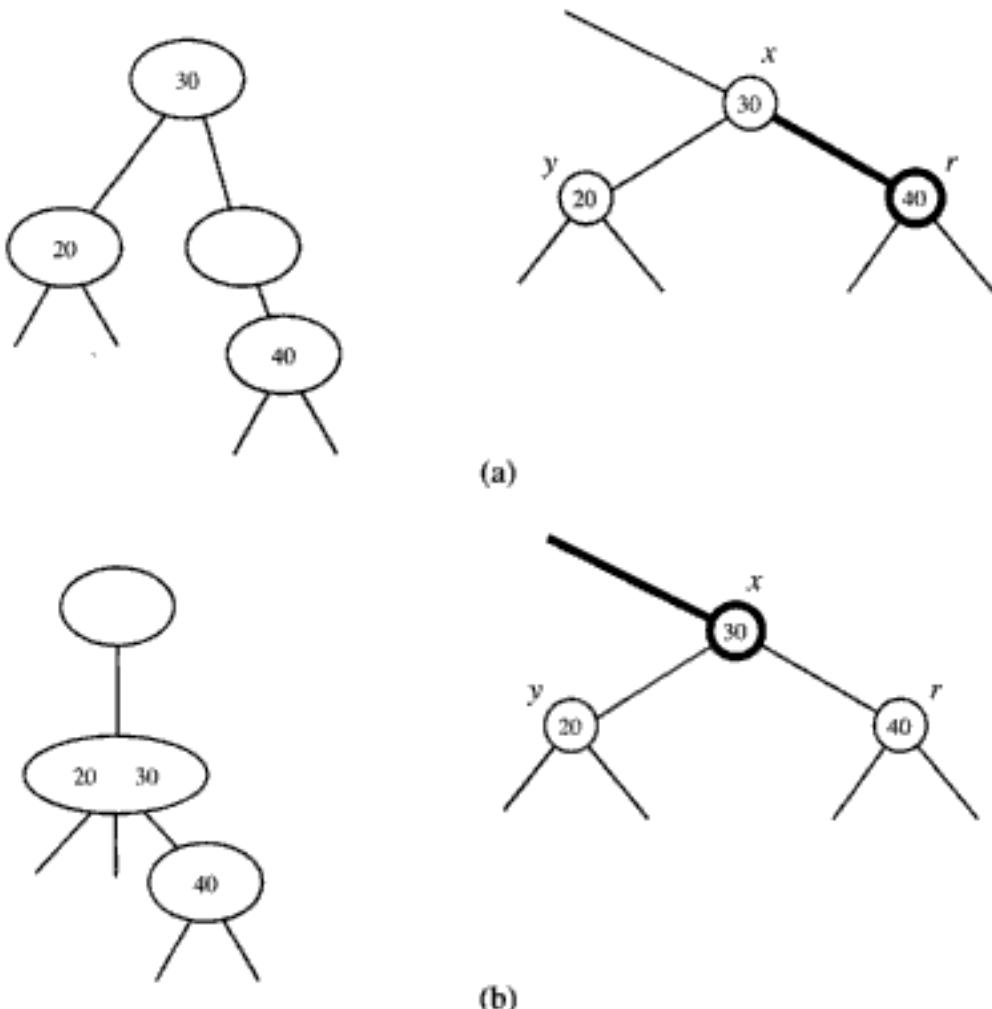
### Desempenho das árvores vermelho-pretas

A Tabela 10.4 resume os tempos de execução das principais operações do dicionário implementado usando uma árvore vermelho-preta. As justificativas para esses limites são apresentadas na Figura 10.38.

Operação	Tempo
size, isEmpty	$O(1)$
find, insert, remove	$O(\log n)$
findAll	$O(\log n + s)$

**Tabela 10.4** Performance de um dicionário de  $n$  elementos implementado, usando uma árvore vermelho-preta em que  $s$  denota o tamanho dos iteradores retornados por findAll. O espaço utilizado é  $O(n)$ .

Desta forma, uma árvore vermelho-preta obtém tempo de execução logarítmico para o pior caso tanto para pesquisa como para atualização em um dicionário. A estrutura da árvore vermelho-preta é ligeiramente mais complicada que a árvore (2,4) correspondente. Apesar disso, uma árvore vermelho-preta tem a vantagem conceitual de requerer apenas um número constante de reestruturações trinodo para restaurar o balanceamento após uma atualização.

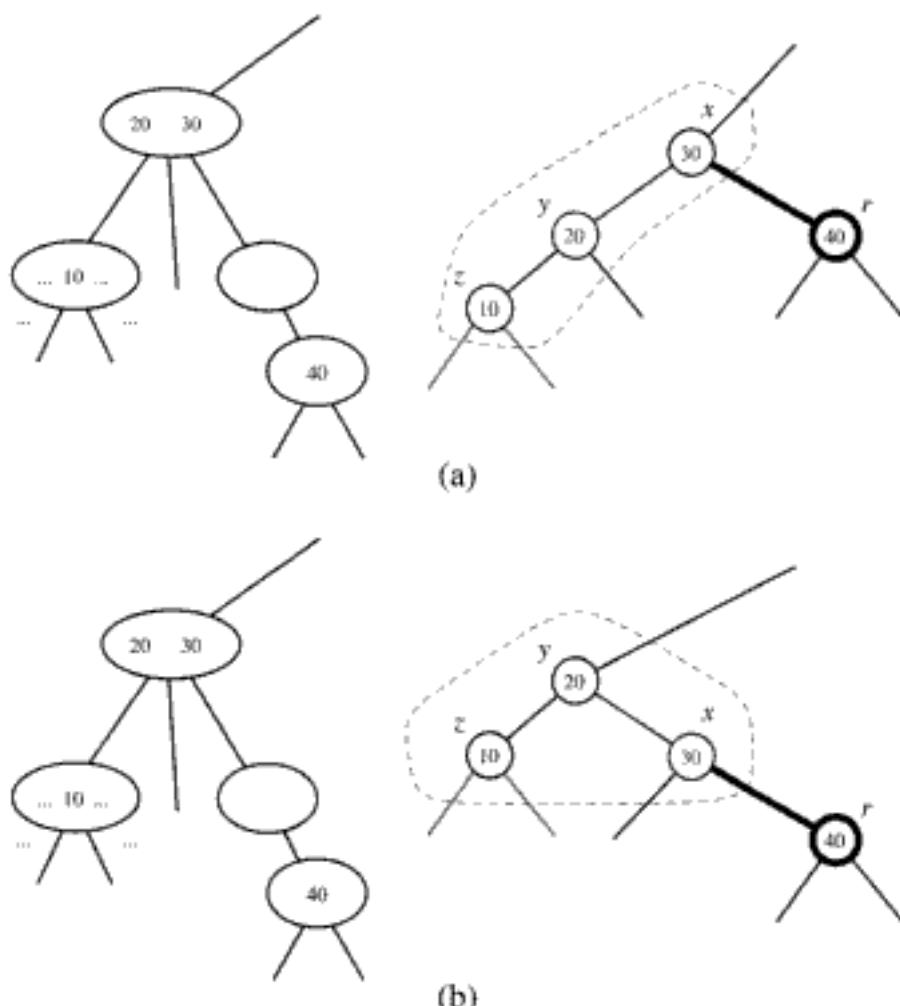


**Figura 10.34** Troca de cores de uma árvore vermelho-preta para propagar o problema do duplo preto: (a) configuração antes da alteração das cores e nodos correspondentes na árvore (2,4) associada antes da fusão (outras configurações similares são possíveis); (b) configuração após a troca de cores e nodos correspondentes na árvore (2,4) associada após a fusão.

### 10.5.2 Implementação Java

Nos Trechos de código 10.9 – 10.11, são apresentados trechos da implementação em Java de um dicionário organizado usando uma árvore vermelho-preta. A classe principal inclui uma classe aninhada, RBNode, mostrada no Trecho de código 10.9, que estende a classe BTNode usada para representar um item chave-valor de uma árvore de binária de pesquisa. Define uma variável de instância adicional isRed, representando a cor do nodo, e métodos para atribuir e retorná-lo.

```
/** Implementação de um dicionário com uma árvore vermelho-preta. */
public class RBTree<K,V>
    extends BinarySearchTree<K,V> implements Dictionary<K,V> {
    public RBTree() { super(); }
    public RBTree(Comparator<K> C) { super(C); }
    /** Classe aninhada para os nodos da árvore vermelho-preta */
    protected static class RBNode<K,V> extends BTNode<Entry<K,V>> {
        protected boolean isRed; // Adiciona-se um campo cor para um BTNode
        RBNode() {/* Construtor padrão*/}
        /** Construtor preferido */
        RBNode(Entry<K,V> element, BTPosition<Entry<K,V>> parent,
```



**Figura 10.35** Ajuste de uma árvore vermelho-preta na presença do problema de um duplo preto: (a) configuração antes do ajuste e nodos correspondentes na árvore (2,4) associada (uma configuração simétrica é possível); (b) configuração após o ajuste com os mesmos nodos correspondentes na árvore (2,4) associada.

```

BTPosition<Entry<K,V>> left, BTPosition<Entry<K,V>> right) {
    super(element, parent, left, right);
    isRed = false;
}
public boolean isRed() {return isRed;}
public void makeRed() {isRed = true;}
public void makeBlack() {isRed = false;}
public void setColor(boolean color) {isRed = color;}
}

```

**Trecho de código 10.9** Variáveis de instância, classe aninhada e construtor para RBTree.

A classe `RBTree` (Trechos de código 10.9 – 10.11) estende a classe `BinarySearchTree` (Trechos de código 10.3 – 10.5). Assume-se que a classe pai suporta o método `restructure` para executar a reestruturação trinodo (rotações); sua implementação foi deixada como exercício (P-10.3). A classe `RBTree` herda os métodos `size`, `isEmpty`, `find` e `findAll` da classe `BinarySearchTree`, mas sobrecarrega os métodos `insert` e `remove`. Implementa estas duas operações, primeiro pela chamada ao método correspondente da classe pai, e então remediando qualquer violação de cor que esta alteração pode ter causado. Vários métodos auxiliares da classe `RBTree` não são mostrados, mas seus nomes sugerem seus significados, e suas implementações são diretas.

Hidden page

heap unificável  $h$  com o presente, destruindo as versões antigas de ambos. Descreva uma implementação concreta para o TAD heap unificável que obtenha performance  $O(\log n)$  para todas as suas operações.

- C-10.19 Considere uma variação da árvore splay chamada *árvores half-splay*, onde a expansão de um nodo com profundidade  $d$  para assim que o nodo consiga a profundidade  $\lfloor d/2 \rfloor$ . Execute uma análise de amortização das árvores half-splay.
- C-10.20 A etapa de expansão padrão requer duas passagens, uma descida para encontrar o nodo  $x$  para expansão, seguida por uma subida para expandir o nodo  $x$ . Descreva um método para expansão e pesquisa pelo nodo  $x$  em um passo de descida. Cada subpasso requer que você considere os próximos dois nodos no caminho abaixo de  $x$ , com um possível subpasso zig executado no final. Descreva como executar os passos zig-zig, zig-zag e zig.
- C-10.21 Descreva uma seqüência de acessos a um nodo  $n$  da árvore splay  $T$ , onde  $n$  é ímpar que resulta em  $T$ , consistindo em uma simples cadeia de nodos internos com filhos que são nodos externos, no qual o caminho do nodo interno abaixo  $T$  alterna entre o filho à esquerda e o filho à direita.
- C-10.22 Explique como implementar um arranjo de  $n$  elementos onde os métodos *add* e *get* levam o tempo  $O(\log n)$  no pior caso (sem a necessidade de um arranjo expansível).

## Projetos

- P-10.1 Simulações de  $n$ -corpos são ferramentas de modelagem importantes na física, astronomia e química. Neste projeto, você tem que escrever um programa que execute uma simples simulação de  $n$ -corpos chamada “Duendes Saltitantes”\*. Esta simulação envolve  $n$  duendes, numerados de 1 até  $n$ . Ela mantém um valor ouro  $g$ , para cada duende  $i$ , e se inicia com cada duende começando com o valor do ouro em um milhão de dólares, isto é,  $g_i = 1\,000\,000$  para cada  $i = 1, 2, \dots, n$ . Além disso, a simulação também mantém, para cada duende  $i$ , um lugar no horizonte, que é representado com um número de ponto flutuante de dupla precisão,  $x_i$ . Em cada iteração da simulação, esta processa os duendes na ordem. O processamento da um duende durante esta iteração inicia pela computação de um novo lugar no horizonte para  $i$ , que é determinado pela seguinte tarefa

$$x_i \leftarrow x_i + rg_i,$$

onde  $r$  é um número de ponto flutuante gerado randomicamente dentro do intervalo  $-1$  e  $1$ . O duende  $i$  então rouba metade do ouro do duende mais próximo de um dos seus lados e adiciona este ouro no seu valor de ouro  $g_i$ . Escreva um programa que possa executar uma série de iterações nesta simulação para um dado número,  $n$ , de duendes. Tente incluir uma visualização dos duendes nesta simulação, incluindo seus valores de ouro e posições no horizonte. Você pode manter o conjunto de posições do horizonte usando uma estrutura de dados de dicionário ordenado descrita neste capítulo.

\* N. de T. O autor utiliza a expressão “Jumping Leprechauns”.

- 
- P-10.2 Estenda a classe `BinarySearchTree` (Trecho de código 10.3 – 10.5) para suportar os métodos de um TAD dicionário ordenado (ver Seção 9.5.2).
  - P-10.3 Implemente um classe `RestructurableNodeBinaryTree` que suporte os métodos de um TAD árvore binária, mais um método `restructure` para execução de uma operação de rotação. Esta classe é um componente da implementação de uma árvore AVL apresentada na Seção 10.2.2.
  - P-10.4 Escreva uma classe Java que implemente todos os métodos de um TAD dicionário ordenado (ver Seção 9.5.2) usando uma árvore AVL.
  - P-10.5 Escreva uma classe Java que implemente todos os métodos de um TAD dicionário ordenado (ver Seção 9.5.2) usando uma árvore (2,4).
  - P-10.6 Escreva uma classe Java que implemente todos os métodos de um TAD dicionário ordenado (ver Seção 9.5.2) usando uma árvore vermelho-preta.
  - P-10.7 Forme uma equipe de três programadores e que cada membro implemente um dos três projetos apresentados anteriormente. Faça um extensivo estudo para comparar a velocidade de cada um destas três implementações. Projete três conjuntos de experimentos, cada um favorecendo uma diferente implementação.
  - P-10.8 Escreve uma classe Java que possa pegar qualquer árvore vermelho-preta e convertê-la em uma árvore (2,4) correspondente e possa pegar qualquer árvore (2,4) e convertê-la em uma árvore vermelho-preta correspondente.
  - P-10.9 Execute um estudo experimental para comparar o desempenho de uma árvore vermelho-preta com uma skip list.
  - P-10.10 Prepare uma implementação de árvores splay que utilizem expansão bottom-up como descrito neste capítulo e outra que utilize uma expansão top-down como descrito no Exercício C-10.20. Execute um estudo experimental extensivo para verificar qual implementação é melhor na prática, se houver.

---

## Observações sobre o capítulo

Algumas das estruturas de dados discutidas neste capítulo são descritas em detalhes por Knuth no seu livro *Sorting and Searching* [63] e por Mehlhorn em [74]. As árvores AVL são atribuídas a Adel'son-Vel'skii e Landis [1], que inventaram essa classe de árvores de pesquisa balanceadas em 1962. Árvores de pesquisa binária, árvores AVL e estruturas de hash são descritas por Knuth no seu livro *Sorting and Searching* [63]. Análises de altura média para árvores de pesquisa binária podem ser encontradas nos livros de Aho, Hopcroft e Ulman [5] e Cormen, Leiserson e Rivest [25]. O manual de Gonnet e Bazea-Yates [41] contém uma boa quantidade de comparações experimentais e teóricas entre implementações de dicionários. Ahos, Hopcroft e Ulman [4], discutem árvores (2,3), que são similares a árvores (2,4). Árvores vermelho-pretas são definidas por Bayer [10]. Variações e propriedades interessantes de árvores vermelho-pretas são apresentadas em um artigo de Guibas e Sedgewick [46]. O leitor interessado em aprender mais sobre diferentes tipos de estruturas de árvores balanceadas deve procurar os livros de Mehlhorn [74] e Tarjan [91] e o capítulo de livro de Mehlhorn e Tsakalidis [76]. Knuth [63] é uma leitura adicional excelente que inclui abordagens mais recentes de árvores balanceadas. Árvores splay foram inventadas por Sleator and Tarjan [86] (ver também [91]).



## Conteúdo

---

<b>11.1 Merge-sort .....</b>	<b>432</b>
11.1.1 Divisão e conquista .....	432
11.1.2 Junção de arranjos e listas .....	433
11.1.3 O tempo de execução do merge-sort .....	438
11.1.4 Implementações Java do merge-sort .....	439
11.1.5 O merge-sort e suas relações de recorrência ★ .....	441
<b>11.2 Quick-sort.....</b>	<b>442</b>
11.2.1 Quick-sort randômico .....	448
11.2.2 Quick-sort in-place .....	450
<b>11.3 Um limite inferior para ordenação .....</b>	<b>452</b>
<b>11.4 Bucket-sort e radix-sort .....</b>	<b>453</b>
11.4.1 Bucket-sort .....	454
11.4.2 Radix-sort .....	455
<b>11.5 Comparando algoritmos de ordenação.....</b>	<b>456</b>
<b>11.6 O TAD conjunto e estruturas union/find.....</b>	<b>458</b>
11.6.1 Uma simples implementação de conjunto .....	458
11.6.2 Partições com operações de union-find .....	461
11.6.3 Uma implementação de partição baseada em árvore ★ .....	462
<b>11.7 Seleção .....</b>	<b>465</b>
11.7.1 Poda e busca .....	465
11.7.2 Quick-select randômico .....	466
11.7.3 Analisando o quick-select randômico .....	467
<b>11.8 Exercícios .....</b>	<b>468</b>

## 11.1 Merge-sort

Nesta seção, será apresentada uma técnica de ordenação chamada merge-sort, que pode ser descrita de uma forma simples e compacta usando recursão.

### 11.1.1 Divisão e conquista

O merge-sort baseia-se em um padrão de projeto chamado **divisão e conquista** (*divide-and-conquer*). O paradigma de divisão e conquista pode ser descrito, de maneira geral, como sendo composto de três fases:

1. **Divisão:** se o tamanho da entrada for menor que um certo limite (por exemplo, um ou dois elementos), resolve-se o problema usando um método direto e retorna-se a solução obtida. Em qualquer outro caso, divide-se os dados de entrada em dois ou mais conjuntos disjuntos.
2. **Recursão:** soluciona-se os problemas associados aos subconjuntos recursivamente.
3. **Conquista:** obtém-se as soluções dos subproblemas e junta-se as mesmas em uma única solução para o problema original.

### Usando divisão e conquista para ordenação

No problema da ordenação tem-se uma coleção de  $n$  objetos, tipicamente armazenados em uma lista ou arranjo, junto com algum comparador que define uma relação total de ordem nesses objetos, e que se deve gerar uma representação ordenada deles. O algoritmo de ordenação será descrito em alto nível para seqüências e será explicado em detalhes o que é preciso para implementá-las será descrito usando listas e arranjos. Para o problema de ordenar uma seqüência de  $n$  elementos, os três passos de divisão e conquista são os seguintes:

1. **Divisão:** se  $S$  tem zero ou um elemento, retorna-se  $S$  imediatamente; já está ordenado. Em qualquer outro caso ( $S$  tem pelo menos dois elementos), removem-se todos os elementos de  $S$  e colocam-se em duas seqüências,  $S_1$  e  $S_2$ , cada uma contendo aproximadamente a metade dos elementos de  $S$ , ou seja,  $S_1$  contém os primeiros  $\lceil n / 2 \rceil$  elementos de  $S$  e  $S_2$  contém os restantes  $\lfloor n / 2 \rfloor$  elementos.
2. **Recursão:** ordenam-se recursivamente as seqüências  $S_1$  e  $S_2$ .
3. **Conquista:** os elementos são colocados de volta em  $S$ , unindo as seqüências  $S_1$  e  $S_2$  em uma seqüência ordenada.

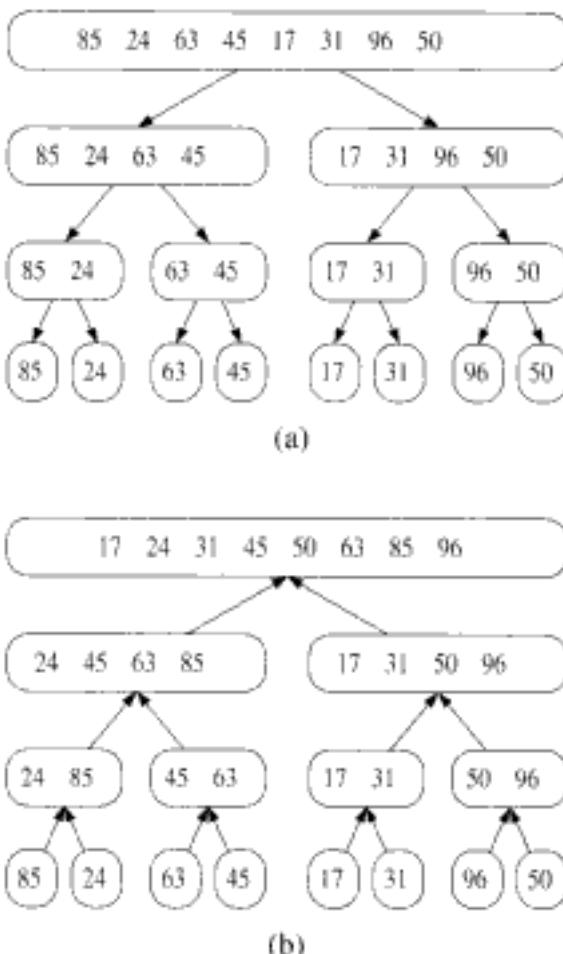
No que se refere ao passo de divisão, é importante lembrar que a notação  $\lceil x \rceil$  indica o **teto** de  $x$ , ou seja, o menor inteiro  $m$  que satisfaz  $x \leq m$ . Da mesma forma, a notação  $\lfloor x \rfloor$  indica o **piso** de  $x$ , ou seja, o maior inteiro  $k$  que satisfaz  $k \leq x$ .

Pode-se visualizar a execução do algoritmo merge-sort usando uma árvore binária  $T$ , chamada de **árvore merge-sort**. Cada nodo de  $T$  representa uma invocação recursiva (ou chamada) do algoritmo merge-sort. Associa-se com cada nodo  $v$  de  $T$  a seqüência  $S$  que é processada pela invocação associada com  $v$ . Os filhos do nodo  $v$  são associados com as chamadas recursivas que processam as subseqüências  $S_1$  e  $S_2$  de  $S$ . Os nodos externos de  $T$  são associados com elementos individuais de  $S$ , correspondendo a instâncias do algoritmo que não fazem chamadas recursivas.

A Figura 11.1 resume uma execução do algoritmo merge-sort, mostrando as seqüências de entrada e saída processadas em cada nodo da árvore merge-sort. A evolução passo a passo dessa árvore é apresentada nas Figuras 11.2 a 11.4.

Esta visualização do algoritmo em termos da árvore merge-sort ajuda a analisar o tempo de execução do algoritmo merge-sort. Em especial, uma vez que o tamanho da seqüência de entrada

é grosseiramente dividido pela metade a cada chamada recursiva do merge-sort, a altura da árvore merge-sort se aproxima de  $\log n$  (lembre que a base de  $\log$  é 2, se omitida).



**Figura 11.1** Árvore merge-sort  $T$  para uma execução do algoritmo merge-sort em uma seqüência com 8 elementos: (a) seqüência de entrada processada em cada nodo de  $T$ ; (b) seqüência de saída geradas em cada nodo de  $T$ .

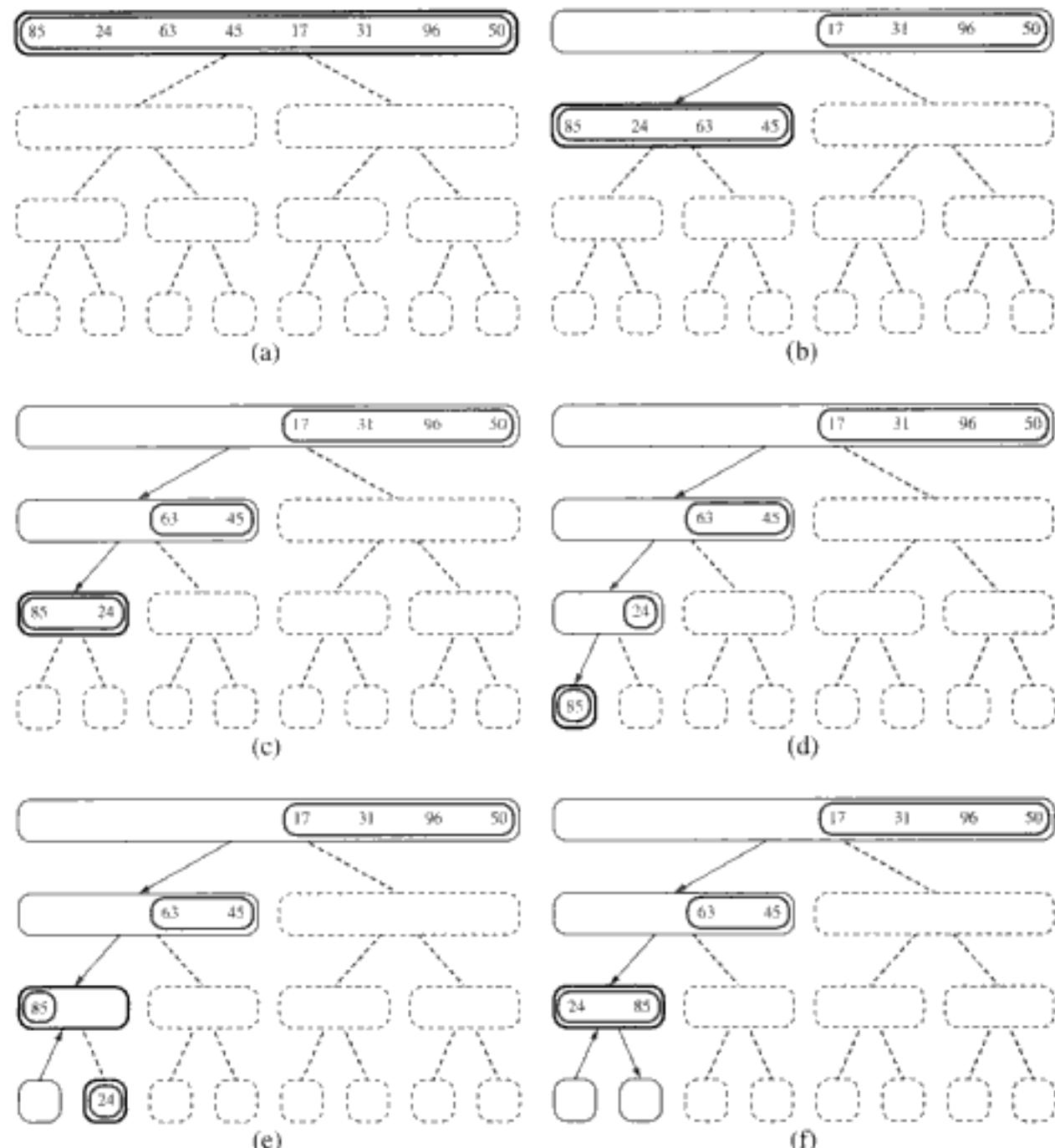
**Proposição 11.1** A árvore merge-sort associada com uma execução do merge-sort a partir de uma seqüência de tamanho  $n$  tem altura  $\lceil \log n \rceil$

A justificativa da Proposição 11.1 é deixada como um exercício simples (R-11.3). Esta proposição será usada para analisar o tempo de execução do algoritmo merge-sort.

A partir de uma visão geral do merge-sort e da ilustração de seu funcionamento, vamos considerar cada um dos passos deste algoritmo de divisão e conquista em maiores detalhes. Os passos de divisão e recursão do algoritmo merge-sort são simples; dividir uma seqüência de tamanho  $n$  envolve separá-la no elemento de ordem  $\lceil n/2 \rceil$  e as chamadas recursivas compreendem simplesmente passar essas seqüências menores como parâmetro. O passo difícil é o de conquista que faz a junção de duas seqüências ordenadas em uma única. Consequentemente, antes de apresentar a análise do merge-sort, é necessário explicar melhor como isso é feito.

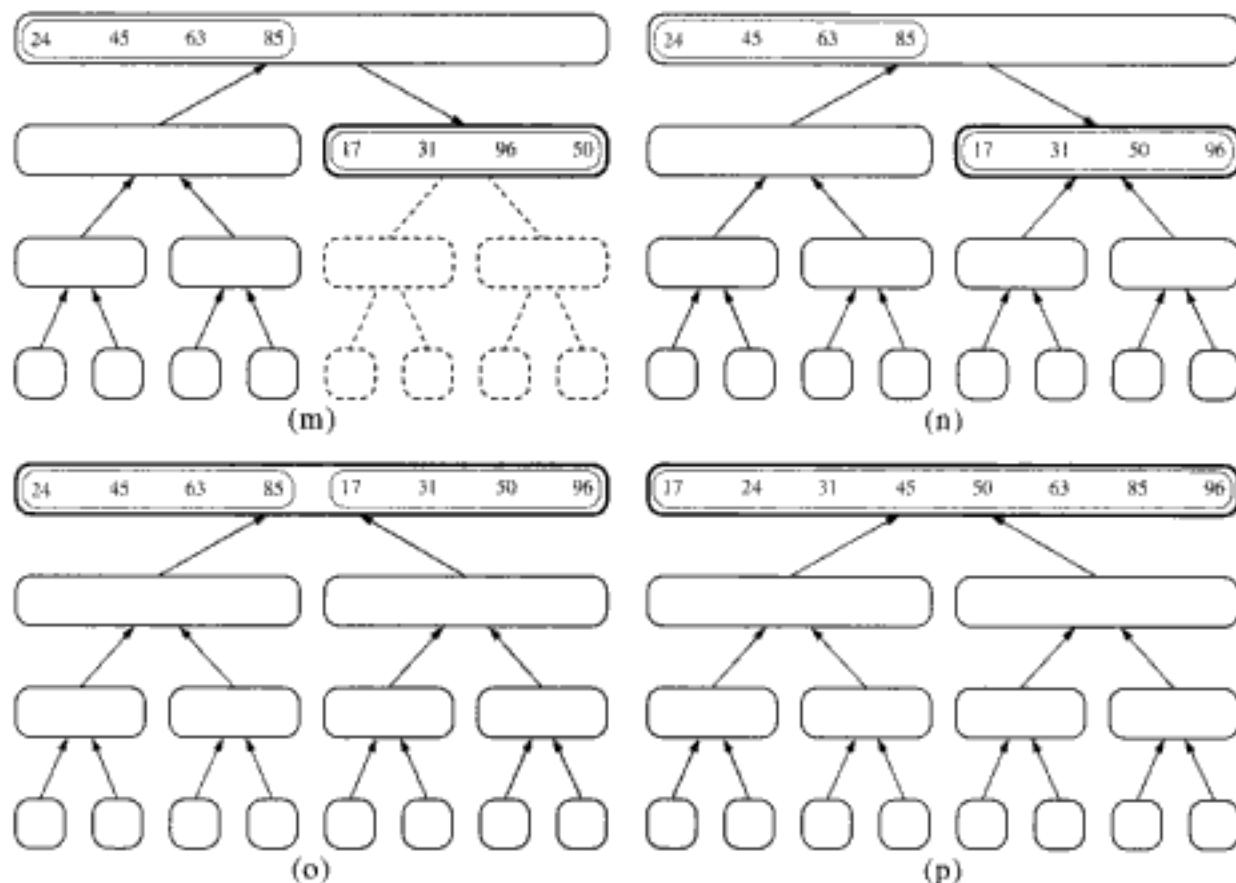
### 11.1.2 Junção de arranjos e listas

Para unir duas seqüências ordenadas, é desejável conhecer se elas foram implementadas como arranjos ou listas. Desta forma, nesta seção são apresentados detalhes do pseudocódigo descrevendo como unir duas seqüências ordenadas representadas como arranjos e como listas encadeadas.



**Figura 11.2** Visualização de uma execução do merge-sort. Cada nodo da árvore representa uma chamada recursiva do merge-sort. Os nodos desenhados com linhas pontilhadas mostram chamadas que ainda não foram feitas. Os nodos desenhados com linhas grossas representam as chamadas correntes. Os nodos vazios desenhados com linhas finas indicam chamadas completadas. Os nodos restantes (desenhados com linhas finas e que não estão vazios) representam chamadas que estão esperando pela invocação dos filhos para retornar. (Continua na Figura 11.3.)

Hidden page



**Figura 11.4** Visualização de uma execução do merge-sort. Várias invocações são omitidas entre (l) e (m) e entre (m) e (n). Vide o passo da conquista no passo (p). (Continuação da Figura 11.3.)

### Unindo dois arranjos ordenados

Inicia-se com a implementação com arranjo, apresentada no Trecho de código 11.1. Um passo em uma junção de dois arranjos ordenados é ilustrado na Figura 11.5.

**Algoritmo**  $\text{merge}(S_1, S_2, S)$ :

**Entrada:** Seqüências ordenadas  $S_1$  e  $S_2$  e uma seqüência vazia  $S$ , todos implementados como arranjos.

**Saída:** Seqüência ordenada  $S$  contendo os elementos de  $S_1$  e  $S_2$ .

$i \leftarrow j \leftarrow 0$

**enquanto**  $i < S_1.\text{size}()$  e  $j < S_2.\text{size}()$  **faça**

**se**  $S_1.\text{get}(i) \leq S_2.\text{get}(j)$  **então**

$S.\text{addLast}(S_1.\text{get}(i))$  {copia o  $i$ -nésimo elemento de  $S_1$  para o final de  $S$ }

$i \leftarrow i + 1$

**senão**

$S.\text{addLast}(S_2.\text{get}(j))$  {copia o  $j$ -nésimo elemento de  $S_2$  para o final de  $S$ }

$j \leftarrow j + 1$

**enquanto**  $i < S_1.\text{size}()$  **faça** {copia os elementos restantes de  $S_1$  para  $S$ }

$S.\text{addLast}(S_1.\text{get}(i))$

$i \leftarrow i + 1$

**enquanto**  $j < S_2.\text{size}()$  **faça** {copia os elementos restantes de  $S_2$  para  $S$ }

$S.\text{addLast}(S_2.\text{get}(j))$

$j \leftarrow j + 1$

**Trecho de código 11.1** Algoritmo para junção de dois arranjos ordenados baseados em seqüências.

$S_1$	0 1 2 3 4 5 6
	2   5   8   11   12   14   15
	$i$
$S_2$	0 1 2 3 4 5 6

$S_2$	0 1 2 3 4 5 6
	3   9   10   18   19   22   25
	$j$
$S$	0 1 2 3 4 5 6 7 8 9 10 11 12

(a)

$S_1$	0 1 2 3 4 5 6
	2   5   8   11   12   14   15
	$i$
$S_2$	0 1 2 3 4 5 6

$S_2$	0 1 2 3 4 5 6
	3   9   10   18   19   22   25
	$j$
$S$	0 1 2 3 4 5 6 7 8 9 10 11 12

(b)

**Figura 11.5** Um passo na junção de dois arranjos ordenados. São mostrados os arranjos antes do passo da cópia (a) e depois deste passo (b).

### Unindo duas seqüências ordenadas

O algoritmo merge, no Trecho de código 10.1, faz a junção de duas seqüências ordenadas,  $S_1$  e  $S_2$ , implementada como lista encadeada. A idéia principal é remover iterativamente o menor elemento das duas, acrescentando-o no final da seqüência resultante,  $S$ , até que uma das duas seqüências esteja vazia, momento a partir do qual se copia o restante da outra seqüência para a resultante  $S$ . Um exemplo da execução desta versão do algoritmo merge é mostrado na Figura 11.6.

#### Algoritmo merge( $S_1$ , $S_2$ , $S$ ):

**Entrada:** Seqüências ordenadas  $S_1$  e  $S_2$  e uma seqüência vazia  $S$ , implementada como listas encadeadas.

**Saída:** Seqüência ordenada  $S$  contendo os elementos de  $S_1$  e  $S_2$ .

**enquanto**  $S_1$  não estiver vazia e  $S_2$  não estiver vazia **faça**

**se**  $S_1.\text{first}().\text{element}() < S_2.\text{first}().\text{element}()$  **então**

        { move o primeiro elemento de  $S_1$  para o final de  $S$ }

$S.\text{addLast}(S_1.\text{remove}(S_1.\text{first}()))$

**senão**

        { move o primeiro elemento de  $S_2$  para o final de  $S$ }

$S.\text{addLast}(S_2.\text{remove}(S_2.\text{first}()))$

    { move os elementos restantes de  $S_1$  para  $S$ }

**enquanto**  $S_1$  não estiver vazia **faça**

$S.\text{addLast}(S_1.\text{remove}(S_1.\text{first}()))$

    { move os elementos restantes de  $S_2$  para  $S$ }

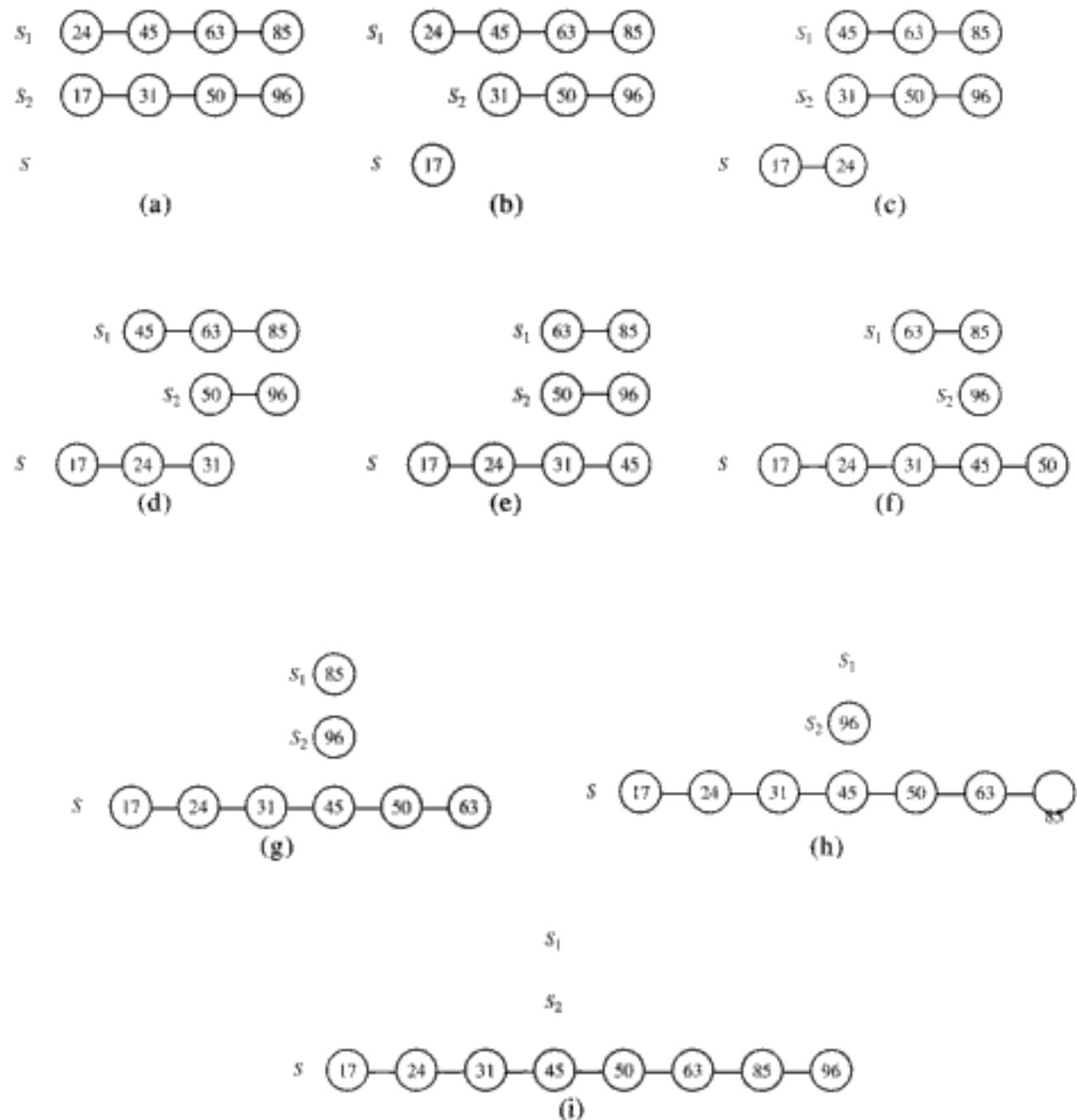
**enquanto**  $S_2$  não estiver vazia **faça**

$S.\text{addLast}(S_2.\text{remove}(S_2.\text{first}()))$

**Trecho de código 11.2** Algoritmo de merge para a junção de duas seqüências ordenadas implementadas como listas encadeadas.

### Tempo de execução para a junção

Analisa-se o tempo de execução do algoritmo merge para fazer algumas observações. Seja  $n_1$  e  $n_2$  o número de elementos de  $S_1$  e  $S_2$ , respectivamente. O algoritmo merge tem três laços while. Independente de se analisar a versão baseada em arranjo ou a versão baseada em lista, as operações são executadas dentro de cada laço, e levam o tempo  $O(1)$  cada. A observação chave é que durante cada iteração de um dos laços, um elemento é copiado ou movido ou de  $S_1$  ou  $S_2$  para  $S$  (e que o elemento é considerado até o momento). Desde que inserções não são executadas em  $S_1$  e  $S_2$ , esta observação implica que o número completo dos três laços é  $n_1 + n_2$ . Desta forma, o tempo de execução do algoritmo merge é  $O(n_1 + n_2)$ .



**Figura 11.6** Exemplo de uma execução do algoritmo de merge mostrado no Trecho de código 11.2.

### 11.1.3 O tempo de execução do merge-sort

Agora que se têm os detalhes do algoritmo merge-sort e analisou-se o tempo de execução do algoritmo decisivo, merge, usado na etapa de conquista, se examinará o tempo de execução do algoritmo merge-sort completo, considerando que é fornecida uma seqüência de entrada com  $n$  elementos. Para facilitar, restringe-se a atenção ao caso em que  $n$  é potência de 2. Deixa-se um exercício (R-11.6) para mostrar que o resultado da análise também pode ser aplicado quando  $n$  não é potência de 2.

Como foi feito na análise do algoritmo merge, assume-se que a seqüência de entrada  $S$  e as seqüências auxiliares  $S_1$  e  $S_2$ , criadas por cada chamada recursiva do merge-sort, são implementadas ou com arranjos ou listas encadeadas (o mesmo que  $S$ ), assim a junção de duas seqüências ordenadas podem ser feitas em um tempo linear.

Como se mencionou anteriormente, analisa-se a rotina merge-sort, referenciando a árvore merge-sort  $T$ . (Vide Figuras 11.2 a 11.4.) Chama-se o *tempo gasto em um nodo*  $v$  de  $T$  de tempo de execução de uma chamada recursiva associada com  $v$ , excluindo o tempo gasto esperando pelas chamadas recursivas associadas com os filhos de  $v$  para terminar. Em outras palavras, o tempo gasto no nodo  $v$  inclui os tempos de execução dos passos de divisão e conquista, mas excluem os tempos de execução do passo de recursão. Já se observou que os detalhes do passo de divisão são diretos; esse passo executa em tempo proporcional ao tamanho da seqüência  $v$ . Da mesma forma, o passo de conquista, que consiste na junção de duas subseqüências, consome tempo linear, independentemente de se estar usando arranjos ou listas encadeadas. Ou seja, fazendo  $i$  denotar a profundidade de um nodo  $v$ , o tempo gasto no nodo  $v$  é  $O(n/2^i)$ , uma vez que o tamanho da seqüência manipulada pelas chamadas recursivas associadas com  $v$  é igual a  $n/2^i$ .

Examinando a árvore  $T$  de forma mais global, como mostrado na Figura 11.7, vê-se que, dada nossa definição de “tempo gasto em um nodo”, o tempo de execução do merge-sort é igual à soma dos tempos gastos nos nodos de  $T$ . Vide que  $T$  tem exatamente  $2^i$  nodos na profundidade  $i$ . Essa observação simples tem uma consequência importante, pois implica que o tempo total gasto em todos os nodos de  $T$  na profundidade  $i$  é  $O(2^i \cdot n/2^i)$ , o que corresponde a  $O(n)$ . Pela Proposição 11.1, a altura de  $T$  é  $\lceil \log n \rceil$ . Portanto, uma vez que o tempo gasto em cada um dos  $\lceil \log n \rceil + 1$  níveis de  $T$  é  $O(n)$ , se terá o seguinte resultado:

**Proposição 11.2:** *O algoritmo merge-sort ordena uma seqüência de tamanho  $n$  em tempo  $O(n \log n)$ , assumindo que dois elementos de  $S$  podem ser comparados no tempo  $O(1)$ .*

Em outras palavras, o algoritmo merge-sort combina assintoticamente o tempo mais rápido do algoritmo heap-sort.

#### 11.1.4 Implementações Java do merge-sort

Nesta seção, são apresentadas duas implementações Java do algoritmo de merge-sort, um para listas e outro para arranjos.

##### Uma implementação recursiva do merge-sort baseada em seqüência

No Trecho de código 11.3, mostra-se uma implementação completa em Java do algoritmo de merge-sort baseada em seqüência com um método estático recursivo – mergerSort. Um comparador (ver Seção 8.1.2) é usado para decidir a ordem relativa dos dois elementos.

Nesta implementação, a entrada é uma seqüência  $L$ , e as listas auxiliares  $L1$  e  $L2$  são processadas pelas chamadas recursivas. Cada seqüência é modificada por inserções e remoções nos extremos da seqüência (*head* e *tail*) somente; consequentemente, cada alteração da seqüência leva o tempo  $O(1)$ , assumindo que as seqüências são implementadas com listas duplamente encadeadas (ver Tabela 6.4). No nosso código, usa-se a classe NodeList (Trecho de código 6.9 – 6.11) para as seqüências auxiliares. Assim, para uma lista  $L$  de tamanho  $n$ , o método *mergeSort(L,c)* executa no tempo  $O(n \log n)$  desde que a seqüência  $L$  seja implementada com uma lista duplamente encadeada e o comparador  $c$  possa comparar dois elementos de  $L$  no tempo  $O(1)$ .

```
/*
 * Ordena os elementos da seqüência em ordem não decrescente
 * de acordo com o comparador c, usando o algoritmo merge-sort
 */
public static <E> void mergeSort (PositionList<E> in, Comparator<E> c) {
    int n = in.size();
    if (n < 2)
        return; // a seqüência já está ordenada
    // divide
```

Hidden page

Uma implementação não-recursiva baseada em arranjo do merge-sort

Existe uma versão não recursiva do merge-sort baseada em arranjo, que executa no tempo  $O(n \log n)$ . Ele é, na prática, um pouco mais rápido que o merge-sort recursivo baseado em seqüências, por ele evitar *numerosas chamadas recursivas extras* e criação de nodos. A idéia principal é executar o merge-sort de baixo para cima, executando as junções nível a nível até realizar a junção para a árvore inteira. Dado um arranjo de elementos de entrada, inicia-se pela junção de todo par ímpar de elementos em execuções ordenadas de tamanho dois. Junta-se essas execuções em execuções de quatro, estas novas execuções são agrupadas em execuções de oito e assim por diante, até que o arranjo esteja ordenado. Para manter o uso do espaço razoável, desenvolve-se um arranjo de saída que armazena as execuções de junção (trocando os arranjos de entrada e saída após cada iteração). Uma implementação em Java é apresentada no Trecho de código 11.4, onde se usa o método `System.arraycopy` para copiar um intervalo de células entre dois arranjos.

```
/** Ordena um arranjo com um comparador usando um merge-sort não recursivo. */
public static <E> void mergeSort(E[] orig, Comparator<E> c) {
    E[] in = (E[]) new Object[orig.length]; // cria um novo arranjo temporário
    System.arraycopy(orig, 0, in, 0, in.length); // copia a entrada
    E[] out = (E[]) new Object[in.length]; // arranjo de saída
    E[] temp; // arranjo temporário referência, usado para trocas
    int n = in.length;
    for (int i=1; i < n; i*=2) { // cada iteração ordena todos executando tamanho-2*i vezes
        for (int j=0; j < n; j+=2*i) // cada iteração junta dois pares de tamanhos i
            merge(in, out, c, j, i); // junta in e out
        temp = in; in = out; out = temp; // troca os arranjos para a próxima iteração
    }
    // o arranjo "in" contém o arranjo ordenado, assim ele será recopiado
    System.arraycopy(in, 0, orig, 0, in.length);
}
/** Junta dois subarranjos, especificados por um início e incremento. */
protected static <E> void merge(E[] in, E[] out, Comparator<E> c, int start,
    int inc) { // junta in[start..start+inc-1] e in[start+inc..start+2*inc-1]
    int x = start; // índice para execução #1
    int end1 = Math.min(start+inc, in.length); // limite para execução #1
    int end2 = Math.min(start+2*inc, in.length); // limite para execução #2
    int y = start+inc; // índice para execução #2 (poderia ser além do arranjo limite)
    int z = start; // índice para o arranjo out
    while ((x < end1) && (y < end2))
        if (c.compare(in[x], in[y]) <= 0) out[z++] = in[x++];
        else out[z++] = in[y++];
    if (x < end1) // primeira execução não finaliza
        System.arraycopy(in, x, out, z, end1 - x);
    else if (y < end2) // segunda execução não finaliza
        System.arraycopy(in, y, out, z, end2 - y);
}
```

**Trecho de código 11.4** Uma implementação do algoritmo não-recursivo merge-sort.

### 11.1.5 O merge-sort e suas relações de recorrência \*

Existe outra forma de justificar que o tempo de execução do algoritmo merge-sort é  $O(n \log n)$  (Proposição 11.2). De fato, existe uma justificativa que lida de forma mais direta com a natureza recursiva do algoritmo merge-sort. Nesta seção, apresenta-se esta análise do tempo de execução

do merge-sort e, fazendo isso, o conceito matemático de *equação de recorrência* é introduzido (também conhecido como *relação de recorrência*).

Faça a função  $t(n)$  denotar o pior caso no que diz respeito ao tempo de execução do merge-sort a partir de uma seqüência de entrada de tamanho  $n$ . Uma vez que o merge-sort é recursivo, pode-se caracterizar a função  $t(n)$  em termos das seguintes igualdades, onde a função  $t(n)$  é expressa recursivamente em termos de si mesma. Para simplificar nossa caracterização de  $t(n)$ , restringe-se a atenção ao caso onde  $n$  é uma potência de 2. (O problema de mostrar que a caracterização assintótica ainda permanece no caso geral ficará como um exercício.) Neste caso, pode-se especificar a definição de  $t(n)$  como

$$t(n) = \begin{cases} b & \text{se } n \leq 1 \\ 2t(n/2) + cn & \text{caso contrário} \end{cases}$$

Uma expressão como a apresentada anteriormente é chamada de *equação de recorrência*, uma vez que a função aparece tanto do lado esquerdo como do lado direito da igualdade. Apesar dessa caracterização ser correta e precisa, o que realmente se quer é uma caracterização mais evidente de  $t(n)$  que não envolva a própria função  $t(n)$ . Ou seja, deseja-se uma caracterização de *forma fechada* para  $t(n)$ .

Pode-se obter uma solução *forma fechada* por meio da aplicação da definição de uma equação de recorrência, assumindo que  $n$  é relativamente grande. Por exemplo, após um ou mais aplicações da equação acima, pode-se escrever uma nova recorrência para  $t(n)$  como

$$\begin{aligned} t(n) &= 2(2t(n/2^2) + (cn/2)) + cn \\ &= 2^2t(n/2^2) + 2(cn/2) + cn = 2^2t(n/2^2) + 2cn. \end{aligned}$$

Se a equação for novamente aplicada,  $t(n) = 2^2t(n/2^3) + 3cn$ . Neste ponto, deve-se ver um padrão surgindo, assim após aplicar esta equação  $i$  vezes obtém-se

$$t(n) = 2^it(n/2^i) + icn.$$

O ponto que resta, então, é determinar quando parar esse processo. Para determinar quando parar, considera-se que se troca para a forma fechada  $t(n) = b$  quando  $n \leq 1$ , o que irá ocorrer quando  $2^i = n$ . Em outras palavras, irá ocorrer quando  $i = \log n$ . Procedendo essa substituição, tem-se

$$\begin{aligned} t(n) &= 2^{\log n}t(n/2^{\log n}) + (\log n)cn \\ &= nt(1) + cn\log n \\ &= nb + cn\log n. \end{aligned}$$

Isto é, obtemos uma justificativa alternativa para o fato de que  $t(n)$  é  $O(n \log n)$ .

## 11.2 Quick-sort

O próximo algoritmo de ordenação que será discutido é chamado de *quick-sort*. Da mesma forma que o merge-sort, este algoritmo também se baseia no paradigma de *divisão e conquista*, mas usa esta técnica de forma contrária, uma vez que o trabalho pesado é feito *antes* das chamadas recursivas.

**Descrevendo o quick-sort em alto nível**

O algoritmo quick-sort ordena uma seqüência  $S$  usando uma abordagem recursiva simples. A idéia principal é aplicar a técnica de divisão e conquista dividindo  $S$  em subseqüências e aplicando recursão para ordenar cada subseqüência, para, então, combinar as subseqüências ordenadas por concatenação simples. Em detalhes, o algoritmo quick-sort consiste nos três passos seguintes (ver Figura 11.8):

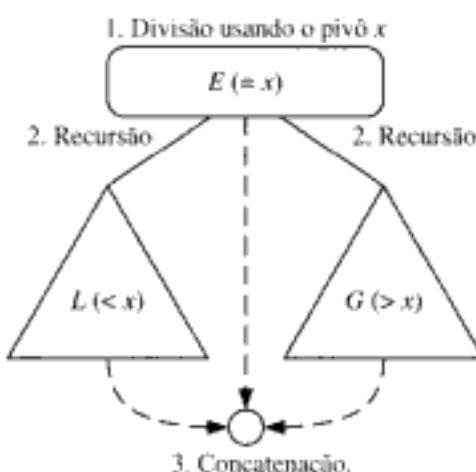
1. **Divisão:** se  $S$  tiver pelo menos dois elementos (nada precisa ser feito se  $S$  tiver zero ou um elemento), escolhe-se um elemento  $x$  de  $S$ , chamado de **pivô**. Normalmente, escolhe-se como pivô  $x$  o último elemento de  $S$ . Removem-se todos os elementos de  $S$  e eles são colocados em três seqüências:

- $L$ , armazenando os elementos de  $S$  menores que  $x$ ;
- $E$ , armazenando os elementos de  $S$  iguais a  $x$ ;
- $G$ , armazenando os elementos de  $S$  maiores que  $x$ .

Naturalmente, se os elementos de  $S$  forem todos diferentes, então  $E$  armazena apenas um elemento, o próprio pivô.

2. **Recursão:** ordenar as seqüências  $L$  e  $G$ , recursivamente.

3. **Conquista:** colocar de volta os elementos em  $S$  em ordem inserindo primeiro os elementos de  $L$ , em seguida os de  $E$  e, por fim, os de  $G$ .



**Figura 11.8** Um esquema visual do algoritmo quick-sort.

Da mesma forma que o merge-sort, a execução do quick-sort pode ser visualizada em termos de uma árvore binária recursiva chamada de **árvore quick-sort**. A Figura 11.9 resume a execução do algoritmo quick-sort mostrando as seqüências de entrada e saída processadas em cada nodo da árvore quick-sort. A evolução passo a passo da árvore quick-sort é apresentada nas Figuras 11.10, 11.11 e 11.12.

Ao contrário do merge-sort, entretanto, a altura da árvore quick-sort associada com a execução do quick-sort é linear no pior caso. Isso acontece, por exemplo, quando a seqüência consiste em  $n$  elementos distintos e já está ordenada. Na verdade, neste caso, a escolha padrão do maior elemento para pivô produz uma subseqüência  $L$  de tamanho  $n - 1$ , enquanto a subseqüência  $E$  tem tamanho 1 e a subseqüência  $G$  tem tamanho 0. A cada invocação do quick-sort sobre a subseqüência  $L$ , o tamanho diminui em 1 unidade. Desta forma, a altura da árvore quick-sort é  $n - 1$ .

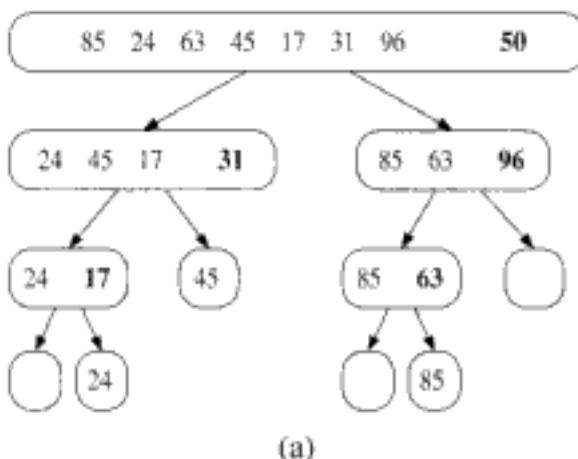
### Execução do quick-sort em arranjos e seqüências

No Trecho de código 11.5, apresenta-se a descrição de um pseudocódigo do algoritmo do quick-sort que é eficiente para seqüências implementadas com arranjos ou listas encadeadas. O algoritmo segue o template do quick-sort apresentado acima, adicionando o detalhe da procura na seqüência  $S$  de entrada iniciando no final para dividi-la nas seqüências  $L$ ,  $E$  e  $G$  de elementos que são respectivamente menores, iguais e maiores que o pivô. Executa-se esta varredura para trás, visto que remover o último elemento na seqüência é uma operação de tempo constante independente se a seqüência for implementada como um arranjo ou uma lista encadeada. Então recorre-se às seqüências  $L$  e  $G$  e copia-se a seqüência ordenada  $L$ ,  $E$  e  $G$  de volta para  $S$ . Executa-se este

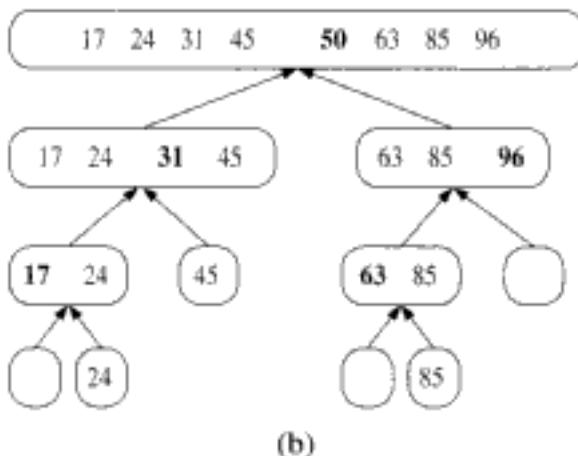
último conjunto de cópias para frente, visto que inserir elementos no final da seqüência é uma operação de tempo constante independente se a seqüência for implementada como um arranjo ou uma lista encadeada.

**Algoritmo QuickSort( $S$ ):**

**Entrada:** Uma seqüência  $S$  implementada como um arranjo ou lista encadeada.  
**Saída:** A seqüência  $S$  ordenada  
**se**  $S.size() \leq 1$  **então**  
 Retorna { $S$  já está ordenada neste caso}  
 $p \leftarrow S.last().element()$  {o pivô}  
 Seja  $L, E$  e  $G$  seqüências baseadas em listas  
**enquanto**  $\neg S.isEmpty()$  **faça** {rastrei  $S$  de trás para a frente, dividindo ela em  $L, E$  e  $G$ }  
**se**  $S.last().element() < p$  **então**  
 $L.addLast(S.remove(S.getLast()))$   
**senão** **se**  $S.last().element() = p$  **então**  
 $E.addLast(S.remove(S.getLast()))$   
**else** {o último elemento de  $S$  é maior que  $p$ }  
 $G.addLast(S.remove(S.getLast()))$   
 QuickSort( $L$ ) {executa novamente com os elementos menores que  $p$ }  
 QuickSort( $G$ ) {executa novamente com os elementos maiores que  $p$ }  
**enquanto**  $\neg L.isEmpty()$  **faça** {copia para o final de  $S$  os elementos ordenados menores que  $p$ }



(a)



(b)

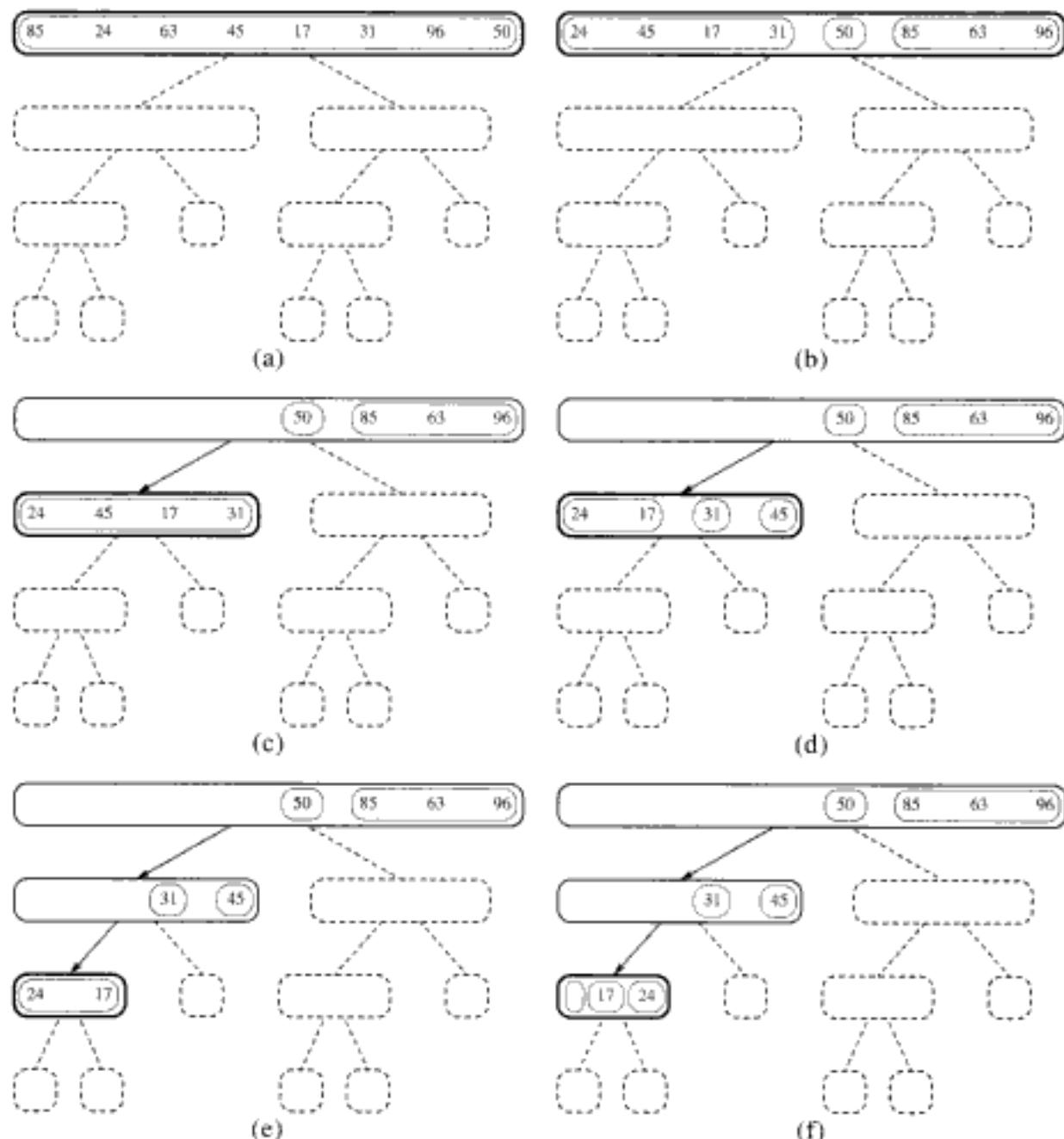
**Figura 11.9** Árvore quick-sort  $T$  correspondente a uma execução do algoritmo quick-sort em uma seqüência de 8 elementos: (a) seqüências de entrada processadas em cada nodo de  $T$ ; (b) seqüências de saída geradas em cada nodo de  $T$ . O pivô usado em cada nível de recursão é mostrado em negrito.

```

S.addLast(L.remove(L.getFirst( )))
enquanto !E.isEmpty() faça { copia para o final de S os elementos iguais a p }
    S.addLast(E.remove(E.getFirst( )))
enquanto !G.isEmpty() faça { copia para o final de S os elementos ordenados maiores que p }
    S.addLast(G.remove(G.getFirst( )))
retorna      { S está agora ordenado }

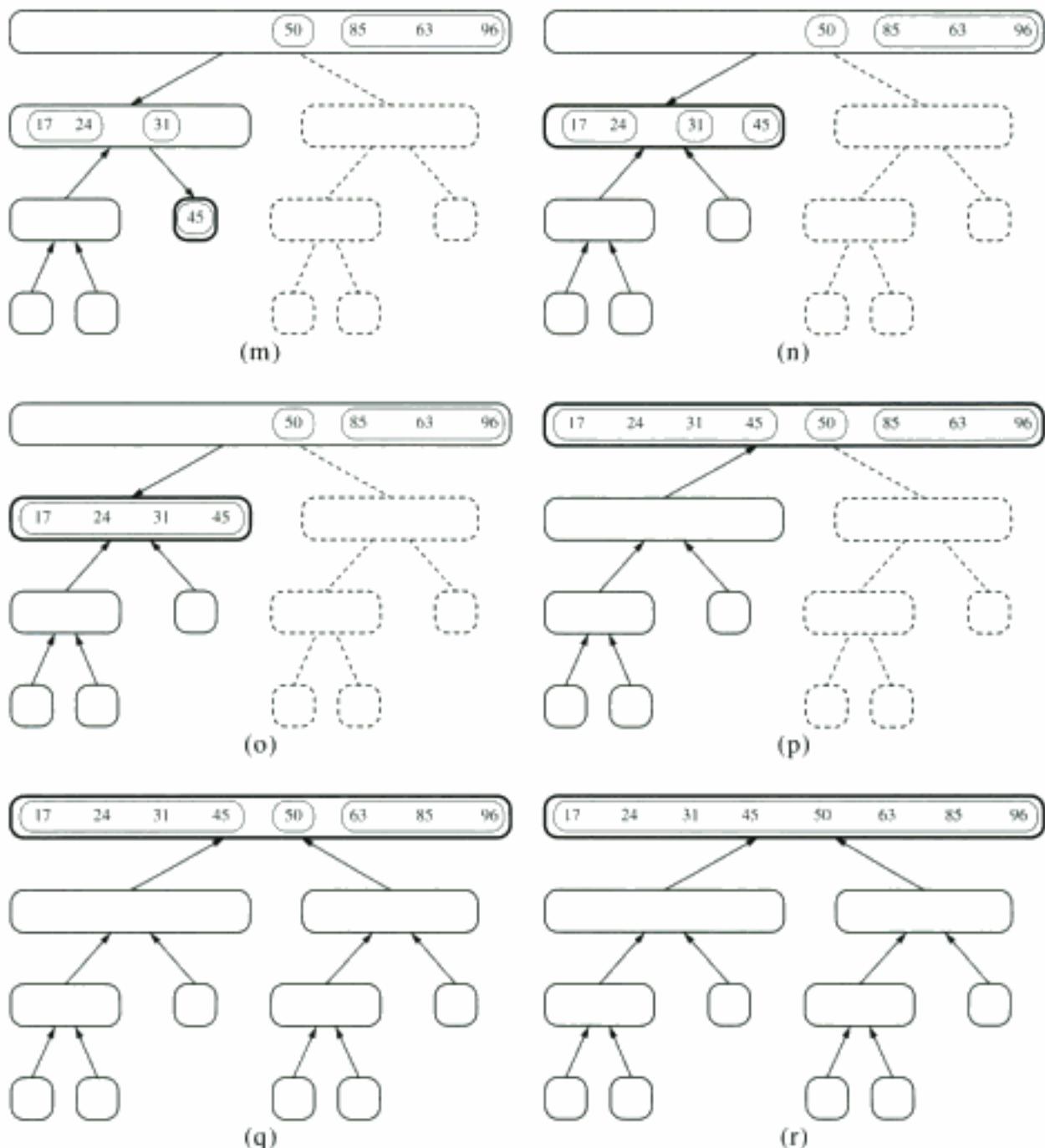
```

**Trecho de código 11.5** Quick-sort para uma seqüência de entrada *S* implementada com uma lista encadeada ou um arranjo.



**Figura 11.10** Visualização do quick-sort. Cada nodo da árvore representa uma chamada recursiva. Os nodos desenhados com linhas pontilhadas indicam chamadas que ainda não foram feitas. Os nodos desenhados com linhas grossas mostram as invocações que ainda estão executando. Os nodos vazios desenhados com linhas finas representam chamadas encerradas. Os nodos restantes indicam chamadas suspensas (isto é, invocações ativas que estão esperando pelo retorno de uma invocação filha). Observa-se os passos de divisão executados em (b), (d) e (f). (Continua na Figura 11.11.)

Hidden page



**Figura 11.12** Visualização de uma execução do *quick-sort*. Várias invocações entre (p) e (q) foram omitidas. Observam-se os passos da conquista executados em (o) e (r). (Continuação da Figura 11.11.)

completa. Da mesma forma,  $s_1 \leq n - 1$ , uma vez que o pivô não é propagado para o filho de  $r$ . Considere-se em seguida  $s_2$ . Se os dois filhos de  $r$  têm tamanho de entrada diferente de zero, então  $s_2 = n - 3$ . Em qualquer outro caso (um filho da raiz tem tamanho zero, o outro tem tamanho  $n - 1$ ),  $s_2 = n - 2$ . Desta forma,  $s_2 \leq n - 2$ . Continuando esta linha de raciocínio, obtém-se que  $s_i \leq n - i$ . Como observado na Seção 10.3, a altura de  $T$  é  $n - 1$  no pior caso. Sendo assim, o pior caso para o tempo de execução do quick-sort é  $O(\sum_{i=0}^{n-1} s_i)$ , que é  $O(\sum_{i=0}^{n-1} (n - i))$ , o qual é  $O(\sum_{i=1}^n i)$ . Pela Proposição 4.3,  $\sum_{i=1}^n i$  é  $O(n^2)$ . Assim, o quick-sort executa no pior caso no tempo  $O(n^2)$ .

Considerando o nome, pode-se esperar que o quick-sort execute de forma rápida. Entretanto, os limites quadráticos mostrados indicam que o quick-sort é lento no pior caso. Paradoxalmente, esse comportamento do pior caso ocorre em situações problemáticas em que a ordenação é sim-

ples – se a seqüência já estiver ordenada. Além disso, pode-se mostrar que o quick-sort tem uma performance pobre, mesmo se a seqüência estiver “quase” ordenada.

Voltando à análise, observe que o melhor caso do quick-sort sobre uma seqüência de elementos distintos ocorre quando as subseqüências  $L$  e  $G$  têm, aproximadamente, o mesmo tamanho. Ou seja, no melhor caso tem-se

$$\begin{aligned}s_0 &= n \\ s_1 &= n - 1 \\ s_2 &= n - (1 + 2) = n - 3 \\ &\vdots \\ s_i &= n - (1 + 2 + 2^2 + \dots + 2^{i-1}) = n - (2^i - 1)\end{aligned}$$

Logo, no melhor caso,  $T$  tem altura  $O(\log n)$  e o quick-sort executa em tempo  $O(n \log n)$ ; deixa-se a justificativa deste fato para o Exercício R-11.11.

A intuição informal por trás do comportamento esperado do quick-sort é que a cada invocação, provavelmente, o pivô irá dividir a seqüência de entrada em partes mais ou menos iguais. Desta forma, espera-se que o tempo médio de execução do quick-sort seja semelhante ao tempo do melhor caso, ou seja,  $O(n \log n)$ . Na próxima seção, será mostrado que a introdução de randomização torna o comportamento do quick-sort exatamente o que acabou de ser descrito.

### 11.2.1 Quick-sort randômico

Uma forma normal de se analisar o quick-sort é supor que o pivô irá sempre dividir a seqüência em partes quase iguais. Esta premissa, entretanto, pressupõe um conhecimento sobre a distribuição da entrada que normalmente não está disponível. Por exemplo, se terá de supor que raramente serão fornecidas seqüências “quase” ordenadas para pôr em ordem, o que pode ser comum em muitas aplicações. Por sorte, essa premissa não é necessária para se combinar nossa intuição com o comportamento do quick-sort.

Em geral, deseja-se alguma forma de fechar com o melhor tempo de execução para o quick-sort. A forma de obter o tempo de execução para o melhor caso é claro, é para o pivô dividir igualmente a seqüência de entrada  $S$ . Se este resultado era para ocorrer, então ele resultaria em um tempo de execução que é assintoticamente o mesmo que o tempo de execução do melhor caso. Isto é, tendo pivôs posicionados no “meio” do conjunto de elementos conduz a um tempo de execução de  $O(n \log n)$  para o quick-sort.

#### Escolhendo pivôs randomicamente

Uma vez que o objetivo do passo de divisão do método quick-sort é dividir a seqüência  $S$  em partes quase iguais, se introduzirá a randomização no algoritmo e escolher como pivô um **elemento randômico** da seqüência de entrada. Ou seja, em vez de escolher como pivô o último elemento de  $S$ , escolhe-se um elemento de  $S$  randomicamente para ser o pivô, mantendo o resto do algoritmo inalterado. Esta variação do quick-sort é chamada de **quick-sort randomizado**. A proposição a seguir mostra que o tempo esperado de execução do quick-sort randomizado em uma seqüência de  $n$  elementos é  $O(n \log n)$ . Esta explicação é tirada a partir de todas as possíveis combinações que o algoritmo pode fazer, e é independente de qualquer premissa sobre a distribuição das possíveis entradas que podem ser fornecidas para o algoritmo.

**Proposição 11.3:** *O tempo esperado de execução para o quick-sort randomizado aplicado em uma seqüência de tamanho  $n$  é  $O(n \log n)$ .*

**Justificativa** Assume-se que dois elementos de  $S$  podem ser comparados no tempo  $O(1)$ . Considere-se agora uma invocação recursiva simples de um quick-sort randomizado, e faça-se  $m$

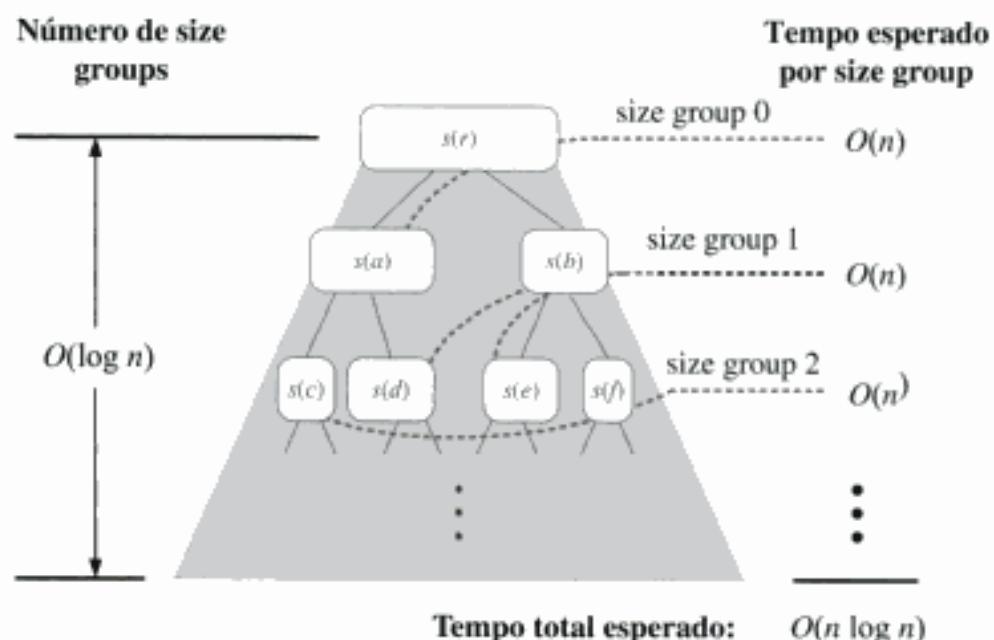
denotar o tamanho da seqüência de entrada para esta invocação. Diz-se que essa invocação é “boa” se o pivô escolhido for tal que as subseqüências  $L$  e  $G$  tenham tamanho no mínimo  $m/4$  e no máximo  $3m/4$  cada; senão, a invocação é “ruim”.

Agora, considerem-se as implicações da escolha de um pivô uniformemente de maneira randômica. Existem  $n/2$  possibilidades de chances boas para que o pivô seja qualquer invocação de tamanho do algoritmo quick-sort randomizado. Assim, a probabilidade que qualquer chamada seja boa será de 50%. Nota-se após que uma boa chamada estará no mínimo em uma partição da seqüência de tamanho  $n$  nas duas seqüências de tamanhos  $3n/4$  e  $n/4$ , e uma chamada ruim poderia ser tão ruim como a produzida com uma chamada simples de tamanho  $n - 1$ .

Considere-se agora uma execução recursiva para o quick-sort randomizado. Esta execução define uma árvore binária  $T$ , em que cada nodo de  $T$  corresponde a uma diferente chamada recursiva de um subproblema de ordenação de uma porção da seqüência original.

Diz-se que um nodo  $v$  em  $T$  está em *size group*  $i$  se o tamanho do subproblema de  $v$  é maior que  $(3/4)^{i+1}$  e no máximo  $(3/4)^i n$ . Analisa-se a seguir o esperado tempo gasto de trabalho em todos os subproblemas para  $s$  nodos no size group  $i$ . Pela linearidade da expectativa (Proposição A.19), o tempo de trabalho esperado de todos estes subproblemas é a soma dos tempos esperados de cada um. Alguns destes nodos correspondem a boas chamadas, e alguns correspondem a chamadas ruins. Porém, nota-se que uma chamada boa ocorre com probabilidade 1/2, e o número esperado de chamadas consecutivas que devem ser feitas antes de conseguir uma boa chamada é 2. Além disso, observa-se que assim que haja uma boa chamada para um nodo no size group  $i$ , seus filhos estarão em size groups maiores que  $i$ . Assim, para qualquer elemento  $x$  de uma seqüência de entrada, o número esperado de nodos no size group  $i$  contendo  $x$  em seus subproblemas é 2. Em outras palavras, o tamanho total esperado de todos os subproblemas no size group  $i$  é  $2n$ . Visto que o trabalho não-recursivo que se executa para qualquer subproblema é proporcional ao seu tamanho, isso implica que o tempo total esperado gasto processando subproblemas para nodos no size group  $i$  é  $O(n)$ .

O número de size groups é  $\log_{4/3} n$ , desde que repetidamente multiplicando por 3/4 é o mesmo que repetidamente por 4/3. Isto é, o número de size groups é  $O(\log n)$ . Então, o tempo de execução total esperado do quick-sort randomizado é  $O(n \log n)$ . (Ver Figura 11.13.) ■



**Figura 11.13:** Uma análise visual do tempo da árvore quick-sort  $T$ . Cada nodo é mostrado rotulado com o tamanho do seu subproblema.

Hidden page

Hidden page

Hidden page

Hidden page

Hidden page

comparado à seqüência de tamanho  $n$ , por exemplo  $N = O(n)$  ou  $N = O(n \log n)$ . Ainda, sua performance se deteriora a medida que  $N$  cresce comparado a  $n$ .

Uma propriedade importante do algoritmo bucket-sort é que ele trabalha corretamente mesmo se existir muitos elementos diferentes com a mesma chave. Na verdade, ele é descrito de forma a antecipar tais ocorrências.

### Ordenação estável

Ao ordenar itens chave-elemento, uma questão importante é como as chaves iguais são tratadas. Seja  $S = ((k_0, e_0), \dots, (k_{n-1}, e_{n-1}))$  uma seqüência de itens. Diz-se que um algoritmo de ordenação é *estável* se para quaisquer dois itens  $(k_i, e_i)$  e  $(k_j, e_j)$  de  $S$ , tal que  $k_i = k_j$  e  $(k_i, e_i)$  precede  $(k_j, e_j)$  em  $S$  antes da ordenação (isto é,  $i < j$ ), e o item  $(k_i, e_i)$  também precede  $(k_j, e_j)$  após a ordenação. A estabilidade é importante para um algoritmo de ordenação, porque as aplicações podem querer preservar a ordenação inicial dos elementos com a mesma chave.

A descrição informal do bucket-sort no Trecho de código 11.8 não garante estabilidade. Esta não é inerente ao método bucket-sort propriamente dito, porém pode-se facilmente modificar a descrição para tornar o bucket-sort estável, ao mesmo tempo em que se preserva seu tempo de execução  $O(n + N)$ . Na verdade, pode-se obter um algoritmo bucket-sort estável removendo sempre o *primeiro* elemento da seqüência  $S$  e das seqüências  $B[i]$  durante a execução do algoritmo.

#### 11.4.2 Radix-sort

Uma das razões pela qual a estabilidade de um algoritmo é importante é que ela permite que a abordagem do bucket-sort seja aplicada a contextos mais gerais do que a ordenação de inteiros. Supondo-se, por exemplo, que se quer ordenar itens que são pares  $(k, l)$ , onde  $k$  e  $l$  são inteiros no intervalo  $[0, N - 1]$  para qualquer inteiro  $N \geq 2$ . Em um contexto como esse, é natural definir a ordenação desses itens usando a convenção *lexicográfica* (do dicionário), onde  $(k_1, l_1) < (k_2, l_2)$  se  $k_1 < k_2$  ou se  $k_1 = k_2$  e  $l_1 < l_2$  (Seção 8.1.2). Esta é uma versão tal qual uma função lexicográfica de comparação, normalmente aplicada em strings de caracteres de mesmo tamanho (e facilmente generalizável para tuplas de  $d$  números com  $d > 2$ ).

O algoritmo *radix-sort* ordena uma seqüência de pares tais como  $S$ , aplicando um bucket-sort estável sobre a seqüência duas vezes; primeiro usando um componente do par como chave de ordenação e, em seguida, empregando o segundo componente. Mas qual é a ordem correta? Deve-se ordenar primeiro pelo  $k$  (o primeiro componente) e em seguida  $l$  (o segundo componente) ou pode-se fazê-lo de outra forma?

Antes de responder essa questão, vide o seguinte exemplo.

**Exemplo 11.5** Considere a seguinte seqüência  $S$ :

$$S = ((3,3),(1,5),(2,5),(1,2),(2,3),(1,7),(3,2),(2,2)).$$

Ordenando  $S$  de forma estável no primeiro componente, então se obtém a seqüência

$$S_1 = ((1,5),(1,2),(1,7),(2,5),(2,3),(2,2),(3,3),(3,2)).$$

Ordenando, então, a seqüência  $S_1$ , usando o segundo componente, segue que

$$S_{1,2} = ((1,2), (2,2), (3,2), (2,3), (3,3), (1,5), (2,5), (1,7)).$$

que não é exatamente uma seqüência ordenada. Por outro lado, se  $S$  for ordenado de forma estável usando o segundo componente, então se obtém a seguinte seqüência

$$S_2 = ((1,2), (3,2), (2,2), (3,3), (2,3), (1,5), (2,5), (1,7)).$$

Ordenando, agora, de forma estável a seqüência  $S_2$ , usando o primeiro componente, obtém-se a seqüência

$$S_{2,1} = ((1,2), (1,5), (1,7), (2,2), (2,3), (2,5), (3,2), (3,3)).$$

que é, de fato, uma seqüência  $S$  lexicograficamente ordenada.

Então, a partir desse exemplo, se é levado a acreditar que é necessário primeiro ordenar usando o segundo componente e, então, ordenar novamente usando o primeiro. Esta intuição é correta. Ordenando de forma estável primeiro pelo segundo componente, e, então, novamente pelo primeiro, garante-se que se dois elementos forem iguais na segunda ordenação (pelo primeiro elemento), então sua ordem relativa na seqüência inicial (que é ordenada pelo segundo componente) será preservada. Sendo assim, a seqüência resultante tem a garantia de ser ordenada lexicograficamente todas as vezes. Deixa-se para um exercício simples (R-10.14) a determinação sobre como esta abordagem pode ser estendida para triplas e outras  $d$ -tuplas de números. Esta seção pode ser resumida como segue:

**Proposição 11.6** *Seja  $S$  uma seqüência de  $n$  itens chave-elemento, cada um dos quais tendo uma chave  $(k_1, k_2, \dots, k_d)$ , onde  $k_i$  é um inteiro no intervalo  $[0, N - 1]$  para qualquer inteiro  $N \geq 2$ . Pode-se ordenar  $S$  lexicograficamente em tempo  $O(d(n + N))$ , usando radix-sort.*

Apesar de ser tão importante, a ordenação não é o único problema interessante que lida com relações de ordem total em um conjunto de elementos. Existem algumas aplicações, por exemplo, que não requerem a listagem ordenada de um conjunto inteiro, mas necessitam de uma quantia de informação ordenada sobre o conjunto. Antes de estudar esse problema (chamado de “seleção”), é preciso retroceder e comparar brevemente todos os algoritmos de ordenação estudados até aqui.

## 11.5 Comparando algoritmos de ordenação

Neste ponto, pode ser útil fazer uma interrupção e analisar todos os algoritmos estudados neste livro para ordenar um arranjo, lista ou uma seqüência de  $n$  elementos.

### Considerando tempo de execução e outros fatores

Vários métodos foram estudados, como a ordenação, o insertion sort e o selection sort, que têm comportamento temporal  $O(n^2)$  na média e no pior caso. Também foram examinados vários métodos com comportamento temporal  $O(n \log n)$  incluindo heap sort, merge-sort e quick-sort. Finalmente, uma classe especial de algoritmos de ordenação foi abordada, a saber, os métodos bucket-sort e radix-sort, que executam em tempo linear para certos tipos de chaves. Certamente, os algoritmos de ordenação da bolha e o selection sort são escolhas pobres em qualquer aplicação, uma vez que eles executam em tempo  $O(n^2)$ , mesmo no melhor caso. Mas entre os algoritmos de ordenação restantes, qual é o melhor?

Como muitas coisas na vida, não existe claramente o “melhor” algoritmo de ordenação entre os candidatos restantes. O algoritmo de ordenação que melhor se aplica para um uso específico depende das várias propriedades do mesmo. Pode-se, entretanto, oferecer algumas diretrizes e observações, baseadas nas propriedades conhecidas de “bons” algoritmos de ordenação.

### Insert-sort

Se bem implementado, o tempo de execução da **insertion-sort** é  $O(n + m)$ , onde  $m$  é o número de **inversões** (isto é, o número de pares de elementos fora de ordem). Sendo assim, o insertion

sort é um algoritmo excelente para ordenar pequenas seqüências (por exemplo, com menos de 50 elementos), porque é simples de programar e seqüências pequenas necessariamente contêm poucas inversões. Além disso, o insertion-sort, é bastante eficiente para ordenar seqüências “quase” ordenadas. Por “quase”, entende-se que o número de inversões é pequeno. Mas a performance  $O(n^2)$  em relação ao tempo do insertion-sort torna-o uma escolha pobre fora dessas situações especiais.

### Merge-sort

**Merge-sort**, por outro lado, executa em tempo  $O(n \log n)$  no pior caso, o que é ótimo para métodos de ordenação baseados em comparações. Além disso, estudos experimentais têm mostrado que, uma vez que é difícil fazer o merge-sort executar in-place, a carga de trabalho necessária para implementá-lo torna-o menos atrativo que as implementações in-place do heap sort e quick-sort para seqüências que cabem inteiras na memória principal do computador. Desta forma, o merge-sort é um algoritmo excelente para situações em que a entrada não cabe toda na memória principal e tem de ser armazenada em blocos em um dispositivo de memória externa, como um disco. Neste contexto, a forma em que o merge-sort executa o processamento dos dados em grandes cadeias faz melhor uso de todos os dados trazidos do disco para a memória principal. Assim, para ordenação em memória externa, o algoritmo merge-sort tende a minimizar o número total de acessos a disco para fazer a leitura ou a escrita necessárias, o que torna o algoritmo merge-sort superior neste contexto.

### Quick-sort

Análises experimentais têm mostrado que se a seqüência de entrada couber inteiramente na memória principal, então as versões in-place do quick-sort e heap-sort executam mais rápido que o merge-sort. De fato, o quick-sort tende, na média, a ser melhor que o heap-sort nesses testes.

Então, o **quick-sort** é uma escolha excelente como algoritmo de ordenação de finalidades genéricas no uso em memória. Na verdade, ele está incluído no utilitário `qsort`, fornecido nas bibliotecas da linguagem C. Entretanto, sua performance temporal  $O(n^2)$  para o pior caso faz do quick-sort uma escolha pobre para aplicações de tempo real em que se tem de apresentar garantias sobre o tempo necessário para completar uma operação de ordenação.

### Heap-sort

Em cenários de tempo real, nos quais se dispõe de um tempo fixo para executar uma operação de ordenação e os dados de entrada cabem na memória, o algoritmo **heap-sort** provavelmente é a melhor escolha. Ele executa em tempo  $O(n \log n)$  no pior caso e pode ser facilmente adaptado para executar in-place.

### Bucket-sort e radix-sort

Finalmente, se esta aplicação envolve ordenação por chaves inteiras ou  $d$ -tuplas de chaves inteiras, então **bucket-sort** ou **radix-sort** são ótimas escolhas, pois executam em tempo  $O(d(n + N))$ , onde  $[0, N - 1]$  é o intervalo de chaves inteiras (e  $d = 1$  para o bucket-sort). Sendo assim, se  $d(n + N)$  está significativamente “abaixo” de  $n \log n$ , então este método de ordenação pode executar mais rápido que o quick-sort ou o heap sort.

Desta forma, o estudo sobre todos estes métodos diferentes proporciona a “caixa de ferramentas” para engenharia de algoritmos com uma coleção versátil de métodos de ordenação.

Hidden page

O método genérico de junção examina e compara iterativamente os elementos correntes  $a$  e  $b$  das seqüências  $A$  e  $B$ , respectivamente, e determina quando  $a < b$ ,  $a = b$  ou  $a > b$ . Então, baseado no resultado desta comparação, determina se pode copiar um ou nenhum dos elementos  $a$  e  $b$  para o fim da seqüência de saída  $C$ . Essa determinação é feita com base na operação específica executada, seja união, interseção ou diferença. Por exemplo, na operação de união procede-se como segue:

- Se  $a < b$ , copia-se  $a$  para o final de  $C$  e avança-se para o próximo elemento de  $A$ .
- Se  $a = b$ , copia-se  $a$  para o final de  $C$  e avança-se para o próximo elemento de  $A$  e de  $B$ .
- Se  $a > b$ , copia-se  $b$  para o final de  $C$  e avança-se para o próximo elemento de  $B$ .

### Desempenho da junção genérica

Analisa-se o tempo de execução do algoritmo genérico de junção. A cada iteração, compara-se dois elementos das seqüências de entrada  $A$  e  $B$ , possivelmente copiando um elemento para a seqüência de saída, e avançando o elemento atual de  $A$ ,  $B$  ou ambos. Admitindo que as comparações e cópias levem tempo  $O(1)$ , o tempo total de execução é  $O(n_A + n_B)$ , onde  $n_A$  é o tamanho de  $A$ , e  $n_B$  é o tamanho de  $B$ ; ou seja, a junção leva um tempo proporcional ao número de elementos envolvidos. Desta forma tem-se:

**Proposição 11.8:** *O TAD conjunto pode ser implementado usando um esquema de seqüência ordenada e junção genérica que suporta as operações union, intersect e subtract em tempo  $O(1)$ , onde  $n$  indica a soma dos tamanhos dos conjuntos envolvidos.*

### Junção genérica usando o padrão do método modelo

O algoritmo genérico de junção é baseado no **padrão do método modelo** (ver Seção 7.3.7). O padrão do método modelo é um padrão de projeto de engenharia de software que descreve um mecanismo genérico de computação que pode ser especializado pela redefinição de certos passos. Neste caso, descreve-se um método que faz a junção de duas seqüências em uma, e que pode ser especializado pelo comportamento de três métodos abstratos.

O Trecho de código 11.9 apresenta a classe Merge provendo uma implementação em Java para o algoritmo de junção genérica.

```
/** Junção genérica para seqüências ordenadas. */
public abstract class Merge<E> {
    private E a, b;                      // elementos atuais em A e B
    private Iterator<E> iterA, iterB;    // iterators para A e B
    /** Método Template */
    public void merge(PositionList<E> A, PositionList<E> B,
                      Comparator<E> comp, PositionList<E> C) {
        iterA = A.iterator();
        iterB = B.iterator();
        boolean aExists = advanceA();    // Teste lógico se existe um atual a
        boolean bExists = advanceB();    // Teste lógico se existe um atual b
        while (aExists && bExists) {      // Laço principal para junção de a e b
            int x = comp.compare(a, b);
            if (x < 0) { alsLess(a, C); aExists = advanceA(); }
            else if (x == 0) {
                bothAreEqual(a, b, C); aExists = advanceA(); bExists = advanceB(); }
            else { blsLess(b, C); bExists = advanceB(); }
        }
        while (aExists) { alsLess(a, C); aExists = advanceA(); }
        while (bExists) { blsLess(b, C); bExists = advanceB(); }
    }
}
```

Hidden page

Hidden page

### Desempenho da implementação de seqüência

A implementação de seqüência anteriormente apresentada é simples, porém ela também é eficiente, como pode ser visto no teorema que segue.

**Proposição 11.9:** *A execução de uma série de  $n$  operações makeSet, union e find, usando a implementação anteriormente apresentada, iniciando a partir de uma partição inicialmente vazia leva o tempo  $O(n \log n)$ .*

**Justificativa:** Usa-se o método contabilizar e assume-se que um ciberdólar pode pagar pelo tempo para executar a operação find, uma operação makeSet ou o movimento de uma posição de uma seqüência para outra na operação union. No caso da operação makeSet ou find, define-se em 1 ciberdólar cada uma. No caso da operação union, especifica-se em 1 ciberdólar para cada posição que se move de um conjunto para outro. Não se especificou para a operação union. Claramente, o total de gastos para as operações find e makeSet somados são  $O(n)$ .

Considere-se, então, o número de gastos criados para posições em nome da operação union. A importante observação é que cada vez que se move uma posição de um conjunto para outro, o tamanho do novo conjunto será pelo menos o dobro. Desta forma, cada posição é movida de um conjunto para outro no máximo  $\log n$  vezes; então, cada posição pode ser definida no máximo com o tempo  $O(\log n)$ . Desde que seja assumido que a partição está inicialmente vazia, existem  $O(n)$  diferentes elementos referenciados em uma dada série de operações, que implicam que o tempo total para todas as operações de união será  $O(n \log n)$ . ■

O tempo de execução amortizado de uma operação em uma série de operações makeSet, union e set, é o tempo total levado para as séries divididas pelo número de operações. Conclui-se, a partir da proposição apresentada anteriormente, que, para uma partição implementada usando seqüências, o tempo de execução amortizado para cada operação será  $O(\log n)$ . Desta forma, pode-se resumir o desempenho da nossa simples implementação da partição baseada em seqüência como segue.

**Proposição 11.10:** *Usando uma implementação baseada em seqüência de uma partição, em uma série de operações makeSet, union e find, iniciando a partir de uma partição inicialmente vazia, o tempo de execução amortizado para cada operação será  $O(\log n)$ .*

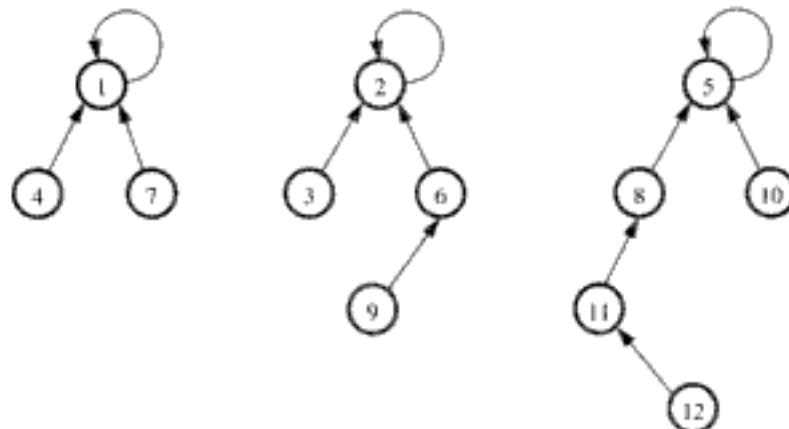
Nesta implementação baseada em seqüência de uma partição, cada operação find leva, no pior caso, o tempo de  $O(1)$ . Isto é, o tempo de execução das operações union, que é o gargalo computacional.

Na próxima seção, descreve-se uma implementação baseada em árvore de uma partição que não garante o tempo constante das operações find, mas tem o tempo amortizado muito melhor que  $O(\log n)$  conforme a operação union.

### 11.6.3 Uma implementação de partição baseada em árvore \*

Uma estrutura de dados alternativa usa uma coleção de árvores para armazenar  $n$  elementos no conjunto, onde cada árvore é associada com um diferente conjunto. (Ver Figura 11.17.) Em particular, implementa-se cada árvore com uma estrutura de dados encadeada, em que os nodos são eles mesmos as posições do conjunto. Vê-se, ainda, cada posição  $p$  como sendo um nodo tendo uma variável, elemento, referindo ao seu elemento  $x$  e uma variável, conjunto, referindo a um conjunto contendo  $x$ , como antes. Porém, agora também se vê cada posição  $p$  como sendo do tipo de dados “conjunto”. Assim, o conjunto referencia que cada posição  $p$  pode apontar para uma posição, que poderia ser o próprio  $p$ . Além disso, implementa-se esta abordagem em que todas as posições e seus respectivos conjuntos definem uma coleção de árvores.

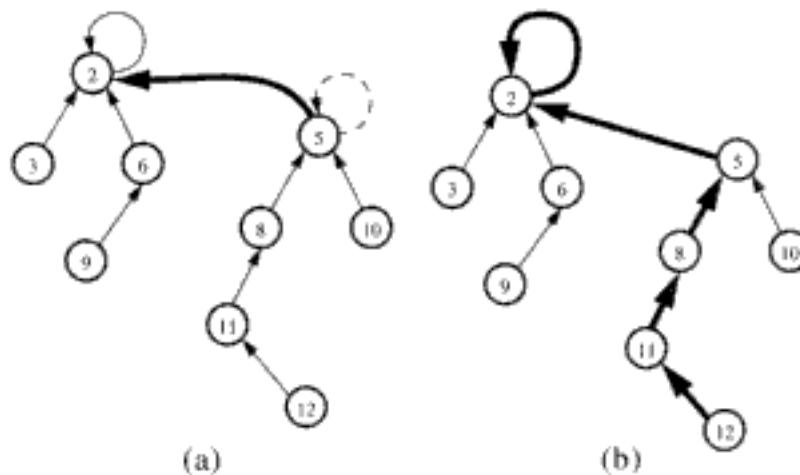
Cada árvore é associada com um conjunto. Para qualquer posição  $p$ , se o conjunto de  $p$  referencia pontos atrás de  $p$ , então  $p$  é a *raiz* desta árvore, e o nome do conjunto contendo  $p$  é “ $p$ ” (isto é, neste caso se estaria usando nomes das posições como nome dos conjuntos). Fora isso, o conjunto referenciado por  $p$  aponta para o pai de  $p$  na sua árvore. Em ambos os casos, o conjunto contendo  $p$  é o associado com a raiz da árvore contendo  $p$ .



**Figura 11.17** Implementação baseada em árvore de uma partição consistindo de três conjuntos separados:  $A = \{1,4,7\}$ ,  $B = \{2,3,6,9\}$  e  $C = \{5,8,10,11,12\}$ .

Com esta estrutura de dados de partição, a operação  $\text{union}(A,B)$  é chamada com os argumentos de posições  $p$  e  $q$ , que respectivamente representam os conjuntos  $A$  e  $B$  (isto é,  $A = p$  e  $B = q$ ). Executa-se esta operação criando uma das árvores como subárvore das outras (Figura 11.18b), que pode ser feito no tempo  $O(1)$  pela definição da referência da raiz de uma árvore para apontar para a raiz da outra árvore. A operação  $\text{find}$  para uma posição  $p$  é executada caminhando para a raiz da árvore que contém a posição  $p$  (Figura 11.18a), o que leva o tempo  $O(n)$  no pior caso.

Esta representação de uma árvore é uma estrutura de dados especializada usada para implementar uma partição, e isso não significa ser uma realização do tipo abstrato de dados árvore (Seção 7.1). Realmente, a representação tem somente conexões “para cima”, e não provê uma forma para acessar os filhos de um determinado nodo.



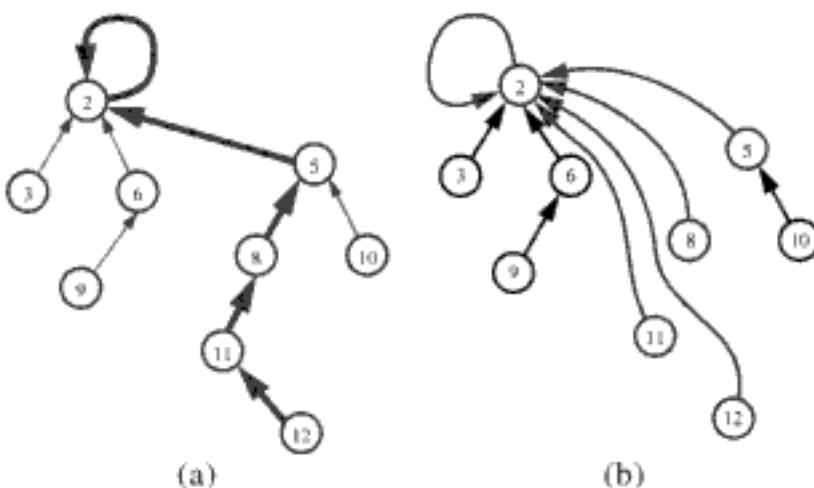
**Figura 11.18** Implementação de uma partição baseada em árvore: (a) operação  $\text{union}(A,B)$ ; (b) operação  $\text{find}(p)$ , onde  $p$  denota a posição do objeto para o elemento 12.

Inicialmente, esta implementação pode não parecer melhor que a estrutura de dados baseado em seqüência, porém, adiciona-se a seguinte heurística simples para fazer com que ele execute mais rápido:

**União-pelo-tamanho:** O tamanho da subárvore enraizada em  $p$  é armazenado com cada nodo posição  $p$ . Na operação union, cria-se a árvore do menor conjunto para se tornar uma subárvore de outra árvore, e atualizar o campo tamanho da raiz da árvore resultante.

**Compressão de caminhos:** Na operação find, para cada nodo  $v$  que a operação find visita, zera o apontador pai de  $v$  para a raiz. (Ver Figura 11.19.)

Estas heurísticas incrementam o tempo de execução de uma operação em um fator constante; porém, como será discutido abaixo, elas melhoraram significativamente o tempo de execução amortizado.



**Figura 11.19** Heurística compressão de caminhos: (a) caminho cruzado pela operação find no elemento 12; (b) árvore reestruturada.

### O log-star e funções ackermann inversas

Uma propriedade surpreendente da estrutura de dados partição baseada em árvore, quando implementada usando as heurísticas união-pelo-tamanho e compressão de caminhos, é que a execução de uma série de  $n$  operações union e find leva o tempo  $O(n \log^* n)$ , onde  $\log^* n$  é a função **log-star**, o qual é o inverso da função **tower-of-two**. Intuitivamente,  $\log^* n$  é o número de vezes que alguém pode, iterativamente, calcular o logaritmo (base 2) de um número antes de obter um número menor que 2. A Tabela 11.1 mostra alguns valores simples.

<b>mínimo <math>n</math></b>	2	$2^2 = 4$	$2^{2^2} = 16$	$2^{2^{2^2}} = 65.536$	$2^{2^{2^{2^2}}} = 2^{65.536}$
$\log^* n$	1	2	3	4	5

**Tabela 11.1** Alguns valores de  $\log^* n$  e valores críticos para os seus inversos.

Como é demonstrado na Tabela 11.1, para todos os propósitos,  $\log^* n \leq 5$ . Ela é uma maravilhosa função de crescimento lento (porém é uma que cresce apesar de tudo).

De fato, o tempo de execução de uma série de  $n$  operações de partição implementadas, como anteriormente mencionado, pode realmente ser apresentado para ser  $O(n\alpha(n))$ , onde  $n\alpha$  é o inverso da função de Ackermann,  $A$ , que cresce assintoticamente mais lenta que  $\log^* n$ . Entretanto, não será provado este fato; a função Ackermann será definida aqui para se apreciar justamente como ela cresce de forma rápida, e, então, como lentamente seus inversos crescem. Primeiro se definirá uma função Ackermann indexada,  $A_i$ , como segue:

$$\begin{aligned}\mathcal{A}_0(n) &= 2n && \text{para } n \geq 0 \\ \mathcal{A}_1(1) &= \mathcal{A}_{i-1}(2) && \text{para } i \geq 1 \\ \mathcal{A}_i(n) &= \mathcal{A}_{i-1}(\mathcal{A}_i(n-1)) && \text{para } i \geq 1 \text{ e } n \geq 2\end{aligned}$$

Em outras palavras, as funções Ackermann definem uma progressão de funções.

- $\mathcal{A}_0(n) = 2n$  é a função múltiplo de dois,
- $\mathcal{A}_1(n) = 2^n$  é a função potência de dois,
- $\mathcal{A}_i(n) = 2^{\dots^2}$  (com vários dois) é a função tower-of-twos,
- e assim por diante.

Então, define-se a função Ackermann como  $\mathcal{A}(n) = \mathcal{A}_n(n)$ , que é uma inacreditável função de crescimento rápido. Da mesma forma, a **função inversa Ackermann**,

$$\alpha(n) = \min\{m: \mathcal{A}(m) \geq n\},$$

é uma inacreditável função de crescimento lento. Ela cresce muito mais lentamente que a função  $\log^* n$  (a qual é a inversa da  $\mathcal{A}_2(n)$ ), por exemplo, e foi notado que  $\log^* n$  é uma função de crescimento muito lento.

## 11.7 Seleção

Existe uma grande quantidade de aplicações nas quais se está interessado em identificar um único elemento em função de sua localização relativa à ordenação de um conjunto inteiro. Exemplos incluem a identificação do maior e do menor elemento, mas também pode-se estar interessado em, por exemplo, identificar o termo **mediano**, ou seja, o elemento tal que metade dos elementos seja menor que ele e a outra metade seja maior. Normalmente, consultas que questionam a respeito da localização de um elemento são chamadas de **estatísticas de ordem**.

### Definindo o problema da seleção

Nesta seção, será discutido o problema geral de estatística de ordem para selecionar o  $k$ -ésimo menor elemento de uma coleção não-ordenada de  $n$  elementos comparáveis. Isso é conhecido como o problema da **seleção**. É claro, pode-se resolver este problema ordenando a coleção, e então acessando a seqüência ordenada na localização  $k - 1$ . Usando o melhor dos algoritmos de ordenação baseado em comparação, esta abordagem irá levar tempo  $O(n \log n)$ , o que é obviamente um absurdo nos casos em que  $k = 1$  ou  $k = n$  (ou mesmo  $k = 2, k = 3, k = n - 1$  ou  $k = n - 5$ ) porque pode-se resolver o problema da seleção para estes valores de  $k$ , com facilidade, em tempo  $O(n)$ . Logo, a questão que surge é como se pode obter um tempo de execução  $O(n)$  para todos os valores de  $k$  (incluindo o caso de encontrar o mediano, onde  $k = \lfloor n / 2 \rfloor$ ).

#### 11.7.1 Poda e busca

Pode ser uma pequena surpresa, mas, na verdade, pode-se solucionar o problema da seleção em tempo  $O(n)$  para qualquer valor de  $k$ . Além disso, a técnica que se usa para obter esse resultado envolve um interessante padrão de projeto de algoritmo. Este padrão de projeto é conhecido como **poda e busca** ou **diminuição e conquista**. Aplicando este padrão de projeto, resolve-se um problema que é definido a partir de uma coleção de  $n$  objetos, podando uma fração destes e recursivamente resolvendo o problema menor. Quando se tiver finalmente reduzido o problema para

um problema definido sobre uma coleção constante de objetos, então é resolvido usando algum método de força bruta. Na medida em que se retorna das chamadas recursivas, a construção se completa. Em alguns casos, pode-se evitar o uso da recursão, caso em que simplesmente itera-se o passo de redução da poda e busca até que se possa aplicar um método de força bruta e parar. Incidentalmente, o método de pesquisa binária descrita na Seção 9.3.3 é um exemplo do padrão de projeto poda e busca.

### 11.7.2 Quick-select randômico

Aplicando-se o padrão poda e busca ao problema de seleção, pode-se projetar um método simples e prático chamado de **quick-select randomizado**, para encontrar o  $k$ -ésimo menor elemento de uma seqüência não-ordenada de  $n$  elementos sobre os quais uma relação de ordem total é definida. O quick-select randomizado executa um tempo **esperado**  $O(n)$ , levando em conta todas as possíveis escolhas randômicas feitas pelo algoritmo, e esta expectativa não depende de qualquer premissa sobre a distribuição de entrada. Observa-se que um quick-select randomizado executa em tempo  $O(n^2)$ , no **pior caso**; a justificativa deste fato aparece em um exercício (R-11.25). Também se inclui um exercício (C-11.31) que propõe modificar o quick-select para obter um algoritmo de seleção **determinístico** que execute em tempo  $O(n)$  no **pior caso**. Entretanto, a existência de um algoritmo determinístico é principalmente de interesse teórico, uma vez que o fator constante escondido pela notação  $O$ , neste caso, é relativamente grande.

Suponha-se uma dada seqüência não-ordenada  $S$  de  $n$  elementos comparáveis, juntamente com um inteiro  $k \in [1, n]$ . No nível superior, o algoritmo quick-select para encontrar o  $k$ -ésimo elemento de  $S$  é similar em estrutura ao algoritmo quick-select randomizado descrito na Seção 10.3.2. Pega-se um elemento  $x$  de  $S$  randomicamente, para usar como “pivô” e subdividir  $S$  em três subseqüências,  $L$ ,  $E$  e  $G$ , armazenando os elementos de  $S$  menores que  $x$ , iguais a  $x$  e maiores que  $x$ , respectivamente. Este é o passo de poda. Então, baseado no valor de  $k$ , determina-se em quais destes conjuntos será aplicada a recursão. O quick-select randomizado é descrito no Trecho de código 11.11.

**Algoritmo** quickSelect( $S, k$ ):

**Entrada:** seqüência  $S$  de  $n$  elementos comparáveis e um inteiro  $k \in [1, n]$ .

**Saída:** o  $k$ -ésimo menor elemento de  $S$ .

**se**  $n = 1$  **então**

**retorna** o (primeiro) elemento de  $S$ .

seleciona um elemento aleatório  $x$  de  $S$  remove todos os elementos de  $S$  e coloca-os em três seqüências:

- $L$ , armazenando os elementos de  $S$  menores que  $x$ ;
- $E$ , armazenando os elementos de  $S$  iguais a  $x$ ;
- $G$ , armazenando os elementos de  $S$  maiores que  $x$ .

**se**  $k \leq |L|$  **então**

quickSelect( $L, k$ )

**senão se**  $k \leq |L| + |E|$  **então**

**retorna**  $x$  {cada elemento em  $E$  é igual a  $x$ }

**senão**

quickSelect( $G, k - |L| - |E|$ ) {observe o novo parâmetro de seleção}

**Trecho de código 11.11** Algoritmo quick-select randomizado.

Hidden page

Hidden page

caso, o algoritmo ordena corretamente a seqüência de entrada, mas o resultado do passo de divisão pode diferir da descrição de alto nível dada na Seção 11.2 e pode resultar em ineficiência. Em particular, o que acontece no passo de partição quando existem elementos iguais ao pivô? A seqüência  $E$  (armazenando os elementos iguais ao pivô) é realmente avaliada? O algoritmo usa as subsequências  $L$  e  $R$  ou outras subsequências? Qual o tempo de execução do algoritmo se todos os elementos da entrada forem iguais?

- R-11.15 Sobre  $n!$  entradas possíveis para um dado algoritmo de ordenação baseado em comparação, qual é o número máximo absoluto de entradas que poderiam ser ordenadas com somente  $n$  comparações?
- R-11.16 Jonathan tem um algoritmo de ordenação baseado em comparação que ordena os primeiros  $k$  elementos de uma seqüência de tamanho  $n$  no tempo  $O(n)$ . Apresente uma caracterização  $O$  do maior valor que  $k$  pode conter?
- R-11.17 O algoritmo merge-sort da Seção 11.1 é estável? Justifique.
- R-11.18 Um algoritmo que ordena itens chave-valor pela chave é chamado *straggling* se, a qualquer tempo, dois itens  $e_i$  e  $e_j$  tenham chaves iguais, mas  $e_i$  aparece antes de  $e_j$  na entrada, então o algoritmo coloca  $e_i$  após  $e_j$  na saída. Descreva uma alteração no algoritmo de merge-sort apresentado na Seção 11.1 para torná-lo straggling.
- R-11.19 Descreva um método de radix-sort para ordenar lexicograficamente uma seqüência  $S$  de triplas  $(k, l, m)$ , onde  $k, l$  e  $m$  sejam inteiros no intervalo  $[0, N - 1]$  para algum  $N \geq 2$ . Como o método pode ser estendido para seqüências de  $d$ -tuplas  $(k_1, k_2, \dots, k_d)$ , onde cada  $k_i$  é um inteiro no intervalo  $[0, N - 1]$ ?
- R-11.20 O algoritmo de bucket-sort é in-place? Justifique.
- R-11.21 Apresente um exemplo de seqüência de entrada que requer merge-sort e heap-sort para ter o tempo de ordenação  $O(n \log n)$ , porém executa uma inserção-ordenação no tempo  $O(n)$ . O que acontece se você reverte esta seqüência?
- R-11.22 Descreva, em pseudocódigo, como executar a compressão de caminho em um caminho de tamanho  $h$  no tempo  $O(h)$  em uma estrutura de partição union/find baseada em árvore.
- R-11.23 George afirma que ele tem uma forma rápida de fazer a compressão de caminho em uma estrutura de partição, iniciando no nodo  $v$ . Ele coloca  $v$  em uma seqüência  $L$  e inicia seguindo os apontadores para os pais. Cada vez que ele encontra um novo nodo,  $u$ , ele adiciona  $u$  em  $L$  e atualiza o apontador pai de cada nodo de  $L$  para apontar para o pai de  $u$ . Mostre que o algoritmo de George executa no tempo  $\Omega(h^2)$  em um caminho de tamanho  $h$ .
- R-11.24 Descreva uma versão in-place do algoritmo quick-select com pseudo código.
- R-11.25 Mostre que o tempo de execução de pior caso do algoritmo de quick-select em uma seqüência de  $n$  elementos é  $\Omega(n^2)$ .

## Criatividade

- C-11.1 Linda afirma ter um algoritmo que pega uma seqüência  $S$  de entrada e produz uma seqüência  $T$  de saída que é uma ordenação de  $n$  elementos de  $S$ .
  - a. Apresente um algoritmo, `isSorted`, para testar no tempo  $O(n)$  se  $T$  está ordenado.

- b. Explique porque o algoritmo `isSorted` não é suficiente para provar que uma saída particular  $T$  do algoritmo de Linda é uma ordenação de  $S$ .
- c. Descreva qual informação adicional o algoritmo de Linda poderia fornecer para que a correção do seu algoritmo pudesse ser estabilizada em qualquer  $S$  e  $T$  no tempo  $O(n)$ .
- C-11.2 Dados dois conjuntos  $A$  e  $B$  representados como seqüências ordenadas, descreva um algoritmo eficiente para realizar  $A \oplus B$ , que é o conjunto de elementos que estão em  $A$  ou em  $B$ , mas não estão em ambos.
- C-11.3 Suponha que se representem conjuntos com árvores de pesquisa balanceadas. Descreva e analise algoritmos para cada método do TAD conjunto, assumindo que um dos dois conjuntos é muito menor que o outro.
- C-11.4 Descreva e analise um método eficiente para remover todas as duplicadas de uma coleção  $A$  de  $n$  elementos.
- C-11.5 Considere conjuntos cujos elementos inteiros no intervalo  $[0, N - 1]$ . Uma forma comum de representar um conjunto  $A$  desse tipo é através de um vetor booleano  $B$ , onde se diz que  $x$  está em  $A$  se e somente se  $B[x] = \text{true}$ . Como cada posição de  $B$  pode ser representada por um bit,  $B$  é às vezes chamado de **vetor de bits**. Descreva algoritmos eficientes para realizar os métodos para o TAD conjunto assumindo essa representação.
- C-11.6 Considere uma versão do quick-sort determinístico na qual pega-se como nosso pivô o mediano dos últimos  $d$  elementos na seqüência de entrada de  $n$  elementos, para um fixo, constante número ímpar  $d \geq 3$ . Argumente informalmente por que este deverá ser uma boa escolha para o pivô. Qual é o assintótico tempo de execução do pior caso do quick-sort neste caso, em termos de  $n$  e  $d$ ?
- C-11.7 Outra forma para analisar o quick-sort randomizado é usar uma **equação de recorrência**. Neste caso, pega-se  $T(n)$  que denota o tempo de execução esperado do quick-sort randomizado e observa-se que, por causa das partições do pior caso para as boas e ruins separações, pode-se escrever:
- $$T(n) \leq \frac{1}{2}(T(3n/4) + \frac{1}{2}(T(n-1)) + bn$$
- onde  $bn$  é o tempo necessário para separar uma seqüência em relação a um dado pivô e contatenar as subsequências resultantes após o retorno das chamadas recursivas. Mostre, por indução, que  $T(n) \in O(n \log n)$ .
- C-11.8 Modifique o algoritmo `inPlaceQuickSort` (Trecho de código 11.6) para tratar eficientemente o caso geral em que a seqüência de entrada  $S$  pode ter chaves repetidas.
- C-11.9 Descreva uma versão não-recursiva e in-place do algoritmo de quick-sort. O algoritmo deve ser baseado na mesma estratégia de divisão e conquista, mas utiliza uma pilha explícita para processar subproblemas. Seu algoritmo deverá garantir que a profundidade da pilha seja no máximo  $O(\log n)$ .
- C-11.10 Mostre que o quick-sort randomizado executa no tempo  $O(n \log n)$  com probabilidade no mínimo  $1 - 1/n$ , isto é, com **alta probabilidade**, respondendo o seguinte:
- Para cada elemento de entrada  $x$ , defina  $C_{i,j}(x)$  para ser uma variável randômica 0/1 que é 1 se e somente se o elemento  $x$  estiver em  $j + 1$

- subproblemas que pertencem ao size group  $i$ . Discuta por que não precisamos definir  $C_{i,j}$  para  $j > n$ .
- Considere  $X_{i,j}$  uma variável randômica que é 1 com probabilidade  $1/2^i$ , independente de qualquer outro evento, e  $L = \lceil \log_{4/3} n \rceil$ . Discuta por que  $\sum_{i=0}^{L-1} \sum_{j=0}^n C_{i,j}(x) \leq \sum_{i=0}^{L-1} \sum_{j=0}^n X_{i,j}$ .
  - Mostre que o valor esperado de  $\sum_{i=0}^{L-1} \sum_{j=0}^n X_{i,j}$  é  $(2 - 1/2^n)L$ .
  - Mostre que a probabilidade de que  $\sum_{i=0}^{L-1} \sum_{j=0}^n X_{i,j} > 4L$  é no máximo  $1/n^2$ , usando o **limite de Chernoff**, que expressa que se  $X$  é a soma de um número finito de variáveis randômicas 0/1 independentes com valor esperado  $\mu > 0$ , então  $\Pr(X > 2\mu) < (4/e)^{-\mu}$ , onde  $e = 2.71828128\dots$ .
  - Discuta por que a afirmação anterior prova que o quick-sort randomizado executa no tempo  $O(n \log n)$  com probabilidade no mínimo  $1 - 1/n$ .
- C-11.11 Dado um arranjo  $A$  de  $n$  elementos com chaves iguais a 0 ou 1, descreva um método in-place para ordenar  $A$  sendo que todos os zeros fiquem antes de todos os uns.
- C-11.12 Suponha que temos uma seqüência  $S$  de  $n$  elementos, de forma que cada elemento em  $S$  representa um voto diferente para líder estudantil, dado como um inteiro representando o número de matrícula do candidato escolhido. Planeje um algoritmo de tempo  $O(n \log n)$  para determinar quem vence a votação representada por  $S$ , supondo que o candidato com o maior número de votos será o escolhido (mesmo se existir  $O(n)$  candidatos).
- C-11.13 Considere o problema de votação do Exercício C-11.12, mas agora suponha que se conhece o número  $k < n$  de candidatos. Descreva um algoritmo de tempo  $O(n \log k)$  para determinar quem vence a eleição.
- C-11.14 Considere o problema de votação do Exercício C-11.12, mas agora suponha que um candidato vence somente se conseguir a maioria dos votos computados. Projete e analise um algoritmo rápido para determinar o vencedor se existir um.
- C-11.15 Mostre que qualquer algoritmo de ordenação baseado em comparações pode ser transformado em um método estável sem afetar o tempo de execução assintótico do algoritmo.
- C-11.16 Suponha que temos duas seqüências  $A$  e  $B$  de  $n$  elementos cada, possivelmente contendo repetições, nas quais existe uma relação de ordem total. Descreva um algoritmo eficiente para determinar se  $A$  e  $B$  contêm o mesmo conjunto de elementos. Qual o tempo de execução deste método?
- C-11.17 Dado um arranjo  $A$  de  $n$  inteiros de um intervalo  $[0, n^2 - 1]$ , descreva um método simples para a ordenação de  $A$  no tempo  $O(n)$ .
- C-11.18 Sejam  $S_1, S_2, \dots, S_k$  seqüências diferentes  $k$ , cujos elementos têm chaves inteiras no intervalo  $[0, N - 1]$  para algum parâmetro  $N \geq 2$ . Descreva um algoritmo com tempo  $O(n + N)$  para ordenar todas as seqüências (sem fazer sua união), onde  $n$  denota o tamanho total de todas as seqüências.
- C-11.19 Dada uma seqüência  $S$  de  $n$  elementos onde existe uma relação de ordem total. Descreva um método eficiente para determinar se existem dois elementos iguais em  $S$ . Qual o tempo de execução de seu algoritmo?
- C-11.20 Seja  $S$  uma seqüência de  $n$  elementos em que está definida uma relação de ordem total. Uma **inversão** em  $S$  é um par de elementos  $x$  e  $y$  tais que  $x$

Hidden page

algoritmo de Karen está correto e analise seu tempo de execução para um caminho de tamanho  $h$ .

- C-11.31 Este problema trata de uma modificação do algoritmo de quick-select para torná-lo determinístico e ainda podendo ser executado em tempo  $O(n)$  em uma seqüência de  $n$  elementos. A idéia é modificar a forma com que se escolhe o pivô de modo que seja escolhido de forma determinística, e não aleatoriamente, como segue:

Divida o conjunto  $S$  em  $\lceil n/5 \rceil$  grupos de tamanho 5 cada (exceto, talvez, um grupo). Ordene cada pequeno conjunto e identifique a mediana do conjunto. Com estas  $\lceil n/5 \rceil$  medianas “pequenas”, aplique o algoritmo de seleção recursivamente para encontrar a mediana das medianas “pequenas”. Use este elemento como pivô e continue como no algoritmo de quick-select.

Mostre que este método determinístico tem tempo  $O(n)$ , respondendo as seguintes questões (ignore pisos e tetos se isto simplificar os cálculos, pois os resultados assintóticos continuam os mesmos):

- Quantas medianas pequenas são menores do que ou iguais ao pivô escolhido? Quantas são maiores ou iguais?
- Para cada mediana pequena menor ou igual ao pivô, quantos elementos são menores ou iguais ao pivô? O mesmo é verdadeiro para aquelas maiores ou iguais ao pivô?
- Discuta por que o método para encontrar o pivô deterministicamente e usá-lo para particionar  $S$  custa tempo  $O(n)$ .
- Baseado nestas estimativas, escreva uma relação de recorrência que limita o tempo de execução de pior caso  $t(n)$  para este algoritmo de seleção. (Note que no pior caso há duas chamadas recursivas – uma para encontrar a mediana das medianas pequenas e outra para fazer a recorrência com o maior valor entre  $L$  e  $G$ .)
- Usando essa relação de recorrência, mostre por indução que  $t(n) = O(n)$ .

## Projetos

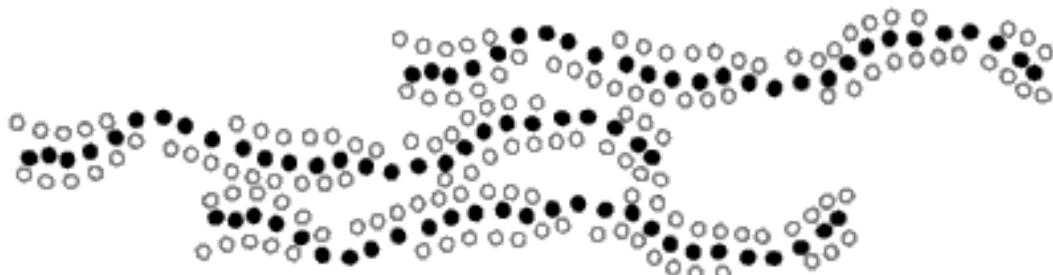
- P-11.1 Experimentalmente, compare o desempenho do quick-sort in-place e uma versão de um quick-sort que não seja in-place.
- P-11.2 Projete e implemente uma versão estável do algoritmo de bucket-sort para ordenar uma seqüência de  $n$  elementos com chaves inteiras no intervalo  $[0, N - 1]$  para  $N \geq 2$ . O algoritmo deve ser executado em tempo  $O(n + N)$ .
- P-11.3 Implemente o merge-sort e o quick-sort determinístico e realize testes para verificar qual dos dois é mais rápido. Seus testes devem incluir seqüências que aparentemente são “aleatórias” e seqüências que parecem “quase” ordenadas.
- P-11.4 Implemente o quick-sort determinístico e sua versão randomizada e realize testes para verificar qual dos dois é mais rápido. Seus testes devem incluir seqüências que aparentemente são “aleatórias” e seqüências que parecem “quase” ordenadas.
- P-11.5 Implemente uma versão in-place do insertion-sort e uma versão in-place do quick-sort. Realize testes para determinar os valores de  $n$  para os quais o quick-sort é melhor (em média) do que o insertion-sort.

- P-11.6 Projete e implemente uma animação para um dos algoritmos de ordenação apresentados neste capítulo. Sua animação deve ilustrar as propriedades essenciais do algoritmo de forma intuitiva.
- P-11.7 Implemente os algoritmos quick-sort randomizado e quick-select, e projete uma série de experimentos para testar suas velocidades relativas.
- P-11.8 Implemente um TAD para conjuntos estendidos, incluindo os métodos `union(B)`, `intersect(B)`, `size()` e `isEmpty()` mais os métodos `equals(B)`, `contains(e)`, `insert(e)` e `remove(e)`.
- P-11.9 Implemente a estrutura de dados de partição union/find baseada em árvores com ambas as heurísticas: união-pelo-tamanho e compressão do caminho.

---

## Observações sobre o capítulo

O clássico texto de Knuth *Sorting and Searching* [63] contém uma extensa história do problema da classificação e algoritmos para resolvê-lo. Huang e Langston [52] descrevem como unificar duas listas ordenadas de uma forma in-place e com tempo linear. Nossa TAD conjunto é derivado do TAD conjunto de Aho, Hopcroft e Ullman [5]. O algoritmo padrão para quick-sort foi feito por Hoare [49]. Uma análise mais profunda do quick-sort randomizado pode ser encontrada no livro de Motwani e Raghavan [79]. A análise do quick-sort apresentada neste capítulo é uma combinação de uma análise apresentada na edição anterior deste livro, e a análise de Kleinberg e Tardos [59]. A análise do quick-sort do Exercício C-11.7 é devido a Littman, Gonnet e Baeza-Yates [41] fornecem comparações experimentais e análises teóricas de uma série de algoritmos diferentes de ordenação. O termo “poda e busca” é originário da literatura de geometria computacional (como no livro de Clarkson [22] e Meggido [72,73]). O termo “diminuição e conquista” é de Levitin [68].

**Conteúdo**

<b>12.1</b>	<b>Operações com strings.....</b>	<b>476</b>
12.1.1	A classe Java String .....	477
12.1.2	A classe Java StringBuffer.....	477
<b>12.2</b>	<b>Algoritmos para procura de padrões.....</b>	<b>478</b>
12.2.1	Força bruta .....	478
12.2.2.	O algoritmo Boyer-Moore .....	480
12.2.3	O algoritmo de Knuth-Morris-Pratt .....	483
<b>12.3</b>	<b>Tries.....</b>	<b>487</b>
12.3.1	Tries-padrão .....	487
12.3.2	Tries comprimidos.....	489
12.3.3	Tries de sufixos .....	492
12.3.4	Mecanismos de busca .....	495
<b>12.4</b>	<b>Compressão de textos.....</b>	<b>495</b>
12.4.1	O algoritmo de codificação de Huffman .....	497
12.4.2	O método guloso.....	497
<b>12.5</b>	<b>Testando a similaridade de textos .....</b>	<b>498</b>
12.5.1	O problema da maior subseqüência comum .....	498
12.5.2	Programação dinâmica .....	499
12.5.3	Aplicando programação dinâmica ao problema da LCS.....	499
<b>12.6</b>	<b>Exercícios .....</b>	<b>502</b>

## 12.1 Operações com strings

O processamento de documentos está se tornando rapidamente uma das funções dominantes dos computadores. Estes são usados para editar documentos, pesquisar documentos, transportar documentos pela Internet e apresentar documentos em telas e impressoras. Pesquisas na Web estão se tornando aplicações importantes e significativas para os computadores, e muito do processamento essencial em todo este processamento de documentos envolve cadeias de caracteres e procura de padrões. Por exemplo, os formatos de documentos para Internet HTML e XML são primeiramente formatos de textos, com rótulos especiais adicionados para comportar conteúdo multimídia. Buscar o significado de muitos terabytes de informação na Internet requer um trabalho considerável de processamento de texto.

Além de ter aplicações interessantes, os tópicos deste capítulo também salientam alguns padrões de projeto importantes. Em particular, na seção sobre procura de padrões, será discutido o **método da força bruta**, que muitas vezes é ineficiente, mas tem larga aplicação. Para compressão de textos, se estudará o **método guloso**, que freqüentemente permite que sejam obtidas soluções aproximadas para problemas difíceis, e que, para alguns problemas (como na compressão de textos), de fato, fornecem algoritmos ótimos. Finalmente, na discussão sobre similaridade de textos, se introduz a **programação dinâmica**, que pode ser aplicada em instâncias especiais para resolver em tempo polinomial problemas que a princípio parecem requerer tempo exponencial para resolução.

### Processando texto

No núcleo dos algoritmos para processamento de texto, estão métodos para lidar com cadeias de caracteres. As cadeias de caracteres podem surgir de uma variedade de origens, incluindo aplicações científicas, lingüísticas e da Internet. De fato, abaixo temos exemplos destas cadeias de caracteres:

```
P = "CGTAAACTGCTTTAATCAAACGC"
S = "http://java.datastructures.net".
```

A primeira cadeia de caracteres,  $P$ , tem origem em aplicações de pesquisa de DNA, enquanto a última cadeia de caracteres,  $S$ , é o endereço (URL) do site da Web que acompanha este livro.

Várias das operações típicas do processamento de cadeias de caracteres envolvem quebrar cadeias de caracteres longas em cadeias menores. Para poder falar dos pedaços que resultam deste tipo de operações, usa-se o termo **substring** de uma cadeia de caracteres  $P$  de comprimento  $m$  para se referir a uma cadeia da forma  $P[i]P[i + 1]\dots P[j]$  para  $0 \leq i \leq j \leq m - 1$ , ou seja, a cadeia formada pelos caracteres em  $P$ , do índice  $i$  ao índice  $j$ , inclusive. Tecnicamente, isso significa que uma cadeia de caracteres é uma substring de si mesma (com  $i = 0$  e  $j = m - 1$ ), por isso, desejando-se excluir esta possibilidade, deve-se restringir a definição às substrings **próprias**, que requerem  $i > 0$  ou  $j < m - 1$ .

Para simplificar a notação de modo fazer referência às substrings, usa-se  $P[i..j]$  para denotar a substring de  $P$  do índice  $i$  ao índice  $j$ , inclusive. Ou seja,

$$P[i..j] = P[i]P[i + 1]\dots P[j].$$

Usa-se a convenção de que se  $i > j$  então  $P[i..j]$  é uma **cadeia vazia**, que tem comprimento 0. Além disso, para distinguir alguns casos especiais de substrings, chama-se qualquer cadeia da forma  $P[0..i]$  para  $0 \leq i \leq m - 1$  de **prefixo** de  $P$  e qualquer cadeia da forma  $P[i..m - 1]$  para  $0 \leq i \leq m - 1$  de **sufixo** de  $P$ . Por exemplo, se  $P$  for a cadeia de DNA dada acima, então "CGTAA" é um prefixo de  $P$ , "CGC" é um sufixo de  $P$  e "TTAAC" é uma substring (própria) de  $P$ . Deve-se observar que uma cadeia vazia é prefixo e sufixo de qualquer outra cadeia.

Para permitir noções gerais do modo correto de um caractere se chegar de uma string, tipicamente não se restringe aos caracteres em  $T$  e  $P$  para explicitamente chegar a um conjunto bem

formado de caracteres, como o conjunto de caracteres Unicode. Em vez disso, usa-se o símbolo  $\Sigma$  para denotar o conjunto de caracteres, ou *alfabeto*, a partir de qual os caracteres podem vir. Desde que muitos algoritmos de processamento de documentos são usados em aplicações nas quais o conjunto de caracteres base é finito, usualmente assume-se que o tamanho do alfabeto  $\Sigma$ , denotado com  $|\Sigma|$ , é uma constante.

As operações sobre cadeias de caracteres são de dois tipos: aquelas que modificam uma cadeia e aquelas que simplesmente retornam informação sobre a cadeia, sem alterá-la. Java torna esta distinção precisa definindo a classe `String`, que representa as *cadeias imutáveis*, e a classe `StringBuffer`, que representa as *cadeias mutáveis*, ou que podem ser modificadas.

### 12.1.1 A classe Java String

As principais operações da classe `String` em Java sobre uma cadeia  $S$  são listadas abaixo:

- `length()`: Retorna o comprimento  $n$  de  $S$ . Entrada: nenhuma. Saída: inteiro.
- `charAt( $i$ )`: Retorna o caractere na posição  $i$  em  $S$ .
- `startsWith( $Q$ )`: Determina se  $Q$  é um prefixo de  $S$ .
- `endsWith( $Q$ )`: Determina se  $Q$  é um sufixo de  $S$ .
- `substring( $i,j$ )`: Retorna a substring  $S[i..j]$ .
- `concat( $Q$ )`: Retorna a concatenação de  $S$  e  $Q$ , ou seja,  $S + Q$ .
- `equals( $Q$ )`: Determina se  $Q$  é igual a  $S$ .
- `indexOf( $Q$ )`: Se  $Q$  for uma substring de  $S$ , retorna o índice da primeira ocorrência de  $Q$  em  $S$ , se não retorna  $-1$ .

Esta coleção forma as operações típicas para cadeias imutáveis.

**Exemplo 12.1** Considere o seguinte conjunto de operações, que são executados na string  $S = "abcdefghijklmnopqrstuvwxyz"$ :

Operação	Saída
<code>length()</code>	16
<code>charAt(5)</code>	'f'
<code>concat("grs")</code>	"abcdefghijklmnopqrstuvwxyzgrs"
<code>endsWith("javapop")</code>	false
<code>indexOf("ghi")</code>	6
<code>startsWith("abcd")</code>	true
<code>substring(4,9)</code>	"efghij"

Com a exceção do método `indexOf( $Q$ )`, que será discutido na Seção 12.2, todos os métodos acima são facilmente implementados simplesmente representando a cadeia como um arranjo de caracteres, que é a implementação-padrão de uma `String` em Java.

### 12.1.2 A classe Java StringBuffer

As principais operações da classe `StringBuffer` em Java sobre uma cadeia  $S$  são listadas abaixo:

- `append( $Q$ )`: Retorna  $S + Q$ , substituindo  $S$  por  $S + Q$ .
- `insert( $i,Q$ )`: Retorna e atualiza  $S$  inserindo  $Q$  em  $S$  na posição de índice  $i$ .

`reverse()`: Reverte e retorna a string  $S$ .

`setCharAt( $i, ch$ )`: Atualiza o caractere na posição  $i$  de  $S$ , mudando-o para  $ch$ .

`charAt( $i$ )`: Retorna o caractere na posição  $i$  de  $S$ .

Condições de erro ocorrem quando o item  $i$  está além dos limites da cadeia. Com a exceção do método `charAt`, a maioria dos métodos da classe `String` não são imediatamente disponíveis a um objeto  $S$  da classe `StringBuffer`. Felizmente, a classe `StringBuffer` provê um método `toString()` que retorna uma versão `String` de  $S$ , que pode ser usada para acessar os métodos da classe `String`.

**Exemplo 12.2** Considere o seguinte conjunto de operações realizadas sobre a cadeia mutável  $S = "abcdefghijklmnopqrstuvwxyz"$ :

Operação	$S$
<code>append("qrs")</code>	"abcdefghijklmnopqrstuvwxyzqrs"
<code>insert(3, "xyz")</code>	"abcxyzdefghijklmnopqrstuvwxyz"
<code>reverse()</code>	"srqponmlkjihgfedzyxcba"
<code>setCharAt(7, 'W')</code>	"srqponmWkjihgfedzyxcba"

## 12.2 Algoritmos para procura de padrões

No problema clássico de *procura de padrões* em cadeias de caracteres, recebe-se uma cadeia de caracteres ou *texto*  $T$ , de comprimento  $n$ , e uma segunda cadeia ou *padrão*  $P$ , de comprimento  $m$ , e deseja-se saber se  $P$  é uma substring de  $T$ . Ou seja, deseja-se saber se existe uma substring de  $T$  iniciando na posição  $i$  que confere com  $P$  caractere a caractere, de forma que  $T[i] = P[0]$ ,  $T[i + 1] = P[1], \dots, T[i + m - 1] = P[m - 1]$ , ou seja,  $P = T[i..i + m - 1]$ . Assim, a saída de um algoritmo de procura de padrões poderia ser uma indicação de que o padrão  $P$  não existe em  $T$ , ou um inteiro indicando a posição ou índice em  $T$  onde inicia uma cadeia igual a  $P$ . Este é exatamente o cálculo feito em Java pelo método `indexOf` na interface `String`. Alternativamente, pode-se desejar encontrar todos os índices em que uma substring de  $T$  seja igual a  $P$ .

Nesta seção, serão apresentados três algoritmos para procura de padrões (com níveis progressivos de dificuldade).

### 12.2.1 Força bruta

O padrão de projeto algorítmico baseado em *força bruta* é uma técnica poderosa para o projeto de algoritmos quando se tem algo a procurar ou quando se deseja otimizar alguma função. Aplicando esta técnica em uma situação genérica, tipicamente enumeram-se todas as possíveis configurações das entradas envolvidas e escolhe-se a melhor das configurações enumeradas.

Aplicando esta técnica para o algoritmo de *procura de padrões por força bruta*, deriva-se o que é provavelmente o primeiro algoritmo em que se pode pensar para resolver o problema da procura de padrões – simplesmente testam-se todas as possíveis colocações de  $P$  em relação a  $T$ . Este algoritmo, mostrado no Trecho de código 12.1, é bastante simples.

**Algoritmo** `BruteForceMatch( $T, P$ )`:

**Entrada:** as cadeias  $T$  (texto) com  $n$  caracteres e  $P$  (padrão) com  $m$  caracteres.

**Saída:** o índice da primeira substring de  $T$  igual a  $P$ , ou uma indicação de que  $P$  não é uma substring de  $T$ .

Hidden page

### 12.2.2. O algoritmo Boyer-Moore

Pode-se pensar que é sempre necessário examinar cada caractere de  $T$  para localizar o padrão  $P$  como uma substring. Isso não acontece sempre, pois o algoritmo de procura de padrões **Boyer-Moore (BM)**, que se estuda nessa seção, consegue evitar comparações entre  $P$  e uma boa parte dos caracteres em  $T$ . O único problema é que, enquanto o algoritmo de força bruta pode trabalhar com um alfabeto ilimitado, o algoritmo Boyer-Moore assume que o alfabeto tem tamanho finito. Ele tem melhor desempenho quando o alfabeto é de tamanho moderado e o padrão é relativamente longo. Assim, o algoritmo BM é ideal para procurar palavras em documentos. Nesta seção, será descrita uma versão simplificada do algoritmo original de Boyer-Moore.

A idéia principal do algoritmo BM é melhorar o tempo de execução do algoritmo de força bruta adicionando a ele duas heurísticas que potencialmente podem economizar tempo. Basicamente, essas heurísticas são as seguintes:

**Heurística do espelho:** quando se testa uma possível colocação de  $P$  em  $T$ , começa-se as comparações de forma invertida, ou seja, pelo final de  $P$ , e recua-se até o início de  $P$ .

**Heurística do salto de caracteres:** durante o teste de uma possível colocação de  $P$  em  $T$ , uma diferença entre o caractere  $T[i] = c$  com o caractere correspondente  $P[j]$  é tratada como segue: se  $c$  não está contido em lugar algum de  $P$ , então move-se  $P$  completamente para depois de  $T[i]$  (pois  $T[i]$  não pode estar em  $P$ ). Caso contrário, move-se  $P$  para frente até que uma ocorrência do caractere  $c$  em  $P$  esteja alinhada com  $T[i]$ .

Estas heurísticas serão formalizadas em breve, mas de forma intuitiva pode-se perceber que elas trabalham de forma integrada. A heurística do espelho permite que a segunda heurística evite comparações entre  $P$  e grupos inteiros de caracteres em  $T$ . Neste caso, ao menos, pode-se chegar ao resultado mais depressa indo de trás para frente, pois se encontra uma diferença entre caracteres quando se procura  $P$  em uma certa posição de  $T$ , então provavelmente se evita uma série de comparações desnecessárias movendo  $P$  de forma significativa em relação a  $T$  através da heurística do salto de caracteres. A heurística do salto de caracteres traz grande vantagem se for aplicada cedo no teste de ocorrência de  $P$  em  $T$ .

Define-se agora como a heurística do salto de caracteres pode ser integrada em um algoritmo de procura de padrões em cadeias de caracteres. Para implementar essa heurística, define-se uma função  $\text{last}(c)$  que recebe um caractere  $c$  do alfabeto e caracteriza o quanto é possível avançar o padrão  $P$  se um caractere igual a  $c$  for encontrado no texto e não fizer parte do padrão. Em particular, define-se  $\text{last}(c)$  como

- Se  $c$  está em  $P$ ,  $\text{last}(c)$  é o índice da última ocorrência (mais à direita) de  $c$  em  $P$ . Senão, convencionase que  $\text{last}(c) = -1$ .

Se os caracteres podem ser usados como índices em arranjos, então a função  $\text{last}$  pode ser implementada facilmente como uma tabela. Deixa-se o método de construção desta tabela em tempo  $O(m + |\Sigma|)$ , dado  $P$ , como um simples exercício (R-12.6). Essa função  $\text{last}$  fornece toda a informação de que se precisa para realizar a heurística do salto de caracteres.

No Trecho de código 12.2, mostra-se o algoritmo Boyer-Moore para procura de padrões.

**Algoritmo BMMatch( $T, P$ ):**

**Entrada:** as cadeias  $T$  (texto) com  $n$  caracteres e  $P$  (padrão) com  $m$  caracteres.

**Saída:** o índice da primeira substring de  $T$  igual a  $P$ , ou uma indicação de que  $P$  não é uma substring de  $T$ .

calcular a função  $\text{last}$

$i \leftarrow m - 1$

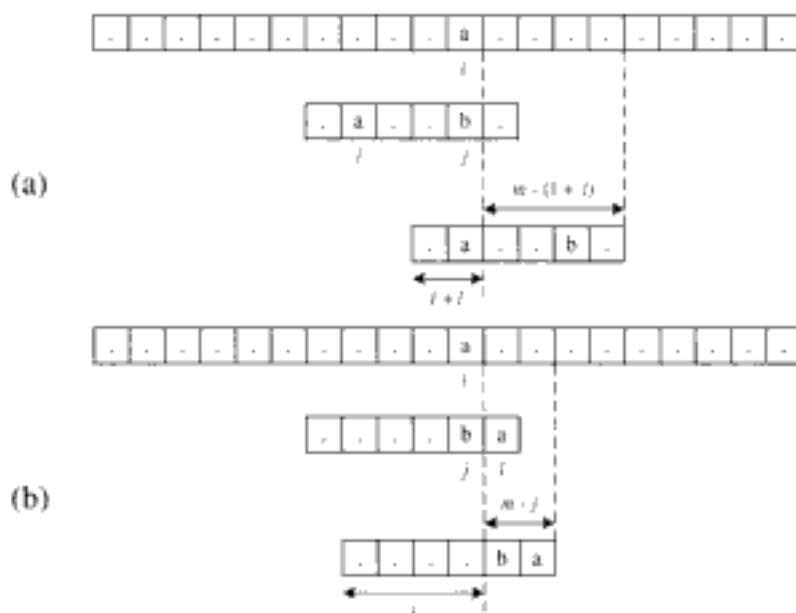
```

 $j \leftarrow m - 1$ 
repita
  se  $P[j] = T[i]$  então
    se  $j = 0$  então
      retorna  $i$  {achado!}
    senão
       $i \leftarrow i - 1$ 
       $j \leftarrow j - 1$ 
    senão
       $i \leftarrow i + m - \min(j, 1 + \text{last}(T[i]))$  { salto }
       $j \leftarrow m - 1$ 
  até  $i > n - 1$ 
retorna "Não existe substring em  $T$  igual a  $P."$ 

```

**Trecho de código 12.2** O algoritmo de procura de padrões Boyer-Moore.

O passo do salto é ilustrado na Figura 12.2.



**Figura 12.2** Ilustração do passo de salto no algoritmo BM (veja o Trecho de código 12.2), onde usa-se a notação  $l = \text{last}(T[i])$ . Diferenciam-se dois casos: (a)  $1 + 1 \leq j$ , onde se move o padrão  $j - 1$  unidades; (b)  $j < 1 + 1$ , onde se move o padrão uma unidade

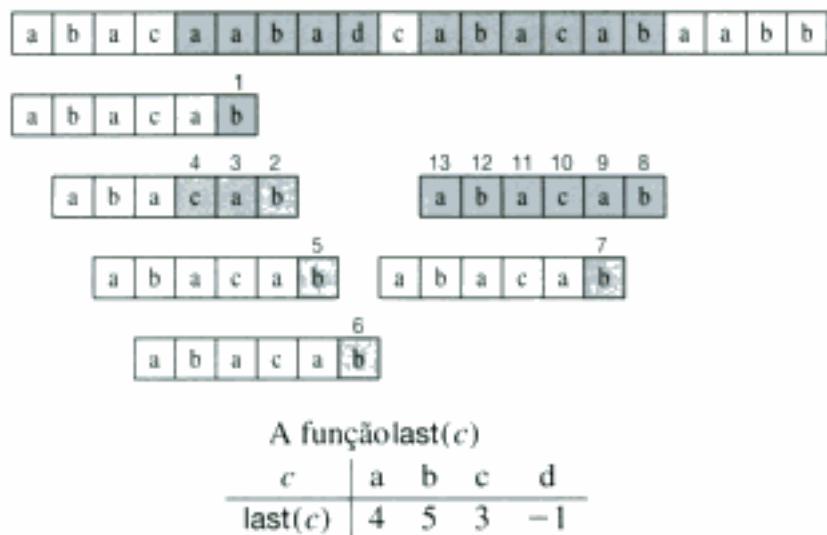
A Figura 12.3 ilustra a execução do algoritmo Boyer-Moore em uma entrada similar à do Exemplo 12.3.

A correção do algoritmo BM para procura de padrões vem do fato de que, a cada vez que o método move o padrão, é garantido que ele não “pula” sobre um possível acerto. Isso acontece porque  $\text{last}(c)$  é o local da *última* ocorrência de  $c$  em  $P$ .

O tempo de execução de pior caso do algoritmo BM é  $O(m + |\Sigma|)$ . O cálculo da função  $\text{last}$  custa tempo  $O(m + |\Sigma|)$  e a procura pelo padrão custa tempo  $O(nm)$  no pior caso, o mesmo que o algoritmo de força bruta. Um exemplo de par texto/padrão que atinge o pior tempo é

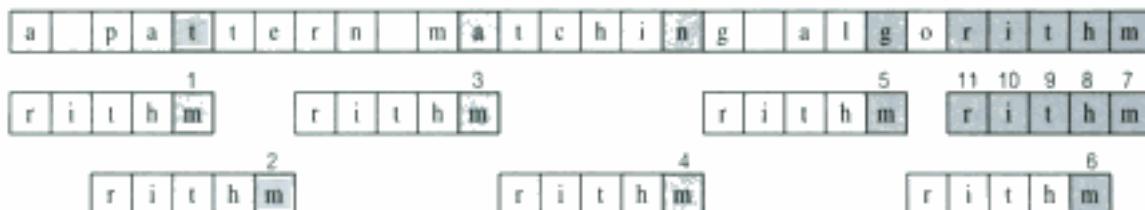
$$T = \overbrace{aaaaaaa\cdots a}^n$$

$$P = \overbrace{baa\cdots a}^{m-1}$$



**Figura 12.3** Uma ilustração do algoritmo BM para procura de padrões. O algoritmo realiza 13 comparações entre caracteres, que são indicadas pela numeração.

O pior desempenho, no entanto, é difícil de ser obtido com texto em língua inglesa. Neste caso, o algoritmo BM é freqüentemente habilitado para saltar grandes porções do texto. (Ver Figura 12.4.) Evidência experimental no texto em inglês mostra que o número médio de comparações feitas por caractere é 0.24 para uma cadeia padrão de 5 caracteres.



**Figura 12.4** Um exemplo de uma execução do algoritmo Boyer-Moore em um texto em inglês.

Uma implementação em Java do algoritmo BM para procura de padrões é mostrada no Trecho de código 12.3.

```
/** Versão simplificada do algoritmo Boyer-Moore (BM), que usa apenas as
 * heurísticas do espelho e do salto de caracteres.
 * @return Índice do começo da ocorrência mais à esquerda do texto igual ao
 * padrão, ou -1 se não há tal ocorrência.*/
public static int BMmatch (String text, String pattern) {
    int[] last = buildLastFunction(pattern);
    int n = text.length();
    int m = pattern.length();
    int i = m - 1;
    if (i > n - 1)
        return -1; // não há ocorrências se o padrão é mais longo do que o texto
    int j = m - 1;
    do {
        if (pattern.charAt(j) == text.charAt(i))
            if (j == 0)
                return i; // achado
            else { // heurística do espelho: do fim para o início
                i--;
                j--;
            }
    } while (i >= 0);
```

```

    }
    else { // heurística do salto de caracteres
        i = i + m - Math.min(j, 1 + last[text.charAt(i)]);
        j = m - 1;
    }
} while (i <= n - 1);
return -1; // não foi achado
}
public static int[] buildLastFunction (String pattern) {
    int[] last = new int[128]; // assume-se o conjunto de caracteres ASCII
    for (int i = 0; i < 128; i++) {
        last[i] = -1; // inicializa-se o arranjo
    }
    for (int i = 0; i < pattern.length(); i++) {
        last[pattern.charAt(i)] = i; // conversão para um código ASCII inteiro
    }
    return last;
}

```

**Trecho de código 12.3** Implementação em Java do algoritmo BM para procura de padrões. O algoritmo é expresso por dois métodos estáticos: o método `BMmatch` realiza a procura por padrões e chama o método auxiliar `buildLastFunction` para calcular a função `last`, expressa por um arranjo indexado pelo código ASCII do caractere. O método `BMmatch` indica a ausência de uma ocorrência retornando o valor convencional `-1`.

Na realidade, é apresentada uma versão simplificada do algoritmo Boyer-Moore (BM). O algoritmo BM original tem um desempenho  $O(n + m + |\Sigma|)$ , usando uma heurística alternativa de saltos à frente até o texto já parcialmente encontrado, sempre que ele avança o padrão mais do que a heurística de salto de caracteres apresentada aqui. Esta heurística alternativa para o salto à frente é baseada na aplicação da idéia principal do algoritmo Knuth-Morris-Pratt de procura de padrões, que será discutido a seguir.

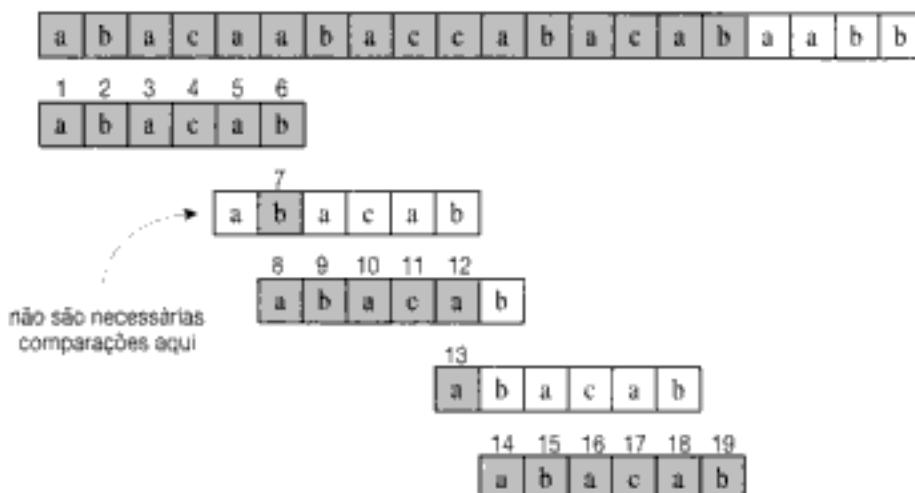
### 12.2.3 O algoritmo de Knuth-Morris-Pratt

Estudando o desempenho de pior caso dos algoritmos de força bruta e Boyer-Moore em instâncias específicas do problema, como a mostrada no Exemplo 12.3, nota-se uma fonte de ineficiência. Especificamente, pode-se realizar muitas comparações enquanto se testa um posicionamento do padrão sobre o texto, mas descobrindo-se um caractere do padrão que falha nesta comparação, então se joga fora toda a informação adquirida pelas comparações anteriores e começa-se novamente (do zero) no ponto em que o padrão for posicionado, mais à frente. O algoritmo Knuth-Morris-Pratt (KMP), discutido nesta seção, evita este desperdício de informação e, ao fazer isso, atinge um tempo de execução de  $O(n + m)$ , que é ótimo no pior caso. Ou seja, no pior caso qualquer algoritmo de procura de padrões terá de examinar todos os caracteres do texto e do padrão ao menos uma vez.

#### A função de falha

A idéia principal do algoritmo KMP é pré-processar  $P$  para calcular uma *função de falha*  $f$ , que indica o quanto se deve avançar  $P$  de forma que seja possível reutilizar as comparações realizadas anteriormente tanto quanto possível. Especificamente, a função de falha  $f(j)$  é definida como o comprimento do prefixo mais longo de  $P$ , que é um sufixo de  $P[1..j]$  (deve-se observar que *não*

Hidden page



**Figura 12.5** Ilustração do algoritmo de procura de padrões Knuth-Morris-Pratt. A função de falha  $f$  para este padrão é dada no Exemplo 12.4. O algoritmo realiza 19 comparações de caracteres, indicadas pela numeração.

### Desempenho

Excluindo o cálculo da função de falha, o tempo de execução do algoritmo Knuth-Morris-Pratt é certamente proporcional ao número de iterações do laço **enquanto**. Para a análise, irá-se definir  $k = i - j$ . Intuitivamente,  $k$  é o total de avanço do padrão  $P$  em relação a  $T$ . Vide que em toda a execução do algoritmo tem-se  $k \leq n$ . Um dos três casos abaixo ocorre a cada iteração do laço.

- Se  $T[i] = P[j]$ , então  $i$  aumenta em 1 e  $k$  não se altera, pois  $j$  também aumenta em 1.
- Se  $T[i] \neq P[j]$  e  $j > 0$ , então  $i$  não se altera e  $k$  aumenta em pelo menos 1, pois neste caso  $k$  muda de  $i - j$  para  $i - f(j - 1)$ , que é maior do que  $j - f(j - 1)$ , que é positivo porque  $f(j - 1) < j$ .
- Se  $T[i] \neq P[j]$  e  $j = 0$ , então  $i$  aumenta em 1 e  $k$  aumenta em 1, pois  $j$  não se altera.

Assim, a cada iteração do laço tem-se que  $i$  ou  $k$  aumentam em ao menos 1 (possivelmente os dois), portanto o número total de iterações do laço **enquanto** no algoritmo KMP é de no máximo  $2n$ . Atingir este número, é claro, pressupõe que a função de falha para  $P$  já foi calculada.

### Construindo a função de falha KMP

Para construir a função de falha, usa-se o método mostrado no Trecho de código 12.5, que é um processo semelhante ao algoritmo KMPmatch. Compara-se o padrão a si mesmo como no algoritmo KMP. A cada vez que se tem dois caracteres iguais, faz-se  $f(i) = j + 1$ . Vide que como se tem  $i > j$  em toda a execução do algoritmo,  $f(j - 1)$  está sempre definida quando é preciso usá-la.

#### Algoritmo KMPFailureFunction( $P$ ):

**Entrada:** o padrão  $P$ , com  $m$  caracteres.

**Saída:** a função de falha  $f$  para  $P$ , mapeando  $j$  para o comprimento do mais longo prefixo de  $P$  que é um sufixo de  $P[1..j]$ .

```

 $i \leftarrow 1$ 
 $j \leftarrow 0$ 
 $f(0) \leftarrow 0$ 
enquanto  $i < m$  faça
  se  $P[j] = P[i]$  então
     $\{$  já foram achados  $j + 1$  caracteres  $\}$ 
  
```

Hidden page

```

    }
    else if (j > 0) // j seguem um prefixo que serve
        j = fail[j - 1];
    else { // não foi achado
        fail[i] = 0;
        i++;
    }
}
return fail;
}

```

**Trecho de código 12.6** Implementação em Java do algoritmo Knuth-Morris-Pratt para procura de padrões. O algoritmo é expresso por dois métodos estáticos: o método KMPmatch realiza a procura e chama o método auxiliar computeFailFunction para calcular a função de falha, armazenada em um arranjo. O método KMPmatch indica a falha na procura retornando o valor convencional – 1.

## 12.3 Tries

Os algoritmos de procura de padrões apresentados na seção anterior aceleram a procura em um texto, pré-processando o padrão (para calcular a função de falha no algoritmo KMP ou a função last no algoritmo BM). Nesta seção, será usada uma abordagem complementar, ou seja, serão apresentados algoritmos de procura que pré-processam o texto. Esta abordagem é adequada para aplicações em que uma série de consultas é realizada em um texto fixo, de forma que o custo inicial de pré-processar o texto é compensado pela aceleração das consultas seguintes (por exemplo, um site na Web que oferece consultas a *Hamlet* ou um mecanismo de busca que oferece páginas da Web sobre o tópico *Hamlet*).

Um *trie* é uma estrutura de dados baseada em árvore para armazenar cadeias de caracteres e suportar uma rápida procura de padrões. A aplicação principal dos tries é na recuperação de informação. De fato, o nome “trie” vem da palavra “retrieval” (recuperação). Em uma aplicação de recuperação de informação, como a procura por uma sequência de DNA em uma base de genomas, tem-se uma coleção  $S$  de cadeias de caracteres definidas usando-se o mesmo alfabeto. As operações primárias de consulta suportadas por tries são procura de padrões e *procura de prefixos*. A última operação envolve receber uma cadeia  $X$  e determinar todas as cadeias em  $S$  que têm  $X$  como prefixo.

### 12.3.1 Tries-padrão

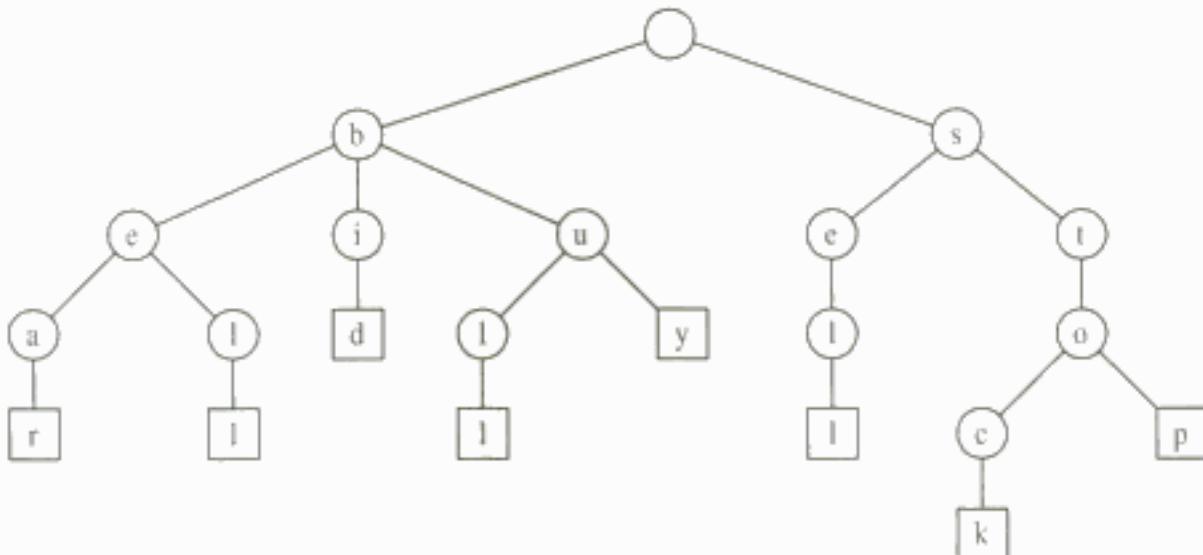
Seja  $S$  um conjunto de  $s$  cadeias de caracteres do alfabeto  $\Sigma$ , de forma que nenhuma cadeia de  $S$  seja um prefixo de outra cadeia. Um *trie-padrão* para  $S$  é uma árvore ordenada  $T$  com as propriedades seguintes (ver Figura 12.6):

- Cada nodo de  $T$ , exceto a raiz, é rotulado com um caractere de  $\Sigma$ .
- A ordem dos filhos de um nodo interno em  $T$  é determinada por uma ordenação canônica do alfabeto  $\Sigma$ .
- $T$  tem  $s$  nodos externos, cada um associado com uma cadeia de  $S$  de tal forma que a concatenação dos rótulos dos nodos no caminho da raiz até um nodo externo  $v$  de  $T$  resulta na cadeia de  $S$  associada com  $v$ .

Assim, um trie  $T$  representa as cadeias de  $S$  em caminhos da raiz até os nodos externos de  $T$ . Observa-se a importância de assumir que nenhuma cadeia de  $S$  é prefixo de outra cadeia, pois

isso garante que toda cadeia seja unicamente associada a um nodo externo de  $T$ . Pode-se sempre satisfazer essa exigência adicionando um caractere especial que não se encontra no alfabeto  $\Sigma$  ao final de uma cadeia.

Um nodo interno em um trie-padrão  $T$  pode ter entre 1 e  $d$  filhos, onde  $d$  é o tamanho do alfabeto. Existe uma aresta indo da raiz  $r$  até um de seus filhos para cada caractere em primeiro lugar em alguma cadeia na coleção  $S$ . Adicionalmente, um caminho da raiz de  $T$  para um nodo interno  $v$  com profundidade  $i$  corresponde a um prefixo  $X[0..i - 1]$  com  $i$  caracteres de uma cadeia  $X$  em  $S$ . De fato, para cada caractere  $c$  que pode seguir o prefixo  $X[0..i - 1]$  em uma cadeia de  $S$ , existe um filho de  $v$  rotulado com o caractere  $c$ . Desta forma, um trie armazena concisamente os prefixos comuns que existem em um conjunto de cadeias.



**Figure 12.6** Trie-padrão para as strings {bear, Bell, bid, Bull, buy, sell, stock, stop}.

Se existem somente dois caracteres no alfabeto, então o trie é essencialmente uma árvore binária, embora alguns nodos internos possam ter somente um filho (ou seja, pode ser uma árvore binária imprópria). Em geral, se existem  $d$  caracteres no alfabeto, então o trie será uma árvore múltipla em que cada nodo interno tem de 1 a  $d$  filhos. Além disso, é provável que existam vários nodos internos em  $T$  que tenham menos do que  $d$  filhos. Por exemplo, o trie mostrado na Figura 12.6 tem vários nodos internos com apenas um filho. É possível implementar um trie com uma árvore armazenando caracteres em seus nodos.

A proposição a seguir descreve algumas propriedades estruturais importantes de um trie-padrão:

**Proposição 12.6** Um trie-padrão que armazena uma coleção  $S$  de cadeia de caracteres de comprimento total  $n$  com um alfabeto de tamanho  $d$  tem as seguintes propriedades:

- Todo nodo interno de  $T$  tem no máximo  $d$  filhos.
- $T$  tem  $s$  nodos externos.
- A altura de  $T$  é igual ao comprimento da maior cadeia em  $S$ .
- O número de nodos em  $T$  é  $O(n)$ .

O pior caso para o número de nodos de um trie ocorre quando nenhum par de cadeias compartilha um prefixo, ou seja, exceto pela raiz, todos os nodos internos têm um filho.

Um trie  $T$  para um conjunto  $S$  de cadeias de caracteres pode ser usado para implementar um dicionário cujas chaves são as cadeias de  $S$ . Realiza-se uma procura por uma cadeia  $X$  em  $T$  iniciando na raiz e seguindo os caracteres de  $X$ . Se este caminho puder ser traçado e terminar em um nodo externo, então  $X$  está no dicionário. Por exemplo, o trie da Figura 12.6, traçando o caminho

para “*bull*”, termina em um nodo externo. Se o caminho não puder ser traçado ou não terminar em um nodo externo, então  $X$  não está no dicionário. No exemplo da Figura 12.6, o caminho para “*bet*” não pode ser traçado, e o caminho para “*be*” termina em um nodo interno. Nenhuma dessas palavras está no dicionário. Deve-se observar que nessa implementação de um dicionário, são comparados caracteres isolados, em vez de toda a cadeia (chave). É fácil perceber que o tempo de execução para a procura de uma cadeia de comprimento  $m$  é  $O(dm)$ , onde  $d$  é o tamanho do alfabeto. De fato, visitam-se no máximo  $m + 1$  nodos de  $T$  e gasta-se tempo  $O(d)$  em cada nodo. Para alguns alfabetos, é possível melhorar o tempo gasto em um nodo para  $O(1)$  ou  $O(\log d)$  usando um dicionário de caracteres implementado com um tabela hash ou outro tipo de tabela. Entretanto, já que  $d$  é uma constante na maior parte das aplicações, pode-se usar uma abordagem mais simples, que usa tempo  $O(d)$  por nodo visitado.

Da discussão anterior, conclui-se que se pode usar um trie para realizar um tipo especial de procura de padrões, chamado de **procura de palavra**, no qual se deseja determinar se um dado padrão é exatamente igual a uma das palavras do texto. (Ver Figura 12.7.) A procura de palavras difere da procura de padrões comum, já que o padrão não pode ser igual a uma substring arbitrária do texto, mas a apenas uma de suas palavras. Usando um trie, a procura de palavras para um padrão de comprimento  $m$  custa tempo  $O(dm)$ , onde  $d$  é o tamanho do alfabeto, independentemente do tamanho do texto. Se o alfabeto tem tamanho constante (como é o caso para linguagens naturais e para DNA), uma consulta custa tempo  $O(m)$ , proporcional ao tamanho do padrão. Uma extensão simples deste esquema suporta procura por prefixos. No entanto, ocorrências arbitrárias do padrão no texto (por exemplo, o padrão é um sufixo próprio de uma palavra ou abrange duas palavras) não podem ser eficientemente realizadas.

Para construir um trie-padrão para um conjunto  $S$  de cadeias de caracteres, pode-se usar um algoritmo incremental que insere as cadeias uma de cada vez. Lembre-se da condição de que nenhuma cadeia de  $S$  é um prefixo de outra cadeia de  $S$ . Para inserir uma cadeia  $X$  no trie corrente  $T$ , primeiro tenta-se traçar o caminho associado com  $X$  em  $T$ . Como  $X$  ainda não está em  $T$ , e não é prefixo de nenhuma outra cadeia, se para de traçar o caminho em um nodo **interno**  $v$  de  $T$  antes de chegar ao final de  $X$ . Então, criamos um novo caminho de nodos descendentes de  $v$  para armazenar os caracteres restantes de  $X$ . O tempo para inserir  $X$  é  $O(dm)$ , onde  $m$  é o comprimento de  $X$ , e  $d$  é o tamanho do alfabeto. Assim, construir um trie completo para o conjunto  $S$  custa tempo  $O(dn)$ , onde  $n$  é o comprimento total das cadeias de caracteres em  $S$ .

Existe uma potencial ineficiência de espaço no trie-padrão, que provocou o desenvolvimento do **trie comprimido**, que é também conhecido (por razões históricas) como **trie Patricia**. Neste caso, existem potencialmente muitos nodos no trie-padrão que têm apenas um filho, e a existência desses nodos é um desperdício. Em seguida, será discutido o trie comprimido.

### 12.3.2 Tries comprimidos

Um **trie comprimido** é similar a um trie-padrão, mas garante que cada nodo interno no trie tenha pelo menos dois filhos. Ele garante esta regra comprimindo cadeias de nodos com apenas um filho em nodos isolados. (Ver Figura 12.8.) Seja  $T$  um trie-padrão. Diz-se que um nodo  $v$  de  $T$  é **redundante** se  $v$  tiver um filho e não for a raiz. Por exemplo, o trie da Figura 12.6 tem 8 nodos redundantes. Agora será mostrado que uma cadeia de  $k \geq 2$  nodos,

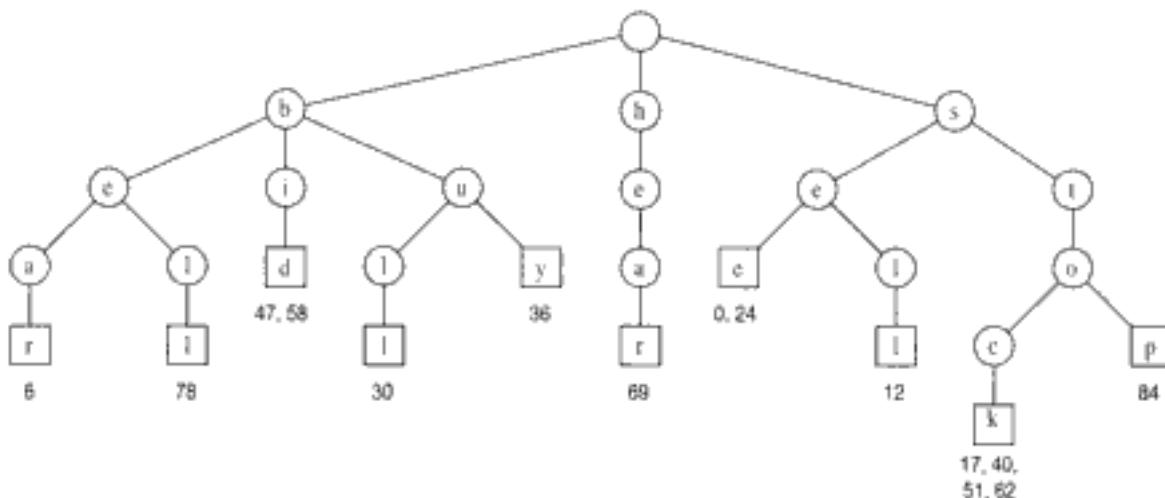
$$(v_0, v_1)(v_1, v_2)\cdots(v_{k-1}, v_k),$$

é **redundante** se:

- $v_i$  for redundante para  $i = 1, \dots, k - 1$ ,
- $v_0$  e  $v_k$  não forem redundantes.

s	e	e	a	b	e	a	r	?		s	e	l	l	s	t	o	c	k	!				
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
s	e	e	a	b	u	l	l	?		b	u	y		s	t	o	c	k	!				
24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	
b	i	d		s	t	o	c	k	!		b	i	d		s	t	o	c	k	!			
47	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63	64	65	66	67	68		
h	e	a	r		t	h	e		b	e	l	l	?		s	t	o	p	!				
69	70	71	72	73	74	75	76	77	78	79	80	81	82	83	84	85	86	87	88				

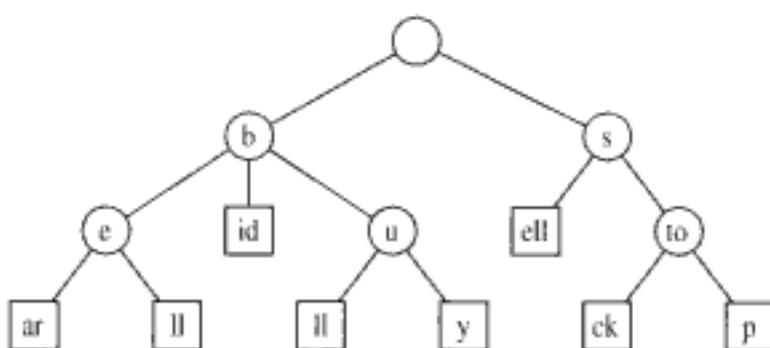
(a)



(b)

**Figura 12.7** Procura de palavras e procura de prefixos com um trie-padrão: (a) texto a ser pesquisado; (b) trie-padrão para as palavras no texto (artigos e preposições, que também são conhecidos como *stop words*, excluídos), com nodos externos aumentados para indicar as posições das palavras.

Pode-se transformar  $T$  em um trie comprimido substituindo cada cadeia redundante  $(v_0, v_1, v_2) \dots (v_{k-1}, v_k)$  com  $k \geq 2$  arestas em uma única aresta  $(v_0, v_k)$ , renomeando  $v_k$  com a concatenação dos rótulos dos nodos  $v_1, \dots, v_k$ .



**Figura 12.8** Trie comprimido para as strings  $\{bear, Bell, bid, Bull, buy, sell, stock, stop\}$ . Compare-o com o trie-padrão mostrado na Figura 12.6.

Hidden page

Hidden page

Hidden page

Hidden page

### 12.3.4 Mecanismos de busca

A World Wide Web contém uma imensa coleção de documentos textuais (páginas). A informação sobre essas páginas é coletada por programas chamados de *Web crawlers*, que armazenam esta informação em bancos de dados especiais de dicionários. Um *mecanismo de busca* na Web permite que os usuários recuperem informações relevantes destes bancos de dados, identificando páginas relevantes na Web que contêm determinadas palavras-chave. Nesta seção, será apresentado um modelo simplificado de um mecanismo de busca.

#### Arquivos invertidos

A informação-base armazenada por um mecanismo de busca é um dicionário, chamado de *índice invertido* ou *arquivo invertido*, o qual armazena pares chave-valor ( $w, L$ ), onde  $w$  é uma palavra e  $L$  é uma coleção de páginas contendo a palavra  $w$ . As chaves (palavras) no dicionário são chamadas de *termos de índice*, e devem ser um conjunto de itens de vocabulário e nomes próprios tão grande quanto possível. Os elementos neste dicionário são chamados de *listas de ocorrências*, e devem cobrir tantas páginas da Web quanto possível.

Pode-se implementar eficientemente um índice invertido com uma estrutura de dados consistindo de:

1. um arranjo armazenando as listas de ocorrências dos termos (sem ordenação);
2. um trie comprimido para o conjunto de termos de índice, no qual cada nodo externo armazene o índice da lista de ocorrência do termo associado.

A razão para armazenar as listas de ocorrências fora do trie é manter seu tamanho pequeno o bastante para caber na memória interna. Em troca, por causa de seu grande tamanho total, as listas de ocorrências são armazenadas em disco.

Com esta estrutura de dados, uma consulta por uma única palavra-chave é similar a uma procura por uma palavra em um texto (ver Seção 12.3.1). Ou seja, se encontra a palavra-chave no trie e se retorna a lista de ocorrências associada.

Quando múltiplas palavras-chave são dadas, e a saída desejada é composta pelas páginas contendo *todas* as palavras, recupera-se a lista de ocorrência de cada palavra-chave usando o trie e retorna-se sua interseção. Para facilitar o cálculo da interseção, cada lista de ocorrências poderia ser implementada como uma seqüência ordenada por endereço ou com um dicionário (ver, por exemplo, a junção discutida na Seção 11.6).

Além da tarefa básica de retornar a lista de páginas contendo as palavras-chave dadas, os mecanismos de busca fornecem um importante serviço adicional *classificando* por relevância as páginas retornadas. Projetar algoritmos de classificação rápidos e precisos para mecanismos de busca é um desafio para pesquisadores em computação e para empresas de comércio eletrônico.

## 12.4 Compressão de textos

Nesta seção, analisa-se outra aplicação do processamento de textos, a *compressão de textos*. Neste problema, tem-se uma cadeia de caracteres  $X$  definida sobre um dado alfabeto como ASCII ou Unicode, e deseja-se codificar  $X$  eficientemente em uma pequena cadeia binária  $Y$  (usando apenas os caracteres 0 e 1). A compressão de textos é útil em qualquer situação em que a comunicação é feita através de um canal de baixa capacidade de transmissão, tais como um modem ou conexão infravermelha, e deseja-se minimizar o tempo necessário para transmitir o texto. De forma similar, a compressão de texto é útil para armazenar coleções de documentos mais eficientemente, permitindo que um meio de armazenamento de capacidade fixa comporte tantos documentos quanto possível.

Hidden page

### 12.4.1 O algoritmo de codificação de Huffman

O algoritmo de codificação de Huffman começa com cada um dos  $d$  caracteres diferentes da cadeia  $X$  a codificar como raízes de árvores binárias de um único nodo. O algoritmo prossegue em uma série de rodadas: em cada rodada, o algoritmo unifica as duas árvores binárias com menor freqüência em uma única árvore binária. Esta operação é repetida até que apenas uma árvore exista. (Ver Trecho de código 12.8.)

Cada iteração do laço **enquanto** no algoritmo de Huffman pode ser implementada em tempo  $O(\log d)$  usando-se uma fila de prioridades  $Q$ , representada com um heap. Além disso, cada iteração retira dois elementos de  $Q$  e adiciona um, em um processo que se repete  $d - 1$  vezes antes de que exatamente um nodo esteja em  $Q$ . Assim, este algoritmo é executado em tempo  $O(n + d \log d)$ . Embora uma justificativa completa da correção desse algoritmo esteja além do escopo deste livro, mostra-se que ele é intuitivo por meio de uma idéia simples: qualquer codificação ótima pode ser convertida em uma codificação ótima em que os códigos para os dois caracteres de menor freqüência,  $a$  e  $b$ , difiram apenas em seu último bit. Repetir o argumento para uma cadeia com  $a$  e  $b$  substituídos por um caractere  $c$  fornece a proposição a seguir:

**Proposição 12.10** *O algoritmo de Huffman constrói uma codificação por prefixos ótima para uma cadeia de caracteres de comprimento  $n$  com  $d$  caracteres distintos em tempo  $O(n + d \log d)$ .*

**Algoritmo** Huffman( $X$ ):

**Entrada:** a cadeia de caracteres  $X$  de comprimento  $n$  com  $d$  caracteres distintos

**Saída:** uma árvore com a codificação de  $X$

Calcule a freqüência  $f(c)$  de todo caractere  $c$  em  $X$ .

Initialize uma fila de prioridade  $Q$ .

**para cada** caractere  $c$  em  $X$  **faça**

Crie uma árvore binária  $T$  de um único nodo armazenando  $c$ .

Insira  $T$  em  $Q$  com chave  $f(c)$ .

**enquanto**  $Q.size() > 1$  **faça**

$f_1 \leftarrow Q.min().key()$

$T_1 \leftarrow Q.removeMin()$

$f_2 \leftarrow Q.min().key()$

$T_2 \leftarrow Q.removeMin()$

Crie uma nova árvore binária  $T$  com subárvore esquerda  $T_1$  e subárvore  $T_2$  direita.

Insira  $T$  em  $Q$  com chave  $f_1 + f_2$ .

**retorna**  $Q.removeMin()$

**Trecho de código 12.8** Algoritmo de codificação de Huffman.

### 12.4.2 O método guloso

O algoritmo de Huffman para construção de uma codificação ótima é um exemplo de aplicação de um padrão de projeto chamado de **método guloso**. Este padrão de projeto é aplicado em problemas de otimização em que se tenta construir alguma estrutura enquanto se minimiza ou maximiza alguma propriedade da estrutura.

A forma geral para o método guloso é quase tão simples quanto para o método da força bruta. Para resolver um dado problema de otimização usando o método guloso, é feita uma sequência de escolhas. A sequência inicia com alguma condição inicial bem conhecida e calcula o custo para esta condição inicial. O padrão então exige que iterativamente sejam feitas escolhas adicionais identificando a decisão que atinge o melhor custo dentre todas as escolhas que são possíveis no momento. Esta abordagem nem sempre leva à solução ótima.

Existem vários problemas para os quais esta abordagem funciona, e tais problemas são conhecidos por terem a propriedade da **escolha gulosa**. Esta é a propriedade de que uma solução global ótima pode ser atingida através de uma seqüência de soluções locais ótimas (ou seja, escolhas que são as melhores dentre as disponíveis a cada momento), começando de uma condição inicial bem definida. O problema de calcular uma codificação de prefixos ótima de comprimento variável é apenas um exemplo de um problema que possui a propriedade da escolha gulosa.

## 12.5 Testando a similaridade de textos

Um problema comum em processamento de texto, que surge em genética e em engenharia de software, é testar a similaridade entre duas cadeias de caracteres. Em uma aplicação em genética, as duas cadeias de caracteres poderiam corresponder a duas seqüências de DNA, que poderiam, por exemplo, vir de dois indivíduos considerados geneticamente relacionados se eles tiverem uma longa subseqüência comum em suas seqüências de DNA. Em uma aplicação de engenharia de software, as duas cadeias de caracteres poderiam vir de duas versões do mesmo programa, e se pode desejar determinar quais mudanças foram feitas de uma versão para a outra. De fato, determinar a similaridade entre duas cadeias de caracteres é considerada uma operação tão comum que os sistemas operacionais Unix e Linux são fornecidos com um programa, chamado `diff`, para comparar arquivos de texto.

### 12.5.1 O problema da maior subseqüência comum

Existem várias formas diferentes de definir a similaridade entre duas cadeias de caracteres. Mesmo assim, pode-se abstrair uma versão simples, mas comum, deste problema usando cadeias de caracteres e suas subseqüências. Dada uma cadeia  $X = x_0x_1x_2 \dots x_{n-1}$ , uma subseqüência de  $X$  é qualquer cadeia da forma  $x_i x_{i+1} \dots x_k$ , onde  $i_j < i_{j+1}$ ; ou seja, uma seqüência de caracteres que não são necessariamente contíguos, mas são, mesmo assim, retirados de  $X$  em ordem. Por exemplo, a cadeia AAAG é uma subseqüência da cadeia CGATAATTGAGA. Observa-se que o conceito de **subseqüência** de uma cadeia é diferente do conceito de uma **substring** de uma cadeia, definida na Seção 12.1.

#### Definição do problema

O problema específico de similaridade de textos abordado aqui é o **problema da maior subseqüência comum** (LCS\*). Neste problema, tem-se duas cadeias de caracteres,  $X = x_0x_1x_2 \dots x_{n-1}$  e  $Y = y_0y_1y_2 \dots y_{m-1}$  definidas sobre algum alfabeto (como o alfabeto {A, C, G, T}, comum em genética computacional) e deseja-se encontrar a cadeia mais longa  $S$  que é uma subseqüência de  $X$  e  $Y$ .

Uma maneira de resolver o problema da maior subseqüência comum é enumerar todas as subseqüências de  $X$  e escolher a mais longa que for também uma subseqüência de  $Y$ . Já que cada caractere de  $X$  está ou não na subseqüência, existem potencialmente  $2^n$  subseqüências diferentes de  $X$ , cada uma das quais requer tempo  $O(m)$  para determinar se é uma subseqüência de  $Y$ . Assim, esta abordagem de força bruta fornece um algoritmo exponencial executado em tempo  $O(2^n m)$ , que é muito ineficiente. Nesta seção, se discutirá como usar um padrão de projeto chamado **programação dinâmica** para resolver o problema da maior subseqüência comum muito mais rápido.

\* N. de T. Em inglês, *longest common subsequence problem*.

Hidden page

pode obter uma maior subsequência comum adicionando  $x_i$  ao final. Assim, uma maior subsequência comum de  $X[0..i]$  e  $Y[0..j]$  termina com  $x_i$ . Portanto, faz-se

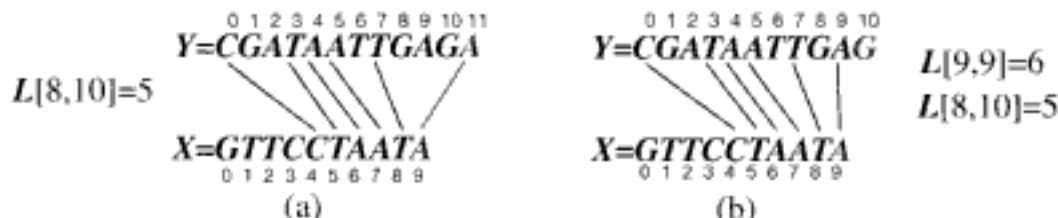
$$L[i, j] = L[i - 1, j - 1] + 1 \text{ se } x_i = y_j$$

- $x_i \neq y_j$ . Neste caso, não se pode ter uma subsequência comum que inclui  $x_i$  e  $y_j$ . Ou seja, pode-se ter uma subsequência comum terminando com  $x_i$  ou uma que termine com  $y_j$  (ou possivelmente nenhum dos dois), mas certamente não ambos. Portanto, faz-se

$$L[i, j] = \max \{L[i - 1, j], L[i, j - 1]\} \text{ se } x_i \neq y_j$$

Para que ambas as equações façam sentido nos casos limite em que  $i = 0$  ou  $j = 0$ , faz-se  $L[i - 1] = 0$  para  $i = -1, 0, 1, \dots, n - 1$  e  $L[-1, j] = 0$  para  $j = -1, 0, 1, \dots, m - 1$ .

A definição acima para  $L[i, j]$  satisfaz a otimização dos subproblemas, pois não é possível ter uma maior subsequência comum sem também ter uma maior subsequência comum para os subproblemas. Ela também usa a interseção de subproblemas, porque a solução de um subproblema  $L[i, j]$  pode ser usada em vários outros problemas (a saber, os problemas  $L[i + 1, j]$ ,  $L[i, j + 1]$  e  $L[i + 1, j + 1]$ ).



**Figura 12.12** Os dois casos no algoritmo de maior subsequência comum: (a)  $x_i = y_j$ ; (b)  $x_i \neq y_j$ . O algoritmo armazena apenas os valores de  $L[i, j]$ , não seus caracteres.

## O algoritmo LCS

Fazer desta definição de  $L[i, j]$  um algoritmo é realmente bastante simples. Inicializa-se um arranjo  $L$  de dimensões  $(n + 1) \times (m + 1)$  com os casos limite em que  $i = 0$  ou  $j = 0$ . Ou seja, inicializa-se  $L[i, -1] = 0$  para  $i = -1, 0, 1, \dots, n - 1$  e  $L[-1, j] = 0$  para  $j = -1, 0, 1, \dots, m - 1$ . (Este é um ligeiro abuso de notação, pois, na realidade, se deveria indexar as linhas e colunas de  $L$  começando em 0.) Então, os valores de  $L$  são construídos iterativamente até que se obtém  $L[n - 1, m - 1]$ , o comprimento da maior subsequência comum entre  $X$  e  $Y$ . Fornece-se uma descrição em pseudocódigo de como esta abordagem resulta em uma solução com programação dinâmica para o problema da maior subsequência comum (LCS) no Trecho de código 12.9.

### Algoritmo LCS( $X, Y$ ):

**Entrada:** as cadeias  $X$  e  $Y$  com  $n$  e  $m$  elementos respectivamente.

**Saída:** para  $i = 0, 1, \dots, n - 1$  e  $j = 0, 1, \dots, m - 1$ , o comprimento  $L[i, j]$  da cadeia mais longa que é uma subsequência tanto de  $X[0..i] = x_0x_1 \dots x_i$  quanto de  $Y[0..j] = y_0y_1 \dots y_j$ .

para  $i \leftarrow -1$  para  $n - 1$  faça

$L[i, -1] \leftarrow 0$

para  $j \leftarrow 0$  para  $m - 1$  faça

$L[-1, j] \leftarrow 0$

para  $i \leftarrow 0$  para  $n - 1$  faça

    para  $j \leftarrow 0$  para  $m - 1$  faça

        se  $x_i = y_j$  então

$L[i, j] \leftarrow L[i - 1, j - 1] + 1$

Hidden page

## 12.6 Exercícios

Para obter o código fonte e auxílio com os exercícios, visite [java.datastructures.net](http://java.datastructures.net)

---

### Reforço

- R-12.1 Quantos prefixos não-vazios da cadeia  $P = "aaabbbaaa"$  são também sufixos de  $P$ ?
- R-12.2 Desenhe uma figura ilustrando as comparações feitas pelo algoritmo de procura de padrões baseado em força bruta, para o caso em que o texto é "aaabaadaabaaa" e o padrão é "aabaaa".
- R-12.3 Repita o problema anterior para o algoritmo BM de procura de padrões, não contando as comparações feitas para calcular a função  $\text{last}(c)$ .
- R-12.4 Repita o problema anterior para o algoritmo KMP de procura de padrões, não contando as comparações feitas para calcular a função de falha.
- R-12.5 Calcule uma tabela representando a função  $\text{last}$  usada no algoritmo BM para o padrão

"the quick brown fox jumped over a lazy cat"

assumindo o seguinte alfabeto (que começa com um espaço em branco):

$$\Sigma = \{ \ ,a,b,c,d,e,f,g,h,i,j,k,l,m,n,o,p,q,r,s,t,u,v,w,x,y,z \}.$$

- R-12.6 Assumindo que os caracteres no alfabeto  $\Sigma$  podem ser enumerados e podem indexar arranjos, forneça um método de tempo  $O(m + |\Sigma|)$  para construir a função  $\text{last}$  a partir de um padrão  $P$  de comprimento  $m$ .
- R-12.7 Calcule uma tabela representando a função de falha KMP para o padrão "cgtacgttcgtac".
- R-12.8 Desenhe um trie-padrão para a seguinte seqüência de cadeias de caracteres:

{abab, baba, ccccc, bbaaaa, caa, bbaacc, cbcc, cbca}.

- R-12.9 Desenhe um trie comprimido para o conjunto de cadeias de caracteres do Exercício R-12.8.
- R-12.10 Desenhe a representação compacta para o trie de sufixos para a cadeia

"minimize minime".

- R-12.11 Qual o mais longo prefixo da cadeia "cgtacgttcgtacg" que também é um sufixo desta cadeia?
- R-12.12 Desenhe o arranjo das freqüências e a árvore de Huffman para a seguinte cadeia:

"dogs do not spot hot pots or cats".

- R-12.13 Mostre o arranjo  $L$  para a maior subseqüência comum para as duas cadeias

$$\begin{aligned} X &= "skullandbones" \\ Y &= "lullabybabies". \end{aligned}$$

Qual é uma maior subseqüência comum entre essas cadeias de caracteres?

---

## Criatividade

- C-12.1 Dê um exemplo de um texto  $T$  de comprimento  $n$  e um padrão  $P$  de comprimento  $m$  que force o algoritmo de procura de padrões por força bruta a um tempo de execução  $\Omega(mn)$ .
- C-12.2 Justifique por que o método KMPFailureFunction (Trecho de código 11.5) precisa de tempo  $O(m)$  em um padrão de comprimento  $m$ .
- C-12.3 Mostre como modificar o algoritmo de procura de padrões KMP de forma que ele ache *todas* as ocorrências de um padrão  $P$  que aparece como substring em  $T$ , ainda sendo executado em tempo  $O(m + n)$ . (Assegure-se de encontrar até as ocorrências que se sobreponem.)
- C-12.4 Seja  $T$  um texto de comprimento  $n$  e  $P$  um padrão de comprimento  $m$ . Descreva um método de tempo  $O(m + n)$  para encontrar o prefixo mais longo de  $P$  que é uma substring de  $T$ .
- C-12.5 Diz-se que um padrão  $P$  de comprimento  $m$  é uma substring *circular* de um texto  $T$  de comprimento  $n$  se existir um índice  $0 \leq i < m$  tal que  $P = T[n - m + i..n - 1] + T[0..i - 1]$ , ou seja, se  $P$  é uma substring (normal) de  $T$  ou se  $P$  é igual à concatenação de um sufixo de  $T$  e um prefixo de  $T$ . Forneça um algoritmo de tempo  $O(m + n)$  para determinar se  $P$  é uma substring circular de  $T$ .
- C-12.6 O algoritmo de procura de padrões KMP pode ser modificado para maior velocidade em cadeias de caracteres binárias redefinindo-se a função de falha como

$$f(j) = \text{o maior } k < j \text{ tal que } P[0..k - 1] \hat{p}_k \text{ é sufixo de } P[1..j],$$

onde  $\hat{p}_k$  denota o complemento do  $k$ -ésimo bit de  $P$ . Descreva como implementar o algoritmo KMP para tirar vantagem desta nova função de falha e forneça um método para avaliar esta nova função de falha. Mostre que este método faz no máximo  $n$  comparações entre o texto e o padrão (contra  $2n$  comparações do algoritmo KMP padrão, dado na Seção 12.2.3).

- C-12.7 Modifique o algoritmo simplificado BM apresentado neste capítulo usando idéias do algoritmo KMP, de forma que ele seja executado em tempo  $O(m + n)$ .
- C-12.8 Dada uma string  $X$  de tamanho  $n$  e uma string  $Y$  de tamanho  $m$ , descreva um algoritmo que execute no tempo  $O(n + m)$  para procurar o mais longo prefixo de  $X$  que é um sufixo de  $Y$ .
- C-12.9 Forneça um algoritmo eficiente para deletar uma cadeia de caracteres de um trie-padrão e analise seu tempo de execução.
- C-12.10 Forneça um algoritmo eficiente para deletar uma cadeia de caracteres de um trie comprimido e analise seu tempo de execução.
- C-12.11 Descreva um algoritmo para construir a representação compacta de um trie de sufixos e analise seu tempo de execução.
- C-12.12 Seja  $T$  uma cadeia de caracteres de comprimento  $n$ . Descreva um método de tempo  $O(n)$  para encontrar o mais longo prefixo de  $T$  que seja uma substring do reverso de  $T$ .

- C-12.13 Descreva um algoritmo eficiente para encontrar o mais longo palíndromo que seja um sufixo de uma cadeia de caracteres  $T$  de comprimento  $n$ . Lembre que um *palíndromo* é uma cadeia de caracteres que é igual a seu reverso. Qual o tempo de execução de seu método?
- C-12.14 Dada uma seqüência  $S = (x_0, x_1, x_2, \dots, x_{n-1})$  de números, descreva um algoritmo de tempo  $O(n^2)$  para achar a mais longa subseqüência  $T = (x_{i_0}, x_{i_1}, x_{i_2}, \dots, x_{i_{k-1}})$  de números, em que  $i_j < i_{j+1}$  e  $x_{i_j} > x_{i_{j+1}}$ . Ou seja,  $T$  é a mais longa subseqüência descendente de  $S$ .
- C-12.15 Defina a *distância de edição* entre duas cadeias de caracteres  $X$  e  $Y$  de comprimentos  $n$  e  $m$ , respectivamente, como sendo o menor número de alterações para transformar  $X$  em  $Y$ . Uma alteração consiste na inserção de caracteres, na deleção de caracteres ou na substituição de caracteres. Por exemplo, as cadeias de caracteres "algorithm" e "rhythm" têm distância de edição 6. Projete um algoritmo de tempo  $O(mn)$  para calcular a distância de edição entre  $X$  e  $Y$ .
- C-12.16 Projete um algoritmo guloso para fazer troco para alguém que compra uma bala que custa  $x$  centavos e entrega ao balonista \$1. Seu algoritmo deve minimizar o número de moedas do troco.
- Mostre que seu algoritmo guloso retorna o número mínimo de moedas se as moedas tiverem os valores de \$0.25, \$0.10, \$0.05 e \$0.01.
  - Forneça um conjunto de moedas para o qual seu algoritmo pode não retornar o menor número de moedas. Inclua um exemplo em que seu algoritmo falha.
- C-12.17 Apresente um algoritmo eficiente para determinar se um padrão  $P$  é uma subseqüência (não substring) de um texto  $T$ . Qual é o tempo de execução do seu algoritmo?
- C-12.18 Seja  $x$  e  $y$  strings de tamanho  $n$  e  $m$  respectivamente. Defina  $B(i, j)$  para ser o tamanho do mais longo substring comum ao sufixo de tamanho  $i$  em  $x$  e do sufixo de tamanho  $j$  em  $y$ . Projete um algoritmo que execute no tempo  $O(mn)$  para computar todos os valores de  $B(i, j)$  para  $i = 1, \dots, n$  e  $j = 1, \dots, m$ .
- C-12.19 Ana acaba de vencer um concurso que permite que ela pegue  $n$  balas de graça em uma loja. Ela tem idade suficiente para saber que algumas são caras, custando alguns dólares e outras são baratas e custam centavos. Os vidros de balas são numerados 0, 1, ...,  $m - 1$  e o vidro  $j$  tem  $n_j$  balas com um preço  $c_j$  por bala. Forneça um algoritmo de tempo  $O(m + n)$  para que Ana maximize o valor das balas que irá retirar. Mostre que seu algoritmo produz o maior valor possível.
- C-12.20 Sejam três arranjos de números inteiros  $A$ ,  $B$  e  $C$ , cada um de comprimento  $n$ . Dado um inteiro arbitrário  $x$ , apresente um algoritmo de tempo  $O(n^2 \log n)$  que determina se existem números  $a \in A$ ,  $b \in B$  e  $c \in C$  tais que  $x = a + b + c$ .
- C-12.21 Forneça um algoritmo de tempo  $O(n^2)$  para o problema anterior.

Hidden page

---

## Observações sobre o capítulo

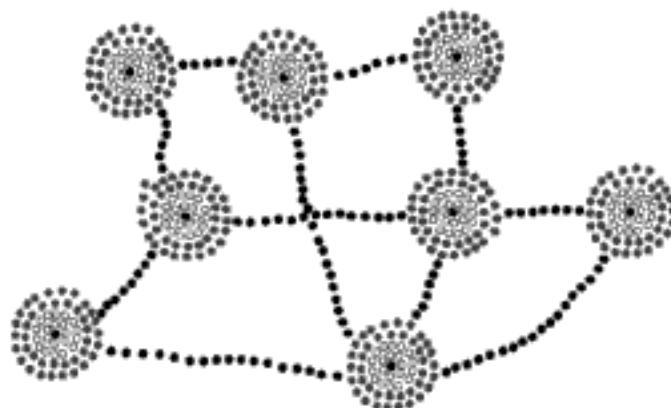
O algoritmo KMP é descrito por Knuth, Morris e Pratt em seu artigo [64], e Boyer e Moore descrevem seu algoritmo em um artigo publicado no mesmo ano [15]. Em seu artigo, no entanto, Knuth *et al.* [64] também provam que o algoritmo BM tem tempo linear. Mais recentemente, Cole [23] mostra que o algoritmo BM realiza no máximo  $3n$  comparações no pior caso e este limite é exato. Todos os algoritmos discutidos acima são discutidos também no livro de Aho [3], embora com uma ênfase para a teoria, incluindo os métodos para procura de expressões regulares. O leitor interessado em outros estudos em procura de padrões em cadeias de caracteres pode usar os livros de Stephen [87] e os capítulos de Aho [3] e Crochemore e Lecroq [27].

O trie foi inventado por Morrison [78], e é discutido intensivamente no livro clássico *Sorting and Searching* de Knuth [63]. O nome “Patricia” abrevia “Practical Algorithm to Retrieve Information Coded in Alphanumeric” [78]. McCreight [70] mostra como construir tries de sufixos em tempo linear. Uma introdução ao campo da recuperação de informação que inclui uma discussão de mecanismos de busca para a Web é dada no livro de Baeza-Yates e Ribeiro-Neto [8].

# Capítulo —

# 13

# Grafos



## Conteúdo

<b>13.1</b>	<b>O tipo abstrato de dados grafo.....</b>	<b>508</b>
13.1.1	O TAD grafo.....	512
<b>13.2</b>	<b>Estruturas de dados para grafos .....</b>	<b>512</b>
13.2.1	A lista de arestas .....	512
13.2.2	A lista de adjacências .....	515
13.2.3	A matriz de adjacência .....	516
<b>13.3</b>	<b>Caminhamento em grafos.....</b>	<b>518</b>
13.3.1	Pesquisa em profundidade .....	518
13.3.2	Implementando a pesquisa em profundidade .....	522
13.3.3	Caminhamento em largura .....	528
<b>13.4</b>	<b>Grafos dirigidos .....</b>	<b>531</b>
13.4.1	Caminhamento em um digrafo .....	532
13.4.2	Fechamento transitivo.....	535
13.4.3	Grafos acíclicos dirigidos .....	536
<b>13.5</b>	<b>Grafos ponderados .....</b>	<b>539</b>
<b>13.6</b>	<b>Caminhos mínimos .....</b>	<b>541</b>
13.6.1	O algoritmo de Dijkstra .....	542
<b>13.7</b>	<b>Árvores de cobertura mínima .....</b>	<b>549</b>
13.7.1	Algoritmo de Kruskal .....	551
13.7.2	O algoritmo Prim-Jarník .....	554
<b>13.8</b>	<b>Exercícios .....</b>	<b>556</b>

### 13.1 O tipo abstrato de dados grafo

Um **grafo** é uma forma de representar relacionamentos que existem entre pares de objetos. Isto é, um conjunto de objetos, chamados de **vértices**, juntamente com uma coleção de conexões entre pares de vértices. A propósito, esta noção de “grafo” não deve ser confundida com o diagrama de barras e funções plots, como estes tipos de “grafos” não são relacionados ao tópico deste capítulo. Grafos têm aplicações em vários domínios diferentes, incluindo mapeamento, transporte, engenharia elétrica e redes de computador.

Visto de forma abstrata, um **grafo**  $G$  é simplesmente um conjunto  $V$  de **vértices** e uma coleção  $E$  de pares de vértices de  $V$ , chamados de **arestas**. Assim, um grafo é uma forma de representar conexões ou relações entre pares de objetos de algum conjunto  $V$ . Alguns livros usam uma terminologia diferente para grafos e referem-se ao que se chama de vértices, como **nodos**; e ao que se chama de arestas, como **arcos**. Serão utilizados aqui os termos “vértices” e “arestas”.

As arestas em um grafo podem ser **dirigidas** ou **não-dirigidas**. Uma aresta  $(u,v)$  é dita **dirigida** de  $u$  para  $v$  se o par  $(u,v)$  for ordenado, com  $u$  precedendo  $v$ . Uma aresta  $(u,v)$  é dita **não-dirigida** se o par  $(u,v)$  não for ordenado. As arestas não-dirigidas são por vezes denotadas como conjuntos  $\{u,v\}$ , mas, para simplificar, se utilizará a notação de pares ordenados  $(u,v)$ , notando que no caso não-dirigido  $(u,v)$  é o mesmo que  $(v,u)$ . Os grafos são visualizados tipicamente desenhando-se os vértices como ovaís ou retângulos e as arestas como segmentos ou curvas conectando pares de ovaís ou retângulos. A seguir, são apresentados alguns exemplos de grafos dirigidos e não-dirigidos.

**Exemplo 13.1** Pode-se visualizar colaborações entre pesquisadores de certa área construindo um grafo cujos vértices são associados com os pesquisadores e cujas arestas conectam pares de vértices associados com os pesquisadores que escreveram juntos um artigo ou livro. (Ver Figura 13.1.) Tais arestas são não-dirigidas porque a co-autoria é uma relação simétrica, ou seja, se  $A$  é co-autor de  $B$ , então necessariamente  $B$  é co-autor de  $A$ .

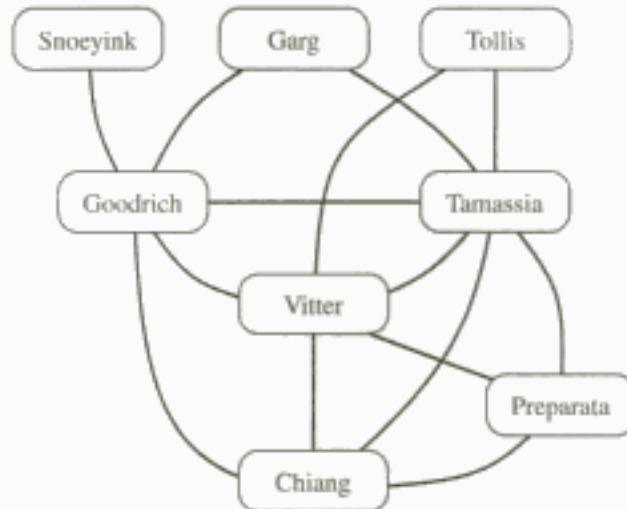


Figura 13.1 Grafo de co-autoria de alguns autores.

**Exemplo 13.2** Pode-se associar a um programa orientado a objetos um grafo cujos vértices representam as classes definidas no programa, e cujas arestas indicam a herança entre as classes. Existe uma aresta de um vértice  $v$  a um vértice  $u$  se a classe para  $v$  estender a classe de  $u$ . Tais arestas são dirigidas porque a relação de herança só existe em uma direção (ou seja, ela é assimétrica).

Se todas as arestas em um grafo forem não-dirigidas, então diz-se que o grafo é um **grafo não-dirigido**. De forma similar, um **grafo dirigido**, ou **dígrafo**, é um grafo em que todas as ares-

tas são dirigidas. Um grafo que tem arestas dirigidas e não-dirigidas é chamado de *grafo misto*. Observe que um grafo não-dirigido ou misto pode ser transformado em um grafo dirigido substituindo-se cada aresta não-dirigida  $(u,v)$  por um par de arestas dirigidas  $(u,v)$  e  $(v,u)$ . No entanto, é frequentemente útil manter grafos não-dirigidos ou mistos em sua forma original, pois estes grafos têm várias aplicações, como a do próximo exemplo.

**Exemplo 13.3** *Um mapa de cidade pode ser modelado como um grafo cujos vértices são cruzamentos ou finais de ruas, e cujas arestas podem ser trechos de ruas sem cruzamentos. Este grafo tem arestas não-dirigidas, representando ruas de dois sentidos, e arestas dirigidas, correspondendo a trechos de um único sentido. Assim, um grafo representando as ruas de uma cidade é um grafo misto.*

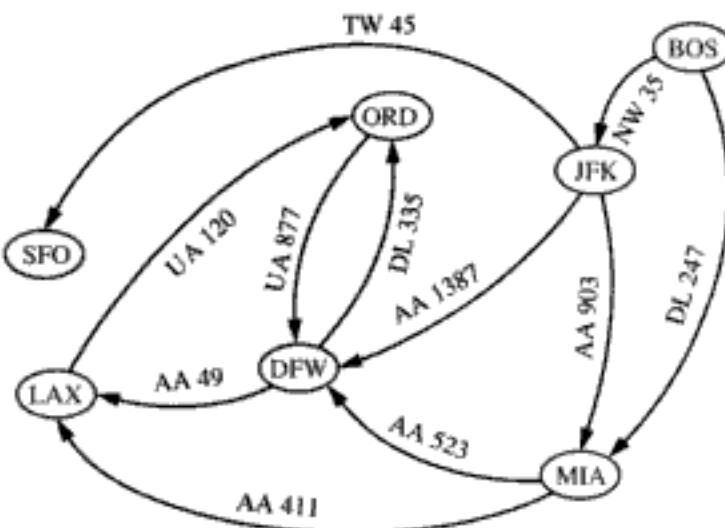
**Exemplo 13.4** *Exemplos físicos de grafos estão presentes nas redes elétricas e de encanamento de um prédio. Tais redes podem ser modeladas como grafos, nos quais cada conector, junção ou saída são vistos como vértices, e cada trecho não-interrompido de fiação ou cano é visto como uma aresta. Tais grafos são, na verdade, componentes de grafos muito maiores, as redes locais de distribuição de energia e de água. Dependendo dos aspectos específicos dos grafos em que estivermos interessados, pode-se considerar suas arestas como dirigidas ou não-dirigidas, pois, em princípio, a água pode fluir em um cano nas duas direções, assim como a corrente em um fio.*

Os dois vértices conectados por uma aresta são chamados de *vértices finais* (ou *pontos finais*) da aresta. Se uma aresta é dirigida, seu primeiro ponto final é sua *origem*, e o outro é seu *destino*. Dois vértices são ditos *adjacentes* se eles forem pontos finais da mesma aresta. Uma aresta é dita *incidente* a um vértice se o vértice for um dos pontos finais da aresta. As *arestas incidentes* de um vértice são as arestas dirigidas cuja origem é aquele vértice. As *arestas incidentes* em um vértice são as arestas dirigidas cujo destino é aquele vértice. O *grau* de um vértice  $v$ , denotado  $\deg(v)$ , é o número de vértices incidentes a  $v$ . O *grau de entrada* e o *grau de saída* de um vértice  $v$  são os números de arestas incidentes em  $v$  e de  $v$ , respectivamente, e são denotados  $\text{indeg}(v)$  e  $\text{outdeg}(v)$ .

**Exemplo 13.5** *Pode-se estudar o transporte aéreo construindo um grafo  $G$  chamado de rede de vôos, cujos vértices são associados a aeroportos e cujas arestas são associadas com vôos. (Ver Figura 13.2.) No grafo  $G$ , as arestas são dirigidas porque um dado vôo tem uma direção específica (do aeroporto de origem ao aeroporto de destino). Os pontos finais de uma aresta e em  $G$  correspondem respectivamente à origem e ao destino do vôo correspondente a  $e$ . Dois aeroportos são adjacentes em  $G$  se existir um vôo entre eles, e uma aresta  $e$  será incidente a um vértice  $v$  de  $G$  se o vôo representado por  $e$  sair do aeroporto ou chegar ao aeroporto representado por  $v$ . As arestas incidentes de um vértice  $v$  correspondem aos vôos que saem do aeroporto de  $v$ , enquanto as arestas incidentes em  $v$  correspondem aos vôos que chegam. Finalmente, o grau de entrada de um vértice  $v$  de  $G$  corresponde ao número de vôos que chegam ao aeroporto de  $v$ , enquanto o grau de saída representa o número de vôos que saem.*

A definição de grafo refere-se ao grupo de arestas como uma *coleção*, não como um *conjunto*, permitindo que duas arestas não-dirigidas tenham os mesmos pontos finais, e que duas arestas dirigidas tenham a mesma origem e mesmo destino. Tais arestas são chamadas de *arestas paralelas* ou *arestas múltiplas*. As arestas paralelas podem estar em uma rede de vôo (Exemplo 13.5) e, neste caso, múltiplas arestas entre o mesmo par de vértices indicam, vôos diferentes operando na mesma rota a diferentes horas do dia. Outro tipo especial de aresta é o que conecta um vértice consigo mesmo. Assim, diz-se que uma aresta (dirigida ou não-dirigida) forma um *laço* se seus pontos finais coincidirem. Um laço pode ocorrer em um grafo associado com um mapa urbano (Exemplo 13.3), onde corresponderia a um “círculo” (uma rua circular que retorna a seu ponto de início).

Com poucas exceções, como as mencionadas acima, os grafos não têm arestas paralelas ou laços, e são ditos *simples*. Assim, pode-se geralmente dizer que as arestas de um grafo simples são um *conjunto* de pares de vértices (e não uma coleção). Neste capítulo, se assumirá que um grafo é simples a não ser que seja especificado de outra forma.



**Figura 13.2** Exemplo de um grafo dirigido representando uma rede de vôos. Os pontos finais da aresta UA120 são LAX e ORD, portanto, LAX e ORD são adjacentes. O grau de entrada de DFW é 3, e o grau de saída de DFW é 2.

Nas proposições a seguir, exploram-se algumas propriedades importantes dos grafos.

**Proposição 13.6** Se  $G$  for um grafo com  $m$  arestas, então

$$\sum_{v \in G} \deg(v) = 2m.$$

**Justificativa** Uma aresta  $(u,v)$  é contada duas vezes na soma acima: uma por seu ponto final  $u$ , e outra por seu ponto final  $v$ . Assim, a contribuição total das arestas para os graus dos vértices é de duas vezes o número de arestas. ■

**Proposição 13.7** Se  $G$  for um grafo dirigido com  $m$  arestas, então

$$\sum_{v \in G} \text{indeg}(v) = \sum_{v \in G} \text{outdeg}(v) = m.$$

**Justificativa** Em um grafo dirigido, uma aresta  $(u,v)$  contribui com uma unidade para o grau de saída de sua origem  $u$ , e uma unidade para o grau de entrada de seu destino  $v$ . Assim, a contribuição total das arestas para os graus de saída dos vértices é igual ao número de arestas e similarmente para os graus de entrada. ■

A seguir, será mostrado que um simples grafo com  $n$  vértices tem  $O(n^2)$  arestas.

**Proposição 13.8** Seja  $G$  um grafo simples com  $n$  vértices e  $m$  arestas. Se  $G$  for não-dirigido, então  $m \leq n(n - 1)/2$ , e se  $G$  for dirigido, então  $m \leq n(n - 1)$ .

**Justificativa** Suponha que  $G$  seja não-dirigido. Como duas arestas não podem ter os mesmos pontos de saída e de chegada, e não há laços, o grau máximo de um vértice em  $G$  é  $n - 1$  neste caso. Assim, pela Proposição 13.6, tem-se  $2m \leq n(n - 1)$ . Agora, suponha que  $G$  seja dirigido. Como duas arestas não podem ter os mesmos pontos de saída e de chegada, e não há laços, o grau máximo de entrada de um vértice em  $G$  é  $n - 1$  neste caso. Assim, pela Proposição 13.7,  $m \leq n(n - 1)$ . ■

Um **caminho** em um grafo é uma sequência alternada de vértices e arestas que se inicia em um vértice e termina em um vértice, de tal forma que cada aresta seja incidente de seu antecessor e incidente em seu sucessor. Um **ciclo** é um caminho em que os vértices de início e fim são os mesmos. Diz-se que um caminho é **simples** se cada vértice no caminho for distinto, e diz-se que um ciclo é **simples** se cada vértice no ciclo for distinto, exceto pelo primeiro e o último. Um **caminho dirigido** é um caminho em que todas as arestas são dirigidas e percorridas em sua direção. Um **ciclo dirigido** é definido de forma similar. Por exemplo, a rede de vôos da Figura 13.2 (BOS, NW35, JFK, AA 1387, DFW) é um caminho dirigido simples e (LAX, UA 120, ORD, UA877, DFW, AA 49, LAX) é um ciclo dirigido simples. Se um caminho  $P$  ou ciclo  $C$  é um simples grafo, pode-se omitir as arestas em  $P$  ou  $C$ , como estas são bem definidas; neste caso,  $P$  é uma lista de vértices adjacentes e  $C$  é um ciclo de vértices adjacentes.

**Exemplo 13.9** Dado um grafo  $G$  representando o mapa de uma cidade (ver Exemplo 13.3), pode-se modelar um casal dirigindo de casa até um restaurante como um caminho em  $G$ . Se eles souberem o caminho e não passarem accidentalmente pelo mesmo cruzamento duas vezes, então eles passam por um caminho simples em  $G$ . Pode-se modelar o caminho completo do casal, de casa ao restaurante e de volta, como um ciclo. Se eles voltam para casa por uma rota completamente diferente da usada para chegar ao restaurante, sem nem passar em um mesmo cruzamento, então toda a viagem de ida e volta é um ciclo simples. Finalmente, se eles só passarem por ruas de mão única, então pode-se modelar sua saída como um ciclo dirigido.

Um **subgrafo** de um grafo  $G$  é um grafo  $H$  cujos vértices e arestas são respectivamente subconjuntos dos vértices e arestas de  $G$ . Por exemplo, na rede de vôos da Figura 13.2, os vértices BOS, JFK e MIA e as arestas AA 903 e DL 247 formam um subgrafo. Um **subgrafo de cobertura** de  $G$  é um subgrafo de  $G$  que contém todos os vértices de  $G$ . Um grafo é **conexo** se, para quaisquer dois vértices, existir um caminho entre eles. Se um grafo  $G$  não for conexo, seus subgrafos conexos maximais são chamados de **componentes conexos** de  $G$ . Uma **floresta** é um grafo sem ciclos. Uma **árvore** é uma floresta conexa, ou seja, um grafo conexo sem ciclos. Observe que esta definição de uma árvore é um pouco diferente da definição fornecida no Capítulo 7. Ou seja, no contexto dos grafos, uma árvore não tem raiz. Sempre que houver ambigüidade, as árvores do Capítulo 7 serão chamadas de **árvores com raiz**, enquanto as árvores deste capítulo serão chamadas de **árvores livres**. Os componentes conexos de uma floresta são árvores (livres). Uma **árvore de cobertura** de um grafo é um subgrafo de cobertura que é uma árvore (livre).

**Exemplo 13.10** Talvez o grafo mais popular atualmente seja a Internet, que pode ser vista como um grafo cujos vértices são computadores e cujas arestas (não-dirigidas) são conexões de comunicação entre pares de computadores na Internet. Os computadores e as conexões em um único domínio como [wiley.com](http://wiley.com) formam um subgrafo da Internet. Se este subgrafo for conexo, então dois usuários em computadores deste domínio podem mandar mensagens um ao outro sem que os pacotes de informação deixem o domínio. Supondo que as arestas deste subgrafo formem uma árvore de cobertura. Isso implica que se uma única conexão for desfeita (por exemplo, porque alguém desliga um dos cabos de rede ou encosta-se a ele e ele sai do lugar) então o subgrafo não será mais conexo.

Existe uma série de propriedades simples de árvores, florestas e grafos conexos. Serão exploradas algumas delas na proposição a seguir.

**Proposição 13.11** Seja  $G$  um grafo não-dirigido com  $n$  vértices e  $m$  arestas. Então tem-se

- Se  $G$  for conexo, então  $m \geq n - 1$ .
- Se  $G$  for uma árvore, então  $m = n - 1$ .
- Se  $G$  for uma floresta, então  $m \leq n - 1$ .

A justificativa desta proposição é deixada como um exercício (C-13.2).

Hidden page

em um contêiner  $V$ , que pode ser tipicamente um arranjo ou lista de nodos. Representando-se  $V$  como um arranjo, por exemplo, então pensa-se naturalmente nos vértices como sendo numerados.

### Objetos vértices

O objeto vértice para um vértice  $v$  armazenando o elemento  $o$  tem variáveis instanciadas para

- uma referência para  $o$ ;
- uma referência para a posição (ou localizador) do objeto vértice na coleção  $V$ .

A principal característica da lista de arestas não é a maneira como ela representa os vértices, mas como ela representa as arestas. Nesta estrutura, uma aresta  $e$  de  $G$ , armazenando um elemento  $o$ , é explicitamente representado por um objeto aresta. Os objetos arestas são armazenados em uma coleção  $E$ , que seria tipicamente um arranjo ou lista de nodos.

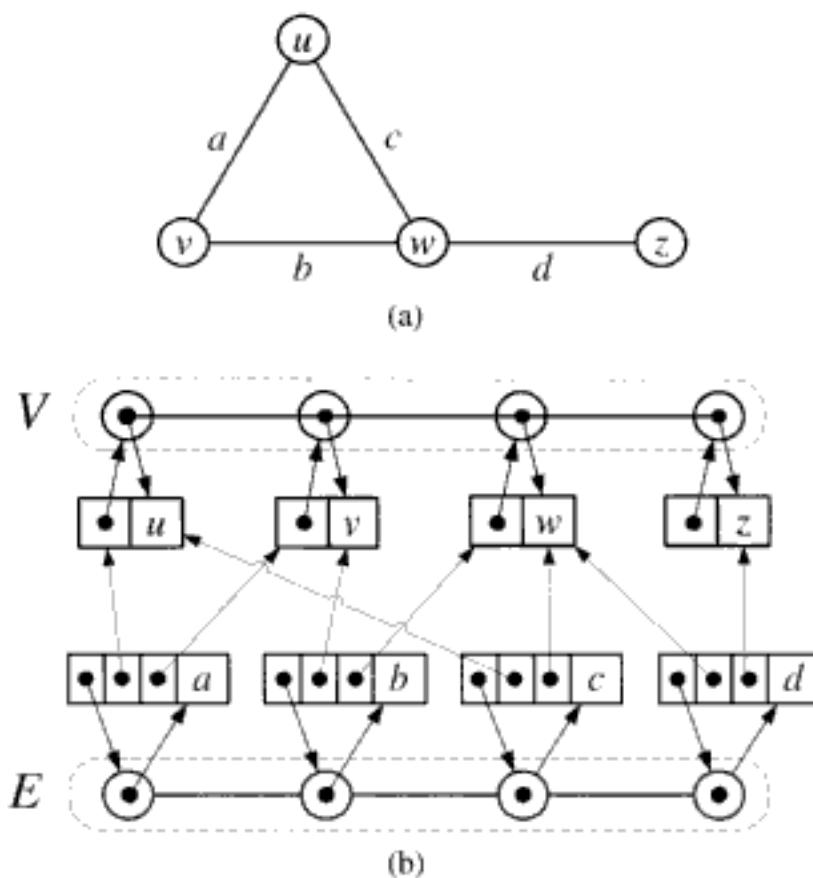
### Objetos arestas

O objeto aresta para uma aresta  $e$  armazenando o objeto  $o$  tem variáveis instanciadas para

- uma referência para  $o$ ;
- referências para os objetos vértice em  $V$  associados com os pontos finais de  $e$ ;
- uma referência para a posição (ou localizador) do objeto aresta na coleção  $E$ .

### Visualização da lista de arestas

Um exemplo de uma lista de arestas para um grafo  $G$  é ilustrado na Figura 13.3.



**Figura 13.3** (a) Um grafo  $G$ ; (b) representação esquemática da lista de arestas para  $G$ . Visualizam-se os elementos armazenados nos objetos vértice e aresta com os nomes dos elementos, em vez de referências reais aos objetos dos elementos.

A razão pela qual esta estrutura é chamada de *lista de arestas* é porque a implementação mais simples e comum da coleção  $E$  é uma lista. Mesmo assim, para poder procurar convenientemente por objetos específicos associados a uma aresta, pode-se desejar implementar  $E$  com um dicionário, apesar de continuar chamando a estrutura de “lista de arestas”. Também pode-se desejar implementar o contêiner  $V$  como um dicionário pela mesma razão. Ainda assim, mantendo a tradição, chama-se a estrutura de “lista de arestas”.

A característica principal da lista de arestas é que ela provê acesso direto das arestas aos vértices nos quais elas são incidentes. Isso permite definir algoritmos simples para os métodos `endVertices(e)` e `opposite(v,e)`.

### Desempenho da lista de arestas

Um método ineficiente para a lista de arestas é o que acessa as arestas incidentes a um vértice. Determinar este conjunto de vértices requer uma inspeção exaustiva de todos os objetos arestas da coleção  $E$ . Isto é, para determinar quais arestas são incidentes a um vértice  $v$ , deve-se examinar todas as arestas na lista de arestas e verificar se cada uma é incidente a  $v$ . Assim, o método `incidentEdges(v)` executa no tempo proporcional ao número de arestas do grafo, e não no tempo proporcional ao grau do vértice  $v$ . Na realidade, até para verificar se os dois vértices  $v$  e  $w$  são adjacentes usando o método `areAdjacent(v,w)`, requer uma pesquisa em todas as arestas da coleção procurando por uma aresta com vértices finais  $v$  e  $w$ . Além disso, assim como remoção de um vértice envolve a remoção de todas as suas arestas incidentes, o método `removeVertex` também requer uma pesquisa completa de todas as arestas da coleção  $E$ .

A Tabela 13.1 resume o desempenho da implementação de um grafo por lista de arestas sob a hipótese de que as coleções  $V$  e  $E$  estejam implementadas com listas duplamente encadeadas (Seção 6.4.2).

Operação	Tempo
<code>vertices</code>	$O(n)$
<code>edges</code>	$O(m)$
<code>endVertices, opposite</code>	$O(1)$
<code>incidentEdges, areAdjacent</code>	$O(m)$
<code>replace</code>	$O(1)$
<code>insertVertex, insertEdge, removeEdge,</code>	$O(1)$
<code>removeVertex</code>	$O(m)$

**Tabela 13.1** Tempos de execução dos métodos para grafos implementados através de uma lista de arestas, onde  $V$  e  $E$  são implementados com listas encadeadas. O espaço usado é  $O(n + m)$ , onde  $n$  é o número de vértices e  $m$  é o número de arestas.

Os detalhes dos métodos selecionados para o TAD grafo são os seguintes:

- Os métodos `vertices()` e `edges()` são implementados chamando `V.iterator()` e `E.iterator()`, respectivamente.
- Os métodos `incidentEdges` e `areAdjacent` custam tempo  $O(m)$ , pois para determinar quais arestas são incidentes a um vértice  $v$  deve-se inspecionar todas as arestas.
- Já que as coleções  $V$  e  $E$  são listas implementadas com uma lista duplamente encadeada, pode-se inserir vértices, e inserir e remover arestas em tempo  $O(1)$ .
- O método `removeVertex(v)` custa tempo  $O(m)$ , pois requer que todas as arestas sejam inspecionadas para encontrar e remover aquelas incidentes a  $v$ .

Desta forma, a representação de lista de arestas é simples, porém tem limitações significantes.

Hidden page

e arestas em ambas as direções permite acelerar o desempenho de uma série de métodos para grafos usando-se uma lista de adjacência no lugar de uma lista de arestas. A Tabela 13.2 resume o desempenho da implementação do grafo com lista de adjacências, assumindo que as coleções  $V$  e  $E$  e as coleções de vértices incidentes são todas implementadas com listas duplamente encadeadas. Para um vértice  $v$ , o espaço usado pela coleção de incidentes de  $v$  é proporcional do grau de  $v$ , isto é, ele é  $O(\deg(v))$ . Assim, pela Proposição 13.6, o espaço requerido da lista de adjacência é  $O(n + m)$ .

Operação	Tempo
<code>vertices</code>	$O(n)$
<code>edges</code>	$O(m)$
<code>endVertices</code> , <code>opposite</code>	$O(1)$
<code>incidentEdges(v)</code>	$O(\deg(v))$
<code>areAdjacent(v, w)</code>	$O(\min(\deg(v), \deg(w)))$
<code>replace</code>	$O(1)$
<code>insertVertex</code> , <code>insertEdge</code> , <code>removeEdge</code> ,	$O(1)$
<code>removeVertex</code>	$O(\deg(v))$

**Tabela 13.2** Tempos de execução dos métodos de um grafo implementado com uma lista de adjacências. O espaço usado é  $O(n + m)$ , onde  $n$  é o número de vértices e  $m$  é o número de arestas.

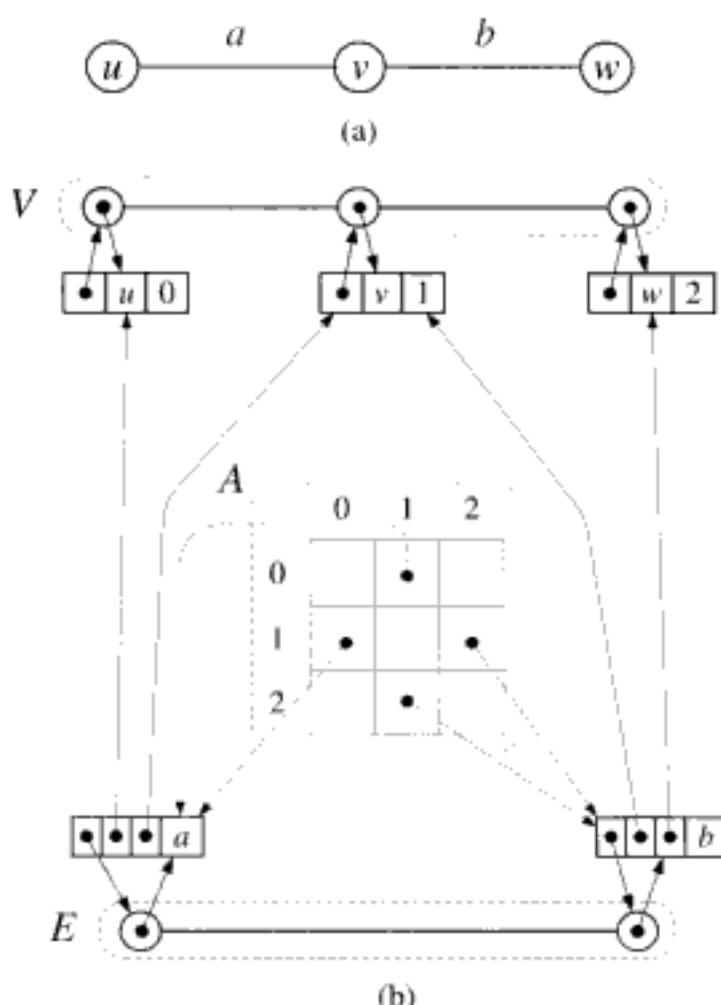
Em contraste com a forma da lista de arestas fazer as coisas, a lista de adjacência provê tempos de execução melhorados para os seguintes métodos:

- Os métodos `incidentEdges(v)` custam o tempo proporcional ao número de vértices incidentes de  $v$ , isto é, tempo de  $O(\deg(v))$ .
- Os métodos `areAdjacent(u, v)` podem ser executados pela inspeção ou pela coleção de incidentes de  $u$  ou de  $v$ . Pela escolha do menor dos dois, tem-se o tempo de execução de  $O(\min(\deg(u), \deg(v)))$ .
- O método `removeVertex(v)` custa o tempo de  $O(\deg(v))$ .

### 13.2.3 A matriz de adjacência

Como a lista de adjacências, a representação de um grafo por **matriz de adjacência** estende a estrutura de armazenamento das arestas com um componente adicional. Neste caso, aumenta-se a lista de arestas com uma matriz (um arranjo de duas dimensões)  $A$ , que permite que se determine adjacências entre pares de vértices em tempo constante. Na matriz de adjacência, consideraremos os vértices como sendo os inteiros no conjunto  $\{0, 1, \dots, n - 1\}$ , e as arestas como sendo pares desses inteiros. Isso permite armazenar referências para as arestas nas células de um arranjo de duas dimensões  $A[n \times n]$ . Especificamente, a representação por matriz de adjacência estende a lista de arestas da maneira a seguir (ver Figura 13.5):

- Um objeto vértice  $v$  também armazena uma chave inteira única entre 0 e  $n - 1$ , chamada de **índice** de  $v$ .
- Mantém-se um arranjo de duas dimensões  $A[n \times n]$  tal que a célula  $A[i, j]$  contenha uma referência para o objeto aresta  $(v, w)$ , se ela existir, onde  $v$  é o vértice com índice  $i$  e  $w$  é o vértice com índice  $j$ . Se não houver aresta, então  $A[i, j] = \text{nulo}$ .



**Figura 13.5** (a) um grafo  $G$  sem arestas paralelas; (b) representação esquemática da matriz de adjacência simplificada de  $G$ .

### Desempenho da matriz de adjacência

Para grafos com arestas paralelas, a representação da matriz de adjacências deve ser estendida de forma que, em vez de ter  $A[i, j]$  armazenando um ponteiro para uma aresta associada  $(v, w)$ , ela deve armazenar um ponteiro para uma coleção de incidentes  $I(v, w)$ , que armazena todas as arestas de  $v$  a  $w$ . Por que a maioria dos grafos considerados são simples, esta complicação não será considerada aqui.

A (simples) matriz de adjacência  $A$  permite executar o método `areAdjacent(v, w)` no tempo  $O(1)$ . Alcança-se este tempo de execução pelo acesso aos vértices  $v$  e  $w$  para determinar seus respectivos índices  $i$  e  $j$ , e então testar se  $A[i, j]$  é **nulo** ou não. O ótimo desempenho do método `areAdjacent` é cancelado por um incremento do espaço usado, o qual é agora  $O(n^2)$ , e do tempo de execução de outros métodos. Por exemplo, o método `incidentEdges(v)` agora requer que se examine toda uma linha ou coluna do arranjo  $A$ , e, assim, executar no tempo  $O(n)$ . Além disso, qualquer inserção ou remoção de vértice agora requer a criação de todo um novo arranjo  $A$ , de maior ou menor tamanho, respectivamente, o que leva o tempo de  $O(n^2)$ .

A Tabela 13.3 resume o desempenho da implementação de um grafo com matriz de adjacência. A partir desta tabela, observa-se que a lista de adjacência é superior a matriz de adjacência no espaço e é superior no tempo para todos os métodos, exceto para o método `areAdjacent`.

Operação	Tempo
vertices	$O(n)$
edges	$O(m)$
endVertices, opposite, areAdjacent	$O(1)$
incidentEdges( $v$ )	$O(n + \deg(v))$
replace, insertEdge, removeEdge,	$O(1)$
insertVertex, removeVertex	$O(n^2)$

**Tabela 13.3** Tempos de execução para um grafo implementado com matriz de adjacência.

Historicamente, a matriz de adjacência booleana foi a primeira representação usada para grafos (de forma que  $A[i, j] = \text{verdadeiro}$  se e somente se  $(i, j)$  for uma aresta). Não se deve considerar este fato surpreendente, no entanto, pois a matriz de adjacência tem um apelo natural como estrutura matemática (por exemplo, um grafo não-dirigido tem uma matriz de adjacência simétrica). A lista de adjacências veio depois, com seu apelo natural para cálculos devido a seus métodos mais rápidos para a maioria dos algoritmos (muitos algoritmos não usam o método `areAdjacent`) e sua eficiência em termos de espaço.

A maior parte dos algoritmos de grafos examinados neste livro serão eficientes quando estiverem agindo sobre um grafo armazenado usando uma lista de adjacências. Em alguns casos, no entanto, a situação pode se balancear, pois grafos com poucas arestas são processados mais eficientemente com uma lista de adjacência, enquanto grafos com muitas arestas são processados mais eficientemente com uma matriz de adjacência.

### 13.3 Caminhamento em grafos

A mitologia grega fala de um elaborado labirinto construído para abrigar o monstruoso Minotauro, parte homem e parte touro. Este labirinto era tão complexo que nenhum animal ou homem podia escapar dele. Até que o herói grego Teseu, com a ajuda da filha do rei, Ariadne, decidiu implementar um algoritmo de **caminhamento em grafos**. Teseu amarrou um novelo de linha na porta do labirinto e o desenrolou à medida que caminhava pelas tortuosas passagens à procura do monstro. Evidentemente, ele sabia sobre o bom projeto de algoritmos, pois após encontrar e vencer o Minotauro ele facilmente seguiu o fio de volta à porta e dos braços de Ariadne. Formalmente, um **caminhamento** é um procedimento sistemático para a exploração de um grafo pelo exame de todos os seus vértices e arestas.

#### 13.3.1 Pesquisa em profundidade

O primeiro algoritmo de caminhamento analisado é o caminhamento em profundidade (DFS, em inglês, **depth-first search**) em um grafo não-dirigido. O caminhamento em profundidade é útil em uma variedade de tarefas com grafos, incluindo encontrar um caminho de um vértice a outro, determinar se um grafo é conexo ou não e achar uma árvore de cobertura de um grafo conexo.

O caminhamento em profundidade em um grafo não-dirigido  $G$  é análogo a caminhar em um labirinto com um fio e uma lata de tinta, sem se perder. Começa-se em um vértice específico  $s$  em  $G$ , o qual se inicializa amarrando nosso fio a  $s$  e pintando  $s$  para marcar-lo como “visitado”. O vértice  $s$  é agora o vértice “atual”— se chamará o vértice atual de  $u$  daqui para frente. Percorre-se  $G$  considerando uma aresta arbitrária  $(u, v)$  incidente ao vértice atual  $u$ . Se a aresta  $(u, v)$  levar a um

vértice  $v$  já visitado (ou seja, pintado), retorna-se imediatamente ao vértice  $u$ . Se, por outro lado,  $(u,v)$  levar a um vértice  $v$  não-visitado, então desenrola-se um pouco de nosso fio e vai-se para  $v$ . Pinta-se  $v$  como “visitado” e faz-se com que ele passe a ser o vértice atual, repetindo a operação. Mais cedo ou mais tarde, se irá a um local sem saída, ou seja, o vértice atual  $u$  tal que todas as arestas incidentes a  $u$  levem a vértices já visitados. Para sair desse impasse, enrola-se um pouco do nosso fio, retornando pela aresta que nos levou a  $u$  até um vértice já visitado  $v$ . Faz-se de  $v$  nosso vértice atual e repete-se a operação acima para quaisquer arestas incidentes a  $v$  que não se tenha explorado antes. Se todas as arestas incidentes a  $v$  conduzirem a vértices já visitados, então enrola-se mais um pouco do fio e retorna-se ao vértice anterior a  $v$  no caminho que se percorreu e repete-se a operação naquele vértice. Assim, continua-se a retornar ao longo do caminho que já se percorreu até achar um vértice que ainda tenha uma aresta não-explorada, então se seguirá esta aresta e se continuará o caminhamento. O processo termina quando nosso retorno nos trouxer de volta ao vértice inicial  $s$ , e não houver mais arestas inexploradas incidentes a  $s$ .

Esse processo simples percorre as arestas de  $G$ . (Ver Figura 13.6.)

### Arestas de descoberta e arestas de retorno

É possível visualizar um caminhamento em profundidade ao orientar as arestas pela direção em que são exploradas durante o caminhamento, distinguindo as arestas usadas para descobrir novos vértices, chamadas de **arestas de descoberta** ou **aresta de árvore**, daquelas que levam a vértices já visitados, chamadas **arestas de retorno**. (Ver Figura 13.6f.) Na analogia acima, as arestas de descoberta são as arestas em que se desenrola o fio quando são percorridas, e as arestas de retorno são as arestas em que se retorna imediatamente sem desenrolar o fio. Como será visto, as arestas de descoberta formam uma árvore de cobertura do componente conexo do vértice de início  $s$ . Chamam-se as arestas que não estão nesta árvore de “arestas de retorno”, porque assumindo que a árvore tem o vértice inicial como raiz, então cada uma dessas arestas leva de um vértice na árvore a um de seus ancestrais.

O pseudocódigo do caminhamento em profundidade iniciando em um vértice  $v$  segue a analogia do fio e da tinta. Usa-se a recursão para implementar a analogia do fio e pressupõe-se ter um mecanismo (a analogia da tinta) para determinar se um vértice ou aresta já foi explorado e para rotular as arestas como sendo de descoberta ou de retorno. Esse mecanismo vai exigir espaço adicional e pode afetar o tempo de execução do algoritmo. Uma descrição em pseudocódigo do algoritmo recursivo do caminhamento em profundidade é mostrada no Trecho de código 13.1.

#### Algoritmo DFS( $G, v$ ):

**Entrada:** um grafo  $G$  e um vértice  $v$  de  $G$ .

**Saída:** as arestas de  $G$  rotuladas como “descoberta” ou “retorno”.

rotule  $v$  como “descoberta”

**para todos** vértices  $e$  em  $G.\text{incidentEdges}(v)$  **faça**

    se a aresta  $e$  for inexplorada **então**

$w \leftarrow G.\text{opposite}(v, e)$

        se vértice  $w$  for inexplorado **então**

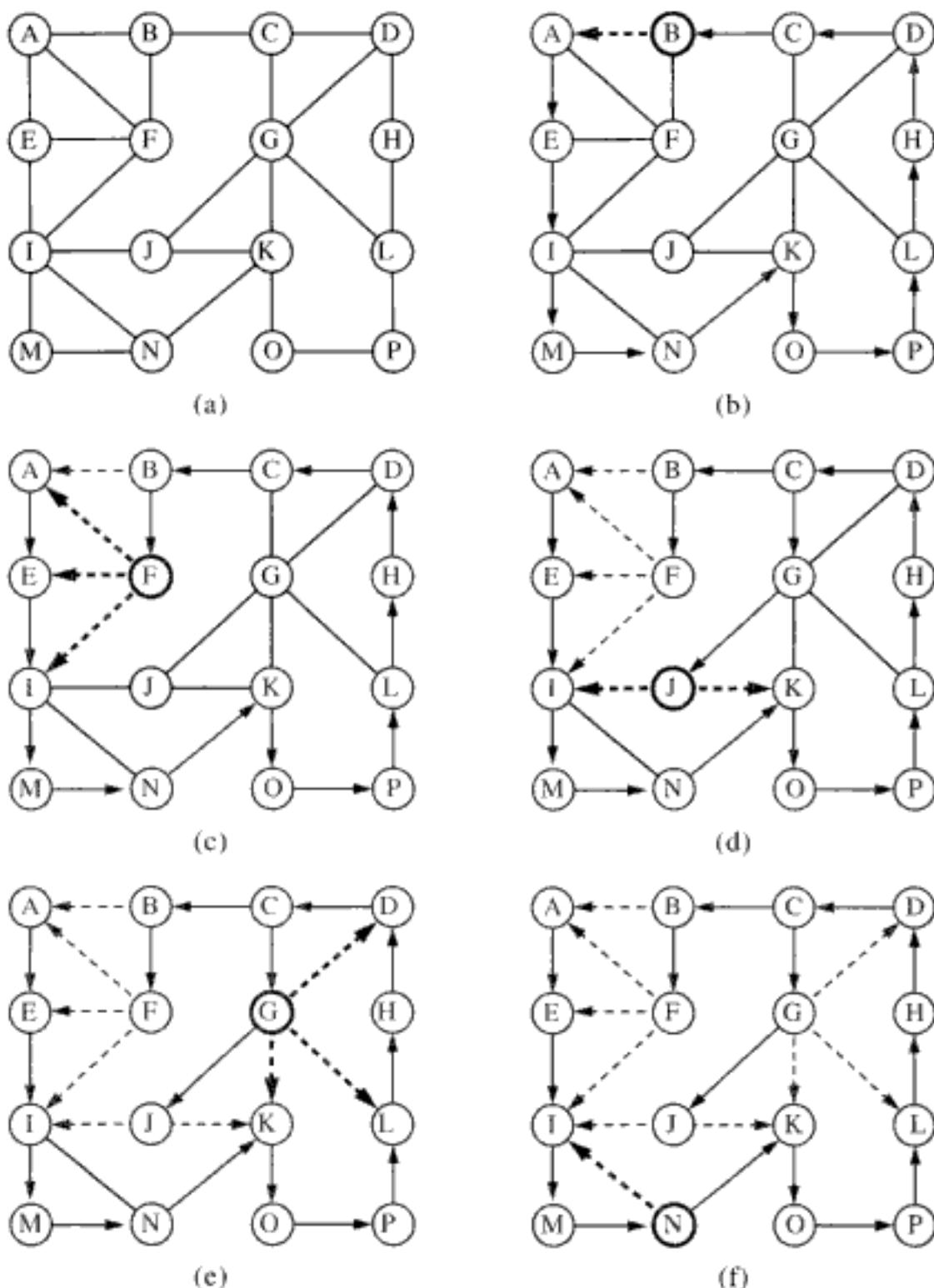
            rotule  $e$  como sendo de “descoberta”

            chame DFS( $G, w$ )

        senão

            rotule  $e$  como sendo de “retorno”

**Trecho de código 13.1** O algoritmo de caminhamento em profundidade.



**Figura 13.6** Exemplo de um caminhamento em profundidade em um grafo, iniciando no vértice A. Arestas de “descoberta” são mostradas em linhas sólidas e arestas de retorno são mostradas em linhas pontilhadas: (a) grafo de entrada; (b) caminho de arestas de descoberta a partir de A até que o retorno (B,A) é atingido; (c) alcança-se F, que é um local sem saída; (d) após retornar a C, continua-se pela aresta (C,G) e chega-se a outro local sem saída; (e) após retornar a G; (f) após retornar a N.

Hidden page

Hidden page

- element( ):** Retorna o elemento armazenado nesta posição.
- put(*k,x*):** Mapeia o valor de decoração *x* para a chave *k*, retornando o valor antigo de *k*, ou **null**, se este for um novo valor para *k*.
- get(*k*):** Busca o valor de decoração *x* assinalado para *k*, ou **null**, se não existir mapeamento para *k*.
- remove(*k*):** Remove o mapeamento da decoração para *k*, retornando o valor antigo, ou **null**, se não existir.
- entries( ):** Retorna todos os pares chave-decoração para esta posição.

Os métodos do mapa de uma posição decorável *p* provê um mecanismo simples para acessar e definir as decorações de *p*. Por exemplo, usa-se *p.get(k)* para obter o valor da decoração com chave *k*, e usa-se *p.put(k,x)* para definir o valor da decoração com chave *k* para *x*. Além disso, a chave *k* pode ser qualquer objeto, incluindo um objeto especial **explored** que o nosso algoritmo de caminhamento em profundidade pode criar. Mostra-se uma interface Java definindo um TAD deste tipo no Trecho de código 13.2.

Pode-se implementar uma posição decorável com um objeto que armazena um elemento e um mapa. Em princípio, os tempos de execução dos métodos de uma posição decorável dependem da implementação baseada em mapa. Entretanto, a maioria dos algoritmos utiliza um pequeno número constante de decorações. Assim, os métodos da posição decorável executarão no tempo  $O(1)$  no pior caso, independentemente de como se implementa o mapa embutido.

```
public interface DecorablePosition<E>
    extends Position<E>, Map<Object, Object> {
} // Sem a necessidade de novos métodos – Esta é uma mistura da interface Position e da interface Map.
```

**Trecho de código 13.2** Uma interface definindo um TAD para posições decoráveis. Não se usam tipos genéricos parametrizados para a herança dos métodos da interface Map, visto que não se sabem antecipadamente os tipos das decorações, e precisa-se permitir decorações de objetos de diferentes tipos.

Usando posições decoráveis, o algoritmo completo de caminhamento em profundidade pode ser descrito em maiores detalhes, como mostrado no Trecho de código 13.3.

#### Algoritmo DFS(*G,v,k*):

**Entrada:** Um grafo *G* com vértices e arestas decoráveis, um vértice *v* de *G* e uma chave de decoração *k*.

**Saída:** Uma decoração dos vértices do componente conexo de *v* com a chave *k* e valor VISITADO e das arestas do componente conexo de *v* com chave *k* e valores DESCOBERTO e RETORNO, de acordo com o caminhamento em profundidade de *G*.

```
v.put(k, VISITADO)
para todas areste e de G.incidentEdges(v) faça
    se e.get(k) = nulo então
        w ← G.opposite(v,e)
        se w.get(k) = nulo então
            e.put(K, DESCOBERTO)
            DFS(G,w,k)
        senão
            e.put(k,RETORNO)
```

**Trecho de código 13.3** Caminhamento em profundidade em um grafo com arestas e vértices decoráveis.

## Implementação em Java de um caminhamento em profundidade genérico

Nos Trechos de código 13.4 e 13.5, mostra-se uma implementação em Java de um caminhamento em profundidade genérico através da classe geral, DFS, que tem um método, `execute`, que pega um grafo de entrada, um vértice inicial e qualquer informação auxiliar necessária, e então inicializa o grafo e as chamadas recursivas ao método recursivo `dfsTraversal`, que ativa o caminhamento DFS. Essa implementação assume que os vértices e arestas são posições decoráveis, e eles usam decorações para falar se os vértices e arestas tem sido visitados ou não. A classe DFS contém os seguintes métodos para permitir que ele faça tarefas especiais durante um caminhamento em profundidade.

- `setup()`: chamado antes de fazer a invocação ao `dfsTraversal()`
- `initResult()`: chamado no início da execução de `dfsTraversal`.
- `startVisit(v)`: chamado no início da visita em `v`.
- `traverseDiscovery(e, v)`: chamado quando uma aresta de descoberta *e* saindo de `v` é percorrida.
- `traverseBack(e, v)`: chamado quando uma aresta de retorno *e* saindo de `v` é percorrida.
- `isDone()`: chamado para determinar se é necessário terminar o caminhamento antecipadamente.
- `finishVisit(v)`: chamado quando todas as arestas incidentes a `v` foram percorridas.
- `result()`: chamado para retornar o resultado de `dfsTraversal`.
- `finalResult(r)`: chamado para retornar a saída do método, dada a saída, `r`, de `dfsTraversal`.

```
/** Caminhamento genérico em profundidade de um grafo usando o patrão template.
 * Uma subclasse deverá sobrepor vários métodos para adicionar funcionalidade.
 A subclass should override various methods to add functionality.
 * Tipos parametrizados:
 * V, o tipo para os elementos armazenados como vértices
 * E, o tipo para os elementos armazenados como arestas
 * I, o tipo para o objeto informação passada para o método executar
 * R, o tipo para o objeto retornado pelo DFS
 */
```

```
public class DFS<V, E, I, R> {
    protected Graph<V, E> graph; // O grafo sendo caminhado
    protected Vertex<V> start; // o vértice inicial para o DFS
    protected I info; // o objeto informação passado ao DFS
    protected R visitResult; // o resultado de uma chamada recursiva
    protected static Object STATUS = new Object(); // o atributo status
    protected static Object VISITED = new Object(); // valor VISITADO
    protected static Object UNVISITED = new Object(); // valor NÃO VISITADO
    /** Marca uma posição (vértice ou aresta) como visitado. */
    protected void visit(DecorablePosition<?> p) { p.put(STATUS, VISITED); }
    /** Marca uma posição (vértice ou aresta) como não visitado. */
    protected void unVisit(DecorablePosition<?> p) { p.put(STATUS, UNVISITED); }
    /** Testa se uma posição (vértice ou aresta) está visitada. */
    protected boolean isVisited(DecorablePosition<?> p) {
        return (p.get(STATUS) == VISITED);
    }
    /** Método setup que é chamado antes da execução do DFS. */
    protected void setup() {}
    /** Resultado inicializado (primeira chamada, uma vez por vértice visitado). */
    protected void initResult() {}
    /** Chamado quando se encontra um vértice (v). */
    protected void startVisit(Vertex<V> v) {}
    /** Chamado após finalizar a visita a um vértice (v). */
}
```

```

protected void finishVisit(Vertex<V> v) { }
/** Chamada quando se cruza uma aresta descoberta (e) a partir de um vértice (from). */
protected void traverseDiscovery(Edge<E> e, Vertex<V> from) { }
/** Chamada quando se cruza uma aresta de retorno (e) a partir de um vértice (from). */
protected void traverseBack(Edge<E> e, Vertex<V> from) { }
/** Determina se o cruzamento foi feito antes. */
protected boolean isDone() { return false; /* valor padrão */ }
/** Retorna um resultado de uma visita (se necessário). */
protected R result() { return null; /* valor padrão */ }
/** Retorna o resultado final da execução do método DFS. */
protected R finalResult(R r) { return r; /* valor padrão */ }

```

**Trecho de código 13.4** Variáveis de instâncias e métodos suportados pela classe DFS, que executa um caminhamento em profundidade. Os métodos visit, unVisit e isVisited estão implementados usando posições decoráveis que são parametrizadas usando o *caractere curinga*, “?”, que pode adaptar ou ao parâmetro V ou E usados para posições decoráveis. (Continua no Trecho de código 13.5)

```

/** Executa um caminhamento em profundidade no grafo g, iniciando
 * a partir de um vértice de início s, passando em um objeto informação (in) */
public R execute(Graph<V, E> g, Vertex<V> s, I in) {
    graph = g;
    start = s;
    info = in;
    for(Vertex<V> v: graph.vertices( )) unVisit(v); // marca os vértices como não-visitados
    for(Edge<E> e: graph.edges( )) unVisit(e); // marca as arestas como não-visitadas
    setup(); // executa qualquer configuração antes do caminhamento DFS
    return finalResult(dfsTraversal(start));
}

/** método recursivo para um caminhamento DFS genérico. */
protected R dfsTraversal(Vertex<V> v) {
    initResult();
    if (!isDone())
        startVisit(v);
    if (!isDone()) {
        visit(v);
        for (Edge<E> e: graph.incidentEdges(v)) {
            if (!isVisited(e)) {
                // encontrou uma aresta inexplorada, explora-a
                visit(e);
                Vertex<V> w = graph.opposite(v, e);
                if (!isVisited(w)) {
                    // w está inexplorada, isto é uma aresta descoberta
                    traverseDiscovery(e, v);
                    if (isDone()) break;
                    visitResult = dfsTraversal(w); // pega o resultado do filho a árvore DFS
                    if (isDone()) break;
                }
            }
        else {
            // w está explorada, isto é uma aresta de retorno
            traverseBack(e, v);
            if (isDone()) break;
        }
    }
}
}

```

```

    if(!isDone( ))
        finishVisit(v);
    return result();
}
} // fim da classe DFS

```

**Trecho de código 13.5** O método principal template `dfsTraversal` da classe `DFS`, que executa um caminhamento em profundidade genérico de um grafo. (Continuação do Trecho de código 13.4.)

### Usando o padrão de templates para o caminhamento

Esse caminhamento em profundidade genérico é baseado no método de templates (ver Seção 7.3.7), que descreve um mecanismo genérico que pode ser especializado redefinindo-se certos passos. O mecanismo usado para identificar os vértices e arestas que já foram visitados durante o caminhamento é encapsulado nas chamadas para os métodos `isVisited`, `visit` e `unVisit`. Para fazer algo interessante, deve-se estender a classe `DFS` e redefinir alguns dos métodos auxiliares. Esta abordagem segue o padrão de métodos de templates. Os Trechos de Código 13.6-13.9 ilustram algumas aplicações do `dfsTraversal`.

A classe `ConnectivityTesterDFS` (Trecho de código 13.6) testa se um grafo é conexo. Ela conta o número de vértices alcançáveis através de um caminhamento em profundidade iniciando em um vértice, e compara este número com o número total de vértices do grafo.

```

/** Esta classe especializa DFS para determinar se o grafo é conexo. */
public class ConnectivityDFS<V, E> extends DFS <V, E, Object, Boolean> {
    protected int reached;
    protected void setup() { reached = 0; }
    protected void startVisit(Vertex<V> v) { reached++; }
    protected Boolean finalResult(Boolean dfsResult) {
        return new Boolean(reached == graph.numVertices());
    }
}


```

**Trecho de código 13.6** Especialização da classe `DFS` para testar se o grafo é conexo.

A classe `ComponentsDFS` (Trecho de código 13.7) procura os componentes conexos de um grafo. Ele rotula cada vértice com o número do componente conexo, usando o padrão decorador, e retorna o número do componente conexo encontrado.

```

/** Esta classe estende DFS para realizar os components conexos de um grafo. */
public class ComponentsDFS<V, E> extends DFS<V, E, Object, Integer> {
    protected Integer compNumber; // número do componente conexo
    protected Object COMPONENT = new Object(); // Selecionador do componente conexo
    protected void setup() { compNumber = 1; }
    protected void startVisit(Vertex<V> v) { v.put(COMPONENT, compNumber); }
    protected Integer finalResult(Integer dfsResult) {
        for (Vertex<V> v : graph.vertices()) // verifica por qualquer vértice não visitado
            if (v.get(STATUS) == UNVISITED) {
                compNumber += 1; // tem-se encontrado outro componente conexo
                dfsTraversal(v); // visita todos os vertices deste componente
            }
        return compNumber;
    }
}


```

**Trecho de código 13.7** Especialização da classe `DFS` para computar os componentes conexos.

A classe `FindPathDFS` (Trecho de código 13.8) encontra um caminho entre um par de vértices dados. Ela realiza um caminhamento em profundidade a partir do vértice inicial. Mantém-se o caminho formado pelas arestas de descoberta a partir do vértice inicial até o vértice atual. Quando se encontra um vértice inexplorado, ele é adicionado ao final do caminho, e quando se termina de processar o vértice, ele é removido do caminho. O caminhamento é finalizado quando o vértice final é encontrado e o caminho é retornado como um iterador de vértices e arestas (ambos tipos de posições em um grafo). Observa-se que o caminho encontrado por esta classe se constitui de arestas de descoberta.

```
/** Classe que especializa DFS para encontrar um caminho entre um vértice de inicio e um
 * determinado vértice. Ela assume que o determinado vértice é passado como o objeto info
 * para o método execute. Ela retorna uma lista de vértices e arestas compreendendo o
 * caminho do inicio até info. O caminho retornado está vazio se info estiver inalcançável a
 * partir do inicio. */
public class FindPathDFS<V, E>
    extends DFS<V, E, Vertex<V>, Iterable<Position>> {
    protected PositionList<Position> path;
    protected boolean done;
    /** Método setup para inicializar o caminho. */
    public void setup() {
        path = new NodePositionList<Position>();
        done = false;
    }
    protected void startVisit(Vertex<V> v) {
        path.addLast(v); // adiciona o vértice v ao caminho
        if (v == info)
            done = true;
    }
    protected void finishVisit(Vertex<V> v) {
        path.remove(path.last()); // remove v do caminho
        if (!path.isEmpty())
            path.remove(path.last()); // se v não for o vértice inicio
        // remove a aresta descoberta em v a partir do caminho
    }
    protected void traverseDiscovery(Edge<E> e, Vertex<V> from) {
        path.addLast(e); // adiciona a aresta e ao caminho
    }
    protected boolean isDone() {
        return done;
    }
    public Iterable<Position> finalResult(Iterable<Position> r) {
        return path;
    }
}
```

**Trecho de código 13.8** Especialização da classe `DFS` para procurar o caminho entre os vértices inicial e final.

A classe `FindCycleDFS` (Trecho de código 13.9) encontra um ciclo no componente conexo de um dado vértice  $v$ , realizando um caminhamento em profundidade a partir de  $v$ , que termina quando uma aresta de retorno é encontrada. Ela retorna um iterador (possivelmente vazio), do ciclo formado pelos vértices e aresta em um ciclo formado pela aresta de retorno encontrada.

Hidden page

dos" os vértices adjacentes ao vértice inicial  $s$  – estes vértices são colocados no nível 1. Na segunda etapa, desenrola-se o fio no comprimento de duas arestas e visitam-se todos os novos vértices que é possível alcançar sem desenrolar mais fio. Esses novos vértices, que são adjacentes aos vértices do nível 1 e não foram associados anteriormente a um nível, são colocados no nível 2, e assim por diante. O caminhamento em largura termina quando todos os vértices tiverem sido visitados.

O pseudocódigo para um caminhamento em largura iniciando em um vértice  $s$  é mostrado no Trecho de código 13.10. Usa-se espaço auxiliar para rotular arestas, marcar vértices visitados e guardar contêineres associados com os níveis. Ou seja, as coleções  $L_0, L_1, L_2$ , etc. armazenam os vértices que estão no nível 0, nível 1, nível 2, e assim por diante. Esses contêineres poderiam, por exemplo, ser implementados como filas. Eles também permitem que o caminhamento em largura não seja recursivo.

#### Algoritmo BFS( $s$ ):

```

Inicia a coleção  $L_0$  para conter o vértice  $s$ 
 $i \leftarrow 0$ 
enquanto  $L_i$  não estiver vazia faça
    Cria coleção  $L_{i+1}$  inicialmente vazia
    para todos os vértices  $v$  em  $L_i$  faça
        para todas as arestas  $e$  em  $G.\text{incidentEdges}(v)$  faça
            se aresta  $e$  estiver inexplorada então
                 $w \leftarrow G.\text{opposite}(v, e)$ 
                se vértice  $w$  estiver inexplorado então
                    rotula  $e$  como uma aresta descoberta
                    insere  $w$  em  $L_{i+1}$ 
                senão
                    rotula  $e$  como uma aresta cruzada
             $i \leftarrow i + 1$ 
```

**Trecho de código 13.10** O algoritmo de caminhamento em largura.

Ilustra-se um caminhamento em largura na Figura 13.7.

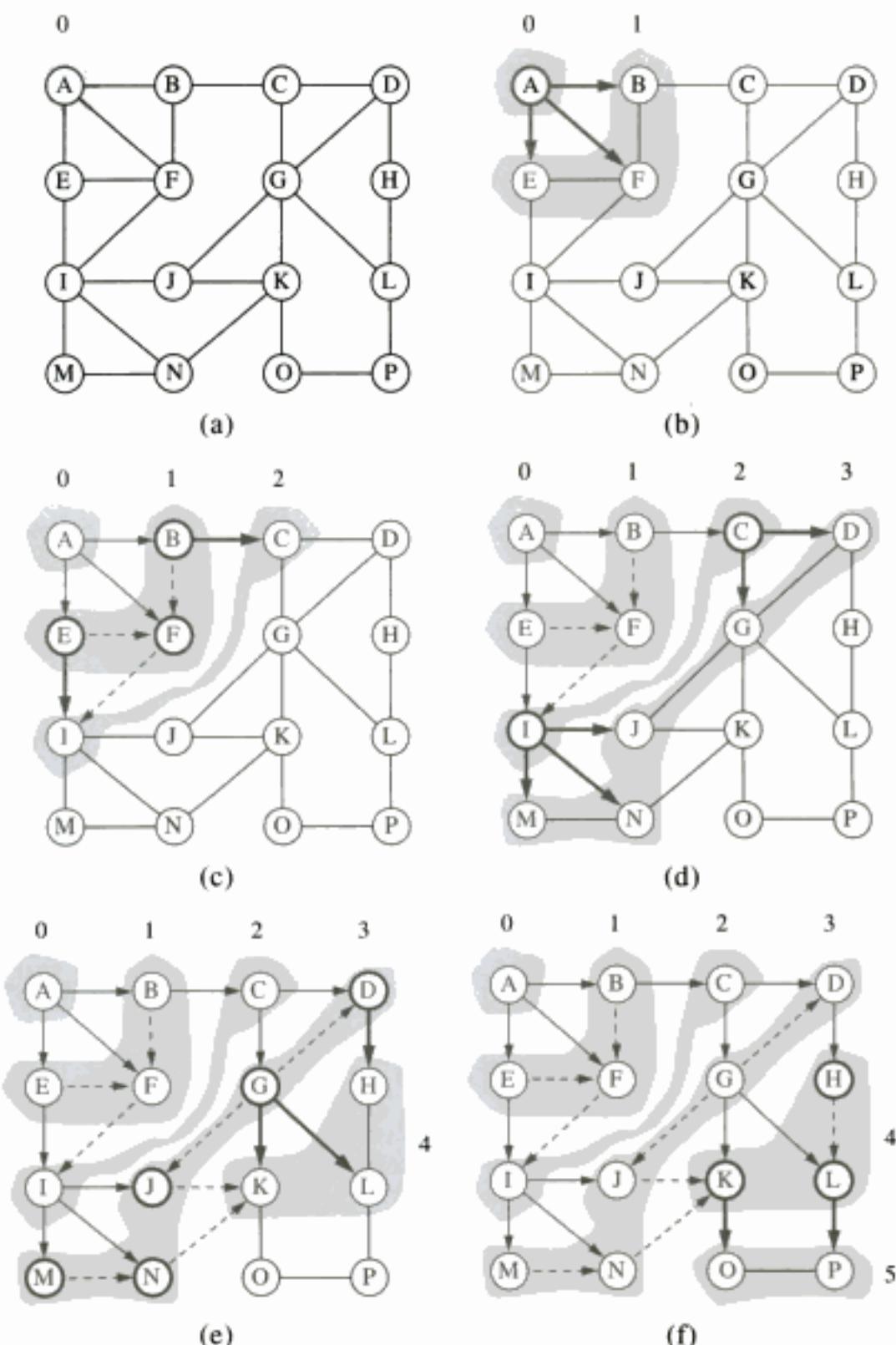
Uma das boas propriedades do caminhamento em largura é que, durante o caminhamento, pode-se rotular cada vértice com o comprimento do menor caminho (em termos de número de arestas) desde o vértice inicial  $s$ . Em particular, se o vértice  $v$  é colocado no nível  $i$  pelo caminhamento iniciado em  $s$ , então o comprimento do menor caminho de  $s$  a  $v$  é  $i$ .

Assim como o caminhamento em profundidade, pode-se visualizar o caminhamento em largura orientando as arestas de acordo com a direção em que são exploradas durante o caminhamento, e distinguindo as arestas usadas para descobrir novos vértices, as chamadas *arestas de descoberta*, e aquelas que levam a vértices já visitados, as *arestas de cruzamento*. (Ver Figura 13.7f.) Assim como o caminhamento em profundidade, as arestas de descoberta formam uma árvore de cobertura, que neste caso chama-se de árvore BFS. No entanto, não se chamam, neste caso, as arestas fora da árvore de "arestas de retorno", pois nenhum deles conecta um vértice a um de seus antecedentes. Cada aresta fora da árvore conecta um vértice  $v$  a outro vértice que não é nem ancestral nem descendente de  $v$ .

O algoritmo de caminhamento em largura tem várias propriedades interessantes, algumas das quais são exploradas na proposição que segue.

**Proposição 13.14** Seja  $G$  um grafo não-dirigido no qual um caminhamento em largura iniciado em um vértice  $s$  foi realizado. Então

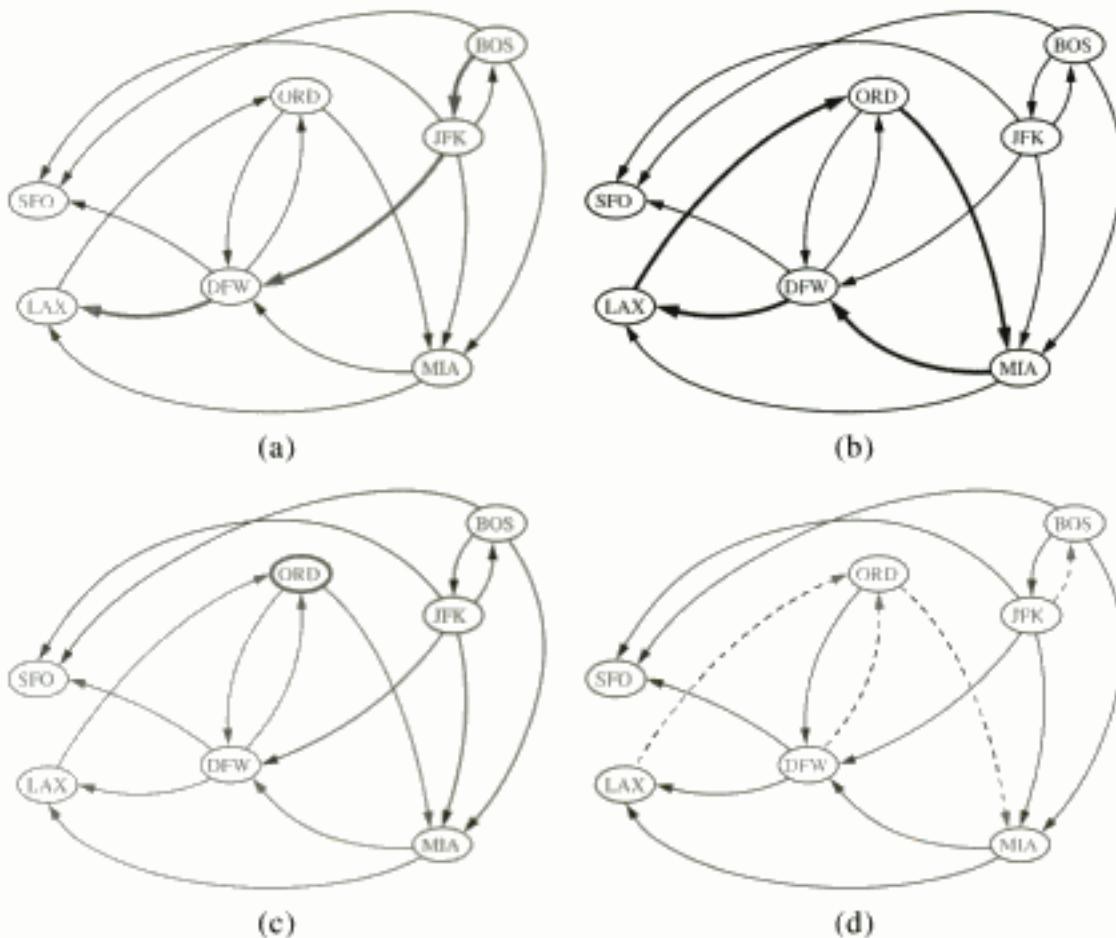
- o caminhamento visita todos os vértices no componente conexo de  $s$ ;
- as arestas de descoberta formam uma árvore de cobertura  $T$  chamada de árvore de cobertura BFS, para o componente conexo de  $s$ ;



**Figura 13.7** Exemplo de caminhamento em largura, onde as arestas incidentes a um vértice são exploradas na ordem alfabética dos vértices adjacentes. As arestas de descoberta são mostradas com linhas sólidas, e as arestas de cruzamento são mostradas com linhas pontilhadas: (a) o grafo antes do caminhamento; (b) descoberta do nível 1; (c) descoberta do nível 2; (d) descoberta do nível 3; (e) descoberta do nível 4; (f) descoberta do nível 5.

Hidden page

O **fechamento transitivo** de um dígrafo  $\tilde{G}$  é o dígrafo  $\tilde{G}^*$ , em que os vértices de  $\tilde{G}^*$  são os mesmos de  $\tilde{G}$ , e  $\tilde{G}^*$  tem um vértice  $(u, v)$  sempre que  $\tilde{G}$  tiver um caminho dirigido de  $u$  para  $v$ . Ou seja, define-se  $\tilde{G}^*$  começando com o dígrafo  $\tilde{G}$  e adicionando uma aresta extra  $(u, v)$  para cada  $u$  e  $v$  tal que  $v$  seja atingível a partir de  $u$  (e não existir ainda uma aresta  $(u, v)$  em  $\tilde{G}$ ).



**Figura 13.8** Exemplos de atingibilidade em um dígrafo: (a) um caminho dirigido de BOS a LAX é desenhado em cinza; (b) um ciclo dirigido (ORD, MIA, DFW, LAX, ORD) é mostrado em cinza; seus vértices formam um subgrafo fortemente conexo; (c) os vértices e arestas atingíveis de ORD são mostrados em cinza; (d) remover as arestas pontilhadas em cinza produz um dígrafo acíclico.

Problemas interessantes que lidam com a atingibilidade em um dígrafo  $\tilde{G}$  incluem os seguintes:

- Dados vértices  $u$  e  $v$ , determinar se  $u$  atinge  $v$ .
- Achar todos os vértices de  $\tilde{G}$  que sejam atingíveis desde um dado vértice  $s$ .
- Determinar se  $\tilde{G}$  é fortemente conexo.
- Determinar se  $\tilde{G}$  é acíclico.
- Determinar o fechamento transitivo  $\tilde{G}^*$  de  $\tilde{G}$ .

No restante desta seção, serão explorados alguns algoritmos eficientes para resolver esses problemas.

#### 13.4.1 Caminhamento em um dígrafo

Assim como grafos não-dirigidos, pode-se explorar um dígrafo de forma sistemática com métodos semelhantes ao caminhamento em largura (BFS) e ao caminhamento em profundidade (DFS) definidos previamente para grafos não-dirigidos (Seções 13.3.1 e 13.3.3). Essas explora-

ções podem ser usadas, por exemplo, para resolver questões de atingibilidade. Os métodos dirigidos para caminhamento em profundidade e caminhamento em largura que se desenvolverão nesta seção para essas tarefas são muito semelhantes aos seus correspondentes não-dirigidos. De fato, a única diferença real é que estes métodos dirigidos para caminhamento em profundidade e caminhamento em largura somente percorrem as arestas de acordo com suas direções respectivas.

A versão dirigida do caminhamento em profundidade (DFS) iniciando no vértice  $v$  pode ser descrito pelo algoritmo recursivo no Trecho de código 13.11 (Ver Figura 13.9.)

#### Algoritmo DirectedDFS( $v$ ):

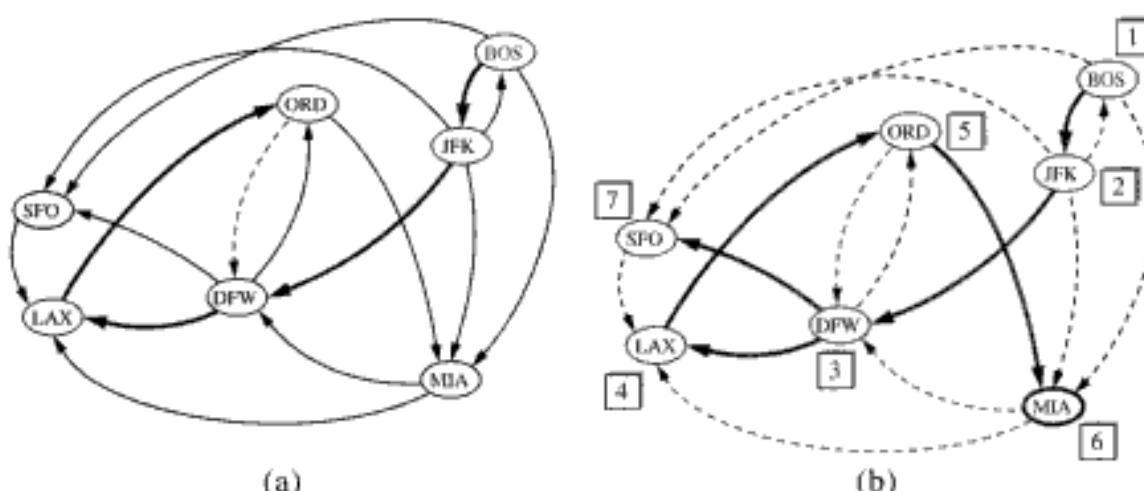
Marque o vértice  $v$  como visitado.

**para** cada aresta  $(v,w)$  saindo de  $v$  **faça**

se vértice  $w$  não foi visitado **então**

Chame recursivamente DirectedDFS( $w$ ).

**Trecho de código 13.11** O algoritmo DirectedDFS.



**Figura 13.9** Um exemplo de caminhamento em profundidade em um dígrafo: (a) passo intermediário, em que pela primeira vez um vértice já visitado (DFW) é alcançado; (b) o caminhamento em profundidade completo. As arestas da árvore são mostradas com linhas sólidas cinzas; as arestas de retorno são mostradas com linhas pontilhadas cinzas; e as arestas de descoberta e de cruzamento são mostradas com linhas pontilhadas pretas. A ordem na qual os vértices são visitados é indicada pelo número próximo ao vértice. A aresta (ORD, DFW) é uma aresta de retorno, mas (DFW, ORD) é uma aresta de descoberta. A aresta (BOS, SFO) é uma aresta de descoberta e (SFO, LAX) é uma aresta de cruzamento.

Um caminhamento em profundidade em um dígrafo  $\tilde{G}$  partitiona as arestas de  $\tilde{G}$  atingíveis pelo vértice inicial em **arestas de árvore** ou **arestas de descoberta**, que levam a descobrir um novo vértice, e em **arestas fora da árvore**, que levam a um vértice visitado previamente. As arestas de árvore formam uma árvore com raiz no vértice inicial, chamada de **árvore de profundidade**, e existem três tipos de arestas fora da árvore:

- **arestas de retorno**, que conectam um vértice a seu antecessor na árvore de profundidade;
- **arestas de descoberta**, que conectam um vértice a um descendente na árvore de profundidade;
- **arestas de cruzamento**, que conectam um vértice a um vértice que não é nem seu antecessor nem seu descendente.

Ver Figura 13.9b para um exemplo de cada tipo de aresta fora da árvore.

**Proposição 13.16** Seja  $\tilde{G}$  um dígrafo. O caminhamento em profundidade em  $\tilde{G}$  iniciando em um vértice  $s$  visita todos os vértices de  $\tilde{G}$  que são atingíveis desde  $s$ . A árvore de profundidade contém caminhos dirigidos de  $s$  a qualquer vértice atingível a partir de  $s$ .

**Justificativa** Seja  $V_s$  o subconjunto de vértices de  $\tilde{G}$  visitados pelo caminhamento em profundidade iniciando em um vértice  $s$ . Deseja-se mostrar que  $V_s$  contém  $s$  e qualquer vértice atingível a partir de  $s$ . Supõe-se, para obter uma contradição, que há um vértice  $w$  atingível desde  $s$  que não está em  $V_s$ . Considere-se um caminho dirigido de  $s$  a  $w$  e seja  $(u,v)$  a primeira aresta deste caminho que nos retira de  $V_s$ , ou seja,  $u$  está em  $V_s$  e  $v$  não está. Quando o caminhamento em profundidade atinge  $u$ , ele explora todas as arestas saindo de  $u$  e, por isso, deve atingir também o vértice  $v$  via a aresta  $(u,v)$ . Portanto  $v$  deve estar em  $V_s$ , e obtém-se uma contradição. Assim,  $V_s$  deve conter cada vértice atingível a partir de  $s$ . ■

Analizar o tempo de execução do caminhamento em profundidade dirigido é um processo análogo ao seu equivalente não-dirigido. Em particular, uma chamada recursiva é feita para cada vértice uma vez e cada aresta é percorrida uma vez (desde sua origem). Assim, se  $n_s$  vértices e  $m_s$  arestas podem ser atingidos a partir de um vértice  $s$ , um caminhamento em profundidade dirigido iniciando em  $s$  é executado em tempo  $O(n_s + m_s)$ , desde que o dígrafo seja representado com uma estrutura de dados que suporte os métodos para vértices e arestas em tempo constante. A estrutura de dados lista de adjacência, por exemplo, satisfaz esta exigência.

Pela Proposição 13.16, usa-se o caminhamento em profundidade para encontrar todos os vértices atingíveis a partir de um dado vértice, e, portanto achar o fechamento transitivo de  $\tilde{G}$ . Ou seja, faz-se um caminhamento em profundidade iniciando de cada vértice  $v$  em  $\tilde{G}$  para verificar quais vértices  $w$  são atingíveis a partir de  $v$ , adicionando uma aresta  $(v,w)$  ao fechamento transitivo para cada um desses vértices  $w$ . De forma similar, percorrendo repetidamente o dígrafo  $\tilde{G}$  com o caminhamento em profundidade, iniciando cada vez em um vértice diferente, testa-se facilmente se  $\tilde{G}$  é fortemente conexo. Ou seja,  $\tilde{G}$  será fortemente conexo se cada caminhamento em profundidade visitar todos os vértices de  $\tilde{G}$ .

Assim, obtém-se imediatamente a proposição a seguir.

**Proposição 13.17** Seja  $\tilde{G}$  um dígrafo com  $n$  vértices e  $m$  arestas. Os problemas a seguir serão solucionados por um algoritmo que percorra  $\tilde{G}$   $n$  vezes usando caminhamento em profundidade, executado em tempo  $O(n(n + m))$  e que usa memória  $O(n)$ :

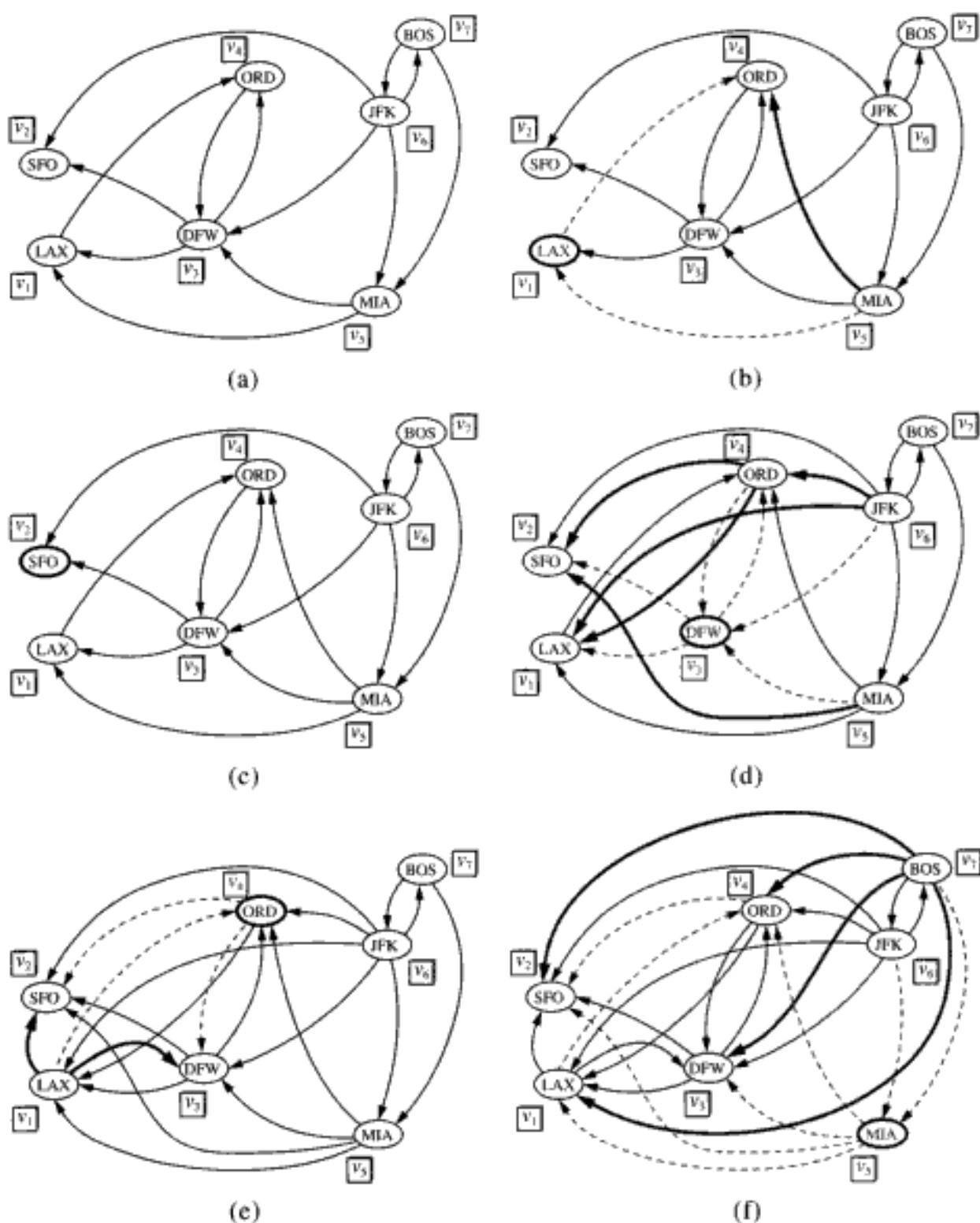
- determinar, para cada vértice  $v$  de  $\tilde{G}$  o subgrafo atingível a partir de  $v$ ;
- testar se  $\tilde{G}$  é fortemente conexo;
- determinar o fechamento transitivo  $\tilde{G}^*$  de  $\tilde{G}$ .

### Testando conexões fortes

Pode-se determinar se um grafo dirigido  $\tilde{G}$  é fortemente conexo muito mais depressa do que a proposição acima determina, usando apenas dois caminhamentos em profundidade. Começa-se realizando um caminhamento em  $\tilde{G}$  iniciando em um vértice arbitrário  $s$ . Se existe algum vértice de  $\tilde{G}$  que não é visitado por este caminhamento e não é atingível a partir de  $s$ , então o grafo não é fortemente conexo. Assim, se este primeiro caminhamento visita cada vértice de  $\tilde{G}$ , então revertem-se todas as arestas de  $\tilde{G}$  (usando o método `reverseDirection`) e realiza-se outro caminhamento em profundidade iniciando em  $s$  neste grafo “revertido”. Se cada vértice de  $\tilde{G}$  é visitado por este segundo caminhamento, então o grafo é fortemente conexo, pois cada um dos vértices visitados no caminhamento pode atingir  $s$ . Já que este algoritmo realiza apenas dois caminhamentos em profundidade sobre  $\tilde{G}$ , ele é executado em tempo  $O(n + m)$ .

Hidden page

Hidden page



**Figura 13.10** Seqüência de dígrafos determinados pelo algoritmo Floyd-Warshall: (a) dígrafo inicial  $\vec{G} = \vec{G}_0$  e enumeração dos vértices; (b) dígrafo  $\vec{G}_1$ ; (c)  $\vec{G}_2$ ; (d)  $\vec{G}_3$ ; (e)  $\vec{G}_4$ ; (f)  $\vec{G}_5$ . Vide que  $\vec{G}_5 = \vec{G}_6 = \vec{G}_7$ . Se o dígrafo  $\vec{G}_{k-1}$  tem as arestas  $(v_i, v_k)$  e  $(v_k, v_j)$ , mas não a aresta  $(v_i, v_j)$ , no desenho do dígrafo  $\vec{G}_k$ , são mostradas as arestas  $(v_i, v_k)$  e  $(v_k, v_j)$  com linhas tracejadas cinzas, e a aresta  $(v_i, v_j)$  com linha contínua cinza.

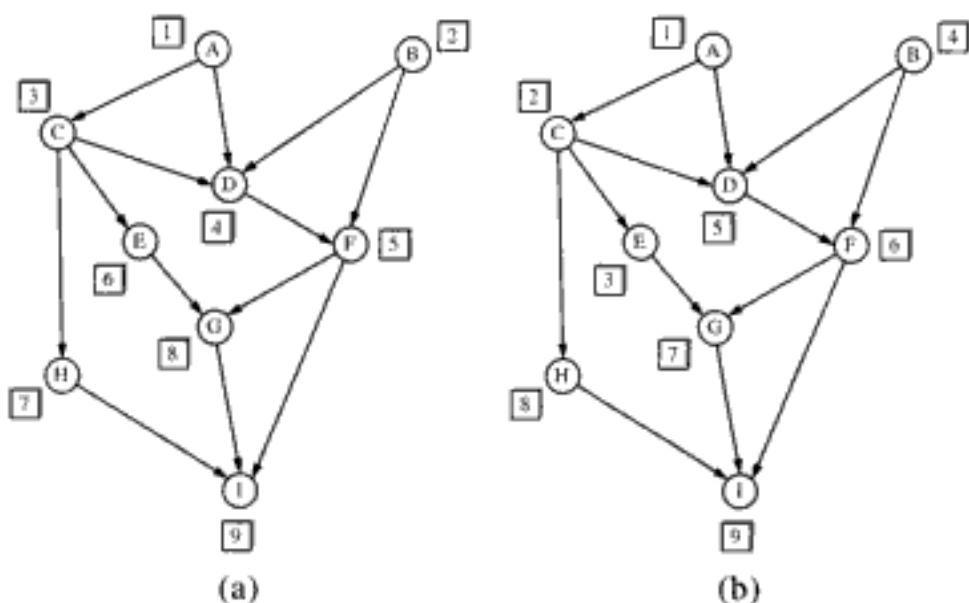


Figura 13.11 Duas ordenações topológicas do grafo cíclico dirigido.

**Proposição 13.21**  $\vec{G}$  tem uma ordenação topológica se e somente se  $\vec{G}$  for acíclico.

**Justificativa** A necessidade (o “somente se” da proposição) é fácil de demonstrar. Suponha que  $\vec{G}$  é topologicamente ordenado. Assume-se, para obter uma contradição, que  $\vec{G}$  tem um ciclo consistindo nas arestas  $(v_{i_0}, v_{i_1}), (v_{i_1}, v_{i_2}), \dots, (v_{i_{k-1}}, v_{i_0})$ . Por causa da ordenação topológica, deve-se ter  $i_0 < i_1 < \dots < i_{k-1} < i_0$ , o que é claramente impossível. Assim,  $\vec{G}$  deve ser acíclico.

Agora, justifica-se a suficiência da condição (o “se” da proposição). Suponha que  $\vec{G}$  seja acíclico. Será fornecida uma descrição algorítmica de como construir uma ordenação topológica para  $\vec{G}$ . Como  $\vec{G}$  é acíclico,  $\vec{G}$  deve ter um vértice ao qual não chega nenhuma aresta (ou seja, um vértice com grau de entrada 0). Seja  $v_1$  um vértice assim. De fato, se  $v_1$  não existir, então ao percorrer um caminho dirigido a partir de um vértice arbitrário se terminaria por encontrar um vértice visitado previamente, o que contradiz o fato de  $\vec{G}$  ser acíclico. Removendo-se  $v_1$  de  $\vec{G}$ , bem como suas arestas de saída, o dígrafo resultante ainda é acíclico. Portanto, o dígrafo resultante também tem um vértice  $v_2$  ao qual não chegam arestas. Repetindo esse processo até que o dígrafo  $\vec{G}$  esteja vazio, obtém-se uma ordenação  $v_1, \dots, v_n$  dos vértices de  $\vec{G}$ . Pela construção acima, se  $(v_i, v_j)$  é uma aresta de  $\vec{G}$ , então  $v_i$  deve ser removido antes que  $v_j$  possa ser removido e, portanto,  $i < j$ . Assim,  $v_1, \dots, v_n$  é uma ordenação topológica. ■

A justificativa da Proposição 13.21 sugere um algoritmo (Trecho de código 13.13) chamado de *ordenação topológica*, para determinar uma ordenação topológica de um dígrafo.

**Algoritmo TopologicalSort ( $\vec{G}$ ):**

**Entrada:** um dígrafo  $\vec{G}$  com  $n$  vértices.

**Saída:** uma ordenação topológica  $v_1, \dots, v_n$  de  $\vec{G}$ .

Seja  $S$  uma pilha inicialmente vazia.

**para todos**  $u$  encontrados em  $\vec{G}.\text{vertices}()$  **faça**

    Seja  $\text{incounter}(u)$  o grau de entrada de  $u$ .

**se**  $\text{incounter}(u) = 0$  **então**

$S.\text{push}(u)$

*i*  $\leftarrow 1$

**enquanto**  $S.\text{isEmpty}()$  **faça**

$u \leftarrow S.\text{pop}()$

    Seja  $u$  o vértice numerado como  $i$  na ordenação topológica.

*i*  $\leftarrow i + 1$

```

para cada aresta de saída  $(u,w)$  de  $u$  faça
     $incounter(w) \leftarrow incounter(w) - 1$ 
    se  $incounter(w) = 0$  então
         $S.push(w)$ 

```

**Trecho de código 13.13** Pseudocódigo para o algoritmo de ordenação topológica. (Um exemplo de aplicação deste algoritmo é mostrado na Figura 13.12.)

**Proposição 13.22** Seja  $\tilde{G}$  um dígrafo com  $n$  vértices e  $m$  arestas. O algoritmo de ordenação topológica é executado em tempo  $O(n + m)$  usando espaço auxiliar  $O(n)$ , e determina uma ordenação topológica de  $\tilde{G}$  ou não consegue numerar algum vértice, o que indica que  $\tilde{G}$  tem um ciclo dirigido.

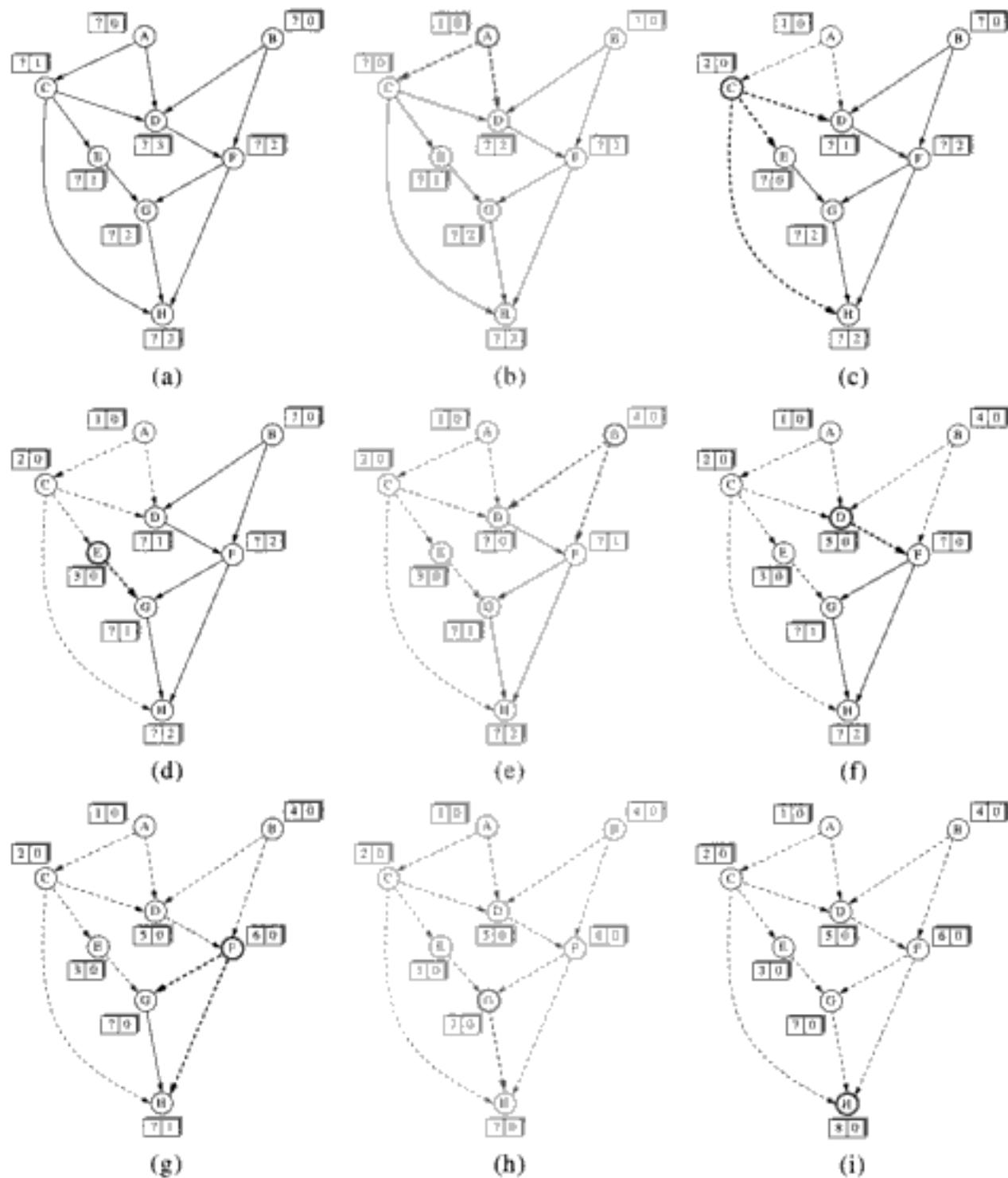
**Justificativa** A determinação inicial dos graus de entrada e preparo das variáveis  $incounter$  podem ser feitos com uma passagem sobre o grafo, que custa tempo  $O(n + m)$ . Usa-se o padrão de decoradores para associar atributos contadores a cada vértice. Um vértice  $u$  é *visitado* pelo algoritmo de ordenação topológica quando  $u$  é removido da pilha  $S$ . Um vértice  $u$  pode ser visitado somente quando  $incounter(u) = 0$ , o que implica que todos os seus predecessores (vértices com arestas que levam a  $u$ ) foram visitados previamente. Como consequência, qualquer vértice que faça parte de um ciclo dirigido nunca será visitado, e qualquer outro vértice será visitado exatamente uma vez. O algoritmo percorre todas as arestas saindo de cada vértice visitado exatamente uma vez, portanto, seu tempo de execução é proporcional ao número de arestas saindo dos vértices visitados. Assim, o algoritmo é executado em tempo  $O(n + m)$ . Quanto ao espaço usado, a pilha  $S$  e as variáveis  $incounter$  associadas aos vértices usam espaço  $O(n)$ . ■

Como um efeito secundário, o algoritmo de ordenação topológica do Trecho de código 13.13 também testa se um grafo  $\tilde{G}$  é acíclico. De fato, se o algoritmo termina sem ordenar todos os vértices, então o subgrafo dos vértices que não foram ordenados deve conter um ciclo dirigido.

## 13.5 Grafos ponderados

Como foi visto na Seção 13.3.3, a estratégia de caminhamento em largura pode ser usada para encontrar um caminho mínimo de algum vértice inicial a cada um dos outros vértices em um grafo conexo. Esta abordagem faz sentido em casos em que cada aresta é tão boa quanto qualquer outra, mas existem muitas situações em que esta abordagem não é apropriada. Por exemplo, pode-se estar usando um grafo para representar uma rede de computadores (como a Internet) e pode-se estar interessado em encontrar o caminho mais rápido para enviar um pacote de dados entre dois computadores. Neste caso, provavelmente não é correto considerar todas as arestas equivalentes, pois algumas conexões na rede são tipicamente muito mais rápidas do que outras (por exemplo, algumas arestas podem representar conexões lentas por linha telefônica, enquanto outras representam ligações de alta velocidade em fibra ótica). Da mesma forma, desejando-se usar um grafo para representar as estradas entre cidades, pode-se estar interessado em achar as distâncias mais curtas entre elas. Neste caso, provavelmente também não é correto considerar todas as arestas equivalentes, pois algumas distâncias serão muito maiores do que outras. Assim, é natural considerar grafos cujas arestas não sejam todas equivalentes.

Um **grafo ponderado** é um grafo que tem um valor numérico  $w(e)$  (por exemplo, um inteiro) associado a cada aresta  $e$ , chamada de **peso** de  $e$ . Um exemplo de um grafo ponderado é mostrado na Figura 13.13.



**Figura 13.12** Exemplo de uma execução do algoritmo TopologicalSort (Trecho de código 12.11): (a) configuração inicial; (b–i) após cada iteração do laço enquanto. Os números nos vértices mostram o número do vértice e o valor corrente de `incounter` para o vértice. As arestas percorridas são mostradas com setas cinzas tracejadas. Linhas espessas mostram o vértice e arestas examinadas na interação corrente.

Hidden page

Supondo que se tem um grafo ponderado  $G$  e se é solicitado a encontrar um caminho mínimo de um vértice  $v$  a cada outro vértice em  $G$ , considerando os pesos das arestas como distâncias. Nesta seção, exploram-se formas eficientes de encontrar todos os caminhos mínimos, se existirem. O primeiro algoritmo que se discute é para o caso simples e comum em que todos os pesos das arestas em  $G$  são não-negativos (ou seja,  $w(e) \geq 0$  para toda aresta  $e$  de  $G$ ); portanto, sabe-se de antemão que não podem haver ciclos negativos em  $G$ . O caso especial de determinar um caminho mínimo quando todos os pesos são 1 foi resolvido com o algoritmo de caminhamento em largura, apresentado na Seção 13.3.3.

Existe uma abordagem interessante para resolver este problema de **origem única** baseado no **método guloso** (Seção 12.4.2). Com este método o problema foi resolvido fazendo repetidamente a melhor escolha entre as disponíveis em cada iteração. Este paradigma pode ser freqüentemente usado em situações em que se está tentando otimizar alguma função de custo em uma coleção de objetos. Pode-se adicionar objetos um de cada vez à essa coleção, sempre escolhendo o próximo que otimiza a função entre aqueles objetos ainda a serem escolhidos.

### 13.6.1 O algoritmo de Dijkstra

A idéia principal na aplicação do método guloso para o problema do caminho mínimo com origem única é realizar um caminhamento em largura “ponderado” iniciando-se em  $v$ . Em particular, pode-se usar o método guloso para desenvolver um algoritmo que iterativamente aumenta uma “nuvem” de vértices em torno de  $v$ , com os vértices entrando na nuvem na seqüência de suas distâncias de  $v$ . Assim, a cada iteração o próximo vértice escolhido é o vértice fora da nuvem e que está mais próximo de  $v$ . O algoritmo termina quando não há mais vértices fora da nuvem e, neste momento, se terá o caminho mais curto de  $v$  para qualquer outro vértice de  $G$ . Esta abordagem é um simples e poderoso exemplo do padrão de projeto baseado no método guloso.

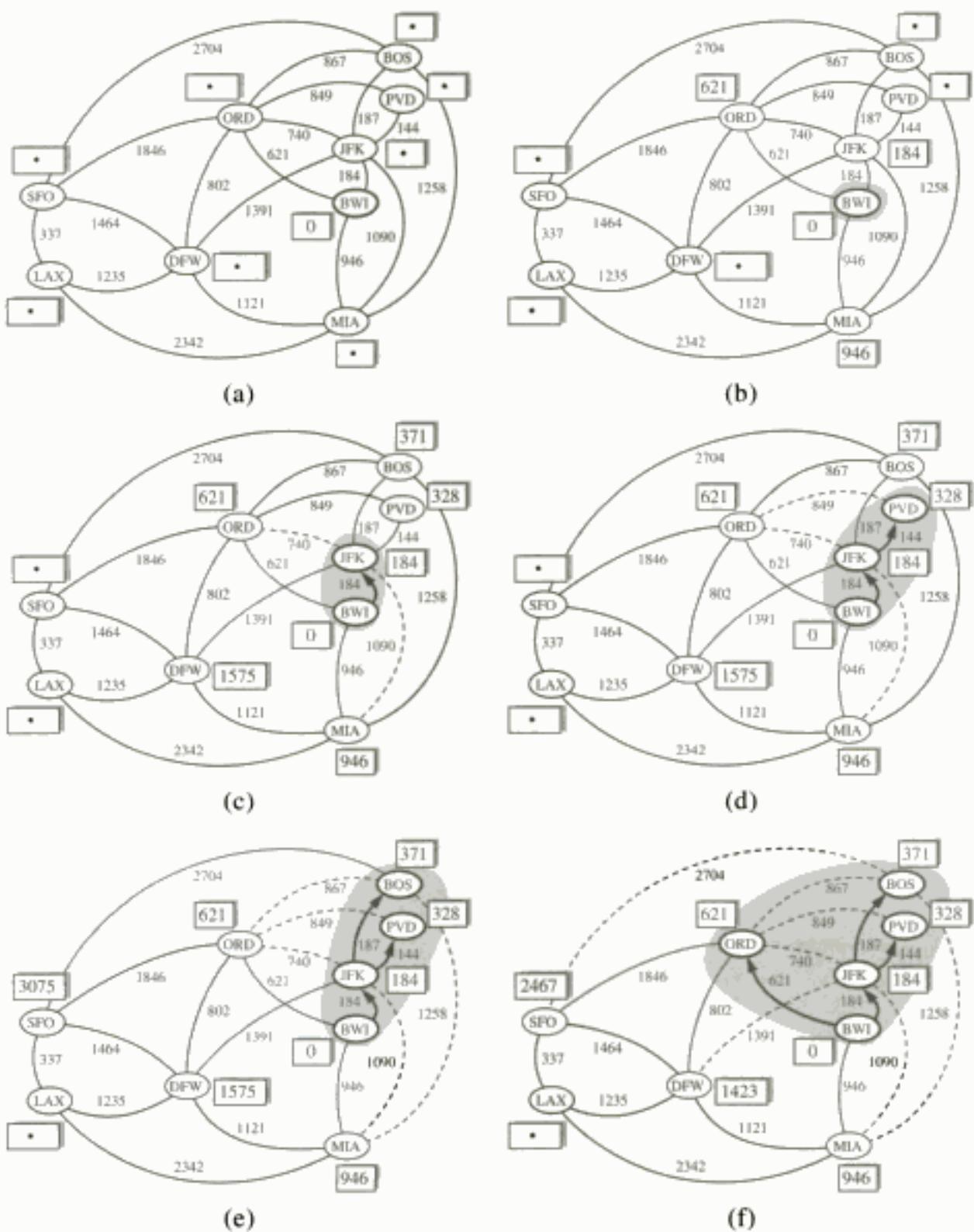
#### Um método guloso para determinar caminhos mínimos

Aplicar o método guloso para o problema do caminho mínimo com origem única resulta em um algoritmo conhecido como **algoritmo de Dijkstra**. Quando é aplicado a outros problemas de grafos, no entanto, o método guloso pode não achar necessariamente a melhor solução (como no caso do **problema do caixeiro viajante**, no qual se deseja encontrar o menor caminho que visita todos os vértices do grafo exatamente uma vez). Mesmo assim, existem várias situações nas quais o método guloso permite determinar a melhor solução. Neste capítulo, duas dessas situações serão discutidas: determinando caminhos mínimos e construindo de uma árvore de cobertura mínima.

Para simplificar a descrição do algoritmo de Dijkstra, pressupõe-se que o grafo de entrada  $G$  seja não-dirigido (ou seja, todas as suas arestas são não-dirigidas) e simples (ou seja, ele não tem arestas paralelas nem vértices com arestas para si mesmos). Assim, denota-se as arestas de  $G$  como pares de vértices não-ordenados  $(u, z)$ .

No algoritmo de Dijkstra, para determinar caminhos mínimos, a função de custo que se deseja otimizar em nossa aplicação do método guloso também é a função que se deseja avaliar – a distância do caminho mínimo. Isso pode parecer um raciocínio circular até se notar que é possível implementar esta abordagem usando um truque para inicializá-la, que consiste em usar uma aproximação para a função de distância que se deseja calcular e que, ao final do processo, será exatamente igual à distância real.

Hidden page



**Figura 13.14** Execução do algoritmo de Dijkstra em um grafo ponderado. O vértice inicial é BWI. Uma caixa próxima a cada vértice  $v$  armazena o rótulo  $D[v]$ . O símbolo  $\bullet$  é usado no lugar de  $+\infty$ . As arestas da árvore do caminho mínimo são desenhadas como linhas espessas cinzas e para cada vértice  $u$  fora da “nuvem” mostram a melhor aresta atual para  $u$  com uma linha sólida cinza. (Continua na Figura 13.15.)

Hidden page

Figura 13.16.) Sabe-se pela escolha de  $z$ , que  $y$  já está em  $C$  neste momento. Além disso,  $D[y] = d(v,y)$  pois  $u$  é o **primeiro** vértice incorreto. Quando  $y$  foi colocado em  $C$ , testa-se (e possivelmente atualiza-se)  $D[z]$  de modo que se teve naquele momento

$$D[z] \leq D[y] + w((y,z)) = d(v,y) + w((y,z)).$$

Mas já que  $z$  é o próximo vértice no caminho mínimo de  $v$  para  $u$ , isto implica que

$$D[z] = d(v,z).$$

Mas se está agora escolhendo  $u$  e não  $z$  para ser colocado em  $C$ , por isso

$$D[u] \leq D[z].$$

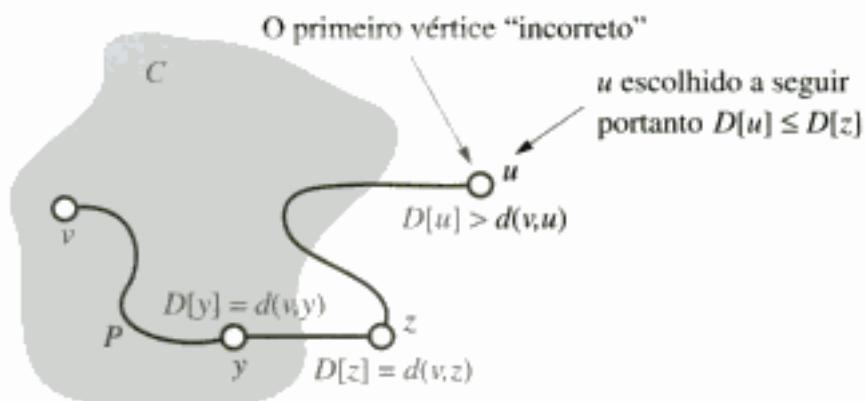
Deve ficar claro que um subcaminho de um caminho mínimo é também um caminho mínimo. Portanto, já que  $z$  está no caminho mínimo de  $v$  até  $u$ ,

$$d(v,z) + d(z,u) = d(v,u).$$

Além disso,  $d(z,u) \geq 0$  porque não há arestas com pesos negativos. Assim,

$$D[u] \leq D[z] = d(v,z) \leq d(v,z) + d(z,u) = d(v,u).$$

Mas isto contradiz a definição de  $u$ , portanto não pode existir tal vértice  $u$ . ■



**Figura 13.16** Uma ilustração esquemática para a justificativa da Proposição 13.23.

### O tempo de execução do algoritmo de Dijkstra

Nesta seção, analisa-se a complexidade do algoritmo de Dijkstra. Denota-se com  $n$  e  $m$  o número de vértices e arestas do grafo  $G$ , respectivamente. Assume-se que os pesos das arestas podem ser somados e comparados em tempo constante. Por causa do alto nível da descrição fornecida para o algoritmo de Dijkstra no Trecho de código 13.14, analisar seu tempo de execução requer que se tenha mais detalhes de sua implementação. Especificamente, deve-se indicar as estruturas de dados usadas e como elas são implementadas.

Assume-se que se está representando o grafo  $G$  com uma lista de adjacência. Esta estrutura de dados permite percorrer os vértices adjacentes a  $u$  durante o passo de relaxamento em tempo proporcional a seu número. Isso ainda não decide todos os detalhes do algoritmo, pois se deve saber mais sobre como se implementa a outra estrutura de dados principal do algoritmo – a fila de prioridade  $Q$ .

Uma implementação eficiente da fila de prioridade  $Q$  usa um heap (ver Seção 8.3). Isso permite extrair o vértice  $u$  com o menor rótulo  $D$  (o método será chamado de `removeMin`) em tempo  $O(\log n)$ . Como dito no pseudocódigo, cada vez que se atualiza o rótulo  $D[z]$  deve-se atualizar a chave de  $z$  na fila de prioridade. Se  $Q$  é implementada como um heap, então essa atualização da chave pode, por exemplo, ser feita retirando e em seguida reinserindo  $z$  com a nova prioridade.

Se a fila de prioridade  $Q$  suporta o padrão de localizadores (ver Seção 8.4), então é possível implementar facilmente essas atualizações de chave em tempo  $O(\log n)$ , pois um localizador para o vértice  $z$  permitiria que  $Q$  tivesse acesso direto ao item armazenando  $z$  no heap (ver Seção 8.4). Assumindo esta implementação de  $Q$ , o algoritmo de Dijkstra é executado em tempo  $O(n + m \log n)$ .

Voltando ao Trecho de código 13.14, os detalhes da análise do tempo de execução são os seguintes:

- A inserção de todos os vértices em  $Q$  com suas chaves iniciais pode ser feita em tempo  $O(n \log n)$  através de inserções repetidas, ou tempo  $O(n)$  usando a construção bottom-up (ver Seção 8.3.6).
- A cada iteração do laço **enquanto** gasta-se tempo  $O(\log n)$  para remover o vértice  $u$  de  $Q$  e tempo  $O(\deg(v) \log n)$  para realizar o relaxamento nas arestas incidentes a  $u$ .
- O tempo total de execução do laço **enquanto** é

$$\sum_{v \in G} (1 + \text{degree}(v)) \log n,$$

que é  $O((n + m) \log n)$  pela Proposição 13.6.

Ao desejar expressar o tempo de execução como uma função de  $n$  apenas, então ela é  $O(n^2 \log n)$  no pior caso.

### Uma implementação alternativa do algoritmo de Dijkstra

Considere uma implementação alternativa para a fila de prioridade  $Q$  usando uma seqüência não-ordenada. Isso, é claro, requer que se gaste tempo  $O(n)$  para retirar o menor elemento, mas permite atualizações de chave muito rápidas desde que  $Q$  suporte o padrão de localizadores (Seção 8.4.2). Especificamente, pode-se implementar cada atualização de chave feita em um passo de relaxamento em tempo  $O(1)$  – simplesmente altera-se o valor da chave depois de localizar o item em  $Q$ . Portanto, essa implementação resulta em um tempo de execução que é  $O(n^2 + m)$ , que pode ser simplificado para  $O(n^2)$  já que  $G$  é simples.

### Comparando as duas implementações

Tem-se duas escolhas para implementar a fila de prioridade no algoritmo de Dijkstra: uma implementação de heap baseada em localizadores, que tem tempo de execução  $O((n + m) \log n)$ , e uma implementação de seqüência não-ordenada baseada em localizadores, que tem tempo de execução  $O(n^2)$ . Já que ambas as implementações seriam relativamente simples de codificar, elas são aproximadamente iguais em termos de sofisticação da programação requerida. Elas também são aproximadamente equivalentes em termos dos fatores constantes em seus tempos de execução de pior caso. Olhando somente para esses tempos de execução de pior caso, prefere-se a implementação baseada em heap quando o número de arestas em um grafo for pequeno (ou seja, quando  $m < n^2 / \log n$ ) e prefere-se a implementação com seqüência quando o número de arestas for grande (ou seja, quando  $m > n^2 / \log n$ ).

**Proposição 13.24** *Dado um grafo simples não-dirigido ponderado  $G$  com  $n$  vértices e  $m$  arestas tal que o peso de cada aresta seja não-negativo e um vértice  $v$  de  $G$ , o algoritmo de Dijkstra determina a distância de  $v$  a todos os outros vértices de  $G$  em tempo  $O((n + m) \log n)$  no pior caso, ou alternativamente em tempo  $O(n^2)$  no pior caso.*

No Exercício R-13.17 explora-se como modificar o algoritmo de Dijkstra para produzir uma árvore  $T$  com raiz em  $v$  tal que o caminho em  $T$  do vértice  $v$  a um vértice  $u$  seja o caminho mínimo em  $G$  de  $v$  para  $u$ .

## Programando o algoritmo de Dijkstra em Java

Tendo fornecido o pseudocódigo do algoritmo de Dijkstra, apresenta-se o código em Java para o algoritmo de Dijkstra pressupondo que se tem um grafo não-dirigido com pesos inteiros positivos. Expressa-se o algoritmo através de uma classe Dijkstra (Trechos de código 13.15–13.16), que declara uma decoração peso para cada aresta  $e$  para acessar o peso da aresta  $e$ . A classe Dijkstra assume que cada aresta tem uma decoração peso.

```

/* Algoritmo de Dijkstra para o problema do menor caminho
 * em um grafo não dirigido cujas arestas têm pesos inteiros não negativos. */
public class Dijkstra<V, E> {
    /** valor infinito. */
    protected static final Integer INFINITE = Integer.MAX_VALUE;
    /** Grafo de entrada. */
    protected Graph<V, E> graph;
    /** Decoração para pesos das arestas */
    protected Object WEIGHT;
    /** Decoração para as distâncias dos vértices */
    protected Object DIST = new Object();
    /** Decoração para os elementos da fila de prioridades */
    protected Object ENTRY = new Object();
    /** Fila de prioridade auxiliar. */
    protected AdaptablePriorityQueue<Integer, Vertex<V>> Q;
    /** Executa o algoritmo de Dijkstra.
     * @param g Grafo de Entrada
     * @param s Vértice
     * @param w Objeto peso */
    public void execute(Graph<V, E> g, Vertex<V> s, Object w) {
        graph = g;
        WEIGHT = w;
        DefaultComparator dc = new DefaultComparator();
        Q = new HeapAdaptablePriorityQueue<Integer, Vertex<V>>(dc);
        dijkstraVisit(s);
    }
    /** Retorna a distância de um vértice a partir do vértice de origem.
     * @param u Vértice inicial da árvore do menor caminho */
    public int getDist(Vertex<V> u) {
        return (Integer) u.get(DIST);
    }
}

```

**Trecho de código 13.15** Classe Dijkstra implementando o algoritmo de Dijkstra. (continua no Trecho de código 13.16.)

A maior tarefa no algoritmo de Dijkstra é realizada pelo método `dijkstraVisit`. É utilizada uma fila de prioridade  $Q$  suportando métodos baseados em localizadores (Seção 8.4.2). Insere-se um vértice  $u$  em  $Q$  com o método `insert`, que retorna o localizador de  $u$  em  $Q$ . Associa-se a  $u$  seu localizador em  $Q$  através do método `setEntry` e recupera-se o localizador de  $u$  através do método `getEntry`. Vide que associar localizadores aos vértices é uma aplicação do padrão de decoradores (Seção 13.3.2). Em vez de usar uma estrutura de dados adicional para os rótulos  $D[u]$ , explora-se o fato de que  $D[u]$  é a chave para o vértice  $u$  em  $Q$  e por isso  $D[u]$  pode ser acessado com o localizador de  $u$  em  $Q$ . Mudar o rótulo de um vértice  $v$  para  $d$  no processo de relaxamento corresponde a chamar o método `replaceKey( $e, d$ )`, onde  $e$  é o localizador de  $v$  em  $Q$ .

```

/** A execução corrente do algoritmo de Dijkstra.
 * @param v vértice.
 */
protected void dijkstraVisit(Vertex<V> v) {
    // armazena todos os vértices em uma fila de prioridades Q
    for (Vertex<V> u: graph.vertices( )) {
        int u_dist;
        if (u==v)
            u_dist = 0;
        else
            u_dist = INFINITE;
        Entry<Integer, Vertex<V>> u_entry = Q.insert(u_dist, u); // autoboxing
        u.put(ENTRY, u_entry);
    }
    // Aumenta a nuvem, um vértice por vez
    while (!Q.isEmpty( )) {
        // remove de Q e insere na nuvem um vértice com distância mínima
        Entry<Integer, Vertex<V>> u_entry = Q.min( );
        Vertex<V> u = u_entry.getValue( );
        int u_dist = u_entry.getKey( );
        Q.remove(u_entry); // remove u da fila de prioridade
        u.put(DIST,u_dist); // a distância de u é final
        u.remove(ENTRY); // remove o elemento decoração de u
        if (u_dist == INFINITE)
            continue; // vértices inalcançáveis não são processados
        // examina todos os vizinhos de u e atualiza suas distâncias
        for (Edge<E> e: graph.incidentEdges(u)) {
            Vertex<V> z = graph.opposite(u,e);
            Entry<Integer, Vertex<V>> z_entry
                = (Entry<Integer, Vertex<V>>) z.get(ENTRY);
            if (z_entry != null) { // verifica que z está em Q, isto é, não está na nuvem.
                int e_weight = (Integer) e.get(WEIGHT);
                int z_dist = z_entry.getKey( );
                if ( u_dist + e_weight < z_dist ) // relaxamento da aresta e = (u,z)
                    Q.replaceKey(z_entry, u_dist + e_weight);
            }
        }
    }
}

```

**Trecho de código 13.16** Método dijkstra da classe Dijkstra. (Continuação do Trecho de código 13.15.)

## 13.7 Árvores de cobertura mínima

Suponha que se deseja conectar todos os computadores em um prédio de escritórios usando a menor quantidade possível de cabos. Pode-se modelar este problema usando um grafo ponderado  $G$  cujos vértices representem os computadores e cujas arestas representem todos os possíveis pares  $(u,v)$  de computadores nos quais o peso  $w(u,v)$  da aresta  $(u,v)$  é igual ao comprimento dos cabos necessários para ligar os computadores  $u$  e  $v$ . Em vez de determinar um caminho mínimo a partir de um dado vértice  $v$ , está-se interessado em encontrar uma árvore (livre)  $T$  que contenha todos os vértices de  $G$  e tenha o peso total mínimo. Os métodos para encontrar essas árvores são o foco desta seção.

### Definição do problema

Dado um grafo não-dirigido ponderado  $G$ , está-se interessado em encontrar uma árvore  $T$  que contenha todos os vértices de  $G$  e minimize a soma

$$w(T) = \sum_{(v,u) \in T} w((v,u)).$$

Uma árvore como esta, que contenha todos os vértices de um grafo conexo  $G$ , é chamada de **árvore de cobertura**, e o problema de encontrar uma árvore de cobertura  $T$  com a menor soma de pesos é conhecido como o problema da **árvore de cobertura mínima (MST)**.

O desenvolvimento de algoritmos eficientes para o problema da árvore de cobertura mínima precede a noção moderna de uma ciência da computação. Nesta seção, serão discutidos dois algoritmos para resolver o problema da MST. Estes algoritmos são aplicações clássicas do **método guloso**, o qual, como foi analisado brevemente na seção anterior, se baseia em escolher objetos para unir a uma coleção crescente escolhendo iterativamente um objeto que minimiza uma dada função de custo. O primeiro algoritmo que será discutido é o algoritmo de Kruskal, que faz a MST “crescer” em grupos considerando as arestas na ordem dada por seus pesos. O segundo algoritmo que será analisado é o algoritmo de Prim-Jarník, que faz a MST crescer a partir de um vértice raiz, de forma semelhante ao algoritmo de Dijkstra para determinação de caminhos mínimos.

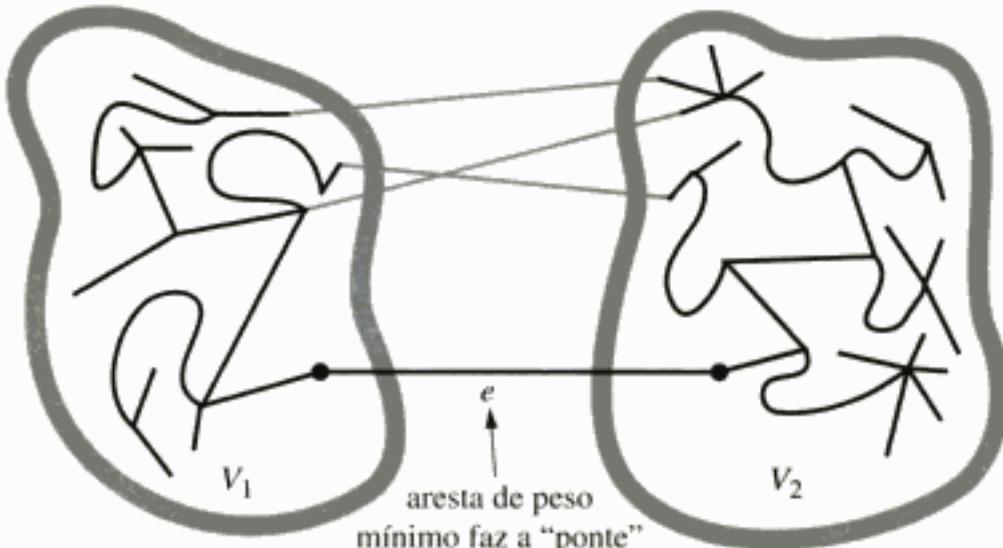
Como na Seção 13.6.1, para simplificar a descrição dos algoritmos, assume-se que o grafo de entrada  $G$  é não-dirigido (ou seja, todas as suas arestas são não-dirigidas) e simples (ou seja, os vértices não são ligados a si mesmos e não a arestas paralelas). Assim, denotam-se as arestas de  $G$  como pares não-ordenados de vértices  $(u,v)$ .

Antes de discutir os detalhes dos algoritmos, no entanto, examina-se um fato crucial sobre as árvores de cobertura mínima que formam a base dos algoritmos.

### Um fato crucial sobre árvores de cobertura mínima

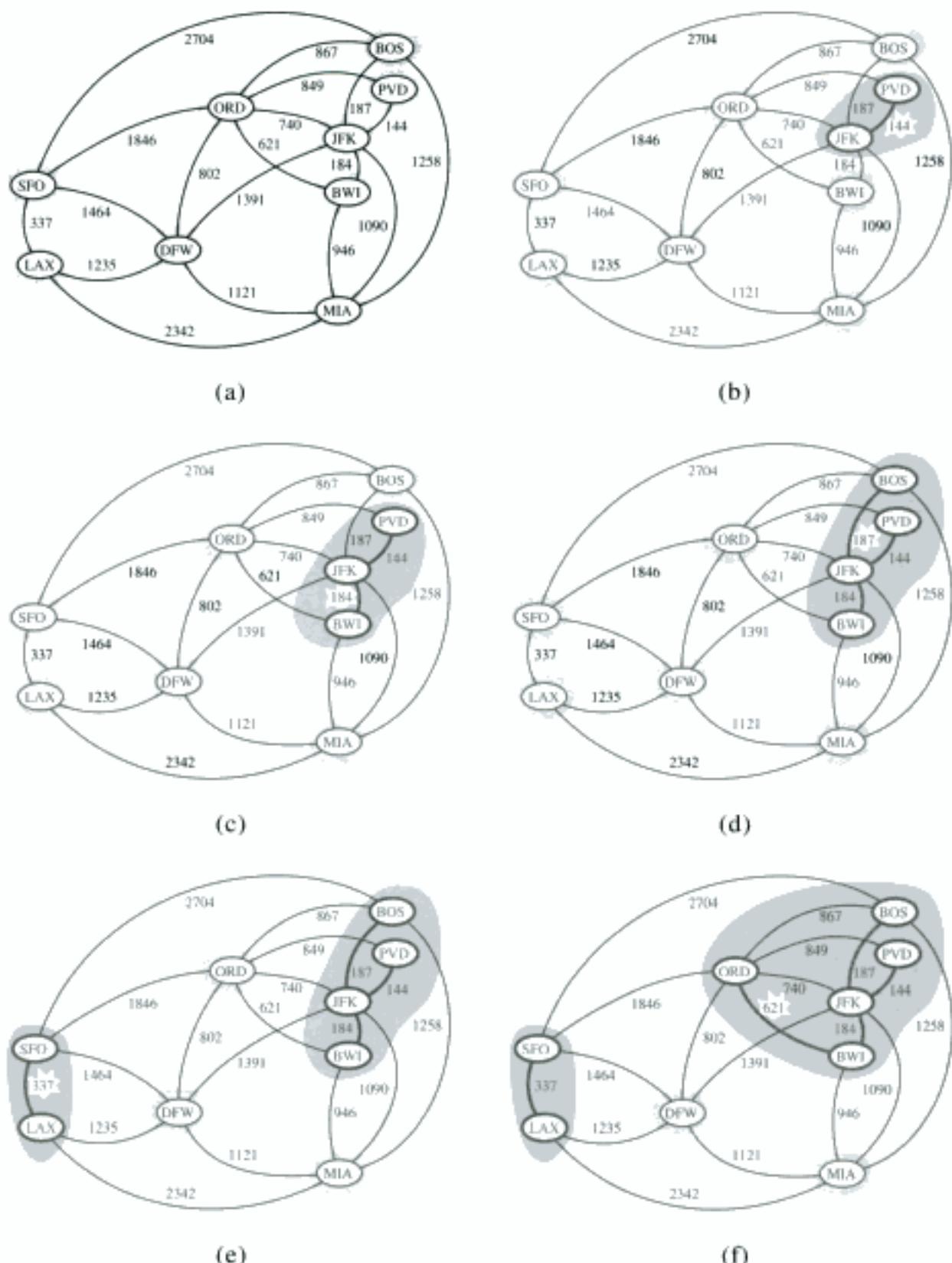
Os dois algoritmos para MST que serão discutidos se baseiam no método guloso, que neste caso depende crucialmente do fato a seguir. (Ver Figura 13.17.)

e pertence a uma árvore de cobertura mínima



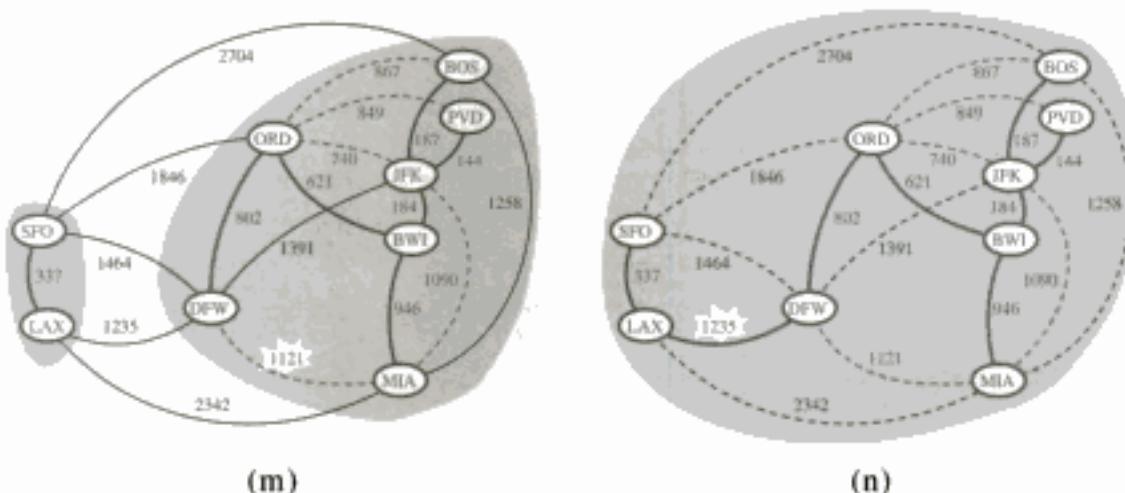
**Figura 13.17** Uma ilustração do fato crucial sobre árvores de cobertura mínima.

Hidden page



**Figura 13.18** Exemplo da execução do algoritmo de Kruskal para MST em um grafo com pesos inteiros. Mostra-se os grupos como regiões sombreadas e salienta-se a aresta sendo examinada a cada iteração. (Continua na Figura 13.19.)

Hidden page



**Figura 13.20** Exemplo da execução do algoritmo de Kruskal para MST (continuação). A aresta considerada em (n) une os dois últimos grupos, o que conclui esta execução do algoritmo de Kruskal. (Continuação da Figura 13.19.)

### O tempo de execução do algoritmo de Kruskal

Nesta seção, será analisado o tempo de execução do algoritmo de Kruskal. Denota-se com  $n$  e  $m$  o número de vértices e arestas do grafo  $G$ , respectivamente. Assume-se que os pesos das arestas podem ser comparados em tempo constante. Por causa do alto nível da descrição fornecido para o algoritmo de Kruskal no Trecho de código 13.17, analisar seu tempo de execução requer que se tenha mais detalhes de sua implementação. Especificamente, deve-se indicar as estruturas de dados usadas e como elas são implementadas.

Implementa-se a fila de prioridade  $Q$  com um heap. Assim, pode-se inicializar  $Q$  em tempo  $O(m \log m)$  através de inserções repetidas, ou em tempo  $O(m)$  usando a construção bottom-up (ver Seção 8.3.6). Adicionalmente, a cada iteração do laço **enquanto** uma aresta de peso mínimo é removida em tempo  $O(\log m)$ , que é na realidade  $O(\log n)$  pois  $G$  é simples. Desta forma, o tempo total gasto para a execução das operações da fila de prioridade não é mais que  $O(m \log n)$ .

Pode-se representar cada grupo  $C$  usando as estruturas de dados de partição união-procura discutida na Seção 11.6.2. Esta estrutura baseada em seqüência permite executar uma série de  $N$  operações union e find no tempo  $O(N \log N)$ , e a versão baseada em árvores pode implementar cada uma das séries de operações no tempo  $O(N \log^* N)$ . Assim, desde que executadas  $n - 1$  chamadas ao método union e no máximo  $m$  chamadas ao método find, o tempo total gasto na união dos grupos e para determinar os grupos que os vértices pertencem não é maior que  $O(m \log n)$  usando uma abordagem baseada em seqüências ou  $O(m \log^* n)$  usando uma abordagem baseada em árvores.

Então, usando argumentos similares a estes para o algoritmo de Dijkstra, conclui-se que o tempo de execução do algoritmo de Kruskal é  $O((n + m) \log n)$ , que pode ser simplificado como  $O(m \log n)$ , desde que  $G$  seja simples e conexo.

### 13.7.2 O Algoritmo Prim-Jarník

No algoritmo de Prim-Jarník, faz-se crescer uma árvore de cobertura mínima a partir de um único grupo iniciando com um vértice “raiz”  $v$ . A idéia principal é similar à do algoritmo de

Dijkstra. Inicia-se com um vértice  $v$ , definindo a “nuvem” inicial de vértices  $C$ . A cada iteração, escolhe-se uma aresta de peso mínimo  $e = (v, u)$  conectando um vértice  $v$  da nuvem  $C$  a um vértice  $u$  fora de  $C$ . O vértice  $u$  é trazido para dentro da nuvem  $C$  e o processo se repete até que uma árvore de cobertura seja formada. De novo, o fato crucial sobre árvores de cobertura mínima entra em ação, pois se sempre for escolhida a aresta de menor peso unindo um vértice de  $C$  com um vértice fora de  $C$ , teremos certeza de estar sempre adicionando uma aresta válida à MST.

Para implementar eficientemente esta abordagem, pode-se usar outra idéia do algoritmo de Dijkstra. Mantém-se um rótulo  $D[u]$  para cada vértice  $u$  fora da nuvem  $C$ , armazenando o peso da melhor aresta atual unindo  $u$  à nuvem  $C$ . Estes rótulos nos permitem reduzir o número de arestas que é necessário analisar para decidir qual vértice deve ser unido à nuvem. O pseudocódigo é fornecido no Trecho de código 13.18.

#### Algoritmo PrimJarnik ( $G$ ):

**Entrada:** um grafo simples, conexo e ponderado  $G$  com  $n$  vértices e  $m$  arestas.

**Saída:** uma árvore de cobertura mínima  $T$  para  $G$

Escolha qualquer vértice  $v$  de  $G$

$D[v] \leftarrow 0$

**para** cada vértice  $u \neq v$  **faça**

$D[u] \leftarrow +\infty$

Inicialize  $T \leftarrow \emptyset$ .

Inicialize uma fila de prioridade  $Q$  com um item  $((u, \text{null}), D[u])$  para cada vértice  $u$  onde  $(u, \text{null})$  é o elemento e  $D[u]$  é a chave.

**enquanto**  $Q$  não está vazia **faça**

$(u, e) \leftarrow Q.\text{removeMin}()$

    Coloque o vértice  $u$  e a aresta  $e$  em  $T$ .

**para** cada vértice  $z$  adjacente a  $u$  tal que  $z$  esteja em  $Q$  **faça**

        [Faça o relaxamento na aresta  $(u, z)$ ]

**se**  $w((u, z)) < D[z]$  **então**

$D[z] \leftarrow w((u, z))$

            Altere para  $(z, (u, z))$  o elemento do vértice  $z$  em  $Q$ .

            Altere para  $D[z]$  a chave do vértice  $z$  em  $Q$ .

**retorna** a árvore  $T$

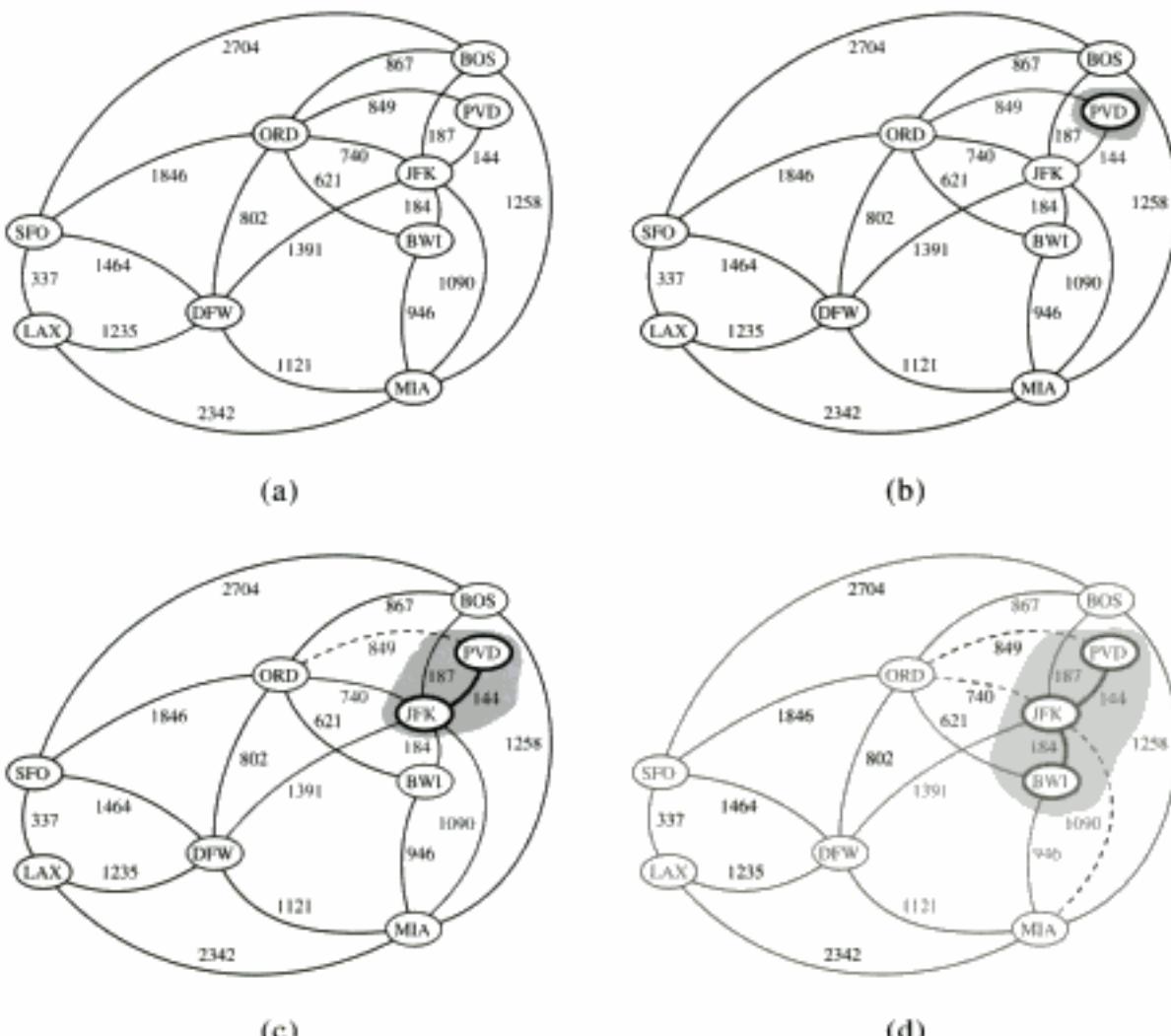
**Trecho de código 13.18** O algoritmo de Prim-Jarník para o problema da MST.

#### Analizando o algoritmo Prim-Jarník

Sejam  $n$  e  $m$  o número de vértices e arestas do grafo de entrada  $G$ . A implementação do algoritmo de Prim-Jarník tem detalhes similares ao algoritmo de Dijkstra. Implementando a fila de prioridade  $Q$  como um heap que suporta os métodos baseados em localizadores (ver Seção 8.4.2), é possível extrair o vértice  $u$  a cada iteração em tempo  $O(\log n)$ . Além disso, pode-se atualizar cada valor de  $D[z]$  em tempo  $O(\log n)$ , o que é feito no máximo uma vez para cada aresta  $(u, z)$ . Os outros passos de cada iteração podem ser implementados em tempo constante. Assim, o tempo de execução total do algoritmo é  $O((n + m) \log n)$ , que é  $O(m \log n)$ .

#### Ilustrando o algoritmo Prim-Jarník

O algoritmo Prim-Jarník é ilustrado nas Figuras 13.21 a 13.22.



**Figura 13.21** Uma ilustração do algoritmo MST Prim-Jarník. (Continua na Figura 13.22.)

## 13.8 Exercícios

Para obter o código fonte e auxílio com os exercícios, visite [java.datastructures.net](http://java.datastructures.net)

### Reforço

- R-13.1 Desenhe um grafo simples não-dirigido  $G$  com 12 vértices, 18 arestas e 3 componentes conexos. Por que seria impossível desenhar  $G$  com 3 componentes conexos se  $G$  tivesse 66 arestas?
- R-13.2 Seja  $G$  um grafo simples conexo com  $n$  vértices e  $m$  arestas. Explique por que  $O(\log m)$  é  $O(\log n)$ .
- R-13.3 Desenhe uma representação de lista de adjacências e uma de matriz de adjacência do grafo não-dirigido mostrado na Figura 13.1.
- R-13.4 Desenhe um grafo simples conexo e dirigido  $G$  com 8 vértices e 16 arestas de forma que o grau de entrada e de saída de cada vértice seja 2. Mostre que existe um único ciclo (não-simples) que inclui todas as arestas do grafo, ou seja, que você pode desenhar todas as arestas em suas direções respectivas sem levantar o lápis do papel (este tipo de ciclo é chamado de *ciclo de Euler*).

Hidden page

Hidden page

Hidden page

Hidden page

- C-13.5 Suponha que se deseja representar um grafo de  $n$  vértices usando uma lista de arestas, assumindo que os vértices com os inteiros do conjunto  $\{0, 1, \dots, n-1\}$  sejam identificados. Descreva como implementar o container  $E$  para suportar desempenho  $O(\log n)$  para o método `areAdjacent`. Como você vai implementar o método neste caso?
- C-13.6 A Universidade Tamarindo e várias outras escolas ao redor do mundo estão envolvidas em um projeto multimídia. Uma rede de computadores é montada para conectar essas escolas usando linhas de comunicação que formam uma árvore livre. As escolas decidem instalar um servidor de arquivos em uma das escolas para compartilhar dados entre todas elas. Como o tempo de transmissão em uma conexão é dominado por sua inicialização e sincronização, o custo da transferência de dados é proporcional ao número de conexões usadas. Assim, é desejável escolher uma escola “central” para o servidor. Dada uma árvore livre  $T$  e um nodo  $v$  de  $T$ , a *excentricidade* de  $v$  é o comprimento do maior caminho de  $v$  a qualquer outro nodo de  $T$ . Um nodo de  $T$  com a menor excentricidade é chamado de *centro* de  $T$ .
  - Projete um algoritmo eficiente que, dada uma árvore livre  $T$  com  $n$  nodos, determina seu centro.
  - O centro é único? Se não for, quantos centros diferentes uma árvore livre pode ter?
- C-13.7 Mostre que se  $T$  é uma árvore produzida por caminhamento em largura para um grafo conexo  $G$ , então, para cada vértice  $v$  no nível  $i$ , o caminho de  $T$  entre  $s$  e  $v$  tem  $i$  arestas, e qualquer outro caminho de  $G$  entre  $s$  e  $v$  tem pelo menos  $i$  arestas.
- C-13.8 O atraso em uma chamada de longa distância pode ser determinado multiplicando-se uma pequena constante pelo número de conexões telefônicas entre os pontos sendo ligados. Suponha que a rede telefônica da companhia RT&T é uma árvore livre. Os engenheiros da RT&T desejam avaliar o maior atraso possível em uma chamada de longa distância. Dada uma árvore livre  $T$ , o *diâmetro* de  $T$  é o comprimento do caminho mais longo entre dois nodos de  $T$ . Forneça um algoritmo eficiente para determinar o diâmetro de  $T$ .
- C-13.9 Uma companhia chamada RT&T tem uma rede de  $n$  estações telefônicas conectadas por  $m$  linhas de alta velocidade. O telefone de cada cliente é conectado diretamente a uma estação em sua área. Os engenheiros da RT&T desenvolveram um protótipo de videofone que permite que dois clientes vejam um ao outro durante uma chamada. Para ter uma imagem de qualidade aceitável, no entanto, o número de conexões usado para transmitir os sinais de vídeo entre as partes não pode exceder 4. Suponha que a rede da RT&T é representada por um grafo. Planeje um algoritmo eficiente que determina, para cada estação, o conjunto de estações que podem ser alcançadas com 4 conexões ou menos.
- C-13.10 Explique por que não existem arestas de descoberta fora da árvore produzida por um caminhamento em largura construído para um grafo dirigido.
- C-13.11 Um *ciclo de Euler* de um grafo dirigido  $\tilde{G}$  com  $n$  vértices e  $m$  arestas é um ciclo que passa por cada aresta de  $\tilde{G}$  exatamente uma vez de acordo com sua direção. Um ciclo deste tipo sempre existe se  $\tilde{G}$  for conexo e o grau de entrada for igual ao grau de saída para cada vértice em  $\tilde{G}$ . Descreva um algoritmo de tempo  $O(n + m)$  para achar um ciclo de Euler em um dígrafo  $\tilde{G}$ .

Hidden page

- C-13.21 Redes de computadores devem evitar pontos de falha, isto é, nodos da rede que podem desconectar a rede se eles falharem. Diz-se que um grafo conexo é **biconexo** se ele não contém vértices que removidos dividiriam  $G$  em dois ou mais componentes conexos. Apresente um algoritmo que execute no tempo  $O(n + m)$  para adicionar no máximo  $n$  arestas a um grafo conexo  $G$  com  $n \geq 3$  vértices e  $m \geq n - 1$  arestas, para garantir que  $G$  seja biconexo.
- C-13.22 A NASA deseja interligar  $n$  estações espalhadas nos Estados Unidos usando canais de comunicação. Cada par de estações tem uma capacidade de transmissão de mensagens diferente, que são conhecidas de antemão. A NASA deseja escolher  $n - 1$  canais (o mínimo possível) de tal forma que todas as estações estejam ligadas pelos canais de transmissão e a capacidade de transmissão total seja máxima. Forneça um algoritmo eficiente para este problema e determine sua complexidade de pior caso. Considere o grafo ponderado  $G = (V, E)$  onde  $V$  é o conjunto de estações e  $E$  é o conjunto de canais entre as estações. Defina o peso  $w(e)$  de uma aresta  $e \in E$  como a capacidade de transmissão do canal correspondente.
- C-13.23 Suponha que você receba uma *tabela de horários* que consiste em:
- um conjunto  $\mathcal{A}$  de  $n$  aeroportos, e para cada aeroporto  $a \in \mathcal{A}$  um tempo mínimo de conexão  $c(a)$ ;
  - um conjunto  $\mathcal{F}$  de  $m$  vôos e para cada vôo  $f \in \mathcal{F}$ :
    - um aeroporto de origem  $a_1(f) \in \mathcal{A}$ ;
    - um aeroporto de destino  $a_2(f) \in \mathcal{A}$ ;
    - hora de saída  $t_1(f)$ ;
    - hora de chegada  $t_2(f)$ .
- Descreva um algoritmo eficiente para o problema do escalonamento dos vôos. Neste problema, recebemos os aeroportos  $a$  e  $b$ , o tempo  $t$  e desejamos calcular a seqüência de vôos que nos permite chegar o mais rápido possível em  $b$  saindo de  $a$  no tempo  $t$  ou mais tarde. O tempo mínimo de conexão nos aeroportos intermediários deve ser observado. Qual o tempo de execução de seu algoritmo em função de  $n$  e  $m$ ?
- C-13.24 No interior do Castelo de Asymptopia existe um labirinto, e em cada passagem do labirinto há uma sacola de moedas de ouro. A quantidade de ouro em cada sacola varia. Você terá a oportunidade de caminhar no labirinto recolhendo sacolas, entrando pela porta marcada “Entrada” e saindo pela porta marcada “Saída” (são portas diferentes). Quando estiver no labirinto, você não poderá voltar em seu caminho. Cada corredor do labirinto tem uma seta pintada na parede, e você só poderá andar seguindo a direção das setas. Não existe maneira de fazer uma “volta” no labirinto. Dado um mapa do labirinto, incluindo as quantidades de ouro e as direções dos corredores, descreva um algoritmo para ajudá-lo a recolher o máximo de ouro.
- C-13.25 Seja  $\tilde{G}$  um dígrafo ponderado com  $n$  vértices. Proponha uma variação do algoritmo de Floyd-Warshall para determinar os comprimentos dos caminhos mínimos de cada vértice a cada outro vértice. Seu algoritmo deve ser executado em tempo  $O(n^3)$ .
- C-13.26 Suponha que recebemos um grafo dirigido  $\tilde{G}$  com  $n$  vértices e seja  $M$  a matriz de adjacência  $n \times n$  correspondente a  $\tilde{G}$ .

- a. Seja o produto de  $M$  consigo mesma ( $M^2$ ) definido, para  $1 \leq i, j \leq n$ , como segue:

$$M^2(i, j) = M(i, 1) \odot M(1, j) \oplus \dots \oplus M(i, n) M(n, j),$$

onde “ $\oplus$ ” é o operador booleano **or** e “ $\odot$ ” é o operador booleano **and**. Dada esta definição, o que é que  $M^2(i, j) = 1$  informa sobre os vértices  $i$  e  $j$ ? E se  $M^2(i, j) = 0$ ?

- b. Suponha que  $M^4$  é o produto de  $M^2$  consigo mesma. O que representam as entradas de  $M^4$ ? E as entradas de  $M^8 = (M^4)(M)$ ? Em geral, que informação está contida na matriz  $M^8$ ?

- c. Suponha que é ponderado e assuma o seguinte:

1. para  $1 \leq i \leq n$ ,  $M(i, i) = 0$ .
2. para  $1 \leq i, j \leq n$ ,  $M(i, j) = w(i, j)$  se  $(i, j) \in E$ .
3. para  $1 \leq i, j \leq n$ ,  $M(i, j) = \infty$  se  $(i, j) \notin E$ .

Também defina  $M^2$  para  $1 \leq i, j \leq n$ , como segue:

$$M^2(i, j) = \min\{M(i, j) + M(1, j), \dots, M(i, n) + M(n, j)\}.$$

Se  $M^2(i, j) = k$ , o que se pode concluir sobre a relação entre os vértices  $i$  e  $j$ ?

- C-13.27 Um grafo  $G$  é **bipartido** se seus vértices podem ser divididos em dois conjuntos  $S$  e  $Y$  sendo que toda aresta de  $G$  tem um vértice final em  $X$  e outro em  $Y$ . Projete e analise um algoritmo eficiente para determinar se um grafo não-dirigido  $G$  é bipartido (sem ter o conhecimento dos conjuntos  $X$  e  $Y$ ).

- C-13.28 Um método MST antigo, chamado *Algoritmo de Baruvka*, trabalha sobre um grafo  $G$  com  $n$  vértices e  $m$  arestas com pesos distintos:

Seja  $T$  um subgrafo de  $G$  inicialmente contendo somente os vértices de  $V$ .  
**enquanto**  $T$  tem menos que  $n - 1$  arestas **faça**  
**para** cada componente conexo  $C_i$  de  $T$  **faça**

Procure a aresta com menor peso  $(v, u)$  de  $E$  com  $v \in C_i$  e  $u \in C_j$ .

Adicione  $(v, u)$  em  $T$  (a menos que ele já esteja em  $T$ ).

**retorne**  $T$

Argumente porque este algoritmo está correto e porque ele executa no tempo  $O(m \log n)$ .

- C-13.29 Seja  $G$  um grafo com  $n$  vértices e  $m$  arestas sendo que todos os pesos das arestas em  $G$  sejam inteiros no intervalo  $[1, n]$ . Apresente um algoritmo para procurar as árvores de cobertura mínimas de  $G$  no tempo  $O(m \log^* n)$ .

## Projetos

- P-13.1 Escreva uma classe implementando um TAD simplificado para grafos que têm os métodos relevantes para grafos não-dirigidos e que não inclui métodos de atualização, usando uma matriz de adjacência. Sua classe deve incluir um método construtor que recebe duas coleções (por exemplo, sequências) — uma coleção  $V$  de vértices e uma coleção  $E$  de pares de vértices — e produz o grafo  $G$  que estas duas coleções representam.
- P-13.2 Implemente o TAD simplificado descrito no Projeto P-13.1 usando uma lista de adjacências.

- P-13.3 Implemente o TAD simplificado descrito no Projeto P-13.1 usando uma lista de arestas.
- P-13.4 Estenda a classe do Projeto P-13.2 para suportar métodos de atualização.
- P-13.5 Estenda a classe do Projeto P-13.2 para suportar todos os métodos do TAD grafo (incluindo métodos para arestas dirigidas).
- P-13.6 Implemente um caminhamento em largura genérico usando o padrão de templates.
- P-13.7 Implemente o algoritmo de ordenação topológica.
- P-13.8 Implemente o algoritmo de Floyd-Warshall para o fechamento transitivo.
- P-13.9 Planeje uma comparação experimental de vários caminhamentos em profundidade em relação ao algoritmo de Floyd-Warshall para determinar o fechamento transitivo de um dígrafo.
- P-13.10 Implemente o algoritmo de Kruskal assumindo que os pesos das arestas sejam inteiros.
- P-13.11 Implemente o algoritmo de Prim-Jarník assumindo que os pesos das arestas sejam inteiros.
- P-13.12 Realize uma comparação experimental de dois dos algoritmos para MST discutidos neste capítulo (Kruskal e Prim-Jarník). Desenvolva um conjunto de experimentos para testar os tempos de execução dos algoritmos usando grafos gerados aleatoriamente.
- P-13.13 Uma forma de construir um *labirinto* inicial com uma matriz de  $n \times n$ , sendo que cada célula da matriz é cercada por quatro paredes de tamanho único. Então removemos duas paredes de tamanho único para representar o inicio e o final. Para cada parede restante que não seja fronteira, definimos um valor randômico e criamos um grafo  $G$ , chamado de *dual*, sendo que cada célula seja um vértice em  $G$  e exista uma aresta ligando os vértices de duas células se e somente se as células compartilharem uma parede em comum. Construímos o labirinto pela procura de uma árvore de cobertura mínima  $T$  em  $G$  e removemos todas as paredes correspondentes as arestas de  $T$ . Escreva um programa que use este algoritmo para gerar labirintos e então solucione-os. De forma resumida, seu programa deverá desenhar o labirinto e, idealmente, ele deverá visualizar a solução do mesmo.
- P-13.14 Escreva um programa que crie as tabelas de roteamento para os nodos de uma rede de computadores, baseado na rota do menor caminho, onde a distância é medida pelo contador de saltos, isto é, o número de aresta em um caminho. A entrada deste problema é a informação de conectividade para todos os nodos em uma rede, como no exemplo a seguir:

241.12.31.14: 241.12.31.15 241.12.31.18 241.12.31.19

que indica três nodos de rede que estão conectados a 241.12.31.14, isto é, três nodos que estão um salto adiante. A tabela de roteamento para o nodo no endereço  $A$  é o conjunto de pares  $(B,C)$ , que indicam que para transferir uma mensagem de  $A$  para  $B$ , o próximo nodo para enviar (no menor caminho entre  $A$  e  $B$ ) é o  $C$ . Seu programa deverá ter como saída a tabela de roteamento para cada nodo em uma rede, dada como entrada a lista de conectividade de nodos, cada qual é a entrada com a sintaxe apresentada acima, uma por linha.

## Observações sobre o capítulo

O método de caminhamento em profundidade é parte do folclore da computação, mas Hopcroft e Tarjan [50, 90] foram aqueles que mostraram o quanto esse algoritmo é útil para resolver vários problemas diferentes de grafos. Knuth [62] discute o problema da ordenação topológica. O algoritmo simples de tempo linear que foi descrito para determinar se um grafo dirigido é fortemente conexo é de Kosaraju. O algoritmo de Floyd-Warshall é descrito em um artigo de Floyd [35], e se baseia em um teorema de Warshall [98]. O método de coleta de lixo por marcação e varredura é um dos muitos algoritmos diferentes para coleta de lixo. Encoraja-se o leitor interessado no estudo da coleta de lixo a consultar o livro de Jones [55]. Para aprender sobre diferentes métodos para desenho de grafos veja o capítulo de Tamassia [88], a bibliografia comentada de Di Battista *et al.* [29] ou o livro de Di Battista *et al.* [30]. O primeiro algoritmo conhecido para a árvore de cobertura mínima é de Baruvka [9] e foi publicado em 1926. O algoritmo de Prim-Jarník foi primeiro publicado em tcheco por Jarník [54] em 1930 e em inglês em 1957 por Prim [82]. Kruskal publicou seu algoritmo para a árvore de cobertura mínima em 1956 [65]. O leitor interessado em mais estudo sobre o problema da árvore de cobertura mínima pode consultar o artigo de Graham e Hell [45]. O algoritmo assintoticamente mais rápido até o momento para a árvore de cobertura mínima é um método randomizado de Karger, Klein e Tarjan [56] que tem tempo esperado  $O(m)$ .

Dijkstra [31] publicou seu algoritmo para caminho mínimo com origem única em 1959. O leitor interessado em um estudo mais profundo sobre grafos pode consultar os livros de Ahuja, Magnanti e Orlin [6], Cormen, Leiserson e Rivest [25], Even [33], Gibbons [39], Mehlhorn [75], Tarjan [91] e o capítulo de van Leeuwen [94]. O tempo de execução para os algoritmos de Prim-Jarník e de Dijkstra pode ser melhorado até tornar-se  $O(n \log(n + m))$  implementando-se a fila de prioridade  $Q$  com uma de duas estruturas de dados mais sofisticadas, o “Fibonacci Heap” [37] ou o “Relaxed Heap” [32].

# Capítulo

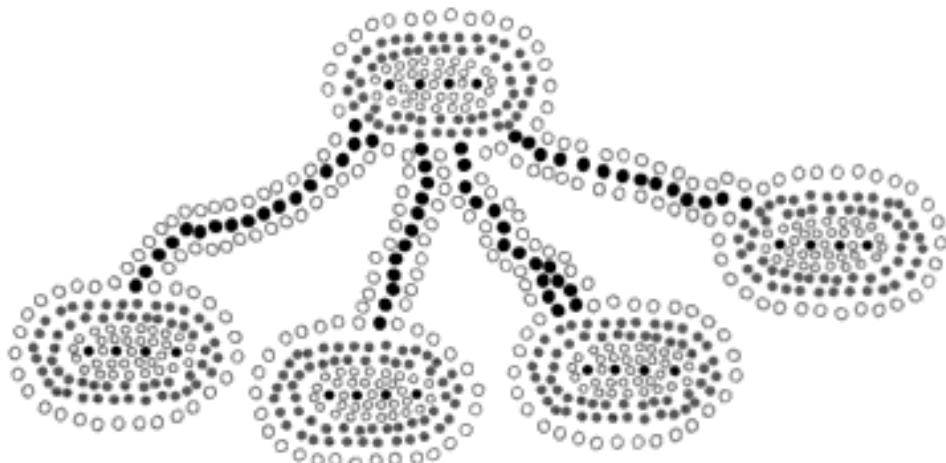
---

# 14

---

# Memória

---



## Conteúdo

---

<b>14.1</b>	<b>Gerenciamento de memória</b>	<b>568</b>
14.1.1	Pilhas na máquina virtual de Java	568
14.1.2	Alocando espaço na memória heap	571
14.1.3	Coleta de lixo	572
<b>14.2</b>	<b>Memória externa e caching</b>	<b>574</b>
14.2.1	A hierarquia de memória	574
14.2.2	Estratégias de cache	575
<b>14.3</b>	<b>Pesquisa externa e árvores-B</b>	<b>579</b>
14.3.1	Árvores $(a,b)$	580
14.3.2	Árvores-B	581
<b>14.4</b>	<b>Ordenando memória externa</b>	<b>582</b>
14.4.1	Merge genérico	583
<b>14.5</b>	<b>Exercícios</b>	<b>584</b>

## 14.1 Gerenciamento de Memória

Para implementar qualquer estrutura de dados em um computador real, precisa-se usar a memória do computador. Memória do computador é simplesmente uma sequências de *palavras* da memória, e cada qual consiste em 4, 8 ou 16 bytes (dependendo do computador). Estas palavras da memória são enumeradas de 0 a  $N - 1$ , onde  $N$  é o número de palavras de memória disponíveis no computador. O número associado com cada memória de computador é conhecido como *endereço*. Assim, a memória em um computador pode ser visualizada como basicamente um arranjo gigante de palavras de memória. Usando esta memória para construir estruturas de dados (e execução de programas) requer que se *gerencie* a memória do computador para prover o espaço necessário para os dados – incluindo variáveis, nodos, apontadores, arranjos e cadeia de caracteres – e para os programas executarem. O básico do gerenciamento de memória será discutido nesta seção.

### 14.1.1 Pilhas na máquina virtual de Java

Um programa Java é tipicamente compilado em uma sequência de códigos byte que são definidos como instruções de “máquina” para um modelo bem formado – a *máquina virtual Java (JVM)*. A definição da JVM é o coração da definição da linguagem Java. Pela compilação do código Java no código de bytes da JVM, preferencialmente na linguagem de máquina de uma CPU específica, um programa Java pode ser executado em qualquer computador, assim como um computador pessoal ou um servidor, que tem um programa que pode emular a JVM. De forma interessante, a estrutura de dados pilha tem um papel central na definição da JVM.

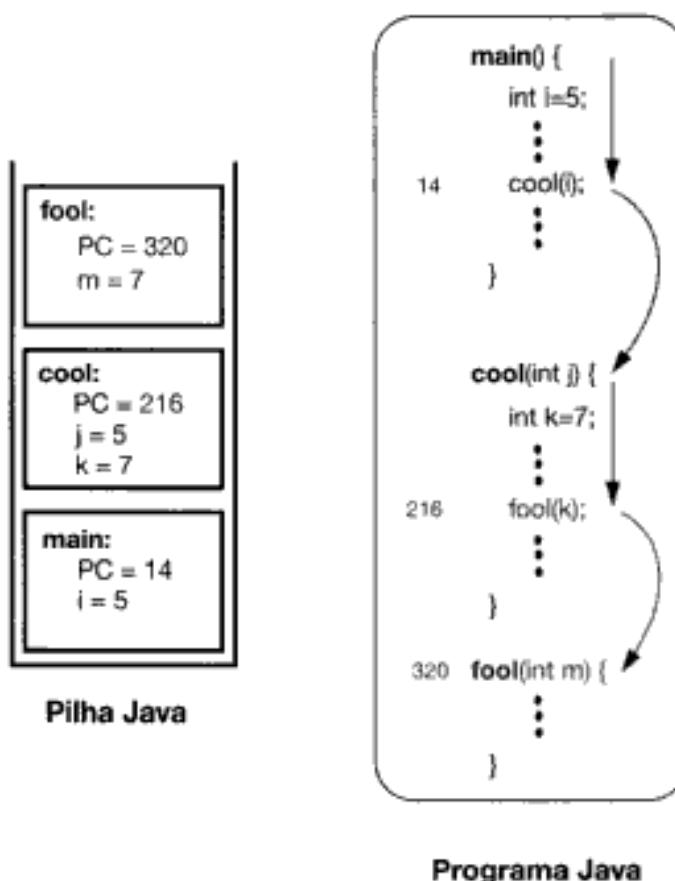
#### A pilha de métodos Java

Pilhas têm uma importante aplicação no ambiente de execução de programas Java. Uma execução de programa Java (mais precisamente, uma execução de uma thread Java) tem uma pilha privada, chamada *pilha de métodos Java*, ou simplesmente *pilha Java*, que é usado para manter a trilha das variáveis locais e outras informações importantes dos métodos como eles são invocados durante a execução. (Ver Figura 14.1.)

Mais especificamente, durante a execução de um programa Java, a máquina virtual Java (JVM) mantém uma pilha cujos elementos são descritores das invocações dos métodos correntes (isto é, não-finalizadas). Estes descritores são chamados *frames*. Um frame para alguma invocação de um método “fool” armazena os valores correntes das variáveis locais e os parâmetros do método fool, bem como as informações do método “cool” que chamou fool e o que necessita ser retornado pelo método “cool”.

#### Mantendo a trilha do contador do programa

A JVM mantém uma variável especial, chamada de *contador do programa*, para manter o endereço do comando que a JVM está executando no momento em um programa. Quando um método “cool” invoca outro método “fool”, o valor corrente do contador do programa é armazenado no frame da invocação corrente de cool (assim a JVM saberá onde retornar quando o método fool for finalizado). No topo da pilha Java está o frame do *método de execução*, isto é, o método que tem o controle da execução. Os elementos restantes da pilha são frames dos *métodos suspensos*, ou seja, métodos que tem invocado outro método e estão esperando para retornar o controle para suas finalizações. A ordem dos elementos na pilha correspondem à cadeia de invocações dos métodos atualmente ativos. Quando um novo método é invocado, um frame para este método é empilhado na pilha. Quando ele termina, seu frame é desempilhado e a JVM retoma o processamento do método anteriormente suspenso.



**Figura 14.1** Um exemplo de uma pilha de métodos Java: método `fool` chamado pelo método `cool` que anteriormente foi invocado pelo método `main`. Observe os valores do contador do programa, parâmetros e variáveis locais armazenadas na pilha de frames. Quando a invocação do método `fool` termina, a invocação do método `cool` será retomada na execução da instrução 217, que é obtida pelo incremento do valor do contador do programa armazenado na pilha de frames.

### Entendendo a passagem de parâmetros por valor

A JVM usa a pilha Java para executar a passagem de parâmetros nos métodos. Especificamente, Java usa o protocolo **passagem por valor** (*call-by-value*). Isso significa que o *valor* corrente de uma variável (ou expressão) é que é passado como um argumento para uma chamada de método.

No caso de uma variável *x* de um tipo primitivo, como um `int` ou `float`, o valor corrente de *x* é simplesmente um número que é associado a *x*. Quando um valor é passado para a chamada do método, ele é assinalado para uma variável local no frame da chamada do método. (Esta simples atribuição também é ilustrada na Figura 14.1.) Se a chamada ao método altera o valor desta variável local, ela *não* alterará o valor da variável na chamada do método.

Entretanto, no caso da uma variável *x* que refere a um objeto o valor corrente de *x* é o endereço de memória do objeto *x*. (Esse assunto será abordado mais profundamente na Seção 14.1.2.) Desta forma, quando o objeto *x* é passado como um parâmetro para algum método, o endereço de *x* é realmente passado. Quando este endereço é atribuído a alguma variável local *y* na chamada do método, *y* referenciará ao mesmo objeto que *x* se refere.

Então, se a chamada ao método altera o estado interno do objeto que *y* se refere, ele simultaneamente será alterado no objeto que *x* se refere (que é o mesmo objeto). Apesar disso, se a chamada do programa altera *y* para se referenciar a outro objeto, *x* continuará inalterado – ele continuará se referindo ao mesmo objeto que ele se referenciava anteriormente.

Desta forma, a pilha de métodos Java é usado pela JVM para implementar chamadas aos métodos e passagem de parâmetros. Incidentalmente, pilhas de métodos não é uma característica

específica do Java. Elas são usadas os ambientes de execução das mais modernas linguagens de programação, incluindo C e C++.

### A pilha de operandos

De forma interessante, existe realmente outro local onde a JVM usa uma pilha. Expressões aritméticas, como  $((a + b) * (c + d))/e$ , são avaliadas pela JVM usando uma pilha de operandos. Uma simples operação binária, como  $a + b$ , é computada pelo empilhamento de  $a$ , empilhamento de  $b$  e então a chamada a uma instrução que desempilha do topo dois itens, executa a operação binária sobre eles e empilha o resultado. Da mesma forma, instruções para escrever e ler elementos na memória envolve o uso dos métodos `pop` e `push` para o operando pilha. Assim, a JVM usa uma pilha para avaliar expressões matemáticas em Java.

Na Seção 7.3.6, foi descrito como avaliar uma expressão matemática usando um caminhamento posfixado, que é exatamente o algoritmo que a JVM usa. Foi descrito que o algoritmo em uma forma recursiva não é uma forma explícita do uso do operando pilha. Todavia, esta descrição recursiva é equivalente a versão não-recursiva baseada no uso de um operando pilha.

### Implementando a recursão

Um dos benefícios do uso de uma pilha para implementar a invocação de métodos é que ela permite que os programas utilizem **recursão**. Isto é, ela permite que um método possa chamar a si mesmo, como discutido na Seção 3.5. De forma interessante, antigas linguagens de programação, como Cobol e Fortran, originalmente não usavam pilhas de execução para implementar chamada a métodos e procedimentos. Porém, por causa da elegância e eficiência que a recursão permite, todas as linguagens de programação modernas, incluindo versões modernas de linguagens clássicas como Cobol e Fortran, utilizam uma pilha de execução para chamada a métodos e procedimentos.

Na execução de um método recursivo, cada caixa de marca de recursão corresponde a um frame da pilha de métodos Java. Também, o conteúdo da pilha de métodos Java corresponde a cadeia de caixas de uma invocação inicial de método a invocação corrente.

Para melhor ilustrar como uma pilha de execução permite métodos recursivos, considera-se uma implementação Java de uma definição clássica recursiva da função factorial.

$$n! = n(n - 1)(n - 2) \cdots 1,$$

como mostrado no Trecho de código 14.1.

```
public static long factorial(long n) {
    if (n <= 1)
        return 1;
    else
        return n * factorial(n - 1);
}
```

**Trecho de código 14.1** Método recursivo para o factorial.

A primeira vez que se chama o método `factorial`, sua pilha se ajusta para incluir uma variável local para armazenar o valor  $n$ . O método `factorial()` chama recursivamente ele próprio para calcular  $(n - 1)!$ , que empilha um novo frame na pilha de execução Java. Um após o outro, estas chamadas recursivas calculam  $(n - 2)!$ , etc. A cadeia de invocações recursivas e desta forma a pilha de execução, somente cresce para o tamanho  $n$ , porque a chamada a `factorial(1)` retorna 1 imediatamente sem invocar ele próprio recursivamente. A pilha de execução permite que o método `factorial()` exista simultaneamente em vários frames ativos (no máximo  $n$  vezes). Cada

frame armazena o valor do seu parâmetro  $n$  bem como o valor retornado. Eventualmente, quando a primeira chamada recursiva termina, ela retorna  $(n - 1)!$ , que é então multiplicado por  $n$  para calcular  $n!$  da chamada original do método factorial.

### 14.1.2 Alocando espaço na memória heap

Tem-se realmente discutido (na Seção 14.1.1) como a máquina virtual Java aloca espaços variáveis locais de métodos nos frames dos métodos na pilha de execução Java. Entretanto, a pilha Java não é somente o tipo de memória disponível para dados do programa em Java.

#### Alocação de memória dinâmica

Memória para um objeto pode também ser alocado dinamicamente durante uma execução de método, tendo o método utilizando um novo operador criado em Java. Por exemplo, a seguinte comando Java cria um arranjo de inteiros cujo tamanho é dado pelo valor da variável  $k$ :

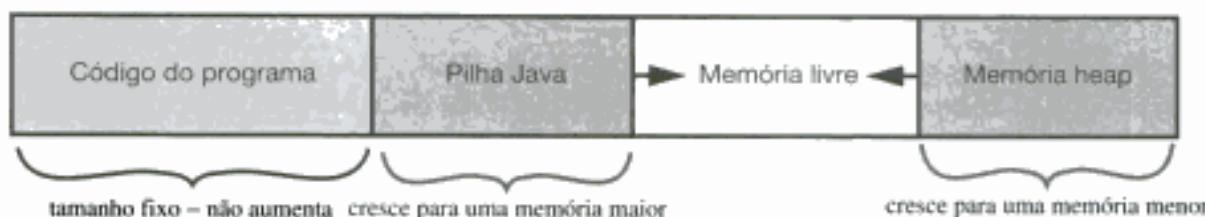
```
int[ ] items = new int[k];
```

O tamanho do arranjo acima é conhecido somente em tempo de execução. Além disso, o arranjo pode continuar a existir mesmo depois do método que o criou terminar. Assim, a memória para este arranjo não pode ser alocada na pilha Java.

#### A memória heap

Em vez de usar a pilha Java para este objeto, Java usa a memória de outra área de armazenamento – a **memória heap** (que não deve ser confundida com a estrutura de dados “heap” apresentada no Capítulo 8). Essa e outras áreas de memória estão ilustradas em uma máquina virtual Java na Figura 14.2. O espaço disponível na memória heap é dividido em **blocos**, que são “pedaços” contíguos de memória como arranjos que podem ter tamanho variável ou fixo.

Para simplificar a discussão, assume-se que os blocos na memória heap são de tamanho fixo, por exemplo, 1.024 bytes, grandes o suficiente para comportar qualquer objeto que se queira criar. (Eficientemente tratando o caso mais geral é realmente um problema de pesquisa interessante.)



**Figura 14.2** Uma visão esquemática do leiaute dos endereços de memória na máquina virtual Java.

#### Algoritmos de alocação de memória

A definição da máquina virtual Java requer que a memória heap esteja disponível para alocar memória rapidamente para novos objetos, porém ela não especifica a estrutura de dados que se deve utilizar para fazer isto. Um método popular é manter “porções” contíguas de memória livre disponível em uma lista duplamente encadeada, chamada **lista livre**. As conexões que unem estas “porções” são armazenadas nas próprias porções, desde que sua memória não esteja sendo usada. À medida que a memória é alocada e desalocada, o conjunto de porções na lista é alterado, e a

memória não utilizada é compartimentada em porções separadas por blocos de memória usada. Esta separação de memória não usada em porções separadas é conhecida como *fragmentação*. Claro que se gostaria de minimizar a fragmentação o máximo possível.

Existem dois tipos de fragmentação que podem ocorrer. *Fragmentação interna* ocorre quando uma porção de um bloco de memória alocado não está realmente sendo usado. Por exemplo, um programa pode requisitar um arranjo de tamanho 1000, porém somente usa as primeiras 100 posições deste arranjo. Não há muito que um ambiente de execução possa fazer para reduzir a fragmentação interna. *Fragmentação externa*, por outro lado, ocorre quando existe uma quantidade significante de memória não usada entre vários blocos contíguos de memória alocada. Desde que o ambiente de execução tenha o controle sobre onde alocar memória quando isto é requisitado (por exemplo, quando a palavra-chave **new** é usada em Java), o ambiente de execução deverá alocar memória em uma forma que tente reduzir a fragmentação externa até que seja possível.

Várias heurísticas têm sido sugeridas para alocação de memória em um heap de forma a minimizar a fragmentação externa. O *algoritmo best-fit* pesquisa toda a lista livre para procurar porções cujo tamanho é o mais próximo da quantidade de memória que está sendo requisitada. O *algoritmo first-fit* pesquisa desde o início da lista livre procurando a primeira porção que é grande o suficiente. O *algoritmo next-fit* é similar, pois também pesquisa a lista livre para buscar a primeira porção que é grande o suficiente, porém ele inicia sua pesquisa por onde encerrou anteriormente, verificando a lista livre como uma lista encadeada circular (Seção 3.4.1). O *algoritmo worst-fit* pesquisa na lista livre para encontrar a maior porção disponível de memória, que pode ser feito mais rápido que uma pesquisa por toda a lista livre se esta lista fosse mantida como uma fila de prioridade (Capítulo 8). Em cada algoritmo, a quantidade requisitada de memória é subtraída da porção de memória escolhida e a parte restante da porção é retornada para a lista livre.

Ainda que possa soar bem em um primeiro momento, o algoritmo best-fit tende a produzir a pior fragmentação externa, desde que as partes restantes das porções escolhidas tendem a serem menores. O algoritmo first-fit é rápido, porém ele tende a produzir uma quantidade grande de fragmentação externa no início da lista livre, que reduzirá a velocidade de pesquisas futuras. O algoritmo next-fit espalha a fragmentação de forma mais justa na memória heap, assim mantém os tempos de pesquisa baixos. Entretanto, este espalhamento também cria maior dificuldade de alocar grandes blocos. O algoritmo worst-fit tenta evitar este problema mantendo seções contínuas de memória livre o máximo possível.

---

#### 14.1.3 Coleta de lixo

Em algumas linguagens, como C e C++, o espaço de memória para objetos pode ser explicitamente desalojado pelo programador, que é uma responsabilidade muitas vezes negligenciada por programadores iniciantes e é a fonte de erros frustrantes de programação até para programadores experientes. Em vez disso, os projetistas da linguagem Java colocaram a carga do gerenciamento de memória totalmente no ambiente de execução.

Como mencionado anteriormente, a memória para objetos é alocada na memória heap e o espaço para as variáveis de instância de um programa Java em execução são colocadas em suas pilhas de métodos, uma para cada processo de execução (para os programas simples discutidos neste livro existe tipicamente um processo executando). Visto que as variáveis de instância em uma pilha de métodos podem se referir a objetos na memória heap, todas as variáveis e objetos na pilha de métodos do processo em execução são chamados *objetos raízes*. Todos estes objetos que podem ser alcançados seguindo as referências dos objetos que iniciam em um objeto raiz são chamados *objetos vivos*. Os objetos vivos são objetos ativos correntemente sendo usados pelo programa em execução; estes objetos *não* devem ser desalojados. Por exemplo, um programa Java em execução pode armazenar, em uma variável, uma referência para uma seqüência *S* que é imple-

mentada usando uma lista duplamente encadeada. A variável de referência para  $S$  é um objeto raiz, onde o objeto para  $S$  é um objeto vivo, como são todos os objetos nodos que são referenciados a partir deste objeto e todos os elementos que são referenciados a partir destes objetos nodos.

De vez enquanto, a máquina virtual Java (JVM) pode notificar que espaço disponível na memória heap está se tornando escasso. Em tempos similares, a JVM pode eleger para recuperar o espaço que está sendo usado por objetos que não viverão muito, e retorna a memória recuperada para a lista livre. Este processo de reparação é conhecido como *coleta de lixo* (*garbage collection*). Existem diferentes algoritmos para a coleta de lixo, porém uma que é a mais utilizada é o *algoritmo mark-sweep*.

No algoritmo de coleta de lixo mark-sweep, associa-se uma pequena "marca" com cada objeto que identifica se este objeto está vivo ou não. Quando se determina, em algum ponto, que a coleta de lixo é necessária, suspendem-se todos os outros processos de execução e limpam-se as pequenas marcas de todos os objetos atualmente alocados na memória heap. Então se traça através da pilha Java dos processos atualmente em execução e marcam-se todos os objetos (raízes) nesta pilha como "vivos". Deve-se então determinar todos os outros objetos vivos – aqueles que são alcançados a partir dos objetos raízes. Para fazer isso eficientemente, pode-se usar a versão do grafo dirigido do caminhamento em profundidade (Seção 13.3.1). Neste caso, cada objeto na memória heap é visto como um vértice em um grafo dirigido e a referência de um objeto para outro é visto como uma aresta dirigida. Executando um caminhamento em profundidade para cada objeto raiz, pode-se corretamente identificar e marcar cada objeto vivo. Este processo é conhecido como fase "mark". Uma vez este processo terminado, se vasculha a memória heap e recupera-se qualquer espaço que está sendo utilizado por um objeto que não tenha sido marcado. Neste ponto, pode-se, também, juntar todos os espaços alocados na memória heap em um simples bloco e, assim, eliminar a fragmentação externa. Este processo de rastreio e recuperação é conhecido como fase "sweep" e quando ele completa, pode-se retomar a execução dos processos suspensos. Assim, o algoritmo de coleta de lixo mark-sweep recuperará espaço não utilizado em um tempo proporcional ao número de objetos vivos e suas referências mais o tamanho da memória heap.

## Executando DFS in-place

O algoritmo mark-sweep corretamente recupera espaço não utilizado na memória heap, porém existe um importante caso que se deve encarar durante a fase de marcação. Desde que se esteja recuperando espaço de memória em um tempo quando a memória disponível é escassa, deve-se cuidar de não usar espaço extra durante a coleta de lixo. A confusão é que o algoritmo DFS, na forma recursiva que se descreve na Seção 13.3.1, pode usar espaço proporcional ao número de vértices no grafo. No caso da coleta de lixo, os vértices no nosso grafo são os objetos na memória heap; então provavelmente não haverá muito desta memória para utilizar. Assim, nossa única alternativa é encontrar uma forma de executar o DFS in-place de preferência recursivamente, isto é, deve-se executar o DFS usando somente uma quantidade constante de armazenamento adicional.

A idéia principal para executar o DFS in-place é simular a recursão na pilha usando as arestas do grafo (que no caso da coleta de lixo correspondem às referências aos objetos). Quando se percorre uma aresta a partir do vértice visitado  $v$  até um novo vértice  $w$ , alterna-se a aresta  $(v,w)$  armazenada na lista de adjacência  $v$  para apontar para o pai de  $v$  na árvore DFS. Quando se retorna para  $v$  (simulando o retorno a partir de uma chamada "recursiva" em  $w$ ), pode-se agora alterar a aresta que foi modificada para que aponte para  $w$ . Claro que é necessário ter alguma forma de identificar qual aresta se precisa alterar. Uma possibilidade é enumerar as referências iniciando em  $v$  como 1, 2, 3 e assim por diante, e armazenar, em adição a pequena marca (que se está usando para marcar como "visitado" na nossa DFS), um contador que fale quais arestas estão sendo modificadas.

Usar um contador requer uma palavra extra armazenada por objeto. Entretanto, esta palavra extra pode ser evitada em algumas implementações. Por exemplo, muitas implementações da máquina virtual Java representam um objeto como uma composição de uma referência com um identificador tipo (que indica se este objeto é um `Integer` ou de algum outro tipo) e como uma referência a outros objetos ou campos de dados para este objeto. Visto que a referência tipo é sempre assumida para ser o primeiro elemento da composição nestas implementações, pode-se usar esta referência para “marcar” a aresta que se altera quando se deixa o objeto `v` e se vai para algum objeto `w`. Simplifica-se a troca da referência em `v` que refere-se ao tipo de `v` com a referência em `v` que refere-se a `w`.

Quando se retorna a `v`, pode-se rapidamente identificar a aresta  $(v, w)$  que se altera, porque ela será a primeira referência na composição de `v` e a posição da referência para o tipo de `v` nos dirá o local onde esta aresta pertence na lista de adjacência de `v`. Assim, usando este truque de troca de arestas ou um contador, pode-se implementar DFS in-place sem afetar assintoticamente o tempo de execução.

## 14.2 Memória externa e caching

Existem várias aplicações de computador que precisam trabalhar com uma grande quantidade de dados. Exemplos incluem a análise de conjuntos de dados científicos, processamento de transações financeiras e organização e manutenção de banco de dados (como uma lista telefônica). De fato, a quantidade de dados que devem ser gerenciadas é muitas vezes muito grande para encaixá-la na memória interna do computador.

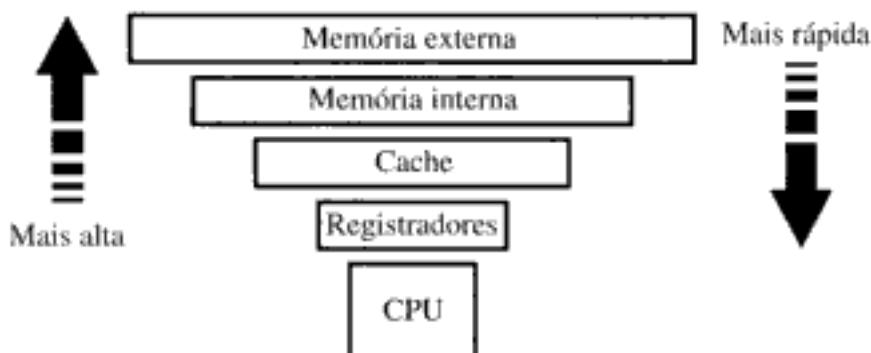
### 14.2.1 A hierarquia de memória

Para acomodar grandes conjuntos de dados, computadores têm uma **hierarquia** de diferentes tipos de memórias, que variam em termos de seus tamanhos e distâncias da CPU. Próximas da CPU são os registradores internos que a CPU utiliza. O acesso a estas memórias é muito rápido, porém existem relativamente poucas delas. Em um segundo nível na hierarquia está a memória **cache**. Esta memória é consideravelmente maior que o conjunto de registrador de uma CPU, porém acessá-la leva tempo (e podem existir algumas vezes múltiplas caches com tempos de acesso progressivamente lentos). No terceiro nível na hierarquia está a **memória interna**, que também é conhecida como **memória principal** ou **core memory**. A memória interna é consideravelmente maior que a memória cache, porém também requer mais tempo de acesso. Finalmente, no mais alto nível na hierarquia tem-se a **memória externa**, que usualmente consiste de discos, drivers de CDS, drives de DVD e/ou *tapes*. Esta memória é muito maior, porém ela também é muito lenta. Desta forma, a hierarquia de memória para computadores pode ser analisada como uma hierarquia consistindo em quatro níveis, cada uma é maior e mais lenta que a apresentada no nível inferior. (Ver Figura 14.3.)

Entretanto, em muitas aplicações somente dois níveis realmente importam – uma que pode agrupar todos os itens de dados e a nível logo abaixo deste. Executar itens de dados de entrada e saída na memória mais alta que possa agrupar todos os itens tipicamente, neste caso, será o gargalo computacional.

#### Caches e discos

Especificamente, os dois níveis que mais importam dependem do tamanho do problema que se está tentando resolver. Para um problema que pode ser todo encaixado na memória principal, os dois mais importantes níveis são a memória cache e a memória interna. O tempo de acesso na memória



**Figura 14.3** A hierarquia de memória.

interna pode ser de 10 a 100 vezes maior que o acesso na memória cache. Por outro lado, para um problema que não se encaixa totalmente na memória principal os dois mais importantes níveis são a memória interna e memória externa. Aqui as diferenças são mais dramáticas para os tempos de acesso em discos, que usualmente tem a proposta geral de ser dispositivo de memória externa, são tipicamente de 100.000 a 1.000.000 vezes mais lenta que o acesso na memória interna.

Para colocar este último citado em perspectiva, imagine existir um estudante em Baltimore que precisa enviar uma mensagem de requisição de dinheiro para seus pais em Chicago. Se o estudante envia a seus pais um email, esta pode chegar ao computador da casa dos pais em segundos. Pense que este modo de comunicação corresponde a um acesso a memória interna pela CPU. Um modo de comunicação que corresponde a um acesso a memória externa que é 500.000 vezes mais lenta seria o fato do estudante caminhar até Chicago e entregar sua mensagem pessoalmente, que levaria por volta de um mês se ele caminhasse 20 milhas por dia na média. Assim, devem ser feitos poucos acessos à memória externa sempre que possível.

#### 14.2.2 Estratégias de cache

Muitos algoritmos não são projetados com a hierarquia de memória em mente, apesar da grande variância entre os tempos de acesso para diferentes níveis. Sem dúvida, todas as análises de algoritmos contidas neste livro, até o momento, tem assumido que todos os acessos a memória são iguais. Esta suposição pode parecer, em um primeiro momento, um grande equívoco – e um sólido endereçado agora no capítulo final – porém existem boas razões do porque ela é realmente uma suposição razoável a ser feita.

Uma justificativa para esta suposição é que ela muitas vezes é necessária para assumir que todo o acesso a memória leva a mesma quantidade de tempo, visto que especificar a informação dependente de dispositivo sobre tamanhos de memória é muitas vezes difícil de se conseguir. De fato, informações de tamanho de memória podem ser impossíveis de se conseguir. Por exemplo, um programa Java que é projetado para executar em diferentes plataformas de computador não pode ser definido em termos de uma configuração de arquitetura específica de computador. Pode-se certamente usar informações específicas de arquitetura, quando se a tem (e será mostrado como explorar esta informação neste capítulo). Porém, uma vez tendo otimizado o software para uma determinada configuração de arquitetura, o software não será mais independente de dispositivo. Felizmente, otimizações não são sempre necessárias, antes de qualquer coisa, por causa da segunda justificativa da suposição de tempos iguais ao acesso à memória.

#### Caching e blocking

Outra justificativa para a suposição de igualdade no acesso a memória é que projetistas de sistemas operacionais têm desenvolvido mecanismos gerais que permitem que a maior parte do

acesso à memória seja rápida. Estes mecanismos são baseados em duas importantes propriedades de *localização-da-referência* que muitos software possuem:

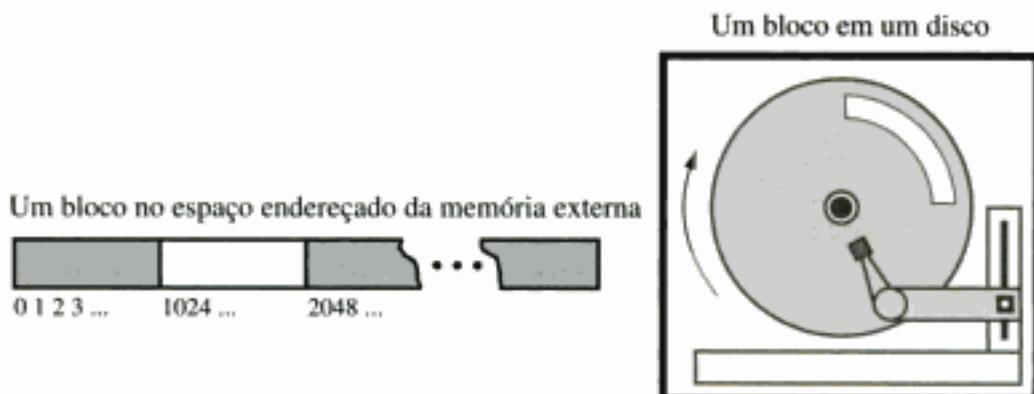
- **Localização temporal:** Se um programa acessa uma certa localidade de memória, então ela provavelmente acessará este local novamente em um futuro próximo. Por exemplo, é bastante comum usar o valor de uma variável contadora em diferentes expressões, incluindo uma para incrementar o valor do contador. De fato, um provérbio comum entre arquitetos de computadores diz que “um programa gasta 90% do seu tempo em 10% do seu código”.
- **Localização espacial:** Se um programa acessa um certo local de memória, então ele provavelmente acessará outras localidades que estão próximas a ela. Por exemplo, um programa usa um arranjo que provavelmente acessará as posições deste arranjo de uma forma seqüencial ou próximo do seqüencial.

Cientistas e engenheiros de computador tem executado extensos experimentos para desenhar o perfil para justificar a afirmativa que muitos softwares possuem ambos os dois tipos de localização-da-referência. Por exemplo, um laço “*for*” usado para rastrear um arranjo exibirá ambos os tipos de localidades.

Localização temporal e espacial tem, um após o outro, causado dois fundamentais escolha de projeto para sistema de memória de computador em dois níveis (que são apresentador na interface entre memória cache e memória interna, e também na interface entre a memória interna e a memória externa).

A primeira escolha de projeto é chamada de **memória virtual**. Este conceito consiste em prover um endereço grande como a capacidade da memória de nível secundário, e de transferir dados localizados na memória de nível secundário para o nível primário, quando os dados são endereçados. Memória virtual não limita o programador para a restrição do tamanho da memória interna. O conceito de trazer dados para a memória primária é chamado de **caching**, e ela é motivada pela localização temporal. Trazendo dados para a memória primária, espera-se que ela será acessada novamente em um breve momento e se estará apto a responder rapidamente para todas as requisições para este dado que chega em um futuro próximo.

A segunda escolha de projeto é motivada pela localidade espacial. Especificamente, se o dado armazenado em um local da memória de segundo nível 1 é acessada, então se leva para a memória de primeiro nível, um grande bloco de locais contíguos que incluem o local 1. (Ver Figura 14.4.) Este conceito é conhecido como **blocking** e é motivado pela expectativa que outros locais próximos a 1 na memória de segundo nível de serem acessados. Na interface entre a memória cache e a memória interna, blocos são muitas vezes chamados de **linhas de cache** e na interface entre a memória interna e a memória externa, blocos são muitas vezes chamados de **páginas**.



**Figura 14.4** Blocos na memória externa.

Quando implementada com caching e clocking, a memória virtual muitas vezes permite perceber a memória de nível secundário como sendo mais rápida que ela realmente é. Entretanto, ainda existe um problema. Memória do nível primário é muito menor que a memória secundária. Além disso, por causa dos sistemas de memória usarem blocking, qualquer programa de substância provavelmente alcançará um ponto onde ele requer dados da memória secundária; porém, a memória primária está realmente cheia de blocos. Para realizar a requisição e manter o uso do caching e blocking, deve-se remover alguns blocos da memória primária para “fazer sala” para um novo bloco advindo da memória secundária, neste caso. Decidir como fazer este despejo cria um número de estruturas de dados interessantes e consequentes projetos de algoritmos.

### Algoritmo de *caching*

Existem várias aplicações Web que devem trabalhar com informações revisitadas apresentadas nas páginas Web. Estas revisitadas tem sido mostradas para exibir ambas as localidades de referências – tempo e espaço. Para explorar estas localidades de referências, elas possuem várias vantagens para armazenar cópias de páginas Web na memória **cache**, assim estas páginas podem ser rapidamente recuperadas quando forem novamente requisitadas. Em particular, supondo que se tenha uma memória cache que tem  $m$  “slots” que podem conter páginas Web. Assume-se que uma página Web pode ser inserida em qualquer slot da cache. Isto é conhecido como cache **completamente associativa**.

Como um navegador executa, ela requisita diferentes páginas Web. Cada vez que o navegador requisita uma página Web  $I$ , o navegador determina (usando um teste rápido) se  $I$  está inalterado e atualmente contido na cache. Se  $I$  está contida na cache, então o navegador satisfaz a requisição utilizando a cópia da cache. Entretanto, se  $I$  não estiver na cache a página para  $I$  é requisitada na Internet e transferida para a cache. Se um dos  $m$  slots da cache estiver disponível, então o navegador atribui  $I$  para os slots vazios. Porém, se todas as células  $m$  da cache estiverem ocupadas, então o computador deve verificar qual foi a página anteriormente vista para despejar antes de buscar  $I$  e liberar seu espaço. Claro que existem diferentes políticas que podem ser usadas para determinar qual página será despejada.

### Algoritmos de substituição de páginas

Algumas das políticas de substituição de páginas melhor conhecidas incluem as seguintes (ver Figura 14.5):

- **First-in, first-out (FIFO):** Despeja a página que está na cache por mais tempo, isto é, a página que foi transferida para a cache no passado mais distante.
- **Least recently used (LRU):** Despeja a página cuja última requisição ocorreu no passado mais distante.

Adicionalmente, pode-se considerar uma simples e pura estratégia randômica:

- **Random:** Escolhe uma página randomicamente para desalojar da cache.

A estratégia Random é uma das políticas mais fáceis de implementar: requer somente um gerador de número randômico ou pseudo-randômico. O resultado elevado envolvido na implementação desta política é uma quantidade de trabalho adicional de  $O(1)$  por página substituída. Além disso, não existe overhead para cada página requisitada, não ser para determinar se uma página requisitada está ou não na cache. Ainda, esta política não cria esforço para levar vantagem de qualquer localização temporal ou espacial que uma navegação de usuário exige.

A estratégia FIFO é muito simples de implementar, visto que ela somente requer uma fila  $Q$  para armazenar referências para as páginas na cache. Páginas são enfileiradas na  $Q$  quando elas são referenciadas por um navegador e então elas são colocadas na cache. Quando uma página precisa



de vista do pior caso, estas políticas são quase sempre as piores possíveis – elas requerem uma substituição de página para toda a página requisitada.

Entretanto, esta análise do pior caso é muito pouco pessimista por ela focar no comportamento de cada protocolo para uma seqüência ruim de requisições de páginas. Uma análise ideal seria comparar estes métodos sobre todas as seqüências possíveis de requisições de páginas. Claro que isso é impossível de fazer exaustivamente, porém existe um grande número de simulações experimentais feitas em seqüências de requisições de páginas derivadas de programas reais. Baseada nestas comparações experimentais, a estratégia LRU tem sido apresentada para ser usualmente superior a estratégia FIFO, que é usualmente melhor que a estratégia Random.

### 14.3 Pesquisa externa e árvores-B

Considere-se o problema de implementar um dicionário ordenado para uma grande coleção de itens que não cabem na memória principal. Visto que uma das principais aplicações de grandes dicionários é em sistemas de banco de dados, referenciam-se os blocos da memória secundária como *bloco de discos*. Da mesma forma, a transferência de blocos entre a memória secundária e a memória principal é chamada de *transferência de disco*. Lembrando que existe uma grande diferença de tempo entre os acessos à memória principal e os acessos a disco, o principal objetivo quando se mantém um dicionário em memória externa é minimizar o número de transferências de disco necessárias para executar uma operação de consulta ou de atualização. De fato, a diferença de velocidade entre o disco e a memória interna é tão grande que se prefere executar um número considerável de acessos à memória interna se eles permitirem evitar algumas transferências de disco. Será analisada, então, a performance de implementações de dicionário pela contagem do número de transferências de disco que cada uma requer para executar as operações-padrão de pesquisa e atualização. Esta contagem é referida como a *complexidade de E/S* dos algoritmos envolvidos.

#### Alguns dicionários de memória externa ineficientes

Considerem-se primeiramente simples implementações de dicionário que usam uma seqüência para armazenar itens. Se a seqüência for implementada como uma lista duplamente encadeada não-ordenada, isto é, um arquivo de log baseado em uma lista, então as inserções podem ser executadas com transferências  $O(1)$ , mas remoções e pesquisas requerem  $n$  transferências no pior caso uma vez que cada ligação executada pode acessar um bloco diferente. Esse tempo de pesquisa pode ser melhorado para  $O(n/B)$  transferências (ver Exercício C-14.1), onde  $B$  denota o número de nodos da lista que cabem em um bloco, mas ainda é uma performance pobre. Pode-se, em vez disso, implementar a seqüência usando um vetor ordenado, isto é, uma tabela de pesquisa. Neste caso, uma pesquisa executa  $O(\log_2 n)$  transferências, usando algoritmos de pesquisa binária, o que é uma pequena melhoria. Porém, esta solução requer  $\Theta(n/B)$  transferências para implementar uma operação de inserção ou remoção no pior caso, no qual será preciso acessar todos os blocos que armazenam o arranjo para mover os elementos para cima ou para baixo. Dessa forma, as implementações baseadas em seqüências de um dicionário não são eficientes do ponto de vista da memória externa.

Uma vez que essas implementações simples são ineficientes em termos de E/S, então talvez seja possível considerar as estratégias com tempo logarítmico usadas com as árvores binárias balanceadas (como as árvores AVL ou as árvores vermelho-pretas) ou outras estruturas de pesquisa com tempo médio para atualização e pesquisa logarítmica, (tais como skip lists). Esses métodos armazenam os itens do dicionário nos nodos de uma árvore binária ou um gráfico (para skip lists). Normalmente, cada nodo acessado em uma consulta ou atualização em uma dessas estruturas se

encontra em um bloco diferente. Sendo assim, esses métodos normalmente requerem  $O(\log_2 n)$  transferências para executar uma operação de consulta ou atualização. Isso é bastante bom, mas pode-se fazer melhor. Em particular, será descrito no restante desta seção como executar consultas e atualizações em um dicionário usando apenas  $O(\log_b n)$ , isto é,  $O(\log n / \log B)$  transferências.

### 14.3.1 Árvores $(a,b)$

Para reduzir a importância da diferença de performance entre acessos à memória interna e acessos à memória externa em pesquisas, pode-se representar nosso dicionário usando uma árvore de pesquisa genérica (Seção 10.4.1). Esta abordagem leva a uma generalização da estrutura de dados árvore  $(2,4)$  em uma estrutura conhecida como árvore  $(a,b)$ .

Uma árvore  $(a,b)$  é uma árvore de pesquisa genérica tal que em cada nodo, entre os filhos  $a$  e  $b$ , são armazenados entre  $a - 1$  e  $b - 1$  itens. Os algoritmos de pesquisa, inserção e remoção de elementos em uma árvore  $(a,b)$ , são generalizações das operações correspondentes para árvores  $(2,4)$ . A vantagem de generalizar árvores  $(2,4)$  em árvores  $(a,b)$  é que a classe de árvores generalizadas oferecem uma estrutura de pesquisa mais flexível, na qual o tamanho dos nodos e o tempo de execução das várias operações sobre o dicionário depende dos parâmetros  $a$  e  $b$ . Ajustando os parâmetros  $a$  e  $b$  adequadamente em relação ao tamanho dos blocos de disco, pode-se derivar uma estrutura de dados que obtém boa performance em memória externa.

#### Definição de uma árvore $(a,b)$

Uma **árvore** onde  $a$  e  $b$  são inteiros, tal que  $2 \leq a \leq (b + 1)/2$ , é uma árvore genérica  $T$  com as seguintes restrições adicionais:

**Propriedade do tamanho:** cada nodo interno tem pelo menos  $a$  filhos, a menos que seja a raiz, e no máximo  $b$  filhos.

**Propriedade da profundidade:** todos os nodos externos têm a mesma profundidade.

**Proposição 14.1** A altura de uma árvore que armazena  $n$  itens é  $\Omega(\log n / \log b)$  e  $O(\log n / \log a)$ .

**Justificativa** Seja  $T$  uma árvore que armazena  $n$  elementos e seja  $h$  a altura de  $T$ . Justifica-se a proposição estabelecendo os seguintes limites para  $h$ :

$$\frac{1}{\log b} \log(n+1) \leq h \leq \frac{1}{\log a} \log \frac{n+1}{2} + 1.$$

Pelas propriedades do tamanho e da profundidade, o número  $n''$  de nodos externos de  $T$  é no mínimo  $2a^{h-1}$  e no máximo  $b^h$ . Pela Proposição 10.7,  $n'' = n + 1$ . Assim,

$$2a^{h-1} \leq n + 1 \leq b^h.$$

Tomando o logaritmo de base 2 para cada termo, obtém-se

$$(h + 1)\log a + 1 \leq \log(n + 1) \leq h\log b.$$

#### Operações de pesquisa e atualização

Deve-se lembrar que em uma árvore de pesquisa genérica  $T$ , cada nodo  $v$  de  $T$  armazena uma estrutura secundária  $D(v)$ , que é também um dicionário (ver Seção 10.4.1). Se  $T$  é uma árvore  $(a,b)$ , então  $D(v)$  armazena no máximo  $b$  itens. Faça  $f(b)$  denotar o tempo para executar uma pesquisa em um dicionário  $D(v)$ . O algoritmo de pesquisa em uma árvore  $(a,b)$  é exatamente igual ao de árvores de pesquisa genérica apresentado na Seção 10.4.1. Logo, pesquisar em uma árvore  $(a,b)$

Hidden page

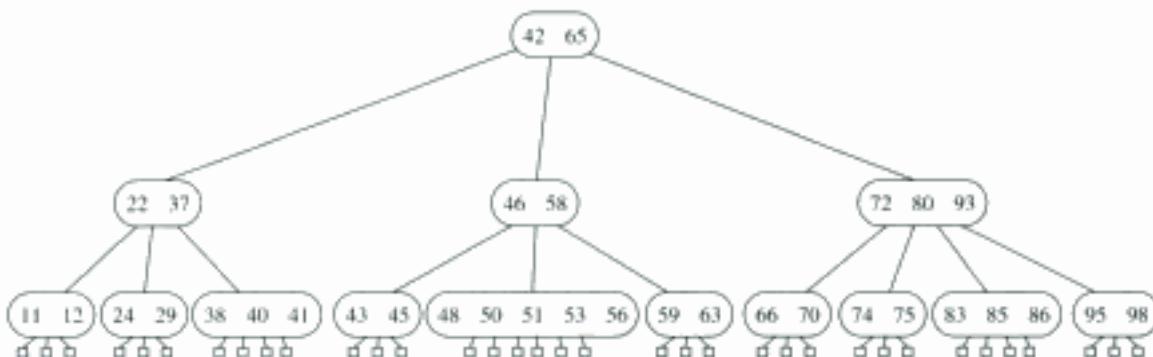


Figura 14.6 Uma árvore-B de ordem 6.

Como foi mencionado antes, cada pesquisa ou atualização requer que se examine no mínimo  $O(1)$  nodos para cada nível da árvore. Consequentemente, qualquer operação de pesquisa ou atualização em uma árvore B requer apenas  $O(\log_{d+1} n)$ , isto é,  $O(\log n / \log b)$  transferências de disco. Por exemplo, uma operação de inserção percorre a árvore para baixo visando localizar o nodo no qual inserir um novo item. Se esta adição for implicar em um *overflow* do nodo (ter  $d + 1$  filhos), então este nodo é dividido em dois nodos que terão  $\lfloor (d + 1)/2 \rfloor$  e  $\lceil (d + 1)/2 \rceil$  filhos, respectivamente. Este processo é então repetido no nível de cima e irá continuar por pelo menos  $O(\log_B n)$  níveis.

Da mesma forma, se uma operação de remoção resultar em um *underflow* de um nodo (com  $\lceil d/2 \rceil - 1$  filhos), então se moverão as referências de um nodo irmão com pelo menos  $\lceil d/2 \rceil + 1$  filhos ou necessitaremos executar uma operação de *fusão* deste nodo com seu irmão (e repetir essa operação para seu pai). Da mesma forma que com a operação de inserção, isso irá continuar pela árvore B acima por pelo menos  $O(\log_B n)$  níveis. O requisito de que cada nodo interno tenha pelo menos  $\lceil d/2 \rceil$  filhos implica que cada bloco de disco usado para suportar uma árvore B esteja pelo menos cheio pela metade. Logo, se terá o seguinte:

**Proposição 14.2** *Uma árvore B com n itens tem complexidade de E/S  $O(\log_B n)$  para operações de pesquisa ou atualização e usa  $O(n/B)$  blocos, onde B é o tamanho de um bloco.*

## 14.4 Ordenando memória externa

Em adição as estruturas de dados, como dicionários, que precisam ser implementados na memória externa, existem vários algoritmos que também podem operar em conjuntos de entrada que são muito grandes para caberem na memória interna. Neste caso, o objetivo é resolver o problema algorítmico usando alguns blocos para transferências, sempre que possível. O domínio mais clássico para algoritmos de memória externa é o problema da ordenação.

### Merge-sort genérico

Uma eficiente forma de ordenar um conjunto  $S$  de  $n$  objetos na memória externa equivale em uma simples variação da família do algoritmo de merge-sort para a memória externa. A idéia principal por trás desta variação é juntar várias listas ordenadas recursivamente num momento, e assim reduzir o número de níveis da recursão. Especificamente, uma descrição deste método *merge-sort genérico* é dividir  $S$  em  $d$  subconjuntos  $S_1, S_2, \dots, S_d$  de tamanhos aproximadamente iguais, recursivamente ordenar cada subconjunto  $S_i$  e então simultaneamente juntar todas as  $d$  listas ordenadas em uma representação ordenada  $S$ . Se é possível executar o processo de junção usando somente  $O(n/B)$  transferências em disco, então para valores grandes suficientes de  $n$ , o total do número de transferências executadas com este algoritmo satisfaz a seguinte recorrência:

$$t(n) = d \cdot t(n/d) + cn/B,$$

Para alguma constante  $c \geq 1$ . Pode-se parar a recursão quando  $n \leq B$ , visto que é possível executar uma transferência simples de bloco neste ponto, pegando todos os objetos na memória interna e então ordenando o conjunto com um algoritmo eficiente de memória interna. Assim, o critério de parada para  $t(n)$  é

$$t(n) = 1 \quad \text{se } n/B \leq 1.$$

Isto implica em uma solução de forma fechada que  $t(n)$  é  $O((n/B)\log_d(n/B))$ , que é

$$O((n/B)\log(n/B)/\log d).$$

Assim, se é possível escolher  $d$  para ser  $\Theta(M/B)$ , então o número de transferências no pior caso executado por este algoritmo de merge-sort genérico será muito baixo. Escolhe-se

$$d = (1/2) M/B.$$

O único aspecto para este algoritmo deixar de especificar é como executar o merge  $d$ -way usando somente  $O(n/B)$  transferências de blocos.

#### 14.4.1 Merge genérico

Executa-se o merge  $d$ -way para apresentar um “torneio”. Seja  $T$  uma árvore binária completa com  $d$  nodos externos e mantém-se  $T$  totalmente na memória interna. Associa-se cada nodo externo  $i$  de  $T$  com uma diferente lista ordenada  $S_i$ . Inicializa-se  $T$  lendo cada nodo externo  $i$ , o primeiro objeto de  $S_i$ . Isso tem o efeito de ler de uma memória interna o primeiro bloco de cada lista ordenada  $S_i$ . Para cada nodo pai interno  $v$  de dois nodos externos, então se compararam os objetos armazenados nos filhos de  $v$  e se associa o menor dos dois com  $v$ . Repete-se este teste de comparação no próximo nível acima de  $T$  e o próximo, e assim por diante. Quando se alcança a raiz  $r$  de  $T$ , se associará o menor objeto entre todos os da lista com  $r$ . Isto completa a inicialização para o merge  $d$ -way. (Ver Figura 14.7)

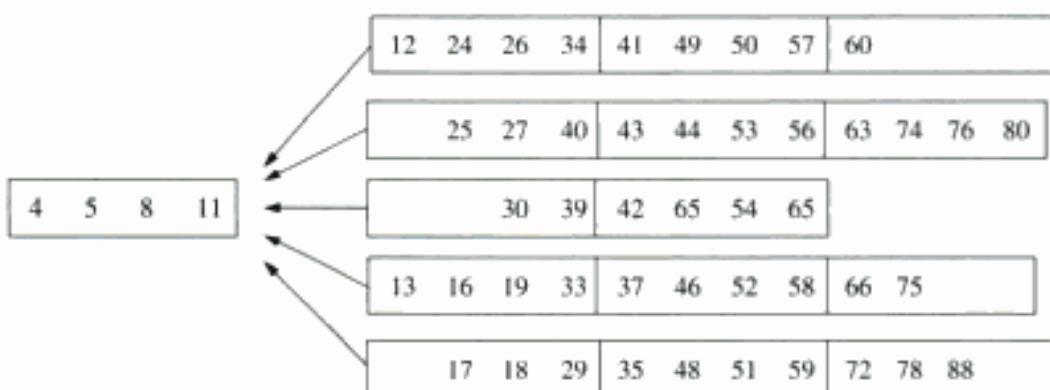


Figura 14.7 Um merge  $d$ -way. Mostramos um merge 5-way com  $B = 4$ .

Em um passo geral do merge  $d$ -way move-se o objeto  $o$  associado com a raiz  $r$  de  $T$  em um arranjo que se está criando para a lista unida  $S'$ . Então segue-se  $T$  para baixo seguindo o caminho do nodo externo  $i$  em que  $o$  surge. Depois, lê-se em  $i$  o próximo objeto na lista  $S_i$ . Se  $o$  não era o último elemento neste bloco, então o próximo objeto está na memória interna. De outra forma, lê-se no próximo bloco de  $S_i$  para acessar este novo objeto (se  $S_i$  agora não está vazio, associa-se o nodo  $i$  com um pseudo-objeto com chave  $+\infty$ ). Então repetem-se as computações mínimas para cada nodo interno a partir de  $i$  até a raiz de  $T$ . Este novamente entrega-nos a árvore completa  $T$ . Então repete-se este processo movendo o objeto a partir da raiz de  $T$  para a lista unida  $S'$ , e reconstrói-se  $T$ , até que  $T$  esteja sem objetos. Cada passo na junção leva o tempo de  $O(\log d)$ ;

então, o tempo interno para o merge  $d$ -way é  $O(n \log d)$ . O número de transferências executadas em um *merger* é  $O(n/B)$ , desde que se vasculhe cada lista  $S_i$  uma vez, e se copie a lista juntada uma vez. Assim tem-se:

**Proposição 14.3** *Dada uma seqüência baseada em arranjo  $S$  de  $n$  elementos armazenados na memória externa, pode-se ordenar  $S$  usando  $O(n/B)\log(n/B)/\log(M/B))$  transferências e o tempo interno de CPU de  $O(n \log n)$ , onde  $M$  é o tamanho da memória interna e  $B$  é o tamanho de um bloco.*

## 14.5 Exercícios

Para obter o código fonte e auxílio com os exercícios, visite [java.datastructures.net](http://java.datastructures.net)

### Reforço

- R-14.1 Descreva em detalhes os algoritmos de inserção e remoção para uma árvore  $(a,b)$ .
- R-14.2 Suponha que  $T$  é uma árvore genérica cujos nodos internos têm pelo menos 5 e no máximo 8 filhos. Para quais valores de  $a$  e  $b$   $T$  é uma árvore válida?
- R-14.3 Para quais valores de  $d$  a árvore  $T$  do exercício anterior corresponde a uma árvore B de ordem  $d$ ?
- R-14.4 Mostre os níveis de recursão da execução de um merge sort tipo four-way da seqüência apresentada no exercício anterior.
- R-14.5 Considere uma memória cache inicialmente vazia de 4 páginas. Quantas páginas faltam para o algoritmo LRU executar em si próprio na seguinte seqüência de requisições: (2,4,1,2,5,1,3,5,4,1,2,3)?
- R-14.6 Considere uma memória cache inicialmente vazia de 4 páginas. Quantas páginas faltam para o algoritmo FIFO executar em si próprio na seguinte seqüência de requisições: (2,4,1,2,5,1,3,5,4,1,2,3)?
- R-14.7 Considere uma memória cache inicialmente vazia de 4 páginas. Quantas páginas faltam para o algoritmo Random executar em si próprio na seguinte seqüência de requisições: (2,4,1,2,5,1,3,5,4,1,2,3)? Mostre todas as escolhas randômicas que o seu algoritmo cria neste caso.
- R-14.8 Desenhe o resultado da inserção, em uma árvore-B inicialmente vazia de ordem 7, elementos com chaves (4,40,23,50,11,34,62,78,66,22,90,59,25,7 22,64,77,39,12), nesta ordem.
- R-14.9 Mostre os níveis de recursão da execução de um merge sort tipo four-way da seqüência apresentada no exercício anterior.

### Criatividade

- C-14.1 Mostre como implementar um dicionário em memória externa usando uma seqüência não-ordenada de forma que as atualizações requeiram apenas  $O(1)$  transferências e que as atualizações requeiram  $O(n/B)$  transferências, no pior caso, onde  $n$  é o número de elementos e  $B$  é o número de nodos da lista que cabem em um bloco de disco.

- C-14.2 Altere as regras que definem árvores vermelho-pretas de forma que cada árvore vermelho-preta  $T$  tenha uma árvore (4,8) correspondente e vice-versa.
- C-14.3 Descreva uma versão modificada do algoritmo de inserção para árvores B de forma que, cada vez que se gera um overflow em virtude da divisão de um nodo  $v$ , redistribui-se chaves entre todos os irmãos de  $v$  de forma que cada irmão armazene, aproximadamente, o mesmo número de chaves (possivelmente propagando a divisão para o pai de  $v$ ). Qual a fração mínima de cada bloco que sempre será preenchida usando este esquema?
- C-14.4 Outra possibilidade de implementação de dicionários usando memória externa é usar uma skip list para organizar em blocos individuais grupos de  $O(B)$  nodos de qualquer nível na skip list. Em especial, define-se uma *skip list B* de *ordem*  $d$  para representar uma estrutura de skip list, na qual cada bloco contém no mínimo  $\lceil d/2 \rceil$  nodos da lista e no máximo  $d$  nodos da lista. Será escolhido também, neste caso,  $d$  para ser o número máximo de nodos da lista de um nível da lista de saltos que cabe em um bloco. Descreva como é possível modificar os algoritmos de inserção e remoção em uma skip list para uma skip list  $B$  de maneira que a altura esperada para a estrutura seja  $O(\log n/\log B)$ .
- C-14.5 Descreva uma estrutura de dados de memória externa para implementar um TAD fila em que o número total de transferências de disco necessárias para processar uma seqüência de  $n$  operações de enfileirar e desenfileirar é  $O(n/B)$ .
- C-14.6 Resolva o problema anterior para um TAD deque.
- C-14.7 Descreva como usar uma árvore-B para implementar o TAD partição (união-procura) (da Seção 11.6.2) em que as operações union e find usam no máximo  $O(\log n/\log B)$  transferências de discos cada uma.
- C-14.8 Suponha que se tem uma seqüência  $S$  de  $n$  elementos com chaves inteiras em que alguns itens em  $S$  são coloridos de “azul” e alguns de “vermelho”. Além disso, diga que um elemento vermelho  $e$  é par de um elemento azul  $f$  se eles tiverem o mesmo valor de chave. Descreva um algoritmo de memória externa eficiente para encontrar todos os pares azul e vermelho de  $S$ . Quantas transferências de disco este algoritmo irá executar?
- C-14.9 Considere o problema de caching de página onde a memória cache pode ter  $m$  páginas, e se envia uma seqüência  $P$  de  $n$  requisições de um pool de  $m + 1$  páginas possíveis. Descreva a estratégia mais eficiente para o algoritmo offline e mostre que isto causa no máximo  $m + n/m$  perdas de páginas no total, iniciando de uma cache vazia.
- C-14.10 Considere a estratégia de caching de páginas baseada na regra ***Menos Frequentemente Usada (LRU***, onde a página na cache que tem sido acessada menos vezes é a que será desalojada quando uma nova página for requisitada. Se existirem amarranções, LFU desalojará a página menos freqüentemente usada que tem estado na cache por mais tempo. Mostre que existe uma seqüência  $P$  de  $n$  requisições que faz com que LRU fracassa  $\Omega(n)$  vezes para um cache de  $m$  páginas, enquanto que o algoritmo mais eficiente fracassará  $\Omega(m)$  vezes.

Hidden page

Hidden page

Hidden page

Hidden page

Hidden page

Uma forma elegante de lidar com eventos é em termos de *variáveis aleatórias*. Intuitivamente, as variáveis aleatórias são variáveis cujos valores dependem do resultado de algum experimento. Formalmente, uma *variável aleatória* é uma função  $X$  que mapeia resultados de um espaço amostral  $S$  a números reais. Uma *variável aleatória indicadora* é uma variável aleatória que mapeia resultados para os valores do conjunto  $\{0, 1\}$ . Freqüentemente, na análise de estruturas de dados e algoritmos usa-se uma variável aleatória  $X$  para caracterizar o tempo de execução de um algoritmo randomizado. Nesse caso, o espaço amostral  $S$  é definido por todos os resultados aleatórios possíveis usados no algoritmo.

Está-se mais interessado no valor típico, médio ou “esperado” de uma variável aleatória. O *valor esperado* de uma variável aleatória  $X$  é definido como

$$E(X) = \sum_x x \Pr(X = x),$$

onde a soma é definida sobre todos os valores em  $X$  (que neste caso é assumido como discreto).

**Proposição A.19 (linearidade do valor esperado)** *Sejam  $X$  e  $Y$  duas variáveis aleatórias arbitrárias. Então*

$$E(X + Y) = E(X) + E(Y) \quad \text{and} \quad E(cX) = cE(X).$$

**Exemplo A.20** *Seja  $X$  uma variável aleatória que associa o resultado do lançamento de dois dados à soma dos valores resultantes. Então  $E(X) = 7$ .*

**Justificativa** Para justificar a afirmação, sejam  $X_1$  e  $X_2$  variáveis aleatórias correspondendo ao número resultante em cada dado. Assim,  $X_1 = X_2$  (isto é, são duas instâncias da mesma função), e  $E(X) = E(X_1 + X_2) = E(X_1) + E(X_2)$ . Cada resultado do lançamento dos dados ocorre com probabilidade  $1/6$ . Portanto

$$E(X_i) = \frac{1}{6} + \frac{2}{6} + \frac{3}{6} + \frac{4}{6} + \frac{5}{6} + \frac{6}{6} = \frac{7}{2},$$

para  $i = 1, 2$ . Por isso,  $E(X) = 7$ . ■

Duas variáveis  $X$  e  $Y$  são *independentes* se

$$\Pr(X = x | Y = y) = \Pr(X = x),$$

para todos os números  $x$  e  $y$ .

**Proposição A.21** *Se duas variáveis aleatórias  $X$  e  $Y$  são independentes, então*

$$E(XY) = E(X)E(Y).$$

**Exemplo A.22** *Seja  $X$  uma variável aleatória relacionando o resultado do lançamento de dois dados ao produto dos valores resultantes. Então  $E(X) = 49/4$ .*

**Justificativa** Sejam  $X_1$  e  $X_2$  variáveis aleatórias correspondendo ao número resultante em cada dado. As variáveis  $X_1$  e  $X_2$  são claramente independentes, portanto

$$E(X) = E(X_1 X_2) = E(X_1)E(X_2) = (7/2)^2 = 49/4. ■$$

Os seguintes limites e resultados são conhecidos como *limites Chernoff*.

**Proposição A.23** *Seja  $X$  a soma de um número finito de variáveis randômicas independentes 0/1 e seja  $\mu > 0$  o valor esperado de  $X$ . Então, para  $\delta > 0$ ,*

$$\Pr(X > (1 + \delta) \mu) < \left[ \frac{e^\delta}{(1 + \delta)^{(1 + \delta)}} \right].$$

## Técnicas matemáticas úteis

Para comparar a medida de crescimento das diferentes funções, às vezes é útil aplicar a seguinte regra:

**Proposição A.24 (regra de L'Hôpital)** *Tendo-se  $\lim_{n \rightarrow \infty} f(n) = +\infty$  e tendo-se  $\lim_{n \rightarrow \infty} g(n) = +\infty$ , então  $\lim_{n \rightarrow \infty} f(n)/g(n) = \lim_{n \rightarrow \infty} f'(n)/g'(n)$ , onde  $f'(n)$  e  $g'(n)$  denotam respectivamente as derivadas de  $f(n)$  e de  $g(n)$ .*

Para derivar um limite superior e inferior para uma soma, é freqüentemente útil *separar uma soma* como a seguir:

$$\sum_{i=1}^n f(i) = \sum_{i=1}^j f(i) + \sum_{i=j+1}^n f(i).$$

Outra técnica útil é *limitar a soma por uma integral*. Se  $f$  é uma função não-decrescente, então, assumindo que os termos a seguir estejam definidos,

$$\int_a^b f(x) dx \leq \sum_{i=a}^b f(i) \leq \int_a^{b+1} f(x) dx.$$

Existe uma forma geral de relação de recorrência que surge na análise de algoritmos de divisão e conquista:

$$T(n) = aT(n/b) + f(n),$$

para constantes  $a \geq 1$  e  $b > 1$ .

**Proposição A.25** *Seja  $T(n)$  definida como acima. Então*

1. Se  $f(n)$  é  $O(n^{\log_b a - \epsilon})$  para alguma constante  $\epsilon > 0$ , então  $T(n)$  é  $\Theta(n^{\log_b a})$ .
2. Se  $f(n)$  é  $\Theta(n^{\log_b a} \log^k n)$  para algum inteiro não-negativo  $k \geq 0$  então,  $T(n)$  é  $\Theta(n^{\log_b a} \log^{k+1} n)$ .
3. Se  $f(n)$  é  $\Omega(n^{\log_b a + \epsilon})$  para alguma constante  $\epsilon > 0$  e se a  $f(n/b) \leq cf(n)$  então,  $T(n)$  é  $\Theta(f(n))$ .

Esta proposição é conhecida como o *método mestre* para caracterizar assintoticamente relações de recorrência obtidas a partir de algoritmos de divisão e conquista.

---

## Bibliografia

- [1] G. M. Adel'son-Vel'skii and Y. M. Landis, "An algorithm for the organization of information," *Doklady Akademii Nauk SSSR*, vol. 146, pp. 263–266, 1962. English translation in *Soviet Math. Dokl.*, 3, 1259–1262.
- [2] A. Aggarwal and J. S. Vitter, "The input/output complexity of sorting and related problems," *Commun. ACM*, vol. 31, pp. 1116–1127, 1988.
- [3] A. V. Aho, "Algorithms for finding patterns in strings," in *Handbook of Theoretical Computer Science* (J. van Leeuwen, ed.), vol. A. Algorithms and Complexity, pp. 255–300, Amsterdam: Elsevier, 1990.
- [4] A. V. Aho, J. E. Hopcroft, and J. D. Ullman, *The Design and Analysis of Computer Algorithms*. Reading, MA: Addison-Wesley, 1974.
- [5] A. V. Aho, J. E. Hopcroft, and J. D. Ullman, *Data Structures and Algorithms*. Reading, MA: Addison-Wesley, 1983.
- [6] R. K. Ahuja, T. L. Magnanti, and J. B. Orlin, *Network Flows: Theory, Algorithms, and Applications*. Englewood Cliffs, NJ: Prentice Hall, 1993.
- [7] K. Arnold and J. Gosling, *The Java Programming Language*. The Java Series, Reading, Mass.: Addison-Wesley, 1996.
- [8] R. Baeza-Yates and B. Ribeiro-Neto, *Modern Information Retrieval*. Reading, Mass.: Addison-Wesley, 1999.
- [9] O. Baruvka, "O jistem problemu minimalním," *Práce Moravské Přírodovedec Společnosti*, vol. 3, pp. 37–58, 1926. (in Czech).
- [10] R. Bayer, "Symmetric binary B-trees: Data structure and maintenance," *Acta Informatica*, vol. 1, no. 4, pp. 290–306, 1972.
- [11] R. Bayer and McCreight, "Organization of large ordered indexes," *Acta Inform.*, vol. 1, pp. 173–189, 1972.
- [12] J. L. Bentley, "Programming pearls: Writing correct programs," *Communications of the ACM*, vol. 26, pp. 1040–1045, 1983.
- [13] J. L. Bentley, "Programming pearls: Thanks, heaps," *Communications of the ACM*, vol. 28, pp. 245–250, 1985.
- [14] G. Booch, *Object-Oriented Analysis and Design with Applications*. Redwood City, CA: Benjamin/Cummings, 1994.
- [15] R. S. Boyer and J. S. Moore, "A fast string searching algorithm," *Communications of the ACM*, vol. 20, no. 10, pp. 762–772, 1977.
- [16] G. Brassard, "Crusade for a better notation," *SIGACT News*, vol. 17, no. 1, pp. 60–64, 1985.
- [17] T. Budd, *An Introduction to Object-Oriented Programming*. Reading, Mass.: Addison-Wesley, 1991.
- [18] D. Burger, J. R. Goodman, and G. S. Sohi, "Memory systems," in *The Computer Science and Engineering Handbook* (A. B. Tucker, Jr., ed.), ch. 18, pp. 447–461, CRC Press, 1997.
- [19] M. Campione and H. Walrath, *The Java Tutorial: Programming for the Internet*. Reading, Mass.: Addison Wesley, 1996.
- [20] L. Cardelli and P. Wegner, "On understanding types, data abstraction and polymorphism," *ACM Computing Surveys*, vol. 17, no. 4, pp. 471–522, 1985.
- [21] S. Carlsson, "Average case results on heapsort," *BIT*, vol. 27, pp. 2–17, 1987.
- [22] K. L. Clarkson, "Linear programming in  $O(n^3 d^2)$  time," *Inform. Process. Lett.*, vol. 22, pp. 21–24, 1986.
- [23] R. Cole, "Tight bounds on the complexity of the Boyer-Moore pattern matching algorithm," *SIAM Journal on Computing*, vol. 23, no. 5, pp. 1075–1091, 1994.

- [24] D. Comer, "The ubiquitous B-tree," *ACM Comput. Surv.*, vol. 11, pp. 121–137, 1979.
- [25] T. H. Cormen, C. E. Leiserson, and R. L. Rivest, *Introduction to Algorithms*. Cambridge, MA: MIT Press, 1990.
- [26] G. Cornell and C. S. Horstmann, *Core Java*. Mountain View, CA: SunSoft Press, 1996.
- [27] M. Crochemore and T. Lecroq, "Pattern matching and text compression algorithms," in *The Computer Science and Engineering Handbook* (A. B. Tucker, Jr., ed.), ch. 8, pp. 162–202, CRC Press, 1997.
- [28] S. A. Demurjian, Sr., "Software design," in *The Computer Science and Engineering Handbook* (A. B. Tucker, Jr., ed.), ch. 108, pp. 2323–2351, CRC Press, 1997.
- [29] G. Di Battista, P. Eades, R. Tamassia, and I. G. Tollis, "Algorithms for drawing graphs: an annotated bibliography," *Comput. Geom. Theory Appl.*, vol. 4, pp. 235–282, 1994.
- [30] G. Di Battista, P. Eades, R. Tamassia, and I. G. Tollis, *GraphDrawing*. Upper Saddle River, NJ: Prentice Hall, 1999.
- [31] E. W. Dijkstra, "A note on two problems in connexion with graphs," *Numerische Mathematik*, vol. 1, pp. 269–271, 1959.
- [32] J. R. Driscoll, H. N. Gabow, R. Shrairaman, and R. E. Tarjan, "Relaxed heaps: An alternative to Fibonacci heaps with applications to parallel computation," *Commun. ACM*, vol. 31, pp. 1343–1354, 1988.
- [33] S. Even, *Graph Algorithms*. Potomac, Maryland: Computer Science Press, 1979.
- [34] D. Flanagan, *Java in a Nutshell*. O'Reilly, 4th ed., 2002.
- [35] R. W. Floyd, "Algorithm 97: Shortest path," *Communications of the ACM*, vol. 5, no. 6, p. 345, 1962.
- [36] R. W. Floyd, "Algorithm 245: Treesort 3," *Communications of the ACM*, vol. 7, no. 12, p. 701, 1964.
- [37] M. L. Fredman and R. E. Tarjan, "Fibonacci heaps and their uses in improved net-work optimization algorithms," *J. ACM*, vol. 34, pp. 596–615, 1987.
- [38] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, Mass.: Addison-Wesley, 1995.
- [39] A. M. Gibbons, *Algorithmic Graph Theory*. Cambridge, UK: Cambridge University Press, 1985.
- [40] A. Goldberg and D. Robson, *Smalltalk-80: The Language*. Reading, Mass.: Addison-Wesley, 1989.
- [41] G. H. Gonnet and R. Baeza-Yates, *Handbook of Algorithms and Data Structures in Pascal and C*. Reading, Mass.: Addison-Wesley, 1991.
- [42] G. H. Gonnet and J. I. Munro, "Heaps on heaps," *SIAM Journal on Computing*, vol. 15, no. 4, pp. 964–971, 1986.
- [43] M. T. Goodrich, M. Handy, B. Hudson, and R. Tamassia, "Accessing the internal organization of data structures in the JDSL library," in *Proc. Workshop on Algorithm Engineering and Experimentation* (M. T. Goodrich and C. C. McGeoch, eds.), vol. 1619 of *Lecture Notes Comput. Sci.*, pp. 124–139, Springer-Verlag, 1999.
- [44] M. T. Goodrich, J.-J. Tsay, D. E. Vengroff, and J. S. Vitter, "External-memory computational geometry," in *Proc. 34th Annu. IEEE Symp. Found. Comput. Sci.*, pp. 714–723, 1993.
- [45] R. L. Graham and P. Hell, "On the history of the minimum spanning tree problem," *Annals of the History of Computing*, vol. 7, no. 1, pp. 43–57, 1985.
- [46] L. J. Guibas and R. Sedgewick, "A dichromatic framework for balanced trees," in *Proc. 19th Annu. IEEE Symp. Found. Comput. Sci.*, Lecture Notes Comput. Sci., pp. 8–21, Springer-Verlag, 1978.
- [47] Y. Gurevich, "What does  $O(n)$  mean?," *SIGACT News*, vol. 17, no. 4, pp. 61–63, 1986.
- [48] J. Hennessy and D. Patterson, *Computer Architecture: A Quantitative Approach*. San Francisco: Morgan Kaufmann, 2nd ed., 1996.
- [49] C. A. R. Hoare, "Quicksort," *The Computer Journal*, vol. 5, pp. 10–15, 1962.
- [50] J. E. Hopcroft and R. E. Tarjan, "Efficient algorithms for graph manipulation," *Communications of the ACM*, vol. 16, no. 6, pp. 372–378, 1973.
- [51] C. S. Horstmann, *Computing Concepts in Java*. New York: John Wiley, and Sons, 1998.
- [52] B. Huang and M. Langston, "Practical in-place merging," *Communications of the ACM*, vol. 31, no. 3, pp. 348–352, 1988.
- [53] J. JaJa, *An Introduction to Parallel Algorithms*. Reading, Mass.: Addison-Wesley, 1992.
- [54] V. Jarník, "O jistém problému minimalním," *Práce Moravské Přírodovedecké Společnosti*, vol. 6, pp. 57–63, 1930. (in Czech).
- [55] R. E. Jones, *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. John Wiley and Sons, 1996.

- [56] D.R.Karger, P.Klein, and R.E.Tarjan, "A randomized linear-time algorithm to find minimum spanning trees," *Journal of the ACM*, vol. 42, pp. 321–328, 1995.
- [57] R. M. Karp and V. Ramachandran, "Parallel algorithms for shared memory machines," in *Handbook of Theoretical Computer Science* (J. van Leeuwen, ed.), pp. 869–941, Amsterdam: Elsevier/The MIT Press, 1990.
- [58] P. Kirschenhofer and H. Prodinger, "The path length of random skip lists," *Acta Informatica*, vol. 31, pp. 775–792, 1994.
- [59] J. Kleinberg and Tardos, MA: Addison-Wesley, E. Tardos, *Algorithm Design*. 2006.
- [60] D. E. Knuth, *Sorting and Searching*, vol. 3 of *The Art of Computer Programming*. Reading, MA: Addison-Wesley, 1973.
- [61] D. E. Knuth, "Big omicron and big omega and big theta," in *SIGACT News*, vol. 8, pp. 18–24, 1976.
- [62] D. E. Knuth, *Fundamental Algorithms*, vol. 1 of *The Art of Computer Programming*. Reading, MA: Addison-Wesley, 3rd ed., 1997.
- [63] D. E. Knuth, *Sorting and Searching*, vol. 3 of *The Art of Computer Programming*. Reading, MA: Addison-Wesley, 2nd ed., 1998.
- [64] D. E. Knuth, J. H. Morris, Jr., and V. R. Pratt, "Fast pattern matching in strings," *SIAM Journal on Computing*, vol. 6, no. 1, pp. 323–350, 1977.
- [65] J. B. Kruskal, Jr., "On the shortest spanning subtree of a graph and the traveling salesman problem," *Proc. Amer. Math. Soc.*, vol. 7, pp. 48–50, 1956.
- [66] N.G. Leveson and C.S. Turner, "An investigation of the Therac-25 accidents," *IEEE Computer*, vol. 26, no. 7, pp. 18–41, 1993.
- [67] R. Levisse, "Some lessons drawn from the history of the binary search algorithm," *The Computer Journal*, vol. 26, pp. 154–163, 1983.
- [68] A. Levitin, "Do we teach the right algorithm design techniques?," in *30th ACM SIGCSE Symp. on Computer Science Education*, pp. 179–183, 1999.
- [69] B. Liskov and J. Guttag, *Abstraction and Specification in Program Development*. Cambridge, Mass./New York: The MIT Press/McGraw-Hill, 1986.
- [70] E. M. McCreight, "A space-economical suffix tree construction algorithm," *Journal of Algorithms*, vol. 23, no. 2, pp. 262–272, 1976.
- [71] C. J. H. McDiarmid and B. A. Reed, "Building heaps fast," *Journal of Algorithms*, vol. 10, no. 3, pp. 352–365, 1989.
- [72] N. Megiddo, "Linear-time algorithms for linear programming in  $R^3$  and related problems," *SIAM J. Comput.*, vol. 12, pp. 759–776, 1983.
- [73] N. Megiddo, "Linear programming in linear time when the dimension is fixed," *J. ACM*, vol. 31, pp. 114–127, 1984.
- [74] K. Mehlhorn, *Data Structures and Algorithms 1: Sorting and Searching*, vol. 1 of *EATCS Monographs on Theoretical Computer Science*. Heidelberg, Germany: Springer-Verlag, 1984.
- [75] K. Mehlhorn, *Data Structures and Algorithms 2: Graph Algorithms and NP-Completeness*, vol. 2 of *EATCS Monographs on Theoretical Computer Science*. Heidelberg, Germany: Springer-Verlag, 1984.
- [76] K. Mehlhorn and A. Tsakalidis, "Data structures," in *Handbook of Theoretical Computer Science* (J. van Leeuwen, ed.), vol. A. Algorithms and Complexity, pp. 301–341, Amsterdam: Elsevier, 1990.
- [77] M. H. Morgan, *Vitruvius: The Ten Books on Architecture*. New York: Dover Publications, Inc., 1960.
- [78] D. R. Morrison, "PATRICIA—practical algorithm to retrieve information coded in alphanumeric," *Journal of the ACM*, vol. 15, no. 4, pp. 514–534, 1968.
- [79] R. Motwani and P. Raghavan, *Randomized Algorithms*. New York, NY: Cambridge University Press, 1995.
- [80] T. Papadakis, J. I. Munro, and P. V. Poblete, "Average search and update costs in skip lists," *BIT*, vol. 32, pp. 316–332, 1992.
- [81] P. V. Poblete, J. I. Munro, and T. Papadakis, "The binomial transform and its application to the analysis of skip lists," in *Proceedings of the European Symposium on Algorithms (ESA)*, pp. 554–569, 1995.
- [82] R. C. Prim, "Shortest connection networks and some generalizations," *Bell Syst. Tech. J.*, vol. 36, pp. 1389–1401, 1957.

- [83] W. Pugh, "Skip lists: a probabilistic alternative to balanced trees," *Commun. ACM*, vol. 33, no. 6, pp. 668–676, 1990.
- [84] H. Samet, *The Design and Analysis of Spatial Data Structures*. Reading, MA: Addison-Wesley, 1990.
- [85] R. Schaffer and R. Sedgewick, "The analysis of heapsort," *Journal of Algorithms*, vol. 15, no. 1, pp. 76–100, 1993.
- [86] D. D. Sleator and R. E. Tarjan, "Self-adjusting binary search trees," *J. ACM*, vol. 32, no. 3, pp. 652–686, 1985.
- [87] G. A. Stephen, *String Searching Algorithms*. World Scientific Press, 1994.
- [88] R. Tamassia, "Graph drawing," in *Handbook of Discrete and Computational Geometry* (J. E. Goodman and J. O'Rourke, eds.), ch. 44, pp. 815–832, Boca Raton, FL: CRC Press LLC, 1997.
- [89] R. Tarjan and U. Vishkin, "An efficient parallel biconnectivity algorithm," *SIAM J. Comput.*, vol. 14, pp. 862–874, 1985.
- [90] R. E. Tarjan, "Depth first search and linear graph algorithms," *SIAM Journal on Computing*, vol. 1, no. 2, pp. 146–160, 1972.
- [91] R. E. Tarjan, *Data Structures and Network Algorithms*, vol. 44 of *CBMS-NSF Regional Conference Series in Applied Mathematics*. Philadelphia, PA: Society for Industrial and Applied Mathematics, 1983.
- [92] A. B. Tucker, Jr., *The Computer Science and Engineering Handbook*. CRC Press, 1997.
- [93] J. D. Ullman, *Principles of Database Systems*. Potomac, MD: Computer Science Press, 1983.
- [94] J. van Leeuwen, "Graph algorithms," in *Handbook of Theoretical Computer Science* (J. van Leeuwen, ed.), vol. A. Algorithms and Complexity, pp. 525–632, Amsterdam: Elsevier, 1990.
- [95] J. S. Vitter, "Efficient memory access in large-scale computation," in *Proc. 8th Sympos. Theoret. Aspects Comput. Sci.*, Lecture Notes Comput. Sci., Springer-Verlag, 1991.
- [96] J. S. Vitter and W. C. Chen, *Design and Analysis of Coalesced Hashing*. New York: Oxford University Press, 1987.
- [97] J.S. Vitter and P. Flajolet, "Average-case analysis of algorithms and data structures," in *Algorithms and Complexity* (J. van Leeuwen, ed.), vol. A of *Handbook of Theoretical Computer Science*, pp. 431–524, Amsterdam: Elsevier, 1990.
- [98] S. Warshall, "A theorem on boolean matrices," *Journal of the ACM*, vol. 9, no. 1, pp. 11–12, 1962.
- [99] J. W. J. Williams, "Algorithm 232: Heapsort," *Communications of the ACM*, vol. 7, no. 6, pp. 347–348, 1964.
- [100] D. Wood, *Data Structures, Algorithms, and Performance*. Reading, Mass.: Addison-Wesley, 1993.

# Índice

- Abstração, 71-72  
Acessibilidade, 531-532  
Acíclico, 531-532  
Adaptabilidade, 70-71  
Adaptador, 209  
Adel'son-Vel'skii, 430  
Adjacente, 508-509  
Aggarwal, 586  
Aho, 204-205, 242-243, 430, 473-474, 506  
Ahuja, 565-566  
Alfabeto, 28-29, 476-477  
Algoritmo, 156-157  
Algoritmo best-fit, 571-572  
Algoritmo de Dijkstra, 542-548  
Algoritmo de Floyd-Warshall, 535-536, 565-566  
Algoritmo de Kruskal, 551-554  
Algoritmo de mark-sweep, 572-573  
Algoritmo de memória externa, 574-584  
Algoritmo de Prim-Jarnik, 554-556  
Algoritmo do crivo, 368  
Algoritmo first-fit, 571-572  
Algoritmo genético de merge, 458-459  
Algoritmo next-fit, 571-572  
Algoritmo worst-fit, 571-572  
Algoritmos de cache, 576-579  
Alocação de memória, 571-572  
Altura, 252-254, 378-379  
Ambiente de desenvolvimento integrado, 61-62  
Amigável, 25-26  
Amortização, 214-215, 461-465  
Análise assintótica, 162-168  
Análise de algoritmo, 156-168  
caso médio, 158-160  
pior caso, 159-160  
Ancestral, 247-248, 529-531  
Anti-simétrica, 292-293  
API, 87-88, 179-180  
Aplicações de missão crítica, 70  
Aproximação de Stirling, 589  
Archimedes, 156-157, 176  
Arco, 508  
Aresta, 248-249, 508  
de partida, 508-509  
destino, 508-509  
incidente, 508-509  
laço com pontos finais coincidentes, 509-510  
múltiplo, 509-510  
origem, 508-509  
paralelo, 509-510  
vértice final, 508-509  
Aresta da árvore, 533-535  
Aresta de descoberta, 519-521, 529-531, 533-535  
Arestas de cruzamento, 529-531, 533-535  
Arestas de entrada, 508-509  
Arestas de retorno, 519-521, 533-535, 560  
Arestas fora da árvore, 533-535  
Ariadne, 518-519  
Aritmética, 39-40  
Armazenamentos associativos, 332  
Arnold, 67-68  
Arquivo de log, 349-350  
Arranjo, 49-53, 102-115  
capacidade, 50-51  
tamanho, 49-50  
Arranjo bi-dimensional, 114-115  
Arranjo de bucket, 335  
Arranjo esparsa, 241-242  
Árvore, 247-290, 511  
altura, 252-254  
aresta, 248-249  
binária, ver Árvore binária  
caminho, 248-249  
de decisão, 259, 374-375, 452-454  
estrutura encadeada, 251  
genérico, 401-404  
nível, 261-262  
nodo, 247  
nodo externo, 247-248  
nodo filho, 247  
nodo interno, 247-248  
nodo pai, 247  
nodo raiz, 247  
ordenado, 248-249  
profundidade, 251-254  
representação de árvore binária, 285  
tipo abstrato de dados, 249-251  
Árvore (2,4), 401-410  
propriedade profundidade, 404-405  
propriedade tamanho, 404-405  
Árvore AVL, 383-392  
fator balanceamento, 388-390  
propriedade altura-balanceamento, 383  
Árvore binária, 259-270, 432-433  
cheia, 250  
completa, 304, 305-310  
estrutura encadeada, 263-270  
filho da esquerda, 259  
filho da direita, 259  
imprópria, 259  
nível, 261-262  
própria, 259  
representação de lista, 270-272  
Árvore binária de pesquisa, 374-380  
inscrição, 376-378  
reestruturação trinodo, 386  
remoção, 377-378  
rotação, 386  
Árvore de cobertura, 511, 519-521, 528-531, 549-550  
Árvore de cobertura mínima, 549-556  
algoritmo de Kruskal, 551-554  
algoritmo de Prim-Jarnik, 554-556  
Árvore de fatias, 285  
Árvore de jogadas, 289  
Árvore de pesquisa, 374  
balanceada, 403-404  
de prioridade, 328  
Árvore de reflexão, 286  
Árvore splay, 392-401  
Árvore transversal, 254-259, 272-282  
caminhamento de Euler, 276-282  
genérico, 278-282  
interfixado, 273-276  
nível, 287  
posfixado, 256-259, 272-274  
prefixado, 254-256, 272-273  
Árvore vermelho-preta, 410-424  
propriedade externa, 410-411  
propriedade interna, 410-411  
propriedade profundidade, 410-411  
propriedade raiz, 410-411  
recolorir, 413-414  
Árvores (*a, b*), 580-582  
Assimétrico, 508-509  
Assinatura, 30-31, 34-35, 37-38, 75-76  
Ataque do estouro do buffer, 50-51  
Ativação dinâmica, 25  
Atributo, 522  
Auto-boxing, 43-44  
Baeza-Yates, 430, 473-474, 506, 586  
Baruva, 564-566  
Bayer, 430, 586  
Bentley, 329-330, 370-371  
BFS, ver Caminhamento em largura  
Blocking, 575-576  
Bloco de comandos, 38  
Bloco try-catch, 85-86  
Booch, 99, 242-243  
Bootstrapping, 403  
Boyer, 506  
Brassard, 176  
B-tree, 581-582  
Bubble-sort, 241-242  
Budd, 99, 242-243  
Buffer, 53-54  
Burger, 586  
Cabeça, 122-123  
Cache, 574-575  
Caesar cipher, 111-112  
Caminhamento em largura, 528-531, 534-535  
Caminhamento interfixado, 374, 377-378, 385-386  
Caminhamento por nível, 287  
Caminhamento pós-ordem, 256-257  
Caminhamento pré-ordem, 254-255  
Caminho de Euler, 556-558

- Caminho, 248-249, 510-511  
dirigido, 510-511  
simples, 510-511  
tamanho, 541-542  
Caminhos mínimos, 541-548  
algoritmo de Dijkstra, 542-548  
Campione, 67-68  
Campo, 24, 31-32, 70  
Capacidade de evolução, 70-71  
Capturada, 85-86  
Caracter curinga, 524  
Cardelli, 99, 204-205  
Carlsson, 329-330  
Cartões CRC, 61-62  
Célula, 49-50  
Chave, 292, 332-333, 348-349, 401-402  
Ciclo de Euler, 556-558, 561  
Ciclo direcionado, 531-532  
Ciclo transversal de Euler, 276-277, 289-290  
Clarkson, 473-474  
Classe, 24-33, 70-72  
Classe aninhada, 233, 299, 388-390, 420-421  
Classe base, 24  
Classe wrapper, 28-29  
Classes numéricas, 28-29  
Clustering, 343  
Codificação, 59-60, 111-112  
Codificação de prefixos, 495-496  
Codificação Huffman, 495-497  
Código de hash, 336-337  
Código de hash polinomial, 337-338  
Coerção, 91-94  
implícito, 43-44  
Cole, 506  
Coleção, 242-243  
Coleção incidente, 515  
Coleta de lixo, 572-574  
mark-sweep, 572-573  
Colocação, 208  
Comer, 586  
Comparador, 293-294  
Complexidade de E/S, 579-580  
Componentes conectados, 511, 521, 529-531  
Compressão de caminhos, 463-464  
Compressão de texto, 495-497  
Concatenação, 29-30, 41-42  
Conjunto independente máximo, 562  
Conjunto, 458-465  
Construtor, 36-37, 78  
Contradição, 168  
Contrapositivo, 168  
Conversão ampliada, 91-92  
Conversão implícita, 43-44  
Conversão reduzida, 91-92  
Core memory, 574-575  
Cormen, 430, 565-566  
Cornell, 67-68  
Criptografia, 111-112  
Crochemore, 506  
Cursor, 128-129, 218, 228-229  
Decodificação, 111-112  
Delimitadores, 55  
Demurjian, 99, 204-205  
Depuração, 59-60  
Deque, 198-199  
implementação com lista encadeada, 199  
tipo abstrato de dados, 198-199  
Descendente, 247-248, 529-531  
Desconexão, 125  
Destino, 508-509  
DFS, ver Pesquisa em profundidade  
Di Battista, 289-290, 565-566  
Diagrama de herança de classe, 24  
Diâmetro, 287  
Dicionário, 348-367  
árvore (2,4), 401-410  
árvore AVL, 383-392  
árvore de binária de pesquisa, 374-380  
árvore vermelho-preta, 410-424  
baseado em lista, 349-351  
não ordenado, 348-351  
operações de atualização, 358-360, 376-378, 384-385, 387-388  
ordenado, 348-349, 363-364, 378-379  
skip list, 356-362  
tabela de pesquisa, 352-356  
tipo abstrato de dados, 348-350, 363-365  
Dígrafo, 531-532  
Dijkstra, 565-566  
Diminuir e conquistar, ver Poda-e-busca  
Distância, 541-542  
Distância de edição, 504, 505  
Divisão, 405-406, 581-582  
Divisão e conquista, 432-436, 447-444  
Down-heap bubbling, 310-312, 318  
Duplo hashing, 343  
Duplo preto, 415-417  
Duplo vermelho, 412-413  
Elemento, 50-51, 292-294, 332  
Empilhar, 214  
Encadeamento de construtores, 77-78  
Encadeamento separado, 340-341  
Encapsulamento, 71-72  
Endereçamento aberto, 342-344  
Enumeração, 34  
Equação recorrente, 441-442, 467, 470  
Escolha gulosa, 497-498  
Escopo, 31-33  
Espaço de probabilidade, 590-591  
Espaço gasto, 156-157, 159-160  
Espaço simples, 590  
Especialização, 76-77  
Estatística de ordem, 465-466  
Estável, 454-455  
Estrutura de dados, 156-157  
secundária, 403-404  
Estrutura de lista de arestas, 513-514  
Estrutura encadeada, 251, 263-264  
Even, 565-566  
Evento, 590-591  
Exceção, 84-86-87  
Expansão binomial, 588  
Exponenciação, 167-168  
Exponencial linear, 590  
Expressão, 39-44  
Extensão, 76-77  
Falha rápida, 241-242  
Fator balanceamento, 388-390  
Fator de círculo, 341-342  
Fator linear de expectativa, 467, 591  
Fatorial, 133-134, 588  
Fechamento transitivo, 531-534  
FIFO, 191  
Fila, 191-199  
implementação com arranjo, 192-195  
implementação com lista encadeada, 195-196  
tipo abstrato de dados, 191-192  
Fila de prioridades, 292-330, 472  
adaptável, 370-371  
implementação com heap, 309-313  
implementação com lista, 298-299  
TAD, 295-296  
Filas com dois finais, ver Deque  
Filho da esquerda, 259  
Filho da direita, 259  
Filhos, 247  
First-in, first-out, 191  
Flanagan, 67-68  
Floresta, 511  
Floyd, 329-330  
Fluxo de controle, 44-49  
Folhas, 247-248  
Fortemente conectado, 531-532  
Fragmentação, 571-572  
Frame, 568  
Framework generics, 94-95  
Função, 35-36  
Função Ackermann, 464-465  
Função constante, 150  
Função cúbica, 152-153  
Função de arredondamento para baixo, 155-156  
Função de arredondamento para cima, 155-156  
Função de compressão, 336, 339-340  
Função de falha, 483-484  
Função de hash, 336, 343  
Função exponencial, 154  
Função linear, 151-152  
Função logarítmica, 150, 587  
natural, 587  
Função  $n \log n$ , 151-152  
Função potência, 167-168  
Função quadrática, 151-152  
Fusão, 409, 580-582  
GAD, ver Grafo dirigido acíclico  
Gamma, 99  
Gauss, 152-153  
Generics, 94-96, 179-180, 187  
Gerador de números pseudorandomicos, 110-111, 356  
Gerenciamento de memória, 568-574, 576-579  
Gibbons, 565-566  
Girar, 111-113  
Golberg, 242-243  
Gonnet, 329-330, 430, 473-474, 586  
Goodrich, 586  
Gosling, 67-68  
Grafo, 508-566  
acíclico, 531-532  
atingibilidade, 531-532, 534-536  
caminhamento em largura, 528-535  
caminhos mínimos, 534-536  
conexo, 511, 529-531  
denso, 522, 536  
dígrafo, 531-532  
dirigido, 508-509, 531-539  
acíclico, 536-539  
fortemente conexo, 531-532  
esparsa, 522  
estruturas de dados, 512-518  
lista de nodos, 512-514  
lista encadeada, 515-516  
matriz adjacente, 516-518  
métodos, 512  
misto, 508-509  
não dirigido, 508-509  
pesquisa em profundidade, 518-528, 532-535

- ponderado, 539-566  
 simples, 509-510  
 tipo abstrato de dados, 508-512  
 transversal, 518-531  
**Grafo biconexo**, 563  
**Grafo bipartido**, 564  
**Grafo completo**, 560  
**Graham**, 565-566  
**Grau**, 152-153, 508-509  
**Grau de entrada**, 508-509  
**Grau de saída**, 508-509  
**Guibas**, 430  
**Guttag**, 99, 204-205, 242-243  
  
**Heap**, 303-320  
 construção bottom-up, 317-320  
**Heap unificado**, 428  
**Heap-sort**, 316-320  
**Hell**, 565-566  
**Hennessy**, 586  
**Herança**, 74-83  
 Herança múltipla, 89-90, 231-232  
**Heurística**, 235-236  
 Heurística do espelho, 480  
 Heurística do salto de caracteres, 480  
 Heurística mover-para-frente, 235-236  
**Hierarquia**, 72-73  
 Hierarquia de memória, 574-575  
**Hoare**, 473-474  
**Hopcroft**, 204-205, 242-243, 430, 473-474, 565-566  
**Horstmann**, 67-68  
**Huang**, 473-474  
  
**Identificador**, 24-25  
**Import**, 59  
**Incidente**, 508-509  
**Independente**, 590-591  
**Índice**, 49-50, 208, 332, 352-353  
 Indução, 168-170  
**In-place**, 325, 450, 573-574  
**Insertion-sort**, 107-108, 131-132, 302-303  
**Interface**, 71-72, 87-93, 179-180  
 Interfixada, 285  
**Internet**, 241-242  
 Interseção dos subproblemas, 499  
 Invariantes de laços, 170-171  
**Inversão**, 302-303, 456-457, 472  
**Irmão**, 247-248  
**Iterador**, 224-230  
  
**JáJá**, 289-290  
**Jarnik**, 565-566  
**Java**, 24-68, 70-96  
 arranjos, 49-53, 102-115  
 conversões, 91-94  
 entrada, 53-55  
 exceções, 84-87  
 expressões, 39-44  
 fluxo de controle, 44-49  
 interfaces, 87-91  
 métodos, 34-38  
 pacotes, 58-59  
 pilha de método, 568-570  
 saída, 49-55  
 tipos, 91-92  
**Java Collections Framework**, 229-230  
**Javadoc**, 62  
**Jogo da Velha**, 114-115, 289  
**Jogos posicionais**, 114-115  
**Jones**, 565-566  
 Junção natural, 241-242  
  
**Karger**, 565-566  
**Karp**, 289-290  
**Klein**, 565-566  
**Kleinberg**, 473-474  
**Knuth**, 147-148, 176, 242-243, 289-290, 329-330, 430, 473-474, 506, 565-566, 586  
**Kosaraju**, 565-566  
**Kruskal**, 565-566  
  
**Laço**, 509-510  
 Laço de for aprimorado, 226  
 Laço for-each, 226  
**Lançar**, 84  
**Landis**, 430  
**Langston**, 473-474  
 Last-in first-out, 178  
**LCS**, ver Maior subsequência comum  
**Lecroq**, 506  
**Lei de DeMorgan**, 168  
**Leiserson**, 430, 565-566  
**Levisse**, 370-371  
 Lexicográfico, 454-455  
**LIFO**, 178  
 Ligação dinâmica, 75  
 Limite de Chernoff, 471, 473-474, 591  
 Linha cache, 575-576  
**Liskov**, 99, 204-205, 242-243  
**Lista**, 208-223, 349-350, 352-353  
 implementação, 209-215  
 tipo abstrato de dados, 208-209  
**Lista circular encadeada**, 128-129, 241-242  
**Lista de adjacência**, 512-513, 515  
**Lista de arestas**, 512-513  
**Lista de favoritos**, 233  
**Lista de nodo**, 216-217  
**Lista encadeada**, 117-133, 185-187, 195-196  
 duplamente encadeada, 121-128, 199, 218-223, 231  
 simplesmente encadeada, 117-121  
**Lista livre**, 571-572  
**Literal**, 39  
**Littman**, 473-474  
**Localizador**, 321  
**Logaritmo natural**, 587  
**Log-star**, 464-465  
  
**Magnanti**, 565-566  
 Maior subsequência comum, 498-501  
**Mapa**, 332-333  
 tabela de hash, 335-349  
 tipo abstrato de dados, 332-334  
  
**Máquina de pesquisa**, 458, 495  
**Máquina virtual Java**, 568, 571-572  
**Marcas**, 62  
**Marcas HTML**, 188-189  
**Matriz**, 114-115  
**Matriz de adjacência**, 512-513, 516-517  
**McCreight**, 506, 586  
**McDiarmid**, 329-330  
**Mediana**, 465-466  
**Megiddo**, 473-474  
**Mehlhorn**, 430, 565-566, 586  
**Membros**, 24  
**Memória externa**, 574-584, 586  
**Memória heap**, 571  
**Memória interna**, 574-575  
**Memória principal**, 574-575  
**Memória virtual**, 575-576  
**Menor ancestral comum**, 287  
**Merge-sort**, 432-442  
 árvore, 432-433  
 genérico, 582-584  
**Método**, 34-38, 70  
 assinatura, 34-35  
 corpo, 34-35  
**Método de Horner**, 175  
**Método division**, 339-340  
**Método guloso**, 497-498, 541-543  
**Método mestre**, 592  
**Métodos de atualização**, 55-56  
**Minimax**, 289  
**Minotauro**, 518-519  
**Mistura**, 89-90  
**Modificadores variáveis**, 31-33  
**Modularidade**, 71-72  
**Módulo**, 194, 588  
**Módulos**, 112-113  
**Moore**, 506  
**Morris**, 506  
**Morrison**, 506  
**MST**, ver Árvore de cobertura mínima  
**Munro**, 329-330  
**Mutuamente independente**, 590-591  
  
**Nível**, 261-262, 528-529  
**Nodo**, 247, 249-250, 508  
 ancestral, 247-248  
 balanceado, 384-385  
 descendente, 247-248  
 externo, 247-248  
 filho, 247  
 interno, 247-248  
 irmão, 247-248  
 não balanceado, 384-385  
 pai, 247  
 raiz, 247  
 redundante, 489-491  
 tamanho, 396-398  
**Nodo-d**, 401-402  
 Notação assintótica, 159-163  
 O, 160-162, 164-168  
 omega, 162-163  
 teta, 162-163  
**Notação pós-fixada**, 202-203, 285  
 Numeração de níveis, 258-259, 270-271  
 Número harmônico, 176, 590  
  
**Objeto**, 24-33, 70  
**Objeto composto**, 35-36  
**Objetos raiz**, 572-573  
**Objetos vivos**, 572-573  
 Operações primitivas, 157-160  
 Operador ponto, 30-31  
 Ordem total, 292-293  
 Ordenação, 107-108, 297-298, 432-455-456  
 bucket-sort, 453-455  
 estável, 454-455  
 fila de prioridades, 297-298  
 heap-sort, 316-320  
 in-place, 316-317, 450  
 insertion-sort, 302-303  
 limite inferior, 452-454  
 memória externa, 582-584  
 merge-sort, 432-442  
 quick-sort, 442-452  
 radix-sort, 454-456  
 selection-sort, 301-302  
 Ordenação topológica, 536-539  
 Origem, 508-509  
**Orlin**, 565-566  
 Otimização dos subproblemas, 499  
**Overflow**, 405-406, 581-582  
  
**Pacote**, 25-26, 58-59  
**Pacote de dados**, 241-242  
**Padrão composição**, 293-294  
**Padrão decorador**, 522-524  
**Padrão Template Method**, 278-282, 459-460, 527  
**Padrões de projeto**, 61-62, 73-74  
 adaptador, 209, 218-219  
 amortização, 214-215  
 comparador, 294-295  
 composição, 293-294  
 decorador, 522-524  
 divisão e conquista, 432-436, 442-444  
 força bruta, 478-479  
 iterador, 224-230  
 método guloso, 497-498  
 método modelo, 278-282, 459-460, 527

- poda e busca, [465-467](#)  
 posição, [216-217](#)  
 programação dinâmica, [499-501](#)
- Pai, [247](#)  
 Palíndromo, [146-147, 504](#)  
 Partição, [461-465](#)  
 Passagem de parâmetro, [36-37](#)  
 Passagem por valor, [569-570](#)  
 Patterson, 586  
 Pesquisa binária, [274-275, 357-356](#)  
 Pesquisa em profundidade, [518-528, 533-534](#)  
 Pilha, [178-191](#)  
   implementação com arranjo, [181-185](#)  
   implementação com lista encadeada, [185-187](#)  
   tipo abstrato de dados, [178-180](#)  
 Pilha de operandos, [569-570](#)  
 Pilha Java, [568](#)  
 Planta baixa futiada, [288](#)  
 Poda-e-busca, [465-467](#)  
 Polimorfismo, [75-75-76](#)  
 Polinomial, [152-153, 175](#)  
 Pontos finais, [508-509](#)  
 Portabilidade, [70-71](#)  
 Posição, [216-217, 249-250, 356-357](#)  
 Posição decorável, [522-523](#)  
 Pratt, [506](#)  
 Prefixo, [476-477](#)  
 Prim, [565-566](#)  
 Probabilidade, [590-591](#)  
 Problema de Josephus, [196-197](#)  
 Problema do caixeiro-viajante, [542-543](#)  
 Procedimento, [35-36](#)  
 Procura de padrões por força bruta, [478-479](#)  
 Profundidade, [251-254](#)  
 Programa contador, [568-569](#)  
 Programação dinâmica, [142-143, 498-501, 535-536](#)  
 Progressão de Fibonacci, [81-82, 589](#)  
 Progressão numérica, [78](#)  
 Projeto orientado a objetos, [70-99](#)  
 Propriedade altura-balanceamento, [383-388](#)  
 Propriedade ordem do heap, [303-304](#)  
 Pseudocódigo, [60-61](#)  
 Pugh, [370-371](#)  
 Quebra-cabeças de soma, [143-144](#)  
 Quick-select randômico, [466](#)  
 Quick-sort, [442-452](#)  
   árvore, [443-444](#)  
 Quick-sort randômico, [448-449](#)  
 Radix-sort, [454-456](#)  
 Raghavan, [370-371, 473-474](#)  
 Raiz, [247](#)  
 Ramachandran, [289-290](#)  
 Randomização, [356-357](#)  
 Rastreamento recursivo, [134](#)  
 Razão ideal, [589](#)  
 Reconhecimento de padrões, [478-486](#)  
   algoritmo de Boyer-Moore, [480-484](#)  
   algoritmo de Knuth-Morris-Pratt, [483-486](#)  
   força bruta, [478-479](#)  
 Recursão, [133-145, 570-571](#)  
   binária, [141-143](#)  
   cauda, [140-141](#)  
   linear, [137-141](#)  
   múltipla, [143-145](#)  
   ordem superior, [141-145](#)  
   rastreamento, [138-139](#)  
 Reed, [329-330](#)  
 Reestrutura, [386](#)  
 Reestruturação trinodo, [385-386, 413-414](#)  
 Referência, [28, 31-33, 38, 75](#)  
 Referência de localização, [235-236](#)  
 Referência-de-localização, [575-576](#)  
 Refinamento, [76-77](#)  
 Reflexível, [292-293](#)  
 Regra de L'Hôpital, [592](#)  
 Rehashing, [347](#)  
 Relaxamento, [543](#)  
 Representação string usando parênteses, [254-255](#)  
 Resolução de colisões, [336, 340-344](#)  
 Reuso, [70-71](#)  
 Ribeiro-Neto, [506](#)  
 Rivest, [430, 565-566](#)  
 Robson, [242-243](#)  
 Robustez, [70](#)  
 Rotação, [386](#)  
   dupla, [386](#)  
   simples, [386](#)  
 Round robin, [196-197](#)  
 Sacola, [369](#)  
 Samet, [586](#)  
 Scanner, [54](#)  
 Schaffer, [329-330](#)  
 Sedgewick, [329-330, 430](#)  
 Seleção, [465-466-467](#)  
 Select-sort, [301-302](#)  
 Semear, [110-111, 356](#)  
 Sentinel, [122-123, 332-333, 348-349](#)  
 Sequência, [208, 229-233](#)  
   tipo abstrato de dados, [231-232](#)  
 Sequência e auditoria, [349-350](#)  
 Símbolo curinga, [524](#)  
 Simétrico, [508](#)  
 Sincronização, [329-330](#)  
 Skip list, [356-362](#)  
 análise, [360-362](#)  
 inserção, [358-359](#)  
 níveis, [356-357](#)  
 operações de atualização, [358-361](#)  
 pesquisa, [357-359](#)  
 remoção, [360-361](#)  
 torres, [356-357](#)  
 Sleator, [430](#)  
 Sobrecarga, [75-76](#)  
 Sobreposição, [75-76](#)  
 Soma geométrica, [590](#)  
 Soma prefixada, [166-167](#)  
 Soma telescópica, [589](#)  
 Somatório, [154, 589](#)  
   geométrico, [154-155](#)  
 Stephen, [506](#)  
 Stop words, [488-489, 505](#)  
 Straggling, [469](#)  
 String, [28-30](#)  
   imutável, [476-477](#)  
   mutável, [477-478](#)  
   nula, [476-477](#)  
   prefixo, [476-477](#)  
   sufixo, [476-477](#)  
   tipo abstrato de dados, [28-30, 476-478](#)  
 Subárvore, [247-248](#)  
 Subárvore direita, [259](#)  
 Subárvore esquerda, [259](#)  
 Subclasse, [74](#)  
 Subgrafo, [511](#)  
 Subseqüência, [498](#)  
 Substituição, [76-77](#)  
 Substring, [476](#)  
 Sufixo, [476-477](#)  
 Superclasse, [74](#)  
 Taarjan, [289-290, 430, 565-566](#)  
 Tabela de hash, [335-349](#)  
   agrupamento, [343](#)  
   arranjo de buckets, [335](#)  
   capacidade, [335](#)  
   colisão, [336](#)  
   encadeamento, [340-341](#)  
   endereçamento aberto, [343-344](#)  
   hashing duplo, [343](#)  
   rehashing, [347](#)  
   resolução de colisões, [340-344](#)  
   teste linear, [342-343](#)  
   teste quadrático, [343](#)  
 Tabela de pesquisa, [352-356](#)  
 TAD, ver Tipo abstrato de dados  
 Tamanho do caminho, [287](#)  
 Tamassia, [289-290, 565-566](#)  
 Tardos, [473-474](#)  
 Tempo de execução, [155-168](#)  
 Teste, [59-60](#)  
 Teste linear, [342-343](#)  
 Teste quadrático, [343](#)  
 Texto cifrado, [111-112](#)  
 Texto limpo, [111-112](#)  
 Theseus, [518-519](#)  
 Tipagem forte, [87-88, 90-91](#)  
 Tipo, [24, 31-32](#)  
 Tipo abstrato de dados, [vii-viii, 71-72](#)  
   árvore, [249-250](#)  
   conjunto, [458-465](#)  
   deque, [198-199](#)  
   dicionário, [348-350, 363-365](#)  
   fila, [191-192](#)  
   fila de prioridade, [292-297](#)  
   grafo, [508-512](#)  
   lista, [208-209](#)  
   lista ou seqüência, [216-219](#)  
   mapa, [321-334](#)  
   partição, [461-465](#)  
   pilha, [178-180](#)  
   seqüência, [231-232](#)  
   string, [28-30, 476-478](#)  
 Tipo base, [26-27, 31-32, 38](#)  
 Tipo genérico, [94-95](#)  
 Tipo primitivo, [26-27](#)  
 Token, [188](#)  
 Tokens, [55](#)  
 Torre dos dois, [464-465](#)  
 Torres de Hanoi, [146-147](#)  
 Trailer, [122-123](#)  
 Transferência, [409](#)  
 Transitivo, [292-293](#)  
 Trie, [497-495](#)  
   comprimido, [489-491](#)  
   padrão, [487-488](#)  
 Tsakalidis, [430](#)  
 Ullman, [204-205, 242-243, 430, 473-474, 586](#)  
 Último nodo, [304](#)  
 Unboxing, [43-44](#)  
 Underflow, [409, 581-582](#)  
 União-pelo-tamanho, [463-464](#)  
 Union-find, [461-465](#)  
 Up-heap bubbling, [310-312](#)  
 Valor esperado, [590-591](#)  
 Van Leeuwen, [565-566](#)  
 Variáveis de instância, [31-32, 237](#)  
 Variável local, [38](#)  
 Variável randômica, [590-591](#)  
 Varredura, [357-358](#)  
 Vértice, [508](#)  
   grau, [508-509](#)  
   grau de entrada, [508-509](#)  
   grau de saída, [508-509](#)  
 Vértices finais, [508-509](#)  
 Vetor, [208](#)  
 Vetor de bits, [470](#)  
 Vishkin, [289-290](#)  
 Vitter, [586](#)  
 Walrath, [67-68](#)  
 Wegner, [99, 204-205](#)  
 Williams, [329-330](#)  
 Wood, [242-243](#)  
 Zig, [393, 398-399](#)  
 Zig-zag, [393, 398-399](#)  
 Zig-zig, [392, 398-399](#)



Hidden page



