



Roberto Ferrari  
Marcela Xavier Ribeiro  
Rafael Loosli Dias  
Maurício Falvo



# Estruturas de dados com Jogos



# Estruturas de Dados com Jogos

---

Roberto Ferrari

Marcela Xavier Ribeiro

Rafael Loosli Dias

Maurício Falvo



---

# Sumário

---

[Capa](#)

[Folha de rosto](#)

[Cadastro](#)

[Copyright](#)

[Apresentação](#)

[Objetivos do livro](#)

[Os jogos e os capítulos](#)

[Materiais Complementares](#)

[Acesso aos Materiais Complementares](#)

[Notação conceitual](#)

[A chave para um bom aproveitamento: praticar](#)

[Encare o desafio!](#)

## **Parte I: Pilhas**

[Desafio 1. Desenvolver uma adaptação do jogo \*FreeCell\*](#)

[Como você desenvolveria um jogo como o \*FreeCell\*?](#)

[Seu desafio: desenvolver uma adaptação do \*FreeCell\*](#)

Por onde começar?

## Capítulo 1. Tipos Abstratos de Dados

Seus objetivos neste capítulo

1.1 Queremos desenvolver um jogo. E agora?

1.2 Definição de Tipos Abstratos de Dados

1.3 Exemplo: *FreeCell*

Exercício 1.1 Transfere Carta

1.4 Qual a melhor maneira de aumentar o volume da televisão?

1.5 O que é um bom programa?

1.6 Vantagens da utilização de Tipos Abstratos de Dados

Referências e leitura adicional

## Capítulo 2. Pilhas com Alocação Sequencial e Estática

Seus objetivos neste capítulo

2.1 O que é uma Pilha?

2.2 Operações de uma Pilha

2.3 Alocação de memória para conjuntos de elementos

2.5 Implementando uma Pilha com Alocação Sequencial e Estática

2.6 Abrir ou não a televisão?

2.7 Projeto *FreeCell*: Pilha Burra ou Pilha Inteligente?

Referências e leitura adicional

## Parte II: Filas

### Desafio 2. Desenvolver uma adaptação do jogo *Snake*

Cobras e Comidas coloridas

Como você desenvolveria um jogo como o *Snake*?

Seu desafio: desenvolver uma adaptação colorida do *Snake*

Por onde começar?

## Capítulo 3. Filas com Alocação Sequencial e Estática

Seus objetivos neste capítulo

3.1 O que é uma Fila?

3.2 O jogo *Snake*

3.3 Operações do tipo abstrato de Dados Fila

3.4 Implementando um TAD Fila com Alocação Sequencial e Estática de Memória

3.5 Botão da televisão ou chave de fenda?

3.6 Projeto *Snake*: Cobra como uma Fila de cores

Referências e leitura adicional

## Capítulo 4. Listas Encadeadas

Seus objetivos neste capítulo

4.1 Alocação Sequencial versus Alocação Encadeada

4.2 Listas Encadeadas: conceito e representação

4.3 Implementando uma Pilha como uma Lista Encadeada

4.4 Implementando uma Fila como uma Lista Encadeada

Referências e leitura adicional

## Capítulo 5. Listas Encadeadas com Alocação Dinâmica

Seus objetivos neste capítulo

5.1 Alocação Dinâmica de Memória para um conjunto de elementos

5.2 Alocação Dinâmica nas linguagens C e C++

5.3 Nós de uma Lista Encadeada alocados dinamicamente

Referências e leitura adicional

## **Parte III: Listas Cadastrais**

Desafio 3. Desenvolver um jogo que use listas de elementos

Como implementar a Lista de Compras?

Por onde começar?

### **Capítulo 6. Listas Cadastrais**

Seus objetivos neste capítulo

6.1 O que é uma Lista Cadastral?

6.2 Operações de um TAD Lista Cadastral

6.3 Implementando uma Lista Cadastral sem elementos repetidos como uma Lista Encadeada Ordenada

6.4 Outras implementações de Lista Cadastral

6.5 Avançando o Projeto *Spider Shopping*

### **Capítulo 7. Generalização de Listas Encadeadas**

Seus objetivos neste capítulo

7.1 Listas Duplamente Encadeadas

7.2 Listas com Nô Header

7.3 Operações de Baixo Nível para Listas Encadeadas

7.4 Generalização de Listas Encadeadas

## **Parte IV: Árvores**

Desafio 4. Desenvolver um game com previsão de jogadas

Como escolher a melhor opção de jogada?

Seu desafio: desenvolver um game com previsão de jogadas

Por onde começar?

## Capítulo 8. Árvores

[Seus objetivos neste capítulo](#)

[8.1 Árvores: conceito e representação](#)

[8.2 Árvores Binárias e Árvores Binárias de Busca](#)

[8.3 Algoritmos recursivos para Árvores Binárias de Busca](#)

[8.4 Árvores Binárias de Busca: inserir e eliminar elementos](#)

[8.5 Por que uma Árvore Binária de Busca é boa?](#)

[8.6 Aplicações de Árvores](#)

[8.7 Avanço de projeto: o Desafio 4](#)

[Links](#)

[Referências e leitura adicional](#)

## Capítulo 9. Árvores Balanceadas

[Seus objetivos neste capítulo](#)

[9.1 Conceito de Balanceamento](#)

[9.2 Inserir elementos em uma ABB Balanceada](#)

[9.3 Remover elementos de uma ABB Balanceada](#)

[Links](#)

[Referências e leitura adicional](#)

[Seu próximo desafio](#)

---

# Cadastro

---



---

# Copyright

---

© 2014, Elsevier Editora Ltda.

Todos os direitos reservados e protegidos pela Lei nº 9.610, de 19/02/1998.  
Nenhuma parte deste livro, sem autorização prévia por escrito da editora, poderá  
ser reproduzida ou transmitida sejam quais forem os meios empregados:  
eletrônicos, mecânicos, fotográficos, gravação ou quaisquer outros.

*Copidesque:* Ivone Teixeira

*Revisão:* Carmem Becker

*Editoração Eletrônica:* Thomson Digital

Elsevier Editora Ltda.

Conhecimento sem Fronteiras

Rua Sete de Setembro, 111 – 16º andar

20050-006 – Centro – Rio de Janeiro – RJ – Brasil

Rua Quintana, 753 – 8º andar

04569-011 – Brooklin – São Paulo – SP

Serviço de Atendimento ao Cliente

0800-026 5340 – [atendimento1@elsevier.com](mailto:atendimento1@elsevier.com)

ISBN: 978-85-352-7804-0

ISBN (versão digital): 978-85-352-7805-7

ISBN (versão eletrônica): 978-8-535-27805-7

---

## Nota

Muito zelo e técnica foram empregados na edição desta obra. No entanto,  
podem ocorrer erros de digitação, impressão ou dúvida conceitual. Em  
qualquer das hipóteses, solicitamos a comunicação ao nosso Serviço de

Atendimento ao Cliente, para que possamos esclarecer ou encaminhar a questão.

Nem a editora nem o autor assumem qualquer responsabilidade por eventuais danos ou perdas a pessoas ou bens, originados do uso desta publicação.

**CIP-BRASIL. CATALOGAÇÃO NA PUBLICAÇÃO SINDICATO NACIONAL DOS EDITORES DE LIVROS, RJ**

E85

Estruturas de dados com jogos / Roberto Ferrari ... [et al.]. - 1. ed. - Rio de Janeiro : Elsevier, 2014.

24 cm.

ISBN 978-85-352-7804-0

1. Jogos por computador. 2. Animação por computador. 3. Estruturas de dados (Computação). 4. Programação (Computadores). 5. Computação. I. Ferrari, Roberto. II. Título.

14-10205      CDD: 005.1

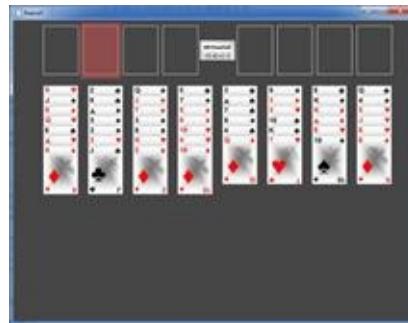
CDU: 004.42

---

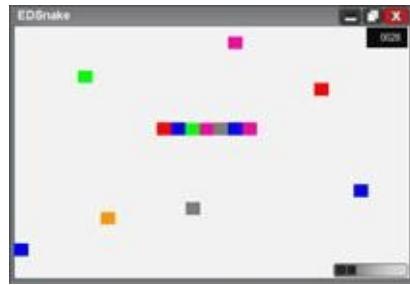
# Apresentação

---

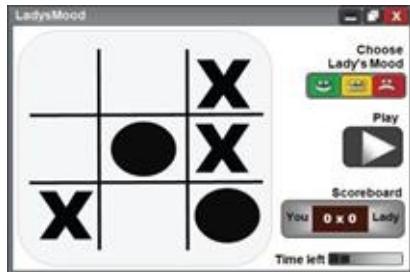
Conhece um jogo chamado ***FreeCell***? É um jogo de paciência, muito conhecido, em que você precisa manipular as cartas de um baralho, com o objetivo de colocá-las em sequência. As cartas são armazenadas em Pilhas e só podem ser movimentadas de acordo com regras bem específicas. Como você desenvolveria um jogo como o FreeCell? Como faria para armazenar as cartas em Pilhas e garantir que elas sejam movimentadas de acordo com as regras do jogo?



E jogos do tipo ***Snake***, você já jogou? O personagem principal desse jogo é uma cobra composta por diversos pedacinhos coloridos. Ao se movimentar e comer alguma coisa, a cobra vai ganhando ou perdendo pedacinhos, crescendo ou diminuindo, e tornando o jogo cada vez mais difícil. É um jogo de habilidade. Se você fosse desenvolver um jogo do tipo *Snake*, como representaria no programa a cobra e cada um de seus pedacinhos? Como faria para retirar ou acrescentar os pedacinhos e para mantê-los na sequência correta?



Pense agora em um jogador humano disputando contra o computador. O jogo em si pode ser simples, como um *Jogo da Velha*, por exemplo, mas não um jogo da velha qualquer; um jogo inteligente, no qual o computador escolhe a melhor jogada em função da jogada do adversário, sempre visando situações com as maiores chances de vitória. Como você implementaria essa inteligência em um jogo? Como faria o seu jogo prever todas as possíveis jogadas e depois escolher, conscientemente, a melhor opção?



## Objetivos do livro

O livro *Estruturas de dados com jogos* tem por objetivo prepará-lo para implementar estruturas de dados para representação e armazenamento de conjuntos de informações em um programa. Conjuntos de informações como uma pilha de cartas, uma fila de espera, uma lista de passageiros ou de compras, por exemplo.

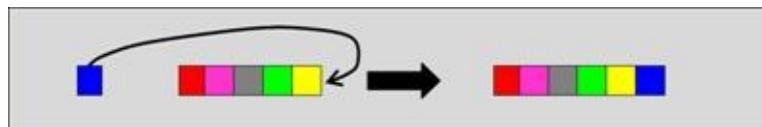
O foco do livro não são os jogos em si. A ideia é que você aprenda estruturas de dados — desenvolva habilidades sólidas de programação, enquanto cria alguns jogos. Os jogos tornarão seu crescimento mais divertido!

## Os jogos e os capítulos

São quatro jogos para você desenvolver. Quatro desafios. O primeiro desafio é desenvolver uma adaptação do *FreeCell*. Para desenvolver um bom jogo, você estudará o [Capítulo 1](#) e o [Capítulo 2](#) de *Estruturas de dados com jogos*. Você vai conhecer uma estrutura de dados chamada **Pilha**. Você vai aprender como implementar uma Pilha de Cartas e como deixar seus jogos bem flexíveis. Assim será fácil ajustar o software do jogo a novas regras e a novas situações.



Seu segundo desafio é desenvolver um jogo do tipo *Snake*. Enquanto desenvolve seu próprio *Snake*, você estudará os [Capítulos 3, 4 e 5](#), e conhecerá outra estrutura de dados chamada **Fila**. Você perceberá que uma Fila é muito útil para implementar um jogo como o *Snake*. Você irá comparar duas técnicas de implementação e escolher uma dessas técnicas para implementar sua própria Fila e seu próprio *Snake*.

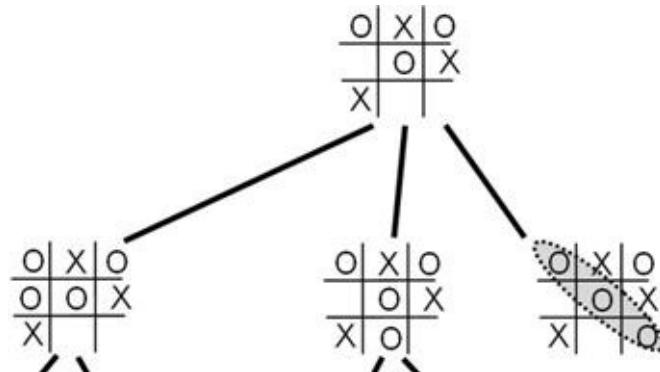


No terceiro desafio, você será apresentado a um jogo chamado ***Spider Shopping***. Seu objetivo será desenvolver uma adaptação desse jogo ou um jogo diferente que mantenha algumas das características do *Spider Shopping*. Para isso, estudará outro tipo de estrutura de dados: a **Lista Cadastral**. Nos [Capítulos 6 e 7](#), você estudará diversas técnicas para implementar uma Lista Cadastral e usará sua criatividade para conceber seu próprio jogo — sua própria aplicação

das Listas Cadastrais.



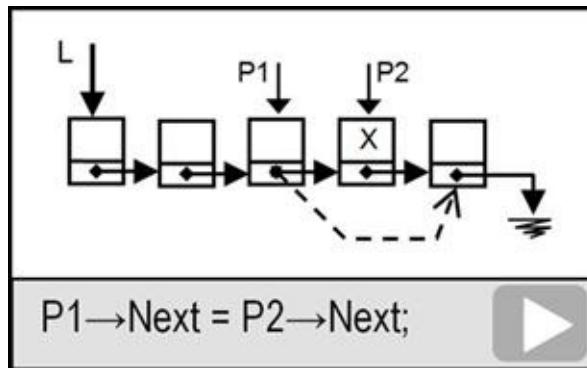
O ponto-chave do quarto desafio é construir a inteligência do jogo. Poderá ser o **Jogo da Velha** ou outro jogo, mas o jogo precisará prever as próximas jogadas, para então optar pela alternativa que ofereça maior chance de vitória. Estudando os [Capítulos 8 e 9](#), você conhecerá uma estrutura de dados chamada **Árvore** e algumas de suas aplicações. Você perceberá que com uma Árvore poderá implementar a **Previsão de Jogadas** e a inteligência do seu jogo.



Nos quatro desafios, você terá a oportunidade de adaptar o jogo sugerido e criar seu próprio jogo: um jogo com personalidade própria; um jogo com a sua cara! Você poderá até propor jogos totalmente novos. Mas, para que você desenvolva as habilidades pretendidas, os jogos terão que manter algumas das características de cada desafio. Em essência, os quatro jogos precisarão ser aplicações de quatro estruturas de dados fundamentais que estudaremos ao longo deste livro: Pilhas, Filas, Listas Cadastrais e Árvores.

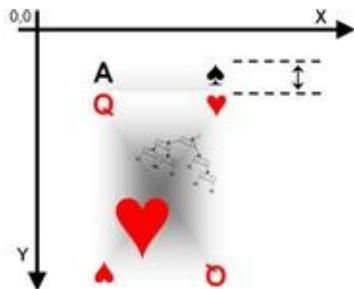
## Materiais Complementares

Para facilitar sua compreensão de alguns dos conceitos e algoritmos, você poderá assistir a **vídeos** com explicações e exemplos. Além dos vídeos, os Materiais Complementares de *Estruturas de dados com jogos* contêm algumas **animações**. As animações lhe mostrarão graficamente a execução de um trecho de programa. Você poderá interagir com as animações, avançando a execução do programa passo a passo enquanto observa uma representação visual do que está acontecendo. O papel dos vídeos e das animações é complementar a leitura, mas nunca substituí-la. Estude o texto, faça os exercícios e complemente seu estudo com os vídeos e animações dos Materiais Complementares.



Se você ainda não tiver prática em programação utilizando interfaces gráficas, poderá consultar nos Materiais Complementares um **Tutorial de Programação Gráfica**. Passo a passo, o tutorial lhe ajudará a instalar uma biblioteca gráfica e a desenvolver seu primeiro jogo. Estude o tutorial em paralelo ao estudo das estruturas de dados. Coloque um foco em seu estudo: viabilizar o desenvolvimento de seus quatro jogos. Será mais produtivo e mais divertido aprender assim. A biblioteca gráfica utilizada no tutorial é apenas uma sugestão;

você poderá utilizar a ferramenta gráfica de sua preferência.



Você encontrará também, nos Materiais Complementares, um **Banco de Jogos**. São **jogos-exemplo**, desenvolvidos por pessoas como você, que aceitaram o desafio de aprender estruturas de dados de um modo bem divertido. E que tal você desenvolver jogos legais e disponibilizá-los nesse banco de jogos? Que tal tornar seus jogos públicos e conhecidos? Que tal participar de uma **competição de jogos**?



## Acesso aos Materiais Complementares

Os Materiais Complementares estarão à sua disposição a qualquer tempo. Consulte-os quando quiser a partir do link a seguir:

<http://www.elsevier.com.br/edcomjogos>

Você também encontrará, ao longo do texto, **ícones** representando os **vídeos**, as **animações**, o **Tutorial de Programação Gráfica** e os **jogos-exemplo**. Quando encontrar um desses ícones, significa que existem materiais complementares pertinentes ao assunto que você está estudando e que esse será um bom momento para você consultar esses materiais, caso desejar. Dependendo de qual versão do livro estiver utilizando, você poderá inclusive acessar os Materiais Complementares clicando nos ícones ou fazendo a leitura do QR-Code associado ao link.

			
Vídeos	Animações	Tutorial de Programação Gráfica	Banco de Jogos

## Notação conceitual

Os algoritmos que estudaremos serão apresentados e discutidos em **notação conceitual**, sem construções de uma linguagem de programação específica. Isso permite que os conceitos sejam compreendidos em sua essência e implementados em qualquer linguagem. Para exemplificar a implementação dos conceitos em uma linguagem de programação, em alguns momentos-chave são apresentados **códigos em C e em C++**. Mas é perfeitamente possível utilizar este livro como referência e implementar os algoritmos em qualquer outra linguagem de programação.

```
PegaOPrimeiro( L1, X, TemElemento );
Enquanto TemElemento == Verdadeiro Faça
{
    Se EstaNaLista(L2, X)
    Então Insere (L3, X, Ok);
    PegaOPróximo( L1, X, TemElemento );
};
```

```
struct Node {  
    char Info;  
    struct Node *Next; };  
typedef struct Node *NodePtr;  
NodePtr P;  
P = new Node;
```

## A chave para um bom aproveitamento: praticar

Não tenha como meta apenas *conhecer* ou *entender*. Isso não é suficiente. Tenha como meta *desenvolver habilidades* para projetar estruturas de armazenamento de dados, para implementar essas estruturas e para utilizá-las na prática, seja no mundo dos games, seja em outro contexto. E para realmente desenvolver essas habilidades, você precisará fazer uma coisa: praticar.

Pratique! Pratique muito! Você encontrará muitos exercícios neste livro e soluções para boa parte deles. Faça os exercícios! Proponha uma primeira solução, erre, faça de novo, erre menos, faça de novo e então acerte! Só consulte as soluções fornecidas após propor sua própria solução ou, pelo menos, após tentar exaustivamente.

## Encare o desafio!

Aprenda estruturas de dados. Aprenda pra valer! Em paralelo a isso, desenvolva seus jogos! Dê personalidade própria a eles! Torne-os divertidos! Projete uma interface legal! Mostre os jogos para seus amigos! Participe de uma competição de jogos! Envolva seus colegas; desenvolva jogos junto com eles; cresça junto com eles! Divulgue seus jogos na internet! Faça seus jogos bombar!

Aprender a programar pode ser divertido!



<http://www.elsevier.com.br/edcomjogos>

---

## **PARTE I**

# Pilhas

### **OUTLINE**

---

Desafio 1 Desenvolver uma adaptação do jogo *FreeCell*

Capítulo 1 Tipos Abstratos de Dados

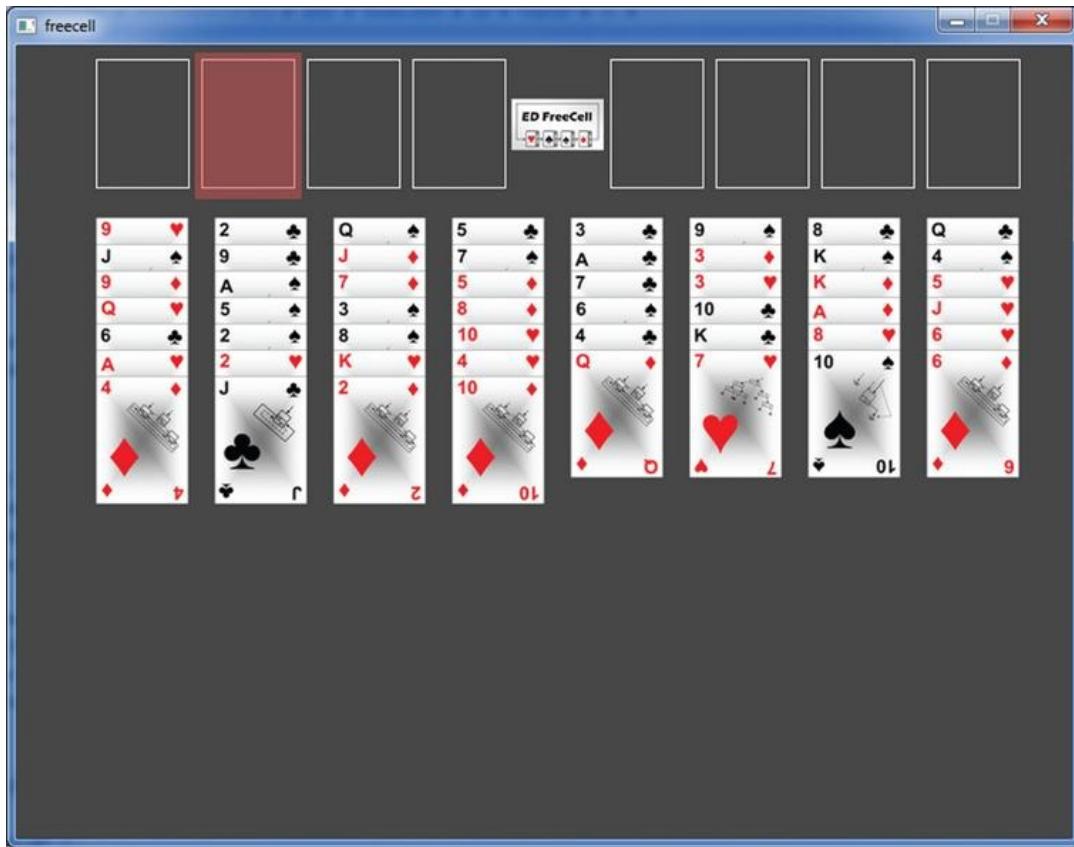
Capítulo 2 Pilhas com Alocação Sequencial e Estática

---

## DESAFIO 1

# Desenvolver uma adaptação do jogo *FreeCell*

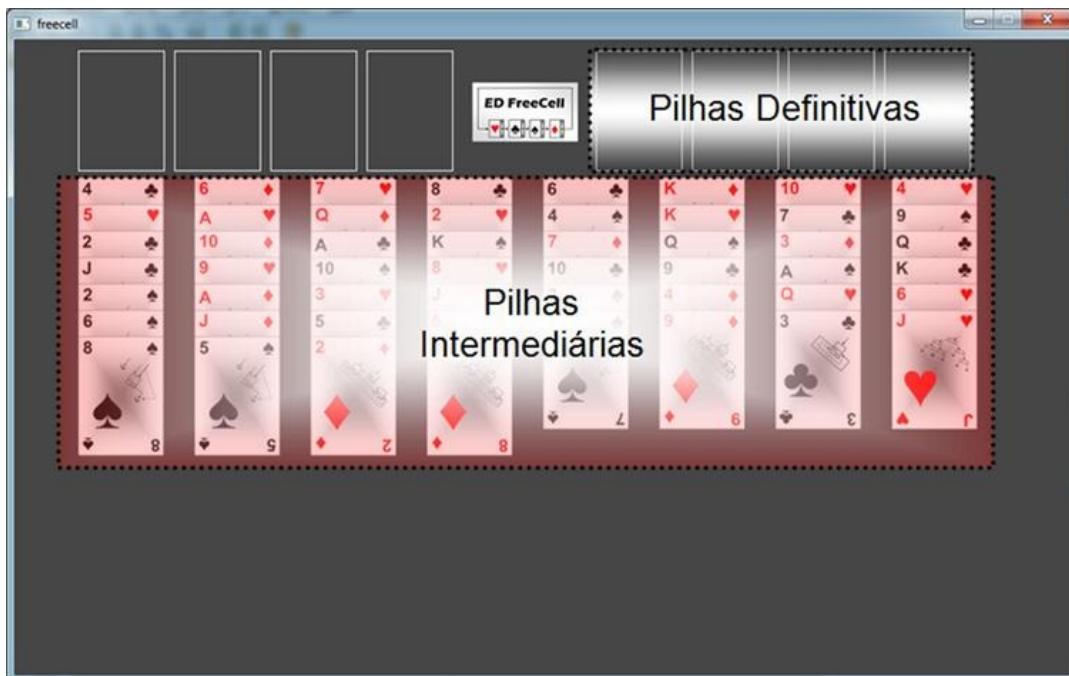
---



**FIGURA D1.1** FreeCell: cartas das oito Pilhas Intermediárias devem ser ordenadas e colocadas em quatro Pilhas Definitivas

O *FreeCell* é um jogo de paciência, que pode ser jogado também com um baralho de verdade, e não apenas no mundo virtual. O *FreeCell* ficou muito conhecido depois que passou a ser distribuído junto com o sistema operacional Windows. Caso você não encontre esse jogo em seu computador, procure em sites de jogos, como o Mania de Jogos (veja o link 1), o [freecell.com.br](http://freecell.com.br) (link 2) ou outros.

Inicialmente, as 52 cartas de um baralho são distribuídas aleatoriamente em oito pilhas de cartas, que ficam na parte de baixo da tela. Essas oito pilhas de cartas são chamadas de *Pilhas Intermediárias*. O objetivo do jogo é empilhar todas as 52 cartas nas quatro pilhas que ficam no canto superior direito da tela, chamadas de *Pilhas Definitivas*. Mas, nas *Pilhas Definitivas*, é preciso empilhar as cartas em sequência (A, 2, 3, 4, 5, 6, 7, 8, 9, 10 J, Q, K) e organizadas por naipes (copas- ♠, espadas- ♦, paus- ♣, e ouros-♦). Há uma *Pilha Definitiva* para cada naipe.



**FIGURA D1.2** *Pilhas Intermediárias* — ordem decrescente e cores alternadas. *Pilhas Definitivas* — cartas do mesmo naipe em ordem crescente.

A movimentação das cartas precisa seguir algumas regras. Só é possível retirar a carta que está no topo da *Pilha Intermediária*. É possível passar cartas de uma *Pilha Intermediária* para outra, mas inserindo as cartas sempre no topo da pilha, em ordem decrescente, e em cores alternadas — pretas sobre vermelhas e vermelhas sobre pretas.

Existem ainda quatro *Espaços para Movimentação de Cartas*, no canto superior esquerdo da tela. Nesses espaços é possível colocar qualquer carta, temporariamente. O uso desses espaços pode ajudar bastante no andamento do jogo.

Para se familiarizar com as regras, jogue algumas partidas no *FreeCell* do seu computador, no *FreeCell* de um site de jogos (consulte, por exemplo, os links 1



ou 2) ou em alguma adaptação do *FreeCell* disponível no Banco de Jogos.

## Como você desenvolveria um jogo como o *FreeCell*?

Se você tivesse que desenvolver um jogo como o *FreeCell*, como começaria o trabalho? Pense, por exemplo, nas pilhas de cartas: oito *Pilhas Intermediárias* e quatro *Pilhas Definitivas*. Como você faria para armazenar os valores e os naipes das cartas? Como armazenaria a sequência das cartas em cada uma das pilhas? Como implementaria as regras que restringem a movimentação de cartas? Qual seria a melhor estratégia para representar, armazenar e manipular as informações referentes a essas pilhas de cartas dentro de seu programa?

## Seu desafio: desenvolver uma adaptação do *FreeCell*

Nesse seu mergulho nas estruturas de dados e nos jogos, seu primeiro desafio é desenvolver uma adaptação do *FreeCell*. Use a criatividade e construa seu próprio *FreeCell*. Faça alguns ajustes nas regras do jogo; dê ao seu jogo um nome criativo! Crie seu próprio jogo, mas mantenha a característica fundamental da manipulação das pilhas de cartas: permita adicionar cartas apenas no topo da pilha e permita a retirada apenas da carta que está no topo da pilha.

## Por onde começar?

Comece estudando os [Capítulos 1 e 2](#) de *Estruturas de dados com jogos*. Esses capítulos o ajudarão a escolher uma estrutura de armazenamento adequada e a definir uma boa estratégia para o desenvolvimento do seu *FreeCell*. Em paralelo,



você pode iniciar o estudo do Tutorial de Programação Gráfica.

Construa seu próprio *FreeCell*! Distribua para os seus amigos! Aprender a programar pode ser divertido!

[Consulte nos Materiais Complementares](#)

Banco de Jogos: Adaptações do *FreeCell*  
Tutorial de Programação Gráfica



<http://www.elsevier.com.br/edcomjogos>

## Links

1. Mania de Jogos. Disponível em: <http://www.maniadejogos.com/jogos/Freecell> (acesso em outubro de 2013).
2. **Freecell.com.br**. Disponível em: <http://freecell.com.br/paciencia-freecell> (acesso em outubro de 2013).

---

## CAPÍTULO 1

---

# Tipos Abstratos de Dados

---

## Seus objetivos neste capítulo

- Entender o conceito de Tipos Abstratos de Dados e o modo de utilizá-lo no desenvolvimento de programas.
- Perceber que o uso de Tipos Abstratos de Dados dá ao software maior portabilidade, maior potencial para reutilização, reduz custos de desenvolvimento e de manutenção.
- Conscientizar-se quanto à importância de adotar uma estratégia que agregue portabilidade e reusabilidade aos jogos que você desenvolverá.

## 1.1 Queremos desenvolver um jogo. E agora?

Suponha que você e mais dois amigos tenham decidido desenvolver um jogo no mundo virtual — um software. E agora? Qual o primeiro passo?

Quando queremos desenvolver um software, o primeiro passo é decidir *o que* desenvolver. O ciclo de vida tradicional do desenvolvimento de software envolve as fases de análise, projeto, implementação, teste e manutenção ([Figura 1.1](#)). Na fase de análise, determinamos *o que* o software precisa fazer, quais problemas deverá resolver e quais informações deverá manipular.



**FIGURA 1.1** Fases do desenvolvimento de software.

Na fase de projeto, o objetivo é definir *como* o software será desenvolvido. Estamos falando de uma definição geral, não detalhada. Na fase de projeto, escolhemos a arquitetura do software, definimos quais serão os principais módulos e, com isso, podemos dividir o trabalho entre os desenvolvedores — cada membro da equipe fica responsável pela implementação de um dos módulos.

As outras fases referem-se à implementação (codificação em uma linguagem de programação), teste (para garantir que o software funcione segundo o esperado) e manutenção (ajuste ou evolução nas funcionalidades do software, ao longo de seu período de utilização). Para uma leitura complementar sobre o ciclo de vida tradicional do desenvolvimento de software, consulte [Sommerville \(2003\)](#) e [Pressman \(1995\)](#).

Vamos supor que você e seus dois amigos aceitaram o Desafio 1 do livro *Estruturas de dados com jogos* e decidiram desenvolver um jogo como o *FreeCell*. Já sabem *o que* desenvolver e estão agora na fase de projeto, definindo *como* desenvolver. E vocês estão decididos a escolher a melhor estratégia para desenvolver esse jogo. É precisamente nesse contexto, na fase de projeto do software, em busca da melhor estratégia, que você precisa conhecer e utilizar o conceito de Tipos Abstratos de Dados.

## 1.2 Definição de Tipos Abstratos de Dados

Um Tipo Abstrato de Dados é formado por uma coleção de **Dados** a serem armazenados e um conjunto de **Operações** ou ainda **Operadores** que podem ser aplicados para manipulação desses Dados ([Langsam, Augenstein e Tenembbaum, 1996](#); [Celes, Cerqueira e Rangel, 2004](#)). Toda manipulação desse conjunto de Dados, para fins de armazenamento e recuperação, deve ser realizada exclusivamente através dos Operadores.

Em que momento devem ser identificados e projetados os Tipos Abstratos de Dados? Na fase de projeto do software, ou seja, no momento em que definimos uma visão geral, não detalhada, de *como* o software será desenvolvido.

**Definição — Tipo Abstrato de Dados (TAD):**

Um Tipo Abstrato de Dados é constituído por um conjunto de **Dados** a serem armazenados e por um grupo de **Operadores** que podem ser aplicados para manipulação desses Dados.

**Manipulação dos Dados armazenados:**

O armazenamento e a recuperação dos Dados devem ser realizados **exclusivamente** através dos Operadores do TAD.

**Em que momento identificar e projetar um TAD:**

Na fase de projeto do software.

**FIGURA 1.2** Definição de Tipo Abstrato de Dados.

### 1.3 Exemplo: *FreeCell*

Como o conceito de Tipos Abstratos de Dados pode ser utilizado no projeto do *FreeCell*, o jogo do Desafio 1 que queremos desenvolver? Temos no *FreeCell* algum conjunto de Dados a serem armazenados? Sim! Temos, por exemplo, os valores e os naipes das cartas e também a sequência dessas cartas nas *Pilhas Intermediárias*.

Um dos Operadores para manipulação desses Dados tem por objetivo retirar a carta que está no topo da pilha. Um segundo Operador poderia ser colocar uma carta no topo da pilha, caso o valor da carta estiver na sequência correta. Nessas *Pilhas Intermediárias*, só é possível inserir cartas no topo da pilha em ordem decrescente e em cores alternadas — pretas sobre vermelhas e vermelhas sobre pretas.

TAD Pilha Intermediária do <i>FreeCell</i>		
K ♣	Coleção de Dados a Serem Armazenados	Operadores Para Manipulação
	<p>Para cada uma das <i>Pilhas Intermediárias</i>:</p> <ul style="list-style-type: none"> <li>• As cartas que estão na pilha - valor e naipe de cada carta;</li> <li>• A sequência das cartas na pilha.</li> </ul>	<ul style="list-style-type: none"> <li>• Retira a carta que está no topo da pilha;</li> <li>• Coloca uma carta no topo da pilha, se o valor estiver na sequência correta.</li> </ul>

**FIGURA 1.3** Exemplo de TAD Pilha Intermediária do *FreeCell*.

A partir da definição do TAD Pilha Intermediária do *FreeCell*, o aplicativo *FreeCell* como um todo deve realizar o armazenamento das cartas nas *Pilhas Intermediárias* de modo abstrato, ou seja, sem se preocupar com os detalhes de como esses dados são efetivamente armazenados. A recuperação dos dados também deve ocorrer de modo abstrato. Funcionaria como se as *Pilhas Intermediárias* fossem *caixas-pretas*, com uma operação para retirar a carta que está no topo da pilha e outra para colocar uma carta no topo da pilha, caso a sequência estiver correta. Os demais módulos do programa simplesmente pediriam “Retire a carta que está no topo desta pilha” ou “Coloque esta carta no topo daquela pilha”, e a *caixa-preta* que cuida disso daria um jeito de atender as solicitações.

Suponha que, na fase de projeto, tenha sido definido o TAD Pilha Intermediária do *FreeCell* e suas operações detalhadas conforme mostra a [Figura 1.4](#).

Operações e parâmetros	Funcionamento
Desempilha (Pilha, Carta, DeuCerto)	Retira a Carta que está no topo da Pilha passada como parâmetro. Se a operação for bem-sucedida, o parâmetro DeuCerto retornará o valor Verdadeiro, e o parâmetro Carta retornará o valor da carta retirada do topo da Pilha. Caso não houver carta a ser desempilhada, o parâmetro DeuCerto retorna o valor Falso.
EmpilhaNaSequência (Pilha, Carta, DeuCerto)	Insere a Carta passada como parâmetro na Pilha passada como parâmetro, caso o valor da Carta estiver na sequência correta para uma <i>Pilha Intermediária</i> — ordem decrescente e em cores alternadas. O parâmetro DeuCerto retornará o valor Verdadeiro se a Carta foi empilhada corretamente, e o valor Falso, caso contrário.
EmpilhaSempre (Pilha, Carta)	Insere a Carta passada como parâmetro no topo da Pilha passada como parâmetro, mesmo se a Carta não estiver na sequência correta para uma <i>Pilha Intermediária</i> .

**FIGURA 1.4** Operações do TAD Pilha Intermediária.

Suponha também que outro projetista de software tenha ficado encarregado de implementar o TAD Pilha Intermediária do *FreeCell*. Coube a você desenvolver a operação para transferir uma carta entre duas *Pilhas Intermediárias*, caso a *Carta* retirada da *Pilha de Origem* estiver na sequência correta na *Pilha de Destino*, conforme as regras do *FreeCell*. Como você desenvolveria essa operação sem ter a menor ideia quanto aos detalhes da implementação da Pilha Intermediária?

## Exercício 1.1 Transfere Carta

Desenvolva uma operação para transferir uma *Carta* entre duas *Pilhas Intermediárias* do *FreeCell*. A transferência só poderá ser realizada se a *Carta* retirada da *PilhaOrigem* estiver na sequência correta na *PilhaDestino*, conforme as regras de funcionamento do *FreeCell*. Se a *Carta* não for efetivamente transferida, deve retornar à *PilhaOrigem*. Essa operação deve ser implementada com o uso dos operadores do TAD Pilha Intermediária do *FreeCell*, conforme

especificado na [Figura 1.4](#).

```
TransfereCarta (parâmetros por referência PilhaOrigem, PilhaDestino do tipo PilhalIntermediária, Parâmetro por referência DeuCerto do tipo Boolean);
/* Transfere uma carta da Pilha Origem para a PilhaDestino, caso a carta estiver na sequência correta na PilhaDestino.
O parâmetro DeuCerto retornará o valor Verdadeiro se uma carta for efetivamente transferida, e o valor Falso caso contrário */
```

A [Figura 1.5](#) apresenta um algoritmo conceitual para a operação, que transfere uma carta entre duas *Pilhas Intermediárias*. Note que a manipulação dos Dados armazenados nas *Pilhas Intermediárias* é feita exclusivamente através dos Operadores definidos na especificação do TAD ([Figura 1.4](#)). Devido a isso, para desenvolver essa solução não foi preciso conhecer detalhes sobre a implementação das Pilhas Intermediárias. Não é legal programar assim?

```
TransfereCarta(parâmetros por referência PilhaOrigem, PilhaDestino do tipo PilhalIntermediária, parâmetro por referência DeuCerto do tipo Boolean) {
    /* Transfere uma carta da Pilha Origem para a PilhaDestino, caso a carta estiver na sequência correta na PilhaDestino. O parâmetro DeuCerto retornará o valor Verdadeiro se uma carta for efetivamente transferida, e o valor Falso caso contrário */

    Variável Carta do tipo Carta-do-Baralho;
    Variável ConseguiuRetirar do tipo Boolean;
    Variável ConseguiuEmpilhar do tipo Boolean;

    /* Tenta retirar Carta do topo da PilhaOrigem */
    Desempilha(PilhaOrigem, Carta, ConseguiuRetirar);
    Se (ConseguiuRetirar == Verdadeiro)
        Então { /* empilha na Pilha Destino, se estiver na sequência correta */
            EmpilhaNaSequência(PilhaDestino, Carta, ConseguiuEmpilhar);
            Se (ConseguiuEmpilhar == Verdadeiro)
                Então DeuCerto = Verdadeiro;
                Senão { /* carta não está na sequência correta e deve retornar à PilhaOrigem */
                    EmpilhaSempre(PilhaOrigem, Carta);
                    DeuCerto = Falso;
                };
        };
        Senão DeuCerto = Falso;
    }
}
```

**FIGURA 1.5** Algoritmo conceitual — Transfere Carta.

As Operações do TAD Pilha Intermediária, conforme constam na [Figura 1.4](#), e

o algoritmo conceitual da [Figura 1.5](#) têm por objetivo apenas exemplificar o conceito de Tipos Abstratos de Dados. A concepção das Operações de uma Pilha deverá ser aprimorada a partir da apresentação de novos conceitos no [Capítulo 2](#).

## 1.4 Qual a melhor maneira de aumentar o volume da televisão?

Temos duas estratégias alternativas para aumentar o volume de uma televisão. Primeira alternativa: podemos aumentar o volume da televisão acionando o botão do volume, no controle remoto ou na própria televisão. Na segunda alternativa, pegamos uma chave de fenda, abrimos a televisão, mexemos com a chave de fenda em um componente eletrônico e aumentamos o volume “na marra”. Qual alternativa você acha mais interessante: botão de volume ou chave de fenda?

Em uma analogia, podemos considerar um Tipo Abstrato de Dados como uma televisão. Podemos desenvolver programas aumentando o volume através do botão da televisão ou podemos desenvolver programas aumentando o volume com uma chave de fenda. Aumentar o volume pelo botão da televisão significa identificar Tipos Abstratos de Dados, definir Dados a serem armazenados, definir Operadores aplicáveis a esses Dados e, a partir de então, só mexer nesses Dados através dos Operadores do TAD. Os Operadores do TAD, em nossa analogia, equivalem aos botões da televisão ([Figura 1.6](#)).

Operadores do TAD ou Botões da TV	
	<b>Pilha de Cartas:</b> <ul style="list-style-type: none"> <li>• Retira a carta que está no topo da pilha;</li> <li>• Coloca uma carta no topo da pilha, se o valor estiver na sequência correta.</li> </ul>
	<b>TV:</b> <ul style="list-style-type: none"> <li>• Aumenta o volume;</li> <li>• Diminui o volume;</li> <li>• Muda de canal (1 canal acima);</li> <li>• Muda de canal (1 canal abaixo).</li> </ul>

**FIGURA 1.6** Analogia: Operadores do TAD e botões da televisão.

Qual o papel de um TAD no projeto de software? Um Tipo Abstrato de Dados é um *modelo abstrato* do armazenamento e manipulação de determinado conjunto de Dados de um programa. Um TAD é uma *caixa-preta*. O que é visível externamente a essa *caixa-preta* é a sua funcionalidade, ou seja, as Operações definidas para o TAD. Os detalhes de implementação ficam escondidos.

O principal propósito desse *modelo abstrato* chamado Tipo Abstrato de Dados é simplificar. Por exemplo, uma vez definido o TAD Pilha de Cartas, ao construir a lógica da aplicação devemos abstrair, nos despreocupar ou simplesmente esquecer como o armazenamento das cartas é efetivamente implementado. Devemos simplesmente acionar as operações da Pilha de Cartas, ou os “botões da televisão”, para construir a lógica da aplicação. É mais simples programar assim, não é?

É possível alterar os dados de uma Pilha de Cartas sem ser por intermédio de uma operação definida na especificação do TAD Pilha de Cartas? Sim, é possível. Mas, fazendo isso, estamos optando pela estratégia da chave de fenda, ou seja, estamos abrindo a televisão com uma chave de fenda para aumentar o volume. Utilizar o botão de volume é uma estratégia melhor!

## 1.5 O que é um bom programa?

Não, um bom programa não é simplesmente um programa que funciona. Na verdade, um programa que não funciona ainda não é um programa. Vamos detalhar a pergunta: temos dois programas e os dois funcionam. Um deles é um programa bom e o outro é um programa ruim. Quais são as possíveis características do programa que é bom?

Você sabe o que significa *portabilidade* de código? Resumidamente, **portabilidade** é a capacidade de um código (trecho de programa) executar em diferentes plataformas de hardware e software. Assim como podemos carregar um computador portátil de um lugar para o outro, podemos carregar um software portátil de uma plataforma para outra, e esse software continuará funcionando. Pense, por exemplo, em um editor de texto. Como ele consegue executar a operação de imprimir, mesmo se você trocar a sua impressora? *Portabilidade* de software: capacidade de executar em diferentes plataformas.

E o termo **reusabilidade**, você sabe o que significa? *Reusabilidade* de código ou software significa reutilizar um software já desenvolvido, em uma segunda situação. Ou seja, você desenvolve o software para uma necessidade e o aproveita (reutiliza) para satisfazer uma segunda necessidade.

**Portabilidade de software:** capacidade de executar em diferentes plataformas de hardware e software.

**Reusabilidade de software:** capacidade de aproveitar (reutilizar) um software já desenvolvido, para satisfazer uma segunda necessidade.

**FIGURA 1.7** Portabilidade e reusabilidade de software.

Outro critério que pode ser utilizado para diferenciar programas bons de programas ruins é a **eficiência** — um programa que executa mais rápido ou utiliza menos memória. Se você tivesse que escolher entre um código com maior portabilidade e reusabilidade, e um código que execute milésimos de segundo mais rápido, o que escolheria?

Existem circunstâncias específicas nas quais certamente é necessário priorizar a eficiência. Por exemplo, quando precisamos garantir que o tempo de resposta de um software satisfaça critérios muito rígidos, para que não ocorram grandes desastres. Mas, na maior parte das circunstâncias, as opções de projeto devem priorizar o que resulta em custo mais baixo no desenvolvimento e na

manutenção de software.

Portabilidade e reusabilidade são características de um software que resultam em custo mais baixo de desenvolvimento e manutenção.

## 1.6 Vantagens da utilização de Tipos Abstratos de Dados

Podemos apontar as seguintes vantagens da utilização do conceito de Tipos Abstratos de Dados:

- **É mais fácil programar, sem se preocupar com detalhes de implantação.**

Por exemplo, no momento de transferir dados de uma *Pilha de Cartas* para outra, você não precisa se preocupar em saber como a *Pilha* é efetivamente implementada. Você precisa apenas saber utilizar as operações que colocam ou retiram cartas em uma *Pilha*.

- **É mais fácil preservar a integridade dos dados.** Apenas as operações do Tipo Abstrato de Dados alteram os dados armazenados. Suponha que as operações do TAD *Pilha de Cartas* estejam corretas, não corrompendo os dados nem ocasionando perda de dados. Se você alterar os Dados pelos Operadores do TAD, os Dados certamente serão preservados. Mas, se você não conhece muito bem a implementação do TAD Pilha de Cartas e decide alterar os Dados *com a chave de fenda* (ou seja, sem usar os botões da televisão), é possível que faça alguma besteira e acabe corrompendo os dados.

- **Maior independência e portabilidade de código.** Alterações na implementação de um TAD não implicam em alterações nas aplicações que o utilizam. Vamos supor que implementamos um TAD *Pilha de Cartas* utilizando determinada técnica de implementação, mas depois decidimos mudar a técnica de implementação. Se uma aplicação que usa o TAD tiver sido implementada exclusivamente através dos Operadores do TAD, essa aplicação continuará funcionando, ainda que a implementação do TAD *Pilha de Cartas* seja completamente alterada. O que mudou foi a implementação do TAD, mas seus Operadores continuam os mesmos! Logo, a aplicação continuará funcionando.

- **Maior potencial de reutilização de código.** Pode-se alterar a lógica de um programa sem necessidade de reconstruir as estruturas de armazenamento. Um mesmo TAD *Pilha de Cartas* pode ser utilizado no desenvolvimento do *FreeCell*, no desenvolvimento de variações do *FreeCell* e também no

desenvolvimento de outras aplicações que utilizam pilhas de cartas.

Se adotarmos uma estratégia de desenvolvimento com o uso de Tipos Abstratos de Dados, ou seja, aumentando o volume da televisão sempre pelos botões e nunca com a chave de fenda, o desenvolvimento será mais fácil, os dados estarão mais seguros, o código terá um potencial maior para executar em diferentes plataformas e para ser reutilizado em outras aplicações. Como consequência, teremos custo menor de desenvolvimento e manutenção. Software bom, bonito e barato!

Vamos adotar Tipos Abstratos de Dados no desenvolvimento de nossos jogos?

### Software bom, bonito e barato

O uso do conceito de Tipos Abstratos de Dados aumenta a portabilidade e o potencial de reutilização do software. Em consequência disso, o custo de desenvolvimento e manutenção é reduzido.



### Consulte no Banco de Jogos

Adaptações do *FreeCell*



<http://www.elsevier.com.br/edcomjogos>

## Exercícios de fixação

**Exercício 1.2** O que é um Tipo Abstrato de Dados (TAD)? Como os dados armazenados em um TAD devem ser manipulados? Em que momento um TAD deve ser identificado e definido?

**Exercício 1.3** Como é possível desenvolver um programa utilizando um TAD Pilha de Cartas, por exemplo, sem conhecer detalhes de sua implementação?

**Exercício 1.4** O que é portabilidade de código? O que é reusabilidade de código?

**Exercício 1.5** Faça uma pesquisa sobre portabilidade e reusabilidade de software. Converse com os colegas sobre o que você considera importante sobre isso. Você concorda que portabilidade e reusabilidade são, na maioria das

situações, características mais importantes para um programa do que executar milésimos de segundo mais rápido?

**Exercício 1.6** Quais as vantagens de programar utilizando o conceito de TAD? Explique com exemplos.

**Exercício 1.7** Consulte em um dicionário o significado do termo “abstrato”. Consulte as palavras “abstrato”, “abstrair” e “abstração”.

## Referências e leitura adicional

1. Celes W, Cerqueira R, Rangel FL. *Introdução a estruturas de dados*. Rio de Janeiro: Elsevier; 2004; p. 126.
2. Langsam Y, Augenstein MJ, Tenenbaum AM. *Data Structures Using C and C++*. 2nd ed. Upper Saddle River. New Jersey: Prentice Hall; 1996; p. 13.
3. Pressman R. *Engenharia de software*. 3. ed. São Paulo: Makron Books; 1995; p. 32-35.
4. Sommerville RI. *Engenharia de software*. 6. ed. São Paulo: Addison Wesley; 2003; p. 35-38.

---

## CAPÍTULO 2

---

# Pilhas com Alocação Sequencial e Estática

---

## Seus objetivos neste capítulo

- Entender o que é e para que serve uma estrutura do tipo Pilha.
- Entender o significado de Alocação Sequencial e Alocação Estática de Memória, no contexto do armazenamento temporário de conjuntos de elementos.
- Desenvolver habilidade para implementar uma estrutura do tipo Pilha, como um Tipo Abstrato de Dados (TAD), com Alocação Sequencial e Estática de Memória.
- Desenvolver habilidade para manipular Pilhas através dos Operadores definidos para o TAD Pilha.
- Iniciar o desenvolvimento do seu primeiro jogo.

## 2.1 O que é uma Pilha?

No contexto deste livro, o termo Pilha diz respeito a uma pilha de pratos, pilha de livros ou pilha de cartas. Ou seja, o sentido é de empilhar uma coisa sobre a outra.

Em uma pilha de pratos, se você tentar tirar um prato do meio da pilha, a pilha poderá desmoronar. Também será bem complicado você inserir um prato no meio da pilha. O mais natural é retirar pratos do topo da pilha e inserir pratos sempre no topo da pilha.

Pilha de Pratos	Pilha de Cartas										
	<table> <tr> <td>8</td> <td>♥</td> </tr> <tr> <td>J</td> <td>♣</td> </tr> <tr> <td>A</td> <td>♠</td> </tr> <tr> <td>7</td> <td>♠</td> </tr> <tr> <td>Q</td> <td>♥</td> </tr> </table> 	8	♥	J	♣	A	♠	7	♠	Q	♥
8	♥										
J	♣										
A	♠										
7	♠										
Q	♥										

**FIGURA 2.1** Ilustração do conceito de Pilha — sentido de empilhar.

Em essência, é assim que funciona uma Pilha: novos elementos entram sempre no topo; se quisermos retirar um elemento, retiramos sempre o elemento do topo.

#### Definição: Pilha

Pilha é uma estrutura para armazenar um conjunto de elementos, que funciona da seguinte forma:

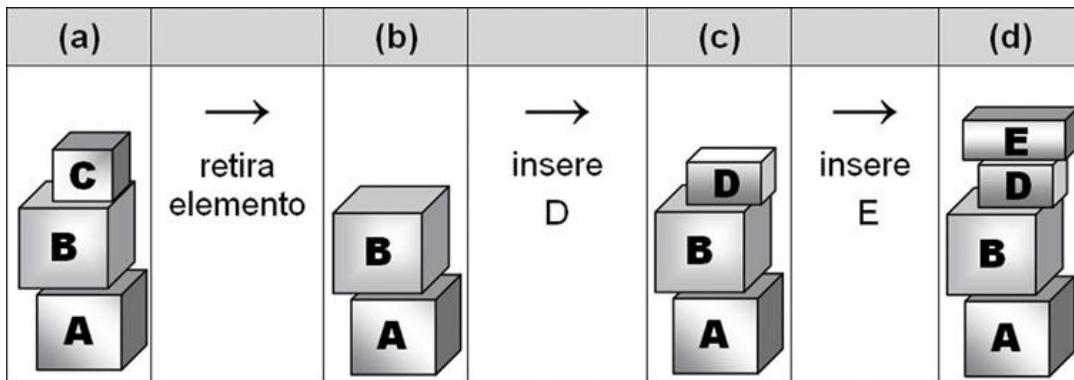
- Novos elementos entram no conjunto sempre no topo da Pilha.
- O único elemento que pode ser retirado da Pilha em dado momento é o elemento do topo.

#### Do inglês Stack, L.I.F.O.

Uma Pilha (em inglês, Stack) é uma estrutura que obedece ao critério *Last In, First Out* (L.I.F.O.; Knuth, 1972, p. 236; Langsam, Augenstein e Tenenbaum, 1996, p. 78). Ou seja, o último elemento que entrou no conjunto será o primeiro elemento a sair do conjunto.

**FIGURA 2.2** Definição de Pilha.

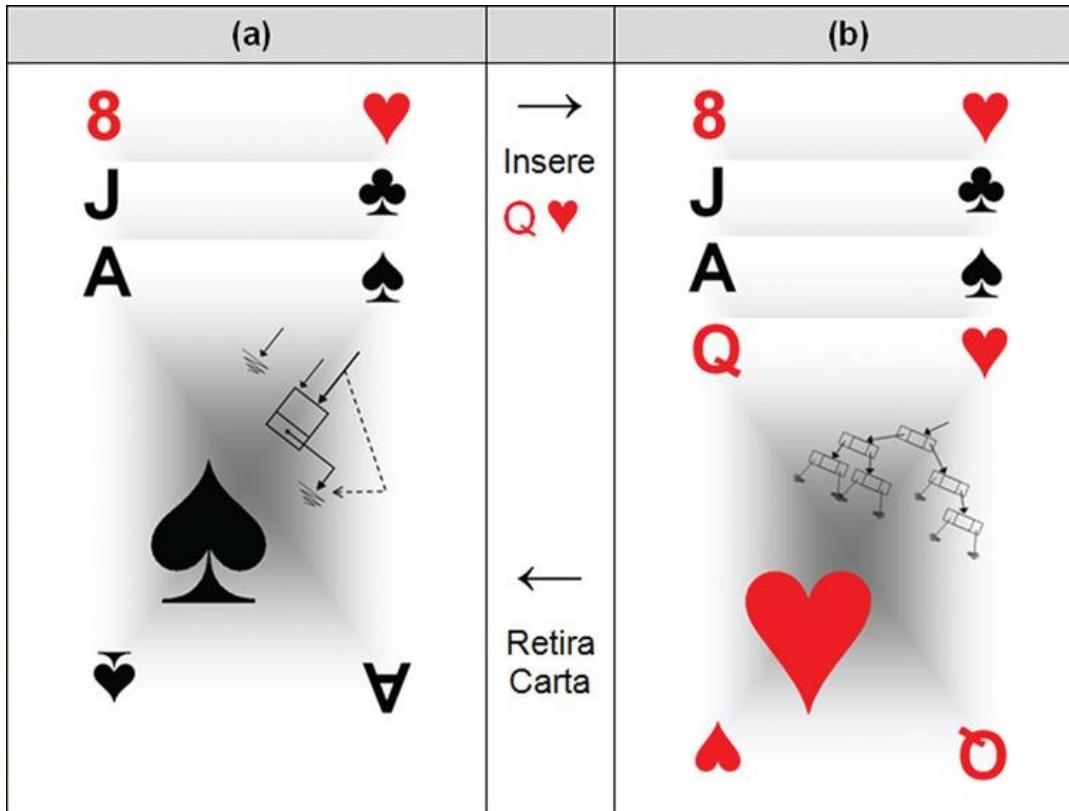
Uma Pilha é um conjunto ordenado de elementos. A ordem de entrada determina a posição dos elementos no conjunto. Se temos três elementos em uma Pilha (A, B e C) e se eles entraram na Pilha na sequência A, depois B e então C, sabemos que o elemento C estará no topo da Pilha (Figura 2.3a).



**FIGURA 2.3** Retirando e inserindo elementos em uma Pilha.

Na situação da [Figura 2.3a](#), o único elemento que podemos retirar é exatamente o elemento que está no topo da Pilha. Ou seja, o elemento C. Retirando o elemento C, a situação da Pilha fica sendo a da [Figura 2.3b](#). Se, em seguida, quisermos inserir o elemento D, esse elemento D passará a ser o topo da Pilha — [Figura 2.3c](#). E, finalmente, se quisermos inserir o elemento E, esse novo elemento passará a ser o elemento do topo — [Figura 2.3d](#). Se a ordem de inserção dos elementos na Pilha tivesse sido outra, a Pilha resultante também seria outra.

O mesmo raciocínio vale para uma Pilha de Cartas, por exemplo. Considere como situação inicial uma Pilha com três cartas: 8 de copas, J de paus e, no topo da Pilha, A de espadas, como mostra a [Figura 2.4a](#). Ao contrário do que fizemos na pilha de cubos da [Figura 2.3](#), agora o topo está representado na parte de baixo da Pilha de Cartas. Essa representação mostra o valor e o naipe de todas as cartas da Pilha, como se estivéssemos empilhando uma carta sobre a outra com um baralho de papel. É a representação usual das Pilhas de Cartas em jogos do tipo *FreeCell*.



**FIGURA 2.4** Inserindo e retirando cartas em uma Pilha.

Se quisermos inserir no conjunto a carta Q de copas, essa nova carta precisará ser inserida no topo da Pilha, resultando na situação da [Figura 2.4b](#). Se, em seguida, quisermos retirar uma carta da Pilha, a única carta que podemos retirar é a carta do topo da Pilha — o próprio Q de copas, que acabou de ser inserido. A Pilha então voltará à situação da [Figura 2.4a](#), tendo apenas três elementos, com o A de espadas novamente no topo.

## 2.2 Operações de uma Pilha

Vamos chamar a operação que insere elementos em uma Pilha de Empilha, e a operação que retira elementos de uma Pilha, de Desempilha. A essas duas Operações, vamos acrescentar outras três: operação para criar uma Pilha, operação para testar se a Pilha está vazia e operação para testar se a Pilha está cheia. Na [Figura 2.5](#) temos a especificação das operações Empilha, Desempilha, Cria, Vazia e Cheia, bem como uma descrição de seus parâmetros e funcionamento.

Operações e parâmetros	Funcionamento
Empilha (P, X, DeuCerto)	Empilha o elemento X, passado como parâmetro, na Pilha P, também passada como parâmetro. O parâmetro DeuCerto indica se a execução da operação foi bem-sucedida ou não.
Desempilha (P, X, DeuCerto)	Desempilha (retira o elemento do topo) da Pilha P passada no parâmetro, retornando o valor do elemento que foi desempilhado no parâmetro X. O parâmetro DeuCerto indica se a operação foi bem-sucedida ou não.
Vazia (P)	Verifica se a Pilha P passada como parâmetro está vazia. Uma Pilha vazia é uma Pilha sem nenhum elemento.
Cheia (P)	Verifica se a Pilha P passada como parâmetro está cheia. Uma Pilha cheia é uma Pilha em que não cabe mais nenhum elemento.
Cria (P)	Cria uma Pilha P, iniciando sua situação como vazia.

**FIGURA 2.5** Operações de uma Pilha.

## Exercício 2.1 Transfere elementos de uma Pilha para outra

Desenvolva um algoritmo para transferir todos os elementos de uma Pilha P1 para uma Pilha P2. Considere que as Pilhas P1 e P2 já existem, ou seja, não precisam ser criadas. Para elaborar esse algoritmo, use os operadores definidos na [Figura 2.5](#). Considere que as Pilhas P1 e P2 possuem elementos do tipo inteiro. Você terá que elaborar esse algoritmo sem saber como o Tipo Abstrato de Dados Pilha é efetivamente implementado. Você encontrará uma solução para este exercício na sequência do texto. Mas, antes de prosseguir a leitura, tente propor uma solução. É assim que desenvolvemos habilidade para manipular Pilhas (e outras estruturas) pelos seus Operadores.

```
TransfereElementos (parâmetros por referência P1, P2 do tipo Pilha);
/* transfere todos os elementos de P1 para P2 */
```

A lógica da solução do [Exercício 2.1](#) é a seguinte: enquanto a Pilha P1 não

estiver vazia, retiramos elementos de P1 através da operação Desempilha. Cada valor desempilhado de P1 deve ser empilhado na Pilha P2 através da operação Empilha. Repetimos essa sequência de comandos até que a Pilha P1 se torne vazia. Veja o algoritmo da [Figura 2.6](#).

```
TransfereElementos(parâmetros por referência P1, P2 do tipo Pilha) {  
    /* transfere todos os elementos de P1 para P2 */  
  
    Variável ElementoDaPilha do tipo Inteiro;  
    Variável DeuCerto do tipo Boolean;  
  
    Enquanto (Vazia(P1) == Falso) Faça {  
        Desempilha(P1, ElementoDaPilha, DeuCerto);           // enquanto P1 não for vazia  
        Empilha(P2, ElementoDaPilha, DeuCerto); }             // desempilha de P1  
    }                                                       // empilha em P2
```

**FIGURA 2.6** Algoritmo para transferir elementos de uma Pilha para outra.

Note que ainda não temos a menor ideia de como essas operações do TAD Pilha (Empilha, Desempilha, Vazia, Cheia, Cria) são efetivamente implementadas. Consideramos as Pilhas como caixas-pretas e manipulamos os valores armazenados apenas através dos Operadores (ou “botões da televisão”). É assim que manipulamos estruturas de armazenamento, como Pilhas, de modo abstrato. É assim que utilizamos os Tipos Abstratos de Dados.

Proponha uma solução para os [Exercícios 2.2, 2.3 e 2.4](#) sempre com a mesma estratégia: manipulando os valores armazenados exclusivamente através dos Operadores do TAD Pilha descritos na [Figura 2.5](#).

## Exercício 2.2 Mais elementos?

Desenvolva um algoritmo para testar se uma Pilha P1 tem mais elementos do que uma Pilha P2. Considere que as Pilhas P1 e P2 já existem e são passadas como parâmetro. Considere também que as Pilhas P1 e P2 possuem elementos do tipo Inteiro. Você pode criar Pilhas auxiliares, se necessário. Você deve preservar os dados de P1 e P2. Ou seja, ao final da execução dessa operação, P1 e P2 precisam conter exatamente os mesmos elementos que continham no início da operação, e na mesma sequência. Para propor a solução, utilize os Operadores da [Figura 2.5](#).

```
Boolean MaisElementos (parâmetros por referência P1, P2 do tipo Pilha);  
/* retorna Verdadeiro se a Pilha P1 tiver mais elementos do que a Pilha P2 */
```

## Exercício 2.3 Algum elemento igual a X?

Desenvolva um algoritmo para verificar se uma Pilha P possui algum elemento igual a um valor X. P e X são passados como parâmetros. Considere que a Pilha P possui elementos do tipo Inteiro. Para propor a solução, utilize os Operadores da [Figura 2.5](#). Você pode criar Pilhas auxiliares, se necessário.

```
Boolean AlgumElementoIgualX (parâmetro por referência P do tipo Pilha, parâmetro X do tipo inteiro);  
/* Verifica se a Pilha P possui algum elemento igual ao valor do parâmetro X */
```

## Exercício 2.4 As Pilhas são iguais?

Desenvolva um algoritmo para testar se duas Pilhas P1 e P2 são iguais. Duas Pilhas são iguais se possuem os mesmos elementos, exatamente na mesma ordem. Para propor sua solução, você pode utilizar Pilhas auxiliares, se necessário. Considere que as Pilhas P1 e P2 já existem e são passadas como parâmetro. Considere que as Pilhas P1 e P2 possuem elementos do tipo Inteiro. Para propor a solução, utilize os Operadores da [Figura 2.5](#).

```
Boolean iguais (parâmetros por referência P1, P2 do tipo Pilha);  
/* retorna Verdadeiro se a Pilha P1 for igual à Pilha P2, ou seja, se P1 e P2 possuírem exatamente os mesmos elementos,  
na mesma ordem. Se ambas as Pilhas forem vazias, serão consideradas iguais */
```

No final deste capítulo você encontrará respostas ou sugestões para alguns dos exercícios. Mas, para desenvolver as habilidades pretendidas, proponha suas próprias soluções antes de consultar as respostas sugeridas.

## 2.3 Alocação de memória para conjuntos de elementos

Alocar memória, em essência, significa reservar uma porção da memória do computador para um uso específico. Alocar memória para um conjunto de elementos, como uma Pilha, envolve reservar espaço de memória para

armazenar cada um dos elementos do conjunto.

## Alocação Sequencial

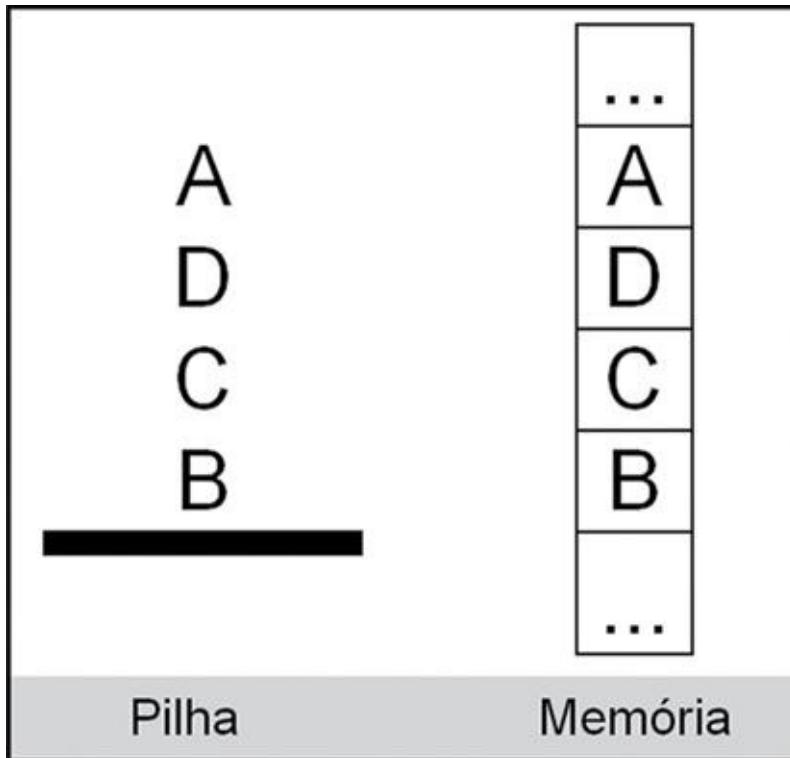
Na Alocação Sequencial de Memória para um conjunto de elementos ([Knuth, 1972](#), p. 240; [Pereira, 1996](#), p. 12), todos os elementos do conjunto serão armazenados juntos, em posições adjacentes de memória. Ou seja, os elementos ficam “em sequência”, um ao lado do outro na memória do computador.

**Definição: Alocação Sequencial de Memória para um conjunto de elementos**

- Os elementos ficam, necessariamente, em posições de memória adjacentes ou “em sequência”.

**FIGURA 2.7** Definição de Alocação Sequencial de Memória para um conjunto de elementos.

A [Figura 2.8](#) mostra uma Pilha com quatro elementos: o elemento A no topo da Pilha e, em seguida, os elementos D, C e B. Na Alocação Sequencial de Memória, a ordem desses quatro elementos é refletida nas posições de memória em que os elementos são armazenados.



**FIGURA 2.8** Alocação Sequencial — armazenamento em posições de memória adjacentes.

## Alocação Estática

Na Alocação Estática de Memória para um conjunto de elementos, estabelecemos o tamanho máximo do conjunto previamente, ao elaborar o programa, e reservamos memória para todos os seus elementos. Esse espaço de memória permanecerá reservado durante toda a execução do programa, mesmo que não esteja sendo totalmente utilizado, isto é, mesmo que em determinado momento a quantidade de elementos no conjunto seja menor que o que seu tamanho máximo. Como o tamanho do espaço de memória é definido previamente, ele não poderá crescer ou diminuir ao longo da execução do programa.

**Definição: Alocação Estática de Memória para um conjunto de elementos**

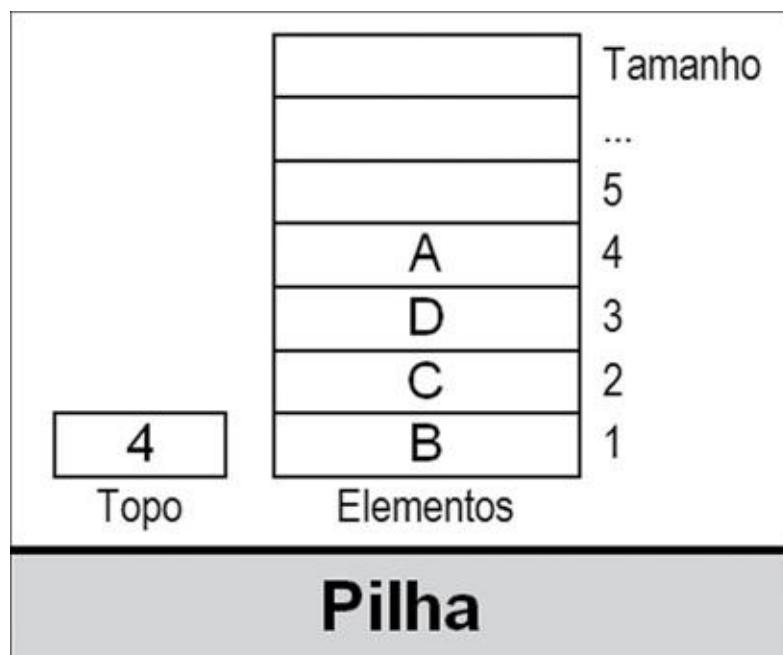
- O espaço de memória para todos os elementos que poderão fazer parte do conjunto é dimensionado previamente.
- Esse espaço de memória permanecerá reservado durante toda a execução do programa, mesmo se não estiver sendo efetivamente utilizado.

**FIGURA 2.9** Definição de Alocação Estática de Memória para um conjunto de elementos.

## 2.5 Implementando uma Pilha com Alocação Sequencial e Estática

Para implementar uma Pilha combinando os conceitos de Alocação Sequencial e Alocação Estática de Memória, ao elaborar o programa precisamos definir o tamanho do espaço de memória a ser reservado e possibilitar que os elementos da Pilha possam ser armazenados em posições adjacentes de memória. Podemos fazer isso através de uma estrutura do tipo vetor.

Na [Figura 2.10](#) temos o esquema de uma Pilha implementada através de um vetor denominado Elementos, que armazena os elementos da Pilha, e de uma variável denominada Topo, que indica o topo da Pilha.



**FIGURA 2.10 Esquema da implementação de uma Pilha com Alocação Sequencial e Estática de Memória.**

Podemos definir o tipo Pilha como um Registro (uma *Struct* na linguagem C) contendo dois campos: Elementos e Topo. Topo do tipo Inteiro e Elementos do tipo vetor de 1 até Tamanho (constante que indica o tamanho da Pilha) ou, ainda, de zero até Tamanho –1, conforme o padrão da linguagem C. Os elementos da Pilha podem ser do tipo *Char*, do tipo Inteiro, do tipo Carta-do-Baralho ou de outro tipo qualquer, conforme necessário.

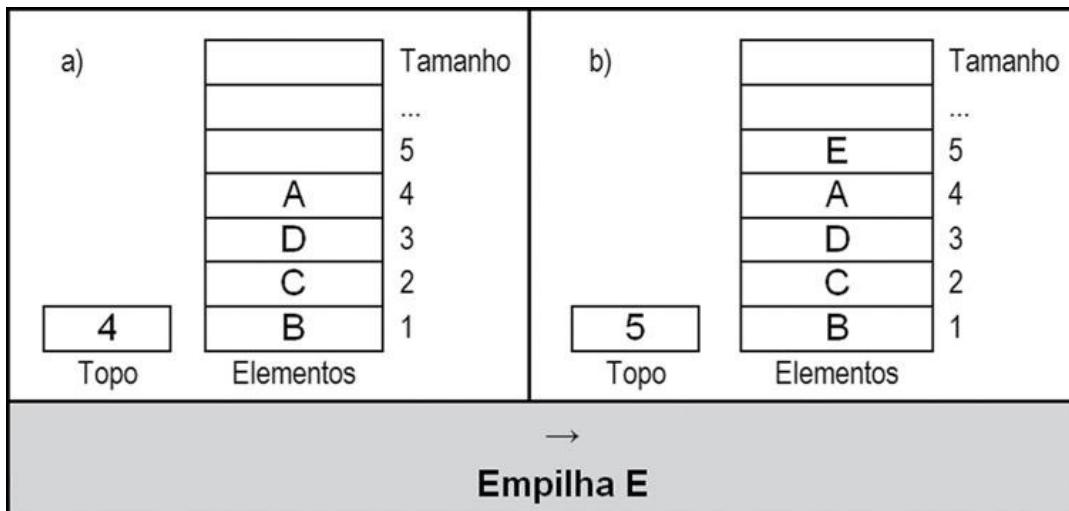
Para implementar um Tipo Abstrato de Dados (TAD) precisamos de uma estrutura de armazenamento e de Operadores que possam ser aplicados sobre os Dados armazenados. Vamos implementar o TAD Pilha com a estrutura de armazenamento definida na [Figura 2.10](#) e com os Operadores definidos na [Figura 2.5](#): Empilha, Desempilha, Cria, Vazia e Cheia.

## Exercício 2.5 Operação Empilha

Conforme especificado na [Figura 2.5](#), a operação Empilha recebe como parâmetros a Pilha e o valor do elemento que queremos empilhar. O elemento só não será inserido se a Pilha já estiver cheia. Você encontrará uma possível solução para este exercício na sequência do texto. Mas tente propor uma solução antes de prosseguir a leitura.

```
Empilha (parâmetro por referência P do tipo Pilha, parâmetro X do tipo Char, parâmetro por referência DeuCerto do tipo Boolean);
/* Empilha o elemento X na Pilha P. O parâmetro DeuCerto deve indicar se a operação foi bem-sucedida ou não. */
```

A [Figura 2.11](#) ilustra o funcionamento da operação Empilha. A partir da situação da [Figura 2.11 a](#), um elemento de valor 'E' é inserido, resultando na situação da [Figura 2.11 b](#). Note que o valor da variável Topo foi incrementado, e o elemento 'E' foi inserido no vetor Elementos na posição indicada pelo valor atualizado da variável Topo.



**FIGURA 2.11** Operação Empilha.

A [Figura 2.12](#) apresenta um algoritmo conceitual para a Operação Empilha. Se a Pilha P estiver cheia, sinalizamos através do parâmetro DeuCerto que o elemento X não foi inserido. Mas, se a Pilha P não estiver cheia, X será inserido da seguinte forma: incrementamos o Topo da Pilha P ( $P.\text{Topo} = P.\text{Topo} + 1$ ) e armazenamos X no vetor Elementos, na posição indicada pelo Topo da Pilha ( $P.\text{Elementos}[ P.\text{Topo} ] = X$ ). O elemento X passará a ser o elemento do Topo da Pilha.

```

Empilha (parâmetro por referência P do tipo Pilha, parâmetro X do tipo Char, parâmetro
por referência DeuCerto do tipo Boolean) {
    /* Empilha o elemento X, passado como parâmetro, na Pilha P também passada como
       parâmetro. O parâmetro DeuCerto deve indicar se a operação foi bem-sucedida ou não.*/
    Se (Cheia(P) == Verdadeiro)           // se a Pilha P estiver cheia...
        Então DeuCerto = Falso            // ... não podemos empilhar
    Senão { P.Topo = P.Topo + 1;          // incrementa o Topo da Pilha P
            P.Elementos[ P.Topo ] = X;    // armazena o valor de X no Topo da Pilha
            DeuCerto = Verdadeiro; }      // a operação deu certo
}

```

**FIGURA 2.12** Algoritmo conceitual — Empilha.

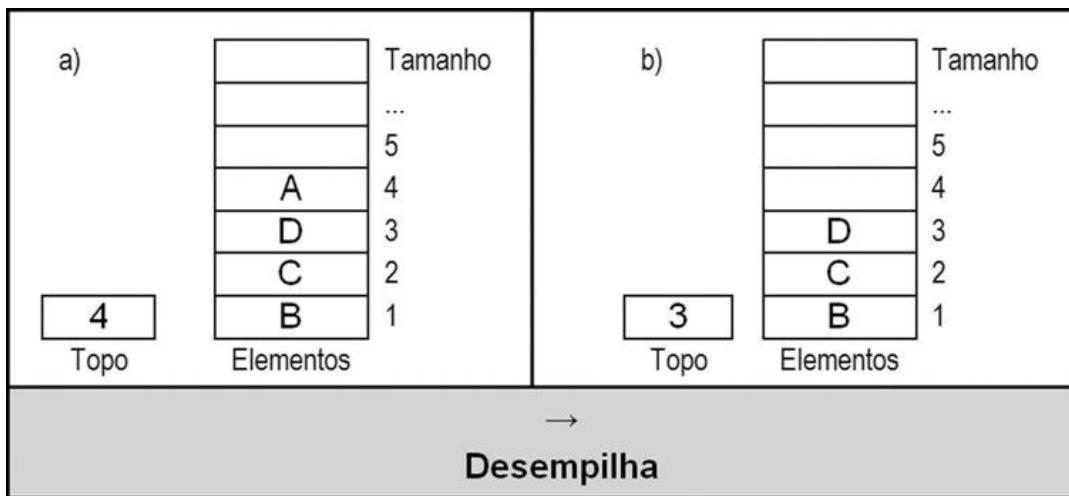
## Exercício 2.6 Operação Desempilha

Conforme especificado na [Figura 2.5](#), a operação Desempilha recebe como parâmetro a Pilha da qual queremos retirar um elemento. Caso a Pilha não

estiver vazia, a operação retorna o valor do elemento retirado. A ação da operação Desempilha, na situação da [Figura 2.13a](#), resultaria na situação da [Figura 2.13b](#).

Desempilha(parâmetro por referência P do tipo Pilha, parâmetro por referência X do tipo Char, parâmetro por referência DeuCerto do tipo Boolean);

/\* Caso a Pilha P não estiver vazia, retira o elemento do Topo e retorna seu valor no parâmetro X. Se a Pilha P estiver vazia, o parâmetro DeuCerto deve retornar Falso \*/



**FIGURA 2.13** Operação Desempilha.

## Exercício 2.7 Operação Cria

Conforme especificado na [Figura 2.5](#), a operação Cria recebe como parâmetro a Pilha que deverá ser criada. Criar a Pilha significa inicializar os valores de modo a indicar que a Pilha está vazia, ou seja, sem nenhum elemento.

Cria (parâmetro por referência P do tipo Pilha);

/\* Cria a Pilha P, inicializando-a como vazia – sem nenhum elemento. \*/

## Exercício 2.8 Operação Vazia

Conforme especificado na [Figura 2.5](#), a operação Vazia testa se a Pilha passada como parâmetro está vazia (sem elementos), retornando o valor Verdadeiro

(Pilha vazia) ou Falso (Pilha não vazia).

Boolean Vazia (parâmetro por referência P do tipo Pilha);

/\* Retorna Verdadeiro se a Pilha P estiver vazia – sem nenhum elemento; Falso caso contrário. \*/

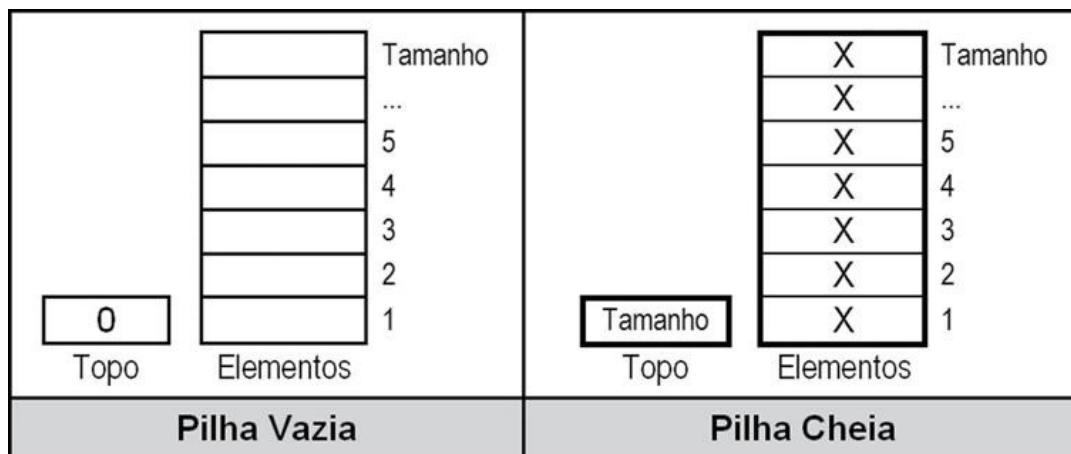
## Exercício 2.9 Operação Cheia

Conforme especificado na [Figura 2.5](#), a operação Cheia testa se a Pilha passada como parâmetro está cheia. A Pilha estará cheia se na estrutura de armazenamento não couber mais nenhum elemento.

Boolean Cheia (parâmetro por referência P do tipo Pilha);

/\* Retorna Verdadeiro se a Pilha P estiver cheia, ou seja, se na estrutura de armazenamento não couber mais nenhum elemento; Retorna Falso caso a Pilha P não estiver cheia. \*/

Conforme mostra a [Figura 2.14](#), na operação Cria inicializamos o Topo da Pilha com o valor zero, pois os elementos começam a ser armazenados no vetor Elementos a partir da posição 1. Assim, com o valor de Topo sendo zero, indicamos que a Pilha está vazia. Note que, em vez de começar em 1, se o vetor começar em zero (esse é o padrão na linguagem C) a situação de Pilha vazia será caracterizada quando o valor do Topo for -1.



**FIGURA 2.14** Pilha Vazia e Pilha Cheia.

A operação Vazia simplesmente testará se o Topo da Pilha está apontando para o valor zero. Se o valor do Topo for zero, a Pilha estará vazia. Se o valor do

Topo não estiver em zero, a Pilha não estará vazia. Se, em vez de começar em 1, o vetor começar em zero (padrão na linguagem C) a Pilha estará vazia quando o valor do Topo for -1.

Analogamente, na operação Cheia verificamos se o valor do Topo é igual à constante Tamanho, que indica o Tamanho da Pilha. Se, em vez de começar em 1, o vetor começar em zero (padrão na linguagem C) a situação de Pilha cheia será caracterizada quando o valor do Topo for igual a Tamanho -1.

No final deste capítulo são apresentadas soluções ou sugestões para alguns dos exercícios. Mas, para desenvolver as habilidades pretendidas, proponha suas próprias soluções antes de consultar as respostas e sugestões.

## 2.6 Abrir ou não a televisão?

Suponha que queiramos desenvolver uma operação para consultar o valor do elemento do topo de uma Pilha. Não queremos retirar o elemento do topo; queremos apenas consultar o seu valor, sem retirá-lo da Pilha. Pense em duas estratégias diferentes para implementar essa operação.

### Exercício 2.10 Duas estratégias para elemento do topo

Desenvolva uma operação que retorne o valor do elemento do topo de uma Pilha sem retirar o elemento da Pilha. Proponha duas soluções diferentes para essa operação e compare as soluções, apontando suas vantagens e desvantagens. Proponha e compare as soluções antes de prosseguir a leitura.

```
Char ElementoDoTopo (parâmetro por referência P do tipo Pilha, parâmetro por referência DeuCerto do tipo Boolean);
/* Retorna o valor do elemento do Topo da Pilha P, caso a Pilha não estiver vazia. Caso a Pilha estiver vazia, DeuCerto
retorna o valor Falso */
```

Podemos implementar a operação Elemento do Topo basicamente de duas maneiras. A primeira maneira é dependente da estrutura de armazenamento, e a segunda maneira é independente da estrutura de armazenamento. Na [Figura 2.15](#) apresentamos duas soluções para a operação Elemento do Topo. A diferença entre as soluções está destacada em negrito. O que muda é o modo de pegar a informação do Topo da Pilha.

```

Char ElementoDoTopo (parâmetro por referência P do tipo Pilha, parâmetro por referência
DeuCerto do tipo Boolean) {
    /* Retorna o valor do elemento do Topo da Pilha P, caso a Pilha não esteja vazia. Caso a
    Pilha esteja vazia, DeuCerto retorna Falso. */
    Variável X do tipo Char; /* X armazenará o valor do elemento do Topo */

    /* primeira solução: abrindo a televisão com uma chave de fenda para aumentar o
    volume */
    Se (Vazia (P)==Verdadeiro)
        Então DeuCerto = Falso
    Senão { X = P.Elementos[ P.Topo ]; /* pega o valor do elemento do Topo */
        DeuCerto = Verdadeiro;
        Retorne X; }

    /* segunda solução: aumentando o volume pelo botão de volume */
    Se (Vazia (P)==Verdadeiro)
        Então DeuCerto = Falso
    Senão { Desempilha ( P, X, DeuCerto); /* pega o valor do elemento do Topo */
        Empilha ( P, X, DeuCerto);
        DeuCerto = Verdadeiro;
        Retorne X; }
}

```

**FIGURA 2.15** Soluções para a operação Elemento do Topo.

Na primeira solução, pegamos o valor do elemento do Topo simplesmente atribuindo a X o valor de P.Elementos[ P.Topo ]. Nessa primeira solução, fica evidente que a estrutura de armazenamento utilizada é um vetor. Ou seja, é uma solução dependente da estrutura de armazenamento; só funcionará na estrutura de armazenamento em questão — um vetor. Na segunda implementação, pegamos o valor do elemento do Topo através da operação Desempilha. A operação Desempilha retorna o valor do elemento do Topo na variável X, mas retira o elemento da Pilha. Por isso, temos que recolocar o elemento na Pilha através da operação Empilha. Nessa segunda solução, não fica evidente a estrutura de armazenamento utilizada. É, portanto, uma solução independente da estrutura de armazenamento.

Quais as vantagens de uma solução e de outra? A primeira solução funcionará exclusivamente nessa implementação que fizemos da Pilha, com Alocação Sequencial e Estática de Memória. Nos capítulos a seguir implementaremos uma Pilha com outras técnicas, e essa primeira solução não continuará funcionando; terá que ser reescrita. A segunda solução para a operação Elemento do Topo é independente da estrutura de armazenamento, pois manipula a Pilha exclusivamente através dos operadores Vazia, Empilha e Desempilha. Assim,

essa segunda implementação continuará funcionando mesmo se adotarmos uma nova estrutura de armazenamento para a Pilha, como faremos nos próximos capítulos. Por ser independente da estrutura de armazenamento, a segunda solução proporciona maior portabilidade de código, conceito que estudamos no [Capítulo 1](#).

Você pode estar pensando: “Para que complicar, se podemos simplesmente consultar o valor do elemento do Topo com um comando  $X = P.Elementos[P.Topo]$ ”? Sim, é uma solução eficiente. Mas é dependente da implementação e não proporciona portabilidade de código.

Pense também nas soluções dos [Exercícios 2.1](#) a [2.4](#). O [Exercício 2.2](#), por exemplo, verifica se uma Pilha  $P1$  possui mais elementos do que uma Pilha  $P2$ . Podemos ter uma solução que simplesmente compara os valores do Topo das Pilhas  $P1$  e  $P2$ , ou seja, se  $P1.Topo$  for maior do que  $P2.Topo$ , a Pilha  $P1$  tem mais elementos do que a Pilha  $P2$ . Essa seria uma solução dependente da estrutura de armazenamento; dependente da implementação. O código dessa solução não seria portável. Seria como abrir uma televisão com uma chave de fenda para aumentar o volume.

Uma segunda solução poderia desempilhar todos os elementos de  $P1$  e de  $P2$ , colocando os elementos em Pilhas auxiliares e contando os elementos de cada Pilha. Depois os elementos poderiam ser devolvidos às Pilhas originais. Essa segunda solução seria independente da estrutura de armazenamento; independente da implementação. Uma solução portável. Conforme estudamos no [Capítulo 1](#), portabilidade e reusabilidade de código são características extremamente desejáveis e, no longo prazo, tornam as soluções mais baratas.

Chave de fenda ou botão da televisão? Qual dessas soluções você considera mais interessante?

## Operações Primitivas e Não Primitivas

Podemos dizer que, em uma Pilha, as operações Empilha, Desempilha, Cria, Vazia e Cheia são *Operações Primitivas*, pois só podem ser implementadas através de uma solução dependente da estrutura de armazenamento. Já as operações que desenvolvemos nos [Exercícios 2.1](#) a [2.4](#) e [2.10](#) (transferir elementos de uma Pilha a outra, verificar se uma Pilha tem mais elementos que outra, verificar se uma Pilha possui um elemento com valor  $X$ , verificar se duas Pilhas são iguais e pegar o valor do Elemento do Topo de uma Pilha) são *Operações Não Primitivas*, pois podem ser implementadas a partir de chamadas a Operações Primitivas do TAD Pilha, resultando em uma implementação

independente da estrutura de armazenamento.

#### Definição: Operações Primitivas e Operações Não Primitivas

- **Operações Primitivas** de um Tipo Abstrato de Dados são aquelas que só podem ser implementadas através de uma solução dependente da estrutura de armazenamento.
- **Operações Não Primitivas** de um Tipo Abstrato de Dados são aquelas que podem ser implementadas através de chamadas a Operações Primitivas, possibilitando uma implementação independente da estrutura de armazenamento.

**FIGURA 2.16** Definição de Operações Primitivas e Não Primitivas.

Visando proporcionar portabilidade e reusabilidade de código, menor custo de desenvolvimento e manutenção, a melhor forma de implementar uma Operação Não Primitiva é de modo abstrato, manipulando o TAD em questão exclusivamente pelas Operações Primitivas ou, ainda, pelos “botões da televisão”.

## 2.7 Projeto *FreeCell*: Pilha Burra ou Pilha Inteligente?

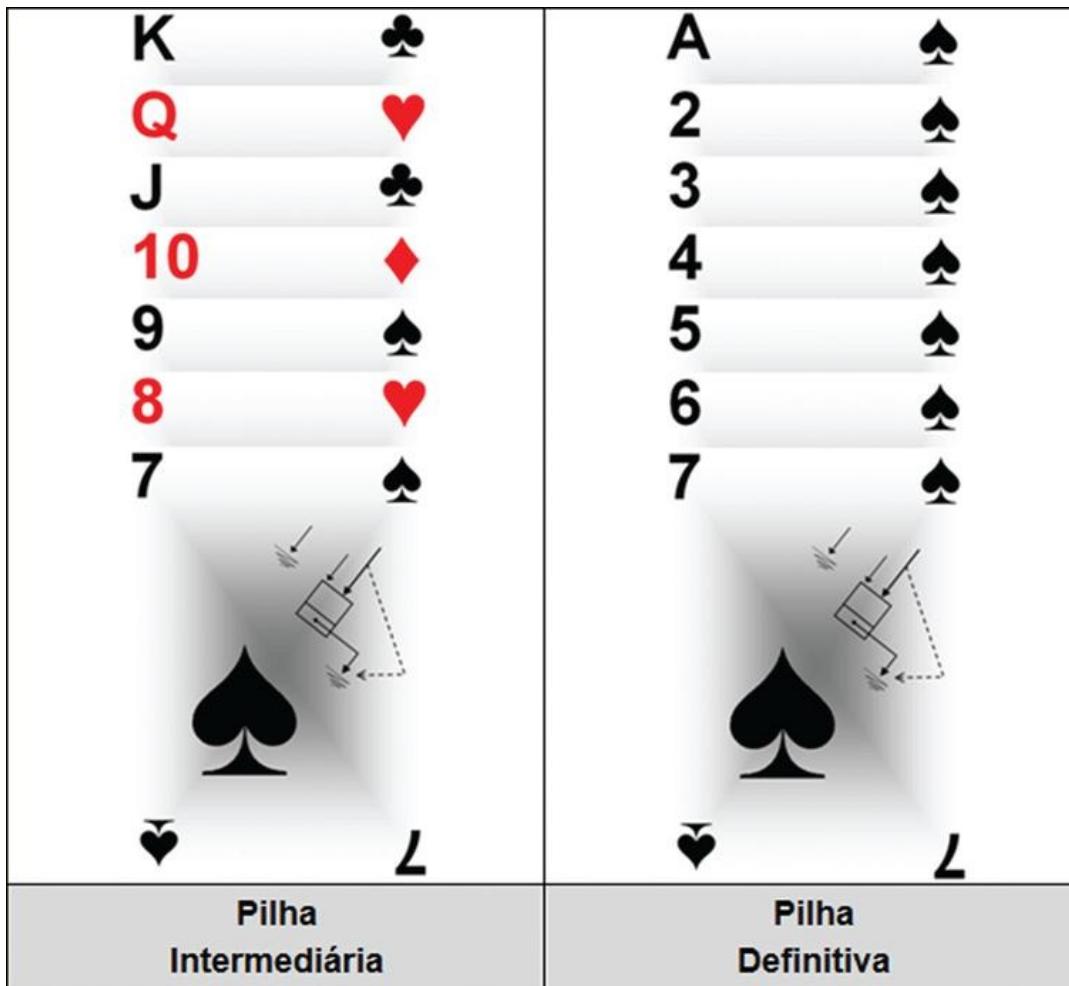
No *FreeCell* temos oito Pilhas Intermediárias que possuem algumas restrições com relação ao valor dos elementos que serão empilhados. Nessas Pilhas Intermediárias só é possível empilhar cartas em ordem decrescente e em cores alternadas — pretas sobre vermelhas e vermelhas sobre pretas.

Temos também no *FreeCell* um segundo tipo de Pilha, as Pilhas Definitivas, que também possuem suas regras com relação ao valor dos elementos que serão empilhados. Nessas quatro Pilhas Definitivas só é possível empilhar cartas de um mesmo naipe e em ordem crescente.

A característica básica de inserir elementos sempre no Topo é válida tanto para as Pilhas Intermediárias quanto para as Pilhas Definitivas. Mas as regras com relação ao valor dos elementos que serão empilhados são diferentes para Pilhas Intermediárias e Pilhas Definitivas.

A Pilha que definimos e implementamos nas seções anteriores é uma “pilha burra”, pois ela não faz qualquer verificação quanto ao valor do elemento que está sendo empilhado. Como você acha que podemos implementar a

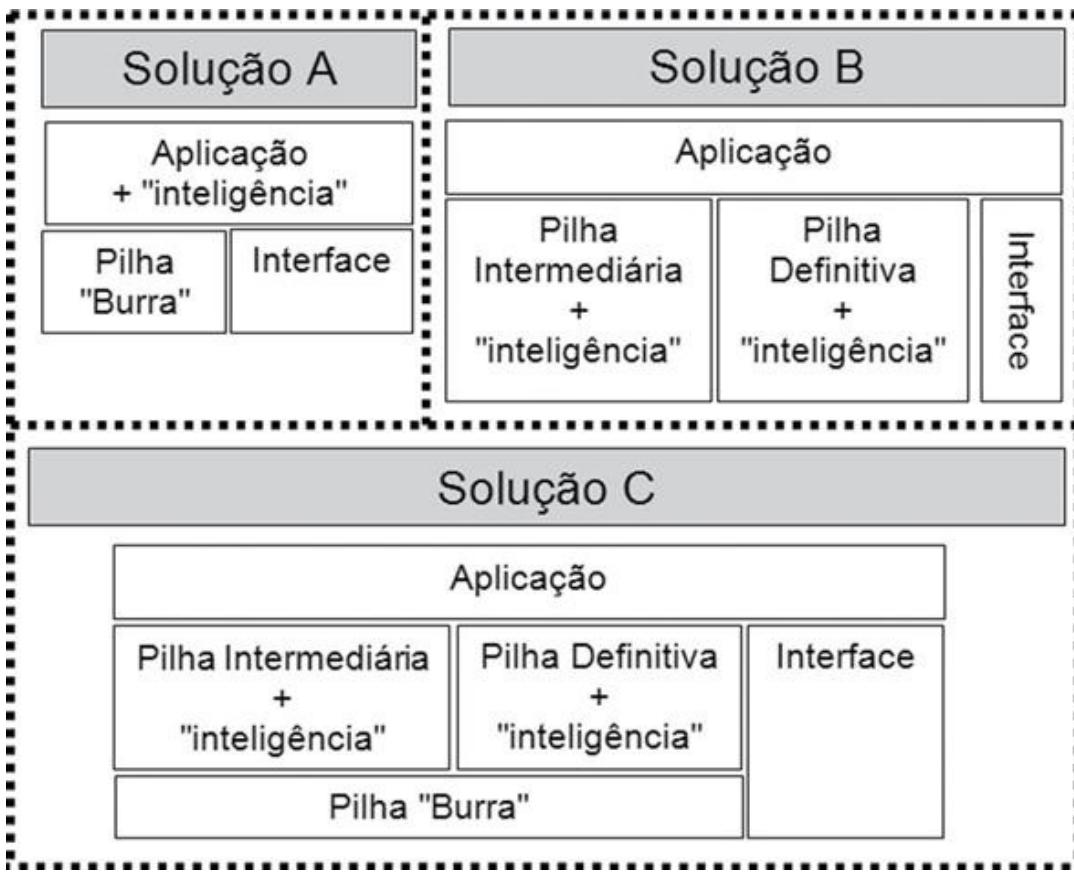
”inteligência“ das Pilhas Intermediárias e das Pilhas Definitivas? Por ”inteligência“ queremos dizer a verificação dos valores que estão sendo empilhados, para saber se podem ser inseridos em uma Pilha Intermediária ou em uma Pilha Definitiva, de acordo com as regras de funcionamento do *FreeCell*.



**FIGURA 2.17** Pilha Intermediária: ordem decrescente e em cores alternadas. Pilha Definitiva: ordem crescente e cartas do mesmo naipe.

A [Figura 2.18](#) mostra três alternativas para implementar a ”inteligência“ das Pilhas. Uma primeira alternativa (solução A) seria utilizar uma Pilha Burra e realizar todas as verificações na aplicação. Ou seja, ”fora“ do TAD Pilha é que iríamos verificar se uma carta pode ou não ser empilhada. Nessa solução A, tanto as Pilhas Intermediárias quanto as Pilhas Definitivas poderiam ser

implementadas através do mesmo TAD Pilha Burra.



**FIGURA 2.18** Projeto *FreeCell* — soluções alternativas quanto à arquitetura do software.

Uma segunda alternativa (solução B) seria desenvolver dois TADs diferentes, Pilha Intermediária e Pilha Definitiva, cada qual incorporando sua própria inteligência. Nessa solução, a aplicação em si não faria qualquer verificação de valores. Toda a verificação seria feita "dentro" das Pilhas. Se, pelas regras do *FreeCell*, uma carta não puder ser empilhada, a própria Pilha vai identificar isso e comunicar à aplicação. Note que, na solução B, temos duas implementações distintas de Pilha: a Pilha Intermediária e a Pilha Definitiva.

A solução C é uma variação da solução B. A diferença é que na solução C a Pilha Intermediária e a Pilha Definitiva são implementadas com base em uma Pilha Burra. Em uma linguagem orientada a objeto, como C++, as Pilhas Intermediárias e as Pilhas Definitivas podem "herdar" características da Pilha Burra e agregar suas características próprias e diferenciadas — a "inteligência". Mesmo sem utilizar o conceito de herança da orientação a objetos, as operações

das Pilhas Intermediária e Definitiva podem ser implementadas através de chamadas às operações da Pilha Burra.

As três soluções incluem um módulo que implementa a interface do sistema com o usuário. Incluir um módulo independente para a interface aumenta a portabilidade da solução e facilita ajustes ou mesmo uma troca radical de interface ou de plataforma, sem grandes mudanças nos demais módulos.

Qual dessas três soluções — A, B ou C — você considera mais adequada ao seu projeto?

## **Exercício 2.11 Avançar o Projeto *FreeCell* — propor uma arquitetura de software**

Pilha Burra? Pilha Inteligente? Proponha uma arquitetura de software para o seu Projeto *FreeCell* identificando os principais módulos do sistema e o relacionamento entre eles. Tome como ponto de partida as soluções alternativas propostas na [Figura 2.18](#). Escolha a alternativa que for mais adequada ao seu projeto ou adapte uma das alternativas, conforme necessário. Sugestão para uso acadêmico: desenvolva o Projeto *FreeCell* em grupo. Promova uma discussão, defina a arquitetura do software e divida o trabalho entre os componentes do grupo.

Seja qual for a arquitetura de software que adotar para o seu Projeto *FreeCell*, visando agregar portabilidade e reusabilidade, observe as seguintes recomendações na implementação:

- Para manipular as Pilhas de Cartas, projete e utilize um TAD Pilha (seja uma Pilha Burra, seja uma Pilha Inteligente).
- A aplicação em si e o TAD Pilha devem estar em unidades de software independentes e em arquivos separados; utilize um arquivo exclusivamente para a implementação do TAD Pilha.
- A aplicação (e outros módulos) deve manipular o TAD Pilha exclusivamente através dos seus Operadores Primitivos: Empilha, Desempilha, Vazia, Cria e Cheia. Aumente o volume da televisão apenas pelo botão de volume.
- Inclua no código do TAD Pilha exclusivamente ações pertinentes ao armazenamento e recuperação das informações sobre as Pilhas de Cartas. Faça o possível para não incluir no TAD Pilha ações relativas à interface ou a qualquer outro aspecto que não seja o armazenamento e a recuperação de informações sobre as Pilhas de Cartas.

## **Exercício 2.12 Implementar uma Pilha Burra**

Implemente um TAD Pilha Burra (Pilha sem qualquer restrição aos valores que estão sendo empilhados) em uma linguagem de programação como C ou C + +, sem utilizar recursos da orientação a objetos. Defina um tipo estruturado e os Operadores. Os elementos da Pilha devem ser do tipo Inteiro. Implemente a Pilha como uma unidade independente. Em um arquivo separado, faça o programa principal bem simples, para testar o funcionamento da Pilha.

### **Exercício 2.13 Implementar uma Pilha Burra como uma classe**

Implemente um TAD Pilha Burra (Pilha sem qualquer restrição aos valores que estão sendo empilhados) em uma linguagem de programação orientada a objetos, como C + +. Implemente a Pilha como uma classe. Os elementos da Pilha devem ser do tipo Inteiro. Implemente a Pilha como uma unidade independente. Em um arquivo separado, faça o programa principal bem simples, para testar o funcionamento da Pilha.

### **Exercício 2.14 Avançar o Projeto *FreeCell* — defina e implemente a Carta do Baralho**

Defina e implemente a Carta do Baralho, como uma classe ou como um tipo estruturado. Não se preocupe, neste momento, com aspectos da interface. Preocupe-se em definir uma carta com naipe (ouro, paus, copas ou espadas) e valor (A, 2, 3, 4, 5, 6, 7, 8, 9, 10, J, Q, K).

### **Exercício 2.15 Avançar o Projeto *FreeCell* — implemente uma Pilha Intermediária Inteligente**

Defina e implemente uma Pilha Intermediária "Inteligente", ou seja, com restrições quanto às cartas que serão empilhadas, conforme as regras do *FreeCell* convencional ou conforme as regras do seu Projeto *FreeCell*. Os elementos da Pilha devem ser do tipo Carta, conforme definido no [Exercício 2.14](#). Para implementar a Pilha Intermediária Inteligente, utilize de algum modo a Pilha Burra implementada no [Exercício 2.12](#) ou no [Exercício 2.13](#). Faça um programa para testar o funcionamento da Pilha Intermediária Inteligente sem se preocupar com a qualidade da interface.

### **Exercício 2.16 Avançar o Projeto *FreeCell* — implemente uma Pilha Definitiva Inteligente**

Defina e implemente uma Pilha Definitiva "Inteligente", ou seja, com restrições quanto às cartas que serão empilhadas, conforme as regras do *FreeCell* convencional. Os elementos da Pilha devem ser do tipo Carta, conforme definido no [Exercício 2.14](#). Para implementar a Pilha Definitiva Inteligente, utilize de algum modo a Pilha Burra implementada no [Exercício 2.12](#) ou no [Exercício 2.13](#). Faça um programa para testar o funcionamento da Pilha Definitiva Inteligente sem se preocupar com a qualidade da interface.

## **Exercício 2.17 Avançar o Projeto *FreeCell* — defina as regras, escolha um nome e inicie o desenvolvimento do seu *FreeCell***

Defina as regras do seu *FreeCell* alterando as regras do *FreeCell* convencional em algum aspecto. Dê personalidade própria ao seu *FreeCell* e escolha para o seu jogo um nome que reflita suas características mais marcantes. Escreva as regras do seu *FreeCell* e coloque em um arquivo-texto, para que possa ser utilizado na documentação do jogo ou em uma opção do jogo que ensine como jogar. Defina a arquitetura do software ([Exercício 2.11](#)), adapte as Pilhas desenvolvidas ([Exercícios 2.12 a 2.16](#)) e inicie o desenvolvimento do seu jogo. Sugestão para uso acadêmico: desenvolva o Projeto *FreeCell* em grupo. Tome as principais decisões em conjunto e divida o trabalho entre os componentes do grupo, cada qual ficando responsável por parte das atividades.

## **Exercício 2.18 Avançar o Projeto *FreeCell* — inicie o projeto da interface**



Em paralelo ao desenvolvimento da lógica do *FreeCell*, escolha e estude uma biblioteca gráfica para ajudar a construir uma interface visual e intuitiva para o seu jogo. Você pode começar estudando o **Tutorial de Programação Gráfica** disponível nos Materiais Complementares de *Estruturas de dados com jogos*, mas fique livre para estudar outros materiais ou adotar outras ferramentas gráficas no seu *FreeCell*. Projete a interface do modo mais independente possível dos demais módulos. Se estiver trabalhando em grupo, um dos membros do grupo pode se dedicar a estudar mais intensamente os aspectos referentes à interface e depois ajudar os demais componentes no aprendizado. Você pode optar também por implementar primeiramente uma interface bem simples, textual, para testar bem os demais componentes do jogo e depois

substituir por uma interface gráfica. Em algum momento, você precisará mesmo ganhar experiência no desenvolvimento de software utilizando bibliotecas gráficas. Por que não agora?

## Comece a desenvolver o seu *FreeCell* agora!

Faça um jogo legal! Faça um jogo que tenha a sua cara! Faça seu jogo ficar atrativo, distribua para os amigos, disponibilize publicamente, faça seu jogo bombar! Aprender a programar pode ser divertido!



### Consulte no Banco de Jogos

Adaptações do *FreeCell*

[www.elsevier.com.br/edcomjogos](http://www.elsevier.com.br/edcomjogos)



### Consulte nos Materiais Complementares

Tutorial de Programação Gráfica



<http://www.elsevier.com.br/edcomjogos>

## Exercícios de fixação

**Exercício 2.19** O que é uma Pilha?

**Exercício 2.20** Para que serve uma Pilha? Em que tipo de situações uma Pilha pode ser utilizada?

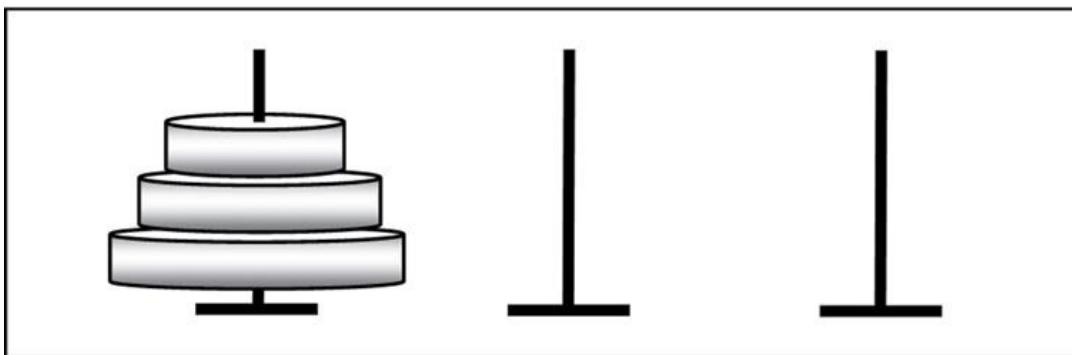
**Exercício 2.21** O que significa Alocação Sequencial de Memória para um conjunto de elementos?

**Exercício 2.22** O que significa Alocação Estática de Memória para um conjunto de elementos?

**Exercício 2.23** Faça o esquema da implementação de uma Pilha com Alocação Sequencial e Estática de Memória, e descreva o seu funcionamento.

**Exercício 2.24 Desafio das Torres de Hanoi.** O Desafio das Torres de Hanoi é um jogo clássico, inspirado em uma antiga lenda. O jogo conta com três hastes verticais. Uma das hastes contém três discos: embaixo, um primeiro disco de

diâmetro maior; em cima desse primeiro disco, há um segundo disco de diâmetro um pouco menor; e em cima desse segundo disco, há um terceiro disco de diâmetro ainda menor, como ilustra a [Figura 2.19](#). O desafio do jogo é passar todos os três discos, nessa mesma sequência, para uma das outras hastes verticais. É possível utilizar as três hastes para auxiliar a movimentação dos discos. Mas existem algumas restrições na movimentação: só é possível movimentar um disco de cada vez; só é possível retirar o disco do topo de uma pilha de discos; só é possível inserir discos no topo de uma pilha de discos; em nenhum momento você pode colocar um disco em cima de outro disco com diâmetro menor. Se não conhece bem o Desafio das Torres de Hanoi, jogue algumas partidas em uma versão on-line (por exemplo, no Games On e no UOL Jogos — links 1 e 2) ou assista a algum vídeo (por exemplo, nos links 3 e 4) para se familiarizar com o jogo. É possível aumentar a quantidade de discos e com isso aumentar o grau de dificuldade. Como você poderia usar uma ou mais Pilhas para implementar um jogo como o Desafio das Torres de Hanoi?



**FIGURA 2.19** Ilustração do Desafio das Torres de Hanoi.

**Exercício 2.25 Identificar outras aplicações de Pilhas.** Identifique e liste aplicações de Pilhas. Identifique e anote alguns jogos que podem ser implementados com o uso de uma Pilha e identifique também outras aplicações de fora do mundo dos games. Sugestão de uso acadêmico: faça uma discussão em grupo. Ao final, cada grupo apresenta a todos os estudantes as aplicações que identificou. Pode ser uma boa fonte de ideias para novos jogos.

## Soluções e sugestões para alguns exercícios

### Exercício 2.2 Mais elementos?

```

Boolean MaisElementos (parâmetros por referência P1, P2 do tipo Pilha) {
    /* retorna Verdadeiro se a Pilha P1 tiver mais elementos do que a Pilha P2 */

    /* A questão deve ser resolvida exclusivamente através dos operadores da Pilha - Empilha,
    Desempilha, Vazia, Cria, e Cheia. A lógica é retirar um a um os elementos de P1 e colocar em uma
    Pilha auxiliar PAux1. A cada elemento retirado, incrementamos um contador. Depois retornamos
    todos os elementos de PAux1 para P1. Fazemos o mesmo para P2. Finalmente, verificamos se o
    número de elementos em P1 é maior do que o número de elementos em P2 */

    Variável DeuCerto do tipo Boolean;
    Variável ElementoDaPilha do tipo Inteiro;
    Variáveis ElementosEmP1, ElementosEmP2 do tipo Inteiro;
    Variáveis PAux1, PAux2 do tipo Pilha;

    /* contando o número de elementos em P1: retira todos de P1 e coloca em PAux1 */
    ElementosEmP1 = 0;
    Cria( PAux1 );
    Enquanto (Vazia(P1) == Falso) Faça {
        Desempilha(P1, ElementoDaPilha, DeuCerto); // retira elemento de P1...
        Empilha(PAux1, ElementoDaPilha, DeuCerto); // ... e insere em PAux1
        ElementosEmP1 = ElementosEmP1 + 1; }; // incrementa o contador

    /* retornando todos os elementos de PAux1 para P1 */
    Enquanto (Vazia(PAux1) == Falso) Faça {
        Desempilha(PAux1, ElementoDaPilha, DeuCerto); // retira elemento de PAux1
        Empilha(P1, ElementoDaPilha, DeuCerto); }; // ... e retorna para P1

    /* contando o número de elementos de P2: retira todos de P2 e coloca em PAux2 */
    ElementosEmP2 = 0;
    Cria( PAux2 );
    Enquanto (Vazia(P2) == Falso) Faça {
        Desempilha(P2, ElementoDaPilha, DeuCerto); // retira elemento de P2...
        Empilha(PAux2, ElementoDaPilha, DeuCerto); // ... e insere em PAux1
        ElementosEmP2 = ElementosEmP2 + 1; }; // incrementa o contador

    /* retornando todos os elementos de PAux2 para P2 */
    Enquanto (Vazia(PAux2) == Falso) Faça {
        Desempilha(PAux2, ElementoDaPilha, DeuCerto); // retira elemento de PAux2
        Empilha(P2, ElementoDaPilha, DeuCerto); }; // ... e retorna para P2

    /* o resultado... */
    Se (ElementosEmP1 > ElementosEmP2) // compara os contadores...
    Então Retorne Verdadeiro;
    Senão Retorne Falso;
} /* fim do procedimento */

```

## Exercício 2.4 As Pilhas são iguais?

```

Boolean Iguais (parâmetros por referência P1, P2 do tipo Pilha) {
    /* retorna o valor Verdadeiro se a Pilha P1 for igual à Pilha P2, ou seja, se possuem exatamente os
    mesmos elementos, na mesma ordem. Se ambas as Pilhas forem vazias, serão consideradas
    iguais */
    /* Resolução: A questão deve ser resolvida exclusivamente através dos operadores da Pilha -
    Empilha, Desempilha, Vazia, Cria, e Cheia */

    Variáveis DeuCerto1, DeuCerto2, PorEnquantoSãoIguais do tipo Boolean;
    Variáveis ElementoDaPilha1, ElementoDaPilha2 do tipo Inteiro;
    Variáveis PAux1, PAux2 do tipo Pilha;

    PorEnquantoSãoIguais = Verdadeiro; // a princípio, consideramos que P1 e P2 são iguais.

    /* tentando encontrar situações em que as Pilhas P1 e P2 não são iguais... */
    /* enquanto houver elemento em P1 E também em P2... retirar 1 elemento de P1 e 1 elemento de
    P2 */

    Enquanto ((Vazia(P1) == Falso) AND Vazia(P2) == Falso) {
        Desempilha(P1, ElementoDaPilha1, DeuCerto1);
        Desempilha(P2, ElementoDaPilha2, DeuCerto2);
        Se (DeuCerto1 == Verdadeiro)
            Então Empilha(PAux1, ElementoDaPilha1, DeuCerto1);
        Se (DeuCerto2 == Verdadeiro)
            Então Empilha(PAux2, ElementoDaPilha2, DeuCerto2);

        /* Para buscar situações em que as Pilhas P1 e P2 não são iguais... em cada "loop" desse
        comando de repetição, quatro casos podem ocorrer:
            DeuCerto1           DeuCerto2
        Caso 1- Falso          Falso
        Caso 2- Falso          Verdadeiro
        Caso 3- Verdadeiro    Falso
        Caso 4- Verdadeiro    Verdadeiro

        No Caso 1, concluímos que as Pilhas continuam sendo iguais. Nos casos 2 e 3 concluímos que as
        Pilhas não são iguais (pois o número de elementos é diferente). No caso 4 se o elemento retirado
        de P1 for diferente do elemento retirado de P2, concluímos que as Pilhas P1 e P2 não são iguais...
        */

        Se ((DeuCerto1 == Verdadeiro) AND (DeuCerto2 == Falso))
            Então PorEnquantoSãoIguais = Falso;                                // caso 2
        Se ((DeuCerto1 == Falso) AND (DeuCerto2 == Verdadeiro))
            Então PorEnquantoSãoIguais = Falso;                                // caso 3
        Se ((DeuCerto1 == Verdadeiro) AND (DeuCerto2 == Verdadeiro)) // caso 4
            Então   Se (ElementoDaPilha1 != ElementoDaPilha2)
                Então PorEnquantoSãoIguais = Falso;
            } /* fim do comando Enquanto */

        /* voltando os elementos de PAux1 para P1 */
        Enquanto (Vazia(PAux1) == Falso) Faça {
            Desempilha(PAux1, ElementoDaPilha1, DeuCerto1);
            Empilha(P1, ElementoDaPilha1, DeuCerto1);};

        /* voltando os elementos de PAux2 para P2 */
        Enquanto (Vazia(PAux2) == Falso) Faça {
            Desempilha(PAux2, ElementoDaPilha2, DeuCerto2);
            Empilha(P2, ElementoDaPilha2, DeuCerto2);};

        /* dando o resultado...*/
        Se (PorEnquantoSãoIguais == Verdadeiro)
            Então Retorne Verdadeiro;
        Senão Retorne Falso;
    } /* fim do procedimento */
}

```

## **Exercício 2.12 Implementação de uma “Pilha Burra”**

```

/* arquivo PilhaSeqEstd.h - implementa o TAD Pilha sem utilizar recursos da orientação a objetos */
#include<conio.h>
#include<stdio.h>

#define Tamanho 3 // Tamanho da Pilha

typedef struct P {
    int Elementos[Tamanho]; // vetor de 0 a Tamanho-1
    int Topo;
} Pilha; // define o Tipo Estruturado Pilha

void Cria(Pilha *p){
    p->Topo = -1; // o vetor começa em 0; logo, inicializamos a pilha em -1
}

bool Vazia(Pilha *p){
    if(p->Topo == -1)
        return true;
    else
        return false;
}

bool Cheia(Pilha *p){
    if(p->Topo == Tamanho-1)
        return true;
    else
        return false;
}

void Empilha(Pilha *p, int x, bool *DeuCerto){
    if(Cheia(p)==1)
        *DeuCerto = false;
    else {
        p->Topo++;
        p->Elementos[p->Topo] = x;
        *DeuCerto = true;
    }
}

void Desempilha(Pilha *p, int *x, bool *DeuCerto){
    if(Vazia(p)==1)
        *DeuCerto = false;
    else {
        *x = p->Elementos[p->Topo];
        p->Topo--;
        *DeuCerto = true;
    }
}

/* arquivo PilhaSeqEstd.cpp - programa para testar o TAD Pilha */
#include "PilhaSeqEstd.h"

void imprime(Pilha *p){ // imprime sem abrir a TV
    int x;
    Pilha PAux;
    bool ok;
    Cria(&PAux);
    while (Vazia(p)==false) {
        Desempilha(p, &x, &ok);

```

```

        if (ok) {
            Empilha(&PAux, x, &ok);
        } // if
    } // while
printf("\n imprimindo a pilha: ");
while (Vazia(&PAux)==false) {
    Desempilha(&PAux, &x, &ok);
    if (ok){
        printf("%d ", x);
        Empilha(p, x, &ok);
    } // if
} // while
printf(" <- topo");
}

int main(){
    Pilha p;
    Cria(&p);
    bool ok;
    int a;
    printf("\n tentando empilhar 4 elementos em uma Pilha em que só cabem 3");
    Empilha(&p,10,&ok); imprime (&p);
    Empilha(&p,20,&ok); imprime (&p);
    Empilha(&p,30,&ok); imprime (&p);
    Empilha(&p,40,&ok); // tamanho da Pilha é 3; 40 não será empilhado
    imprime (&p);
    printf("\n pressione uma tecla... \n"); getch();
    printf("\n tentando desempilhar 4 elementos de uma Pilha que só tem 3");
    Desempilha(&p,&a,&ok); imprime (&p);
    Desempilha(&p,&a,&ok); imprime (&p);
    Desempilha(&p,&a,&ok); imprime (&p);
    Desempilha(&p,&a,&ok); //Pilha vazia; nao vai conseguir desempilhar
    imprime (&p);
    printf("\n pressione uma tecla... \n"); getch(); return(0);
}

```

## Exercício 2.13 Pilha Burra como classe em C + +

```

/* arquivo PilhaBurra.h - implementa um TAD PilhaBurra como uma Classe */
#define Tamanho 3
class PilhaBurra {
private:
    int Topo;
    char Elementos[Tamanho];
public:
    PilhaBurra();
    bool Vazia();
    bool Cheia();
    void Empilha(int, bool &);
    void Desempilha(int &, bool &);
};

PilhaBurra::PilhaBurra () {
    Topo = -1;
}

bool PilhaBurra::Vazia () {
    if (Topo == -1)
        return true;
    else return false;
}

bool PilhaBurra::Cheia () {
    if (Topo == (Tamanho-1))
        return true;
    else return false;
}

void PilhaBurra::Empilha( int X, bool &DeuCerto) {
    if (Cheia())
        DeuCerto = false;
    else {
        DeuCerto = true;
        Topo = Topo + 1;
        Elementos[Topo] = X;
    }
}

void PilhaBurra::Desempilha( int &X, bool &DeuCerto) {
    if (Vazia())
        DeuCerto = false;
    else {
        DeuCerto = true;
        X = Elementos[ Topo ];
        Topo = Topo - 1;
    }
}

```

```

/* arquivo PilhaBurra.cpp */
/* programa para testar a TAD Pilha implementada como Classe */

#include<conio.h>
#include<stdio.h>
#include<iostream>
#include "PilhaBurra.h"

using namespace std;

void imprime(PilhaBurra &p){
    int x;
    PilhaBurra PAux;
    bool ok;
    while (p.Vazia()==false) {
        p.Desempilha(x, ok);
        if (ok) {
            PAux.Empilha(x, ok);
        } // if
    } // while
    cout << "imprimindo a pilha: ";
    while (PAux.Vazia()==false) {
        PAux.Desempilha(x, ok);
        if (ok){
            cout << x << " ";
            p.Empilha(x, ok);
        } // if
    } // while
    cout << " <- topo" << endl;
}

int main(){
    PilhaBurra p;
    bool ok;
    char op = 't';
    int valor;
    while (op != 's') {
        cout << "digite: (e)empilhar,(d)desempilar, (s)sair [enter]" << endl;
        cin >> op;
        switch (op) {
            case 'e' : cout << "digite valor INTEIRO para empilhar empilhar [enter]" << endl;
                        cin >> valor; // CUIDADO: DIGITE UM INTEIRO MESMO!!!
                        p.Empilha(valor, ok);
                        if (ok==true) cout << " valor empilhado" << endl;
                        else cout << " nao conseguiu empilhar" << endl;
                        break;
            case 'd' : p.Desempilha(valor,ok);
                        if (ok==true) cout << "valor desempilhado= " << valor << endl;
                        else cout << "nao conseguiu desempilar" << endl;
                        break;
            default : cout << "saindo..." << endl; op = 's'; break;
        }; // case
        imprime (p);
    } // while
    cout <<"pressione uma tecla..." << endl;
    getch();
    return(0);
}

```

## **Exercício 2.14 Carta do Baralho**

Sugestões:

- A Carta-do-Baralho pode ser um tipo estruturado ou uma classe, com dois campos: Naipe (ouros, copas, espadas e paus) e Valor (A, 2, 3, 4, 5, 6, 7, 8, 9, 10, J, Q, K). O naipe pode ser definido como um tipo enumerado ou do tipo inteiro (nesse caso, estabelecendo uma correlação do tipo ouros = 1, copas = 2, e assim por diante). Valor pode ser um tipo enumerado, do tipo inteiro ou do tipo Char.
- Também é possível definir operações sobre cartas — operação para retornar o naipe, retornar o valor, atualizar o naipe, atualizar o valor, entre outras.

## **Exercício 2.15 e 2.16 Pilha Intermediária Inteligente e Pilha Definitiva Inteligente**

Sugestões:

- Defina uma classe PilhaBurra de elementos do tipo Carta-do-Baralho. Então defina uma classe PilhaIntermediária herdando características da classe PilhaBurra; reescreva a operação Empilha, só permitindo empilhar cartas na sequência correta; em uma PilhaIntermediária podemos empilhar qualquer carta se a Pilha estiver vazia ou em ordem decrescente e em cores alternadas, se a Pilha não estiver vazia.
- Faça o mesmo para a classe PilhaDefinitiva; em uma PilhaDefinitiva podemos empilhar somente a carta de valor A se a Pilha estiver vazia ou em ordem crescente e do mesmo naipe, se a Pilha não estiver vazia.
- Na prática você terá três operações Empilha (Pilha Burra, Pilha Definitiva e Pilha Intermediária). O código das demais operações será comum.

Para exemplificar, segue trecho de implementação de uma PilhaInteligente com elementos do tipo inteiro, na qual só é possível empilhar elementos maiores do que o elemento do topo da pilha.

```

class PilhaInteligente : public PilhaBurra{
public:
    void Empilha(int, bool &);           // Empilha condicionalmente
};

void PilhaInteligente::Empilha(int X, bool &DeuCerto){
/* so empilha se X for maior que o elemento que está no topo da pilha ou se a pilha estiver vazia */
    if (Vazia())
        PilhaBurra::Empilha(X, DeuCerto); // se a pilha estiver vazia, empilha
    else {int Y;
        Desempilha(Y, DeuCerto); // pega o valor do topo - valor em Y
        PilhaBurra::Empilha(Y, DeuCerto); // volta o valor Y para o topo
        if (X > Y)
            PilhaBurra::Empilha(X, DeuCerto); // empilha X apenas se X>Y
        else DeuCerto = false;
    }
}

```

## Links

1. Games On: Torre de Hanoi: <http://www.gameson.com.br/Jogos-Online/ClassicoPuzzle/Torre-de-Hanoi.html> (consulta em setembro de 2013).
2. UOL Jogos: Torre de Hanoi: [http://jogosonline.uol.com.br/torre-de-hanoi\\_1877.html](http://jogosonline.uol.com.br/torre-de-hanoi_1877.html) (consulta em setembro de 2013).
3. YouTube: Torres de Hanoi: Bezerra: <http://www.youtube.com/watch?v=yrNWiFFbcEY> (consulta em setembro de 2013).
4. YouTube: Torres de Hanoi: Neves: [http://www.youtube.com/watch?v=3qTe\\_X1yXEs](http://www.youtube.com/watch?v=3qTe_X1yXEs) (consulta em setembro de 2013).

## Referências e leitura adicional

1. Knuth D. *The Art of Computer Programming*. Volume I: Fundamental Algorithms. Reading, MA: Addison-Wesley, 1972.
2. Langsam Y, Augenstein MJ, Tenenbaum A M. *Data Structures Using C and C++*. 2nd ed. Upper Saddle River. New Jersey: Prentice Hall, 1996.
3. Pereira SL. *Estruturas de dados fundamentais: conceitos e aplicações*. São Paulo: Érica; 1996.

---

## **PARTE II**

# Filas

### **OUTLINE**

---

Desafio 2 Desenvolver uma adaptação do jogo *Snake*

Capítulo 3 Filas com Alocação Sequencial e Estática

Capítulo 4 Listas Encadeadas

Capítulo 5 Listas Encadeadas com Alocação Dinâmica

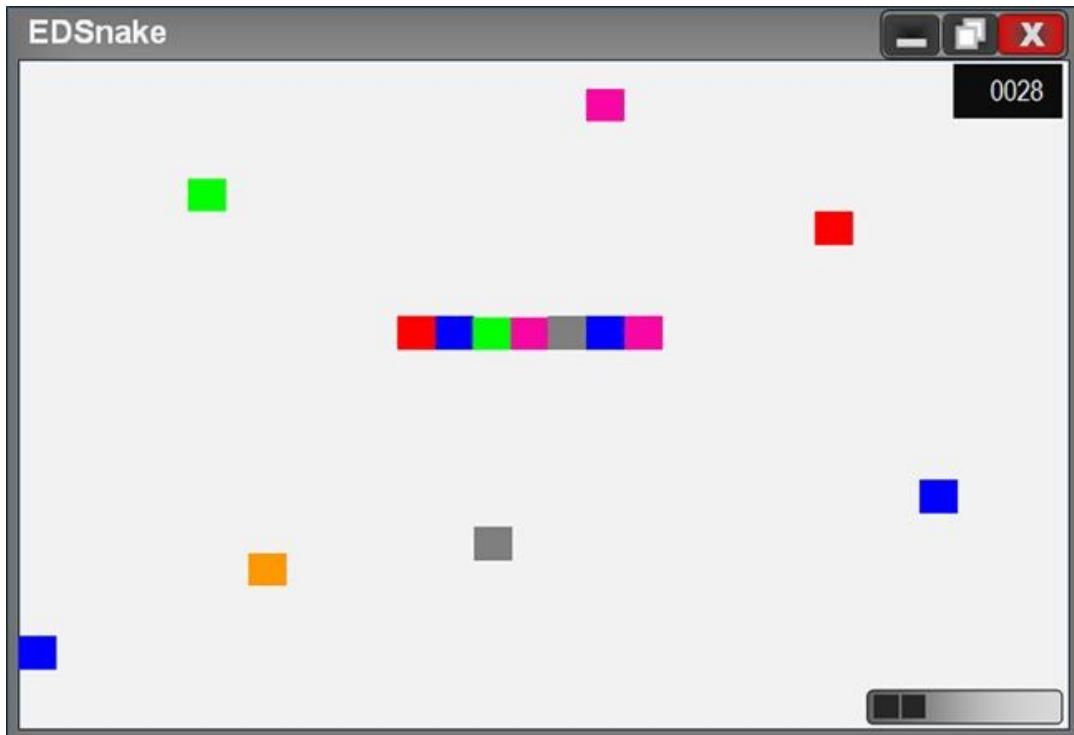
---

## DESAFIO 2

---

# Desenvolver uma adaptação do jogo *Snake*

---



**FIGURA D2.1** EDSnake: cobra formada por quadradinhos coloridos.

Jogos do tipo *Snake* surgiram na época dos primeiros consoles de jogos. Depois apareceram versões do *Snake* para computadores e celulares. Você encontrará implementações do *Snake* em sites de jogos, como o UOL Jogos e o Mania de Jogos (links 1 e 2).

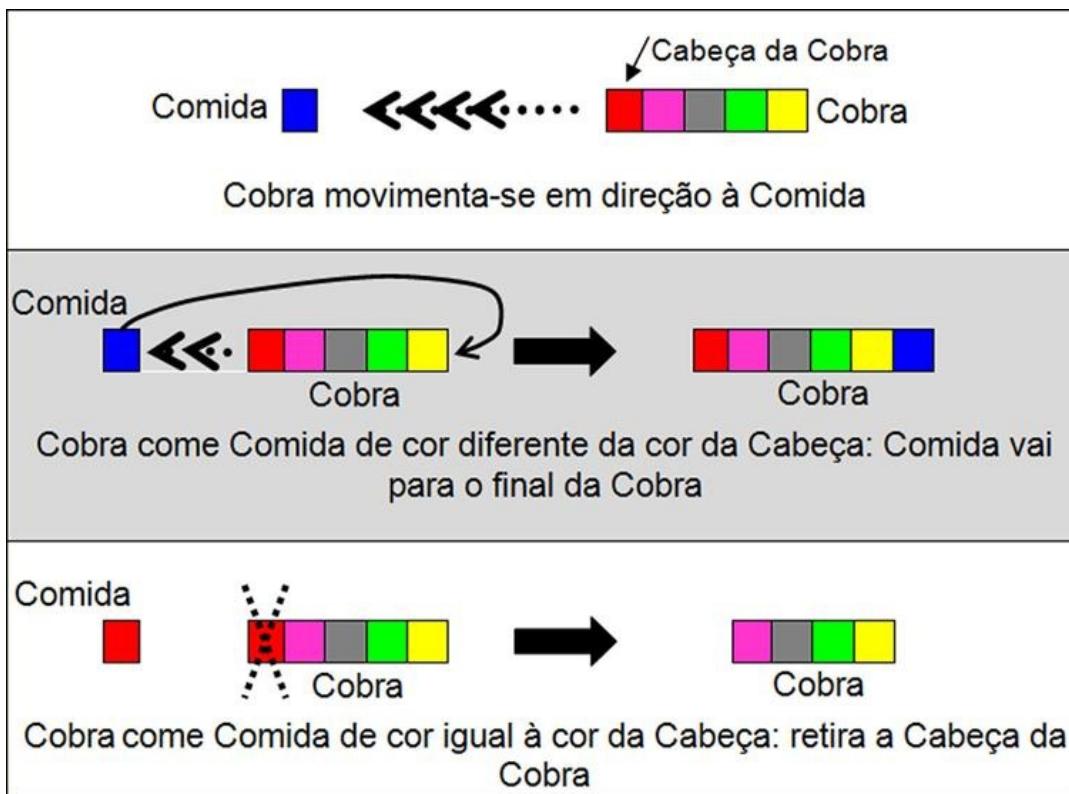
O elemento principal dos jogos *Snake* é uma *Cobra* que vai comendo “comidas” que aparecem na tela. Em geral, quando a *Cobra* come sua comida, o jogador marca pontos, a Comida passa a fazer parte da *Cobra*, e esta fica mais comprida, dificultando um pouco mais a evolução do jogo. É um jogo de habilidade, em que o objetivo é marcar o maior número de pontos possível.



## Cobras e Comidas coloridas

Em algumas versões do Snake, a *Cobra* é formada por pequenos *quadradinhos coloridos*. A tela apresenta também outros quadradinhos coloridos, soltos, que a Cobra poderá comer ao passar por cima. Você encontrará algumas versões do Snake com Cobras e Comidas coloridas no Banco de Jogos.

Nas versões coloridas do Snake, a cor da Cabeça da Cobra e a Cor da Comida são importantes. Por exemplo, segundo uma estratégia proposta no ColorSnake (link 3), se a Cobra come uma Comida de cor diferente da cor de sua Cabeça, o jogador marca pontos, e a Comida é inserida no final da Cobra. Se a Cobra come uma Comida da mesma cor de sua Cabeça, um quadradinho colorido é retirado da Cabeça da Cobra. A cada quadradinho que a Cobra come, outros quadradinhos vão surgindo na tela e, assim, o jogo vai evoluindo.



**FIGURA D2.2** Versões coloridas do *Snake*: *Cobra* formada por quadradinhos coloridos.

Esta descrição é apenas um exemplo. Outras versões coloridas do *Snake* podem mudar o modo de pontuar, o modo de perder ou ganhar, e o modo de

determinar o andamento do jogo. Mas o seguinte ponto da estratégia é importante: os quadradinhos coloridos entram sempre no final da Cobra, e são retirados sempre do início da Cobra. Como se a Cobra fosse uma Fila, com os quadradinhos entrando no final e saindo do começo

## Como você desenvolveria um jogo como o *Snake*?

Pense em desenvolver uma versão colorida do *Snake*. Como você faria para representar a *Cobra* em seu programa? Como armazenaria, a cada momento, a sequência dos *quadradinhos coloridos* que formam a *Cobra*? Como faria para controlar a entrada dos *quadradinhos* no final da *Cobra* e a retirada do quadradinho colorido da *Cabeça da Cobra*?

## Seu desafio: desenvolver uma adaptação colorida do *Snake*

Use sua criatividade e desenvolva uma adaptação colorida do *Snake*. Faça alguns ajustes nas regras do jogo: estabeleça o seu próprio esquema de pontuação, defina fases de dificuldade crescente ou algo assim, mas mantenha a característica fundamental de acrescentar elementos no final da *Cobra* e retirar elementos do início da *Cobra*, como em uma Fila. Seu jogo pode até ficar bem diferente, mas mantendo essa característica principal: um conjunto em que os elementos são inseridos sempre no fim e retirados sempre do começo, como em uma Fila.

## Por onde começar?

Comece estudando os [Capítulos 3, 4 e 5](#) de *Estruturas de dados com jogos*. Esses capítulos o ajudarão a escolher uma Estrutura de Dados adequada para armazenamento e manipulação dos elementos coloridos que formam a *Cobra*.

Construa o seu próprio *Snake*! Dê um nome criativo ao seu jogo! Deixe as pessoas com vontade de jogar! Aprender a programar pode ser divertido!

[Consulte nos Materiais Complementares](#)



Banco de Jogos: Adaptações do Snake

Banco de Jogos: Aplicações de Filas



<http://www.elsevier.com.br/edcomjogos>

## Links

1. UOL Jogos. Disponível em: <http://clickjogos.uol.com.br/Jogos-online/Acao-e-Aventura/Snake> (acesso em setembro de 2013).
2. Mania de Jogos. Disponível em: <http://www.maniadejogos.com/jogos-online/Snake> (acesso em setembro de 2013).
3. ColorSnake. Desenvolvido por Camilo Moreira, João Paulo Soares e Matheus Takata. EDGames 2012. Disponível em: <http://edgames.dc.ufscar.br> (acesso em setembro de 2013).

---

## CAPÍTULO 3

---

# Filas com Alocação Sequencial e Estática

---

## Seus objetivos neste capítulo

- Entender o que é e para que serve uma estrutura do tipo Fila.
- Desenvolver habilidade para implementar uma estrutura do tipo Fila, como um Tipo Abstrato de Dados (TAD), com Alocação Sequencial e Alocação Estática de Memória.
- Desenvolver habilidade para manipular Filas através dos Operadores definidos para o TAD Fila.
- Iniciar o desenvolvimento do seu segundo jogo, referente ao Desafio 2.

### 3.1 O que é uma Fila?

Quando temos um conjunto de pessoas esperando para serem atendidas, temos uma Fila. Fila de clientes no caixa do banco, fila no caixa do supermercado, fila de carros no pedágio, e assim por diante.

Em uma Fila séria, quem chega primeiro é atendido primeiro. Ou seja, novos elementos entram sempre no Final da Fila. E o elemento que sai é sempre o Primeiro da Fila.

### Definição: Fila

Fila é uma estrutura para armazenar um conjunto de elementos que funciona da seguinte forma:

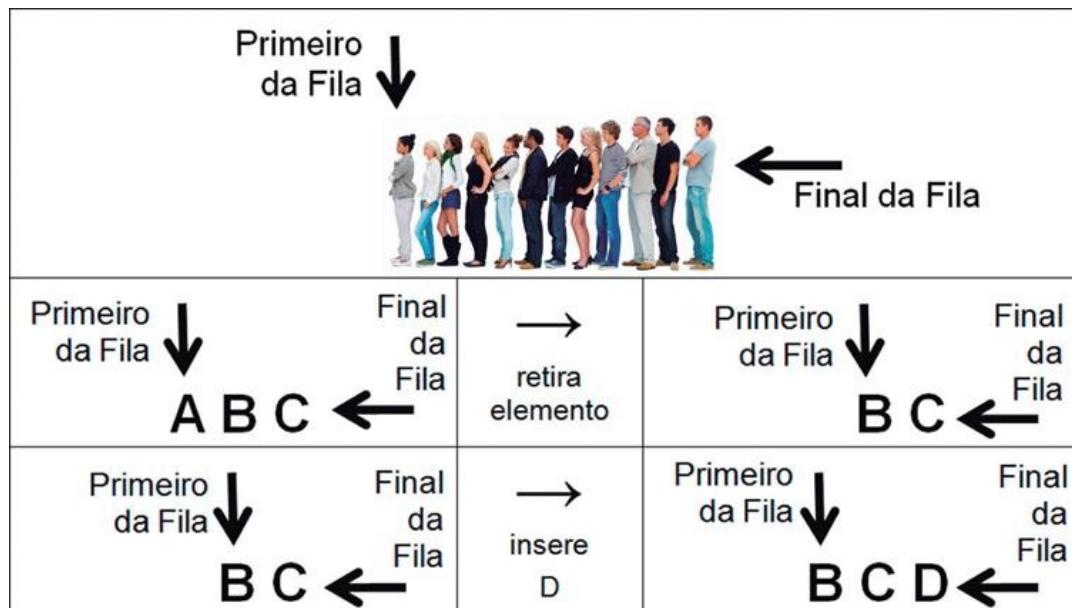
- Novos elementos entram no conjunto sempre no Final da Fila.
- O único elemento que pode ser retirado da Fila em determinado momento é o Primeiro elemento da Fila.

### Do inglês Queue, F.I.F.O.

Uma Fila (em inglês, *Queue*) é uma estrutura que obedece ao critério *First In, First Out* (F.I.F.O.), ou seja, o primeiro elemento que entrou no conjunto será o primeiro elemento a sair do conjunto.

**FIGURA 3.1** Definição de Fila.

Uma Fila é um conjunto ordenado de elementos. A ordem de entrada dos elementos determina sua posição no conjunto. Se temos três elementos em uma Fila (A, B e C) e se eles entraram na Fila na ordem A, depois B e depois C, sabemos que o Primeiro elemento da Fila é A. Sabemos também que, nesse momento, A é o único elemento que podemos retirar da Fila. Se, depois disso, quisermos inserir na Fila o elemento D, esse elemento D passará a ser o Último elemento da Fila ([Figura 3.2](#))

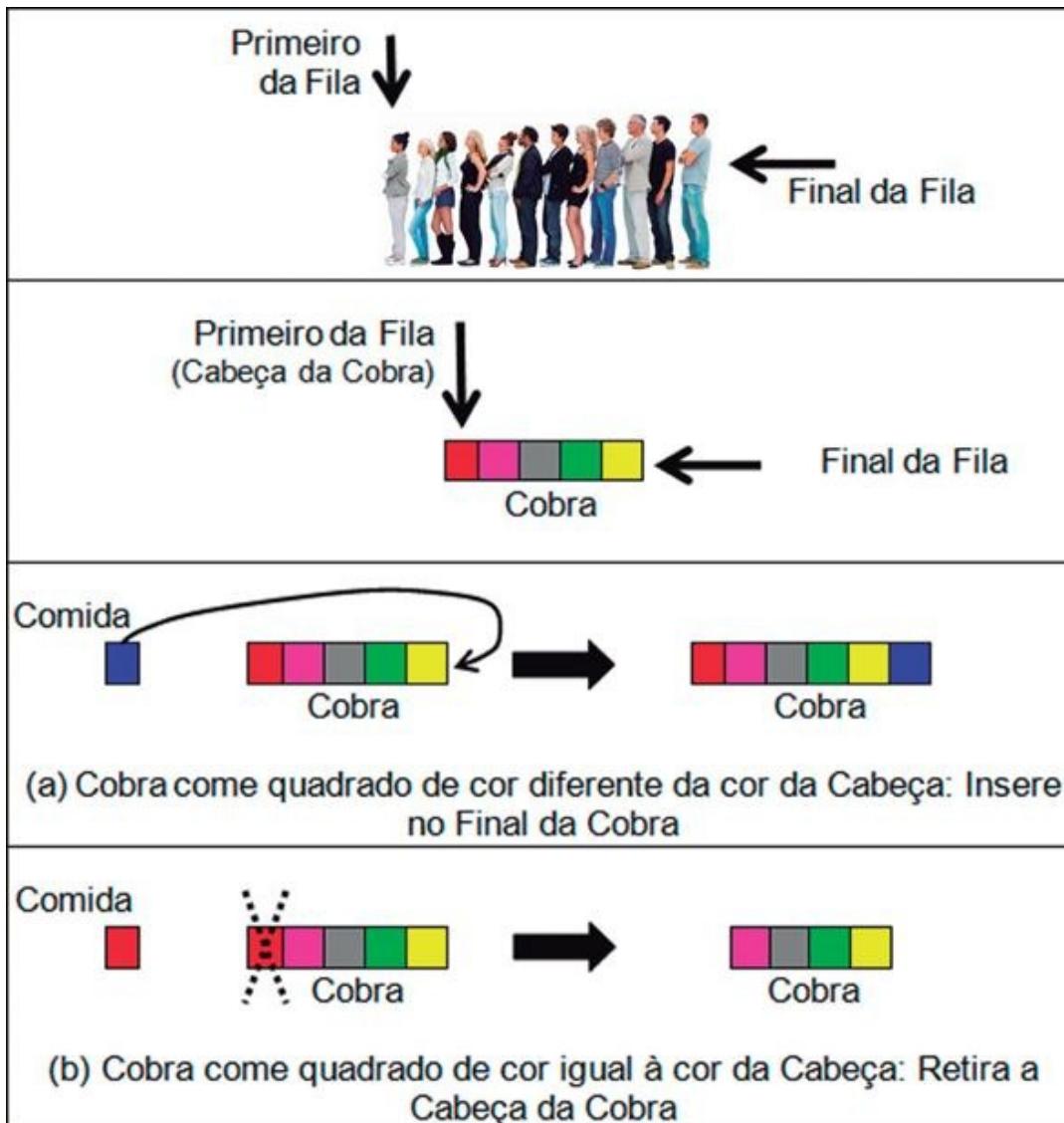


**FIGURA 3.2** Retirando e inserindo elementos em uma Fila.

## 3.2 O jogo *Snake*

O seu segundo desafio é desenvolver uma adaptação do jogo *Snake*. Nas versões coloridas do *Snake*, uma *Cobra* é formada por pequenos *quadrinhos coloridos*. No decorrer do jogo, a *Cobra* vai “comendo” quadrinhos coloridos que estão na tela.

Na versão do *Snake* descrita no Desafio 2, ao comer um quadrinho colorido de cor diferente da cor da Cabeça da *Cobra*, o jogador marca pontos, e o quadrinho comido é inserido no final da *Cobra* ([Figura 3.3a](#)). Se a *Cobra* come um quadrinho colorido da mesma cor de sua Cabeça, o quadrinho colorido que representa a Cabeça da *Cobra* é retirado ([Figura 3.3b](#)).



**FIGURA 3.3** Projeto *Snake*: a Cobra como uma Fila.

Inserir elementos sempre no Final da Cobra; retirar somente o Primeiro elemento da Cobra: essa Cobra funciona como uma Fila!

### 3.3 Operações do tipo abstrato de Dados Fila

A partir da própria definição de Fila, já podemos identificar duas operações: inserir elemento na Fila e retirar elemento da Fila. A Operação para retirar um elemento da Fila deve retirar, precisamente, o Primeiro elemento. E a Operação para inserir um novo elemento deve inseri-lo, precisamente, no Final da Fila. A essas duas Operações vamos acrescentar outras três: Operação para Criar uma Fila, Operação para testar se a Fila está Vazia e Operação para testar se a Fila está Cheia. A [Figura 3.4](#) apresenta uma descrição dos parâmetros e do funcionamento das Operações Insere, Retira, Cria, Vazia e Cheia.

Operações e parâmetros	Funcionamento
Insere (F, X, DeuCerto)	Insere o elemento X, passado como parâmetro, na Fila F, também passada como parâmetro. O parâmetro DeuCerto indica se a execução da operação foi bem-sucedida ou não.
Retira (F, X, DeuCerto)	Retira o primeiro elemento da Fila F, passada no parâmetro, retornando o valor do elemento que foi retirado da Fila no parâmetro X. O parâmetro DeuCerto indica se a operação foi bem-sucedida ou não.
Vazia (F)	Verifica se a Fila F, passada como parâmetro, está vazia. Uma Fila vazia é uma Fila sem elementos.
Cheia (F)	Verifica se a Fila F, passada como parâmetro, está cheia. Uma Fila cheia é uma Fila em que não cabe mais nenhum elemento.
Cria (F)	Cria uma Fila F, iniciando sua situação como vazia.

**FIGURA 3.4** Operações do TAD Fila.

Para elaborar os [Exercícios 3.1](#) e [3.2](#), use os Operadores definidos na [Figura 3.4](#). Você terá que propor soluções sem saber como o Tipo Abstrato de Dados

Fila é efetivamente implementado. Você encontrará uma solução para o [Exercício 3.1](#) na sequência do texto, mas não consulte essa solução ainda. Tente propor uma solução. É assim que desenvolvemos habilidade para manipular Filas e outras estruturas pelos seus Operadores Primitivos ou, ainda, pelos “botões da televisão”.

## Exercício 3.1 Junção dos elementos de duas Filas

Desenvolva um algoritmo para reunir os elementos de duas Filas F1 e F2 em uma terceira Fila F3. F3 deve conter todos os elementos de F1 seguidos de todos os elementos de F2. Considere que as Filas F1 e F2 já existem e são passadas como parâmetros. Preserve os elementos de F1 e F2: ao final da Operação, F1 e F2 devem possuir os mesmos elementos que possuíam no início da Operação, e os elementos devem estar na mesma ordem. Para propor a solução, utilize os Operadores da [Figura 3.4](#).

```
JunçãoDosElementos (parâmetro por referência F1, F2, F3 do tipo Fila);
/* recebe F1 e F2, e cria F3, com os elementos de F1 seguidos dos elementos de F2. Ao final da Operação, F1 e F2 mantêm
seus elementos originais, na mesma ordem */
```

Conseguiu propor uma solução? Muito bom se conseguiu! Se não conseguiu, consulte a solução na [Figura 3.5](#) e fique atento ao modo como o exercício foi resolvido. A lógica do algoritmo é a seguinte: retiramos um a um todos os elementos da Fila F1. Cada elemento retirado de F1 é inserido em F3 e também em uma Fila Auxiliar. Depois os elementos retornam da Fila Auxiliar para F1. Fazemos o mesmo para F2 e, então, temos F1 e F2 em seu estado inicial (mesmos elementos, na mesma ordem), e F3 com os elementos de F1 seguidos dos elementos de F2.

```

JunçãoDosElementos (parâmetro por referência F1, F2, F3 do tipo Fila) {
    /* recebe F1 e F2, e cria F3 com os elementos de F1 seguidos dos elementos de F2. Ao
    final da Operação, F1 e F2 mantêm seus elementos originais, na mesma ordem */
    Variável ElementoDaFila do tipo Char;
    Variável FilaAuxiliar do tipo Fila;

    Cria (FilaAuxiliar);
    Cria (F3);

    /* Passando os elementos de F1 para F3 e também para FilaAuxiliar */
    Enquanto (Vazia(F1) == Falso) Faça { // enquanto F1 não for vazia...
        Retira(F1, X, DeuCerto);           // retira elemento de F1 - retorna valor em X
        Insere(F3, X, DeuCerto);          // insere X em F3
        Insere(FilaAuxiliar, X, DeuCerto); // insere X na FilaAuxiliar

    /* Retornando os elementos da FilaAuxiliar para F1 */
    Enquanto (Vazia(FilaAuxiliar) == Falso) Faça { // enquanto FilaAuxiliar tiver elemento
        Retira(FilaAuxiliar, X, DeuCerto);           // retira X da FilaAuxiliar
        Insere(F1, X, DeuCerto);                     // insere X em F1

    /* Passando os elementos de F2 para F3 e também para FilaAuxiliar */
    Enquanto (Vazia(F2) == Falso) Faça { // enquanto F2 não for vazia
        Retira(F2, X, DeuCerto);           // retira elemento de F2 - retorna valor em X
        Insere(F3, X, DeuCerto);          // insere X em F3
        Insere(FilaAuxiliar, X, DeuCerto); // insere X na FilaAuxiliar

    /* Retornando os elementos da FilaAuxiliar para F2 */
    Enquanto (Vazia(FilaAuxiliar) == Falso) Faça { // enquanto FilaAuxiliar tiver elemento
        Retira(FilaAuxiliar, X, DeuCerto);           // retira X da FilaAuxiliar
        Insere(F2, X, DeuCerto);                     // insere X em F2
    } /* fim do procedimento JunçãoDosElementos */
}

```

**FIGURA 3.5** Junção dos elementos de duas Filas.

Note que ainda não temos a menor ideia de como as Operações Primitivas do TAD Fila (Insere, Retira, Vazia, Cheia e Cria) são efetivamente implementadas. Consideramos as Filas como caixas-pretas e manipulamos os valores armazenados apenas através de seus Operadores (ou “botões da televisão”). É assim que manipulamos estruturas de armazenamento de modo abstrato. É assim que utilizamos os Tipos Abstratos de Dados.

Proponha uma solução para o [Exercício 3.2](#) com a mesma estratégia: manipulando os valores armazenados exclusivamente através dos Operadores do TAD Fila descritos na [Figura 3.4](#).

## Exercício 3.2 Troca dos elementos entre duas Filas

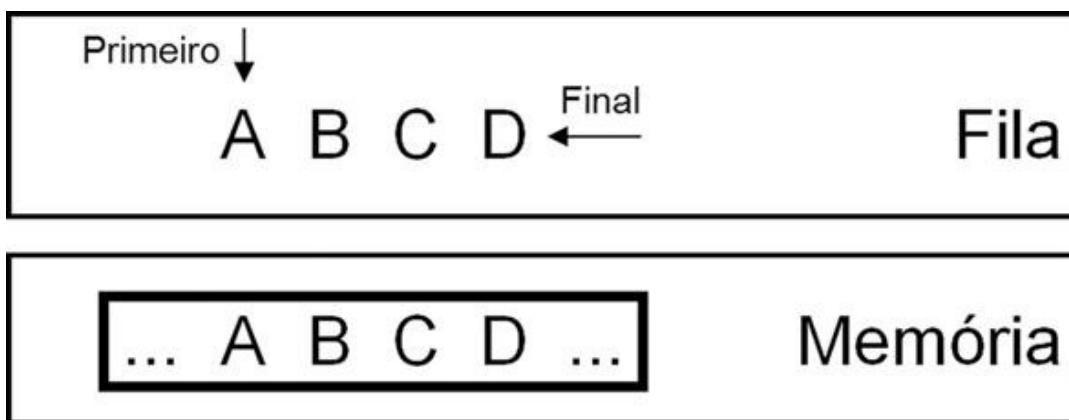
Desenvolva um algoritmo para trocar os elementos de duas Filas F1 e F2. Ao final da operação, a Fila F2 deve conter os elementos que estavam em F1 antes do início da operação, na mesma ordem, e a Fila F1 deve conter os elementos que estavam em F2 antes do início da operação, também na mesma ordem. Considere que F1 e F2 já existem e são passadas como parâmetros. Para propor a solução, utilize os Operadores da [Figura 3.4](#).

```
TrocaDosElementos (parâmetros por referência F1, F2 do tipo Fila);  
/* recebe F1 e F2, e troca os elementos: ao final da operação, F1 deverá ter os elementos que estavam em F2 antes do início  
da operação, na mesma ordem; e F2 deverá ter os elementos que estavam em F1 antes do início da operação, também  
na mesma ordem */
```

### 3.4 Implementando um TAD Fila com Alocação Sequencial e Estática de Memória

No [Capítulo 2](#) implementamos uma Pilha com Alocação Sequencial e Estática. Como poderíamos implementar uma Fila com a mesma técnica de alocação?

Conforme estudamos no [Capítulo 2](#), na Alocação Sequencial os elementos do conjunto são armazenados juntos, em posições de memória adjacentes. A ordem dos elementos no conjunto é refletida na sequência em que são armazenados na memória ([Figura 3.6](#)).



**FIGURA 3.6** Fila com Alocação Sequencial — armazenamento em posições de memória adjacentes.

Na Alocação Estática de Memória, definimos previamente o número máximo de elementos que poderão fazer parte do conjunto e então reservamos espaço

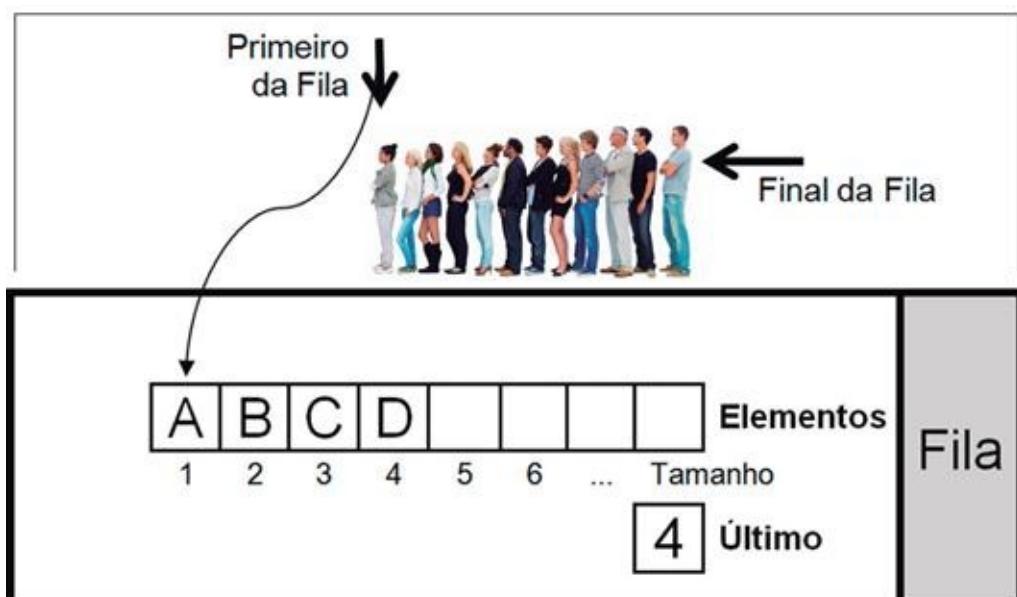
para todos eles.

### Exercício 3.3 Esquema da implementação de uma Fila

Revise a Figura 2.11, que apresenta o esquema da implementação de uma Pilha com Alocação Sequencial e Estática. Em seguida, faça um esquema da implementação de uma Fila com Alocação Sequencial e Estática de Memória. Faça um desenho da estrutura adotada e descreva o funcionamento das Operações Insere, Retira, Cria, Vazia e Cheia. Não prossiga a leitura antes de refletir e propor uma solução.

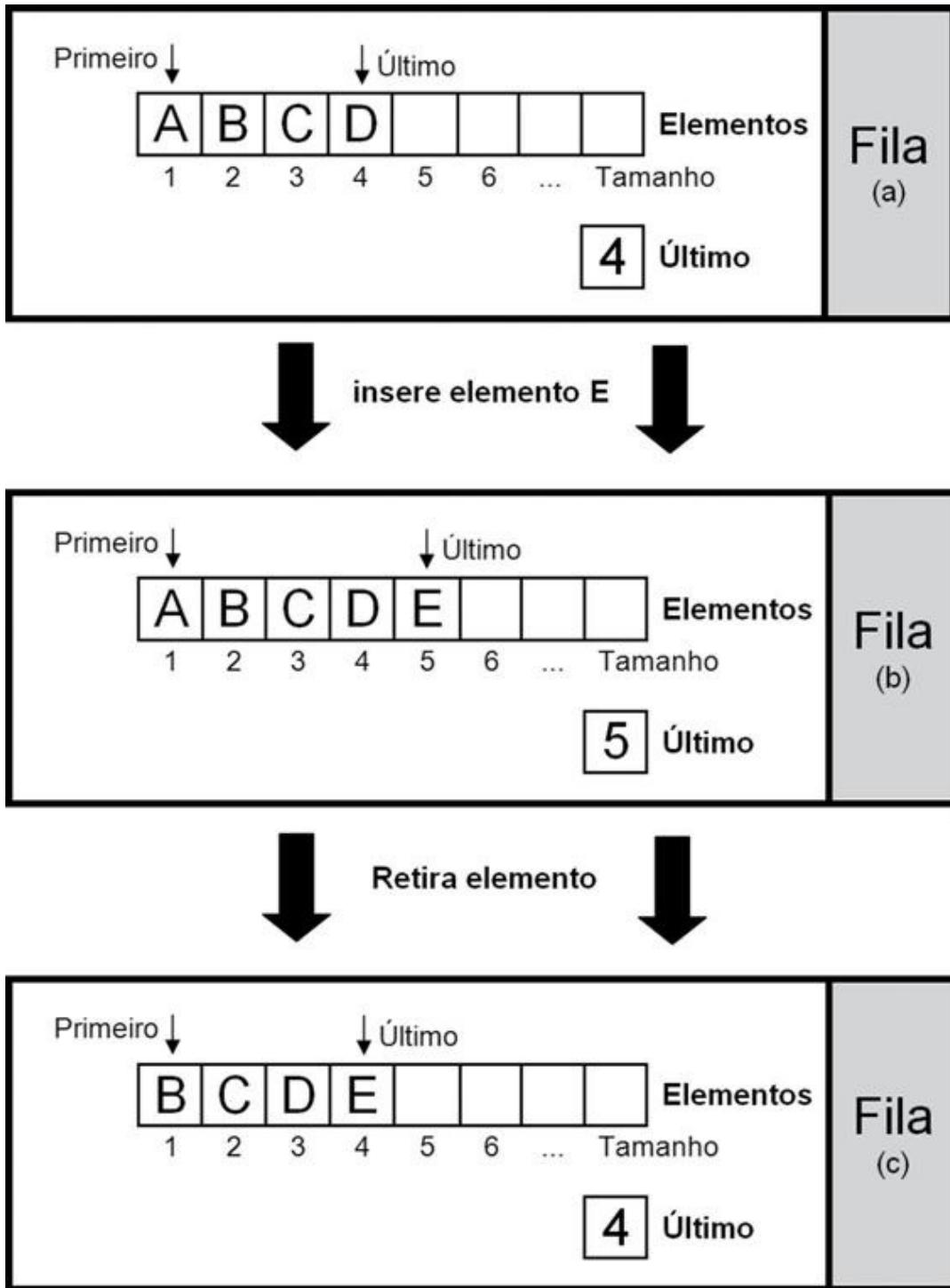
#### 3.4.1 Primeira solução: Fila com realocação dos elementos

A Figura 3.7 apresenta o esquema de uma primeira implementação de Fila através de um vetor denominado Elementos e de uma variável denominada Último. A variável Último indica a posição do Último elemento da Fila no vetor. Nessa primeira implementação de Fila, o Primeiro elemento ficará sempre na primeira posição do vetor. Estamos considerando o início do vetor Elementos em 1. Mas o início do vetor poderá ser ajustado para zero caso este for o padrão na linguagem de programação escolhida.



**FIGURA 3.7** Esquema da implementação de uma Fila com Alocação Sequencial e Estática, e realocação de elementos.

Se quisermos inserir um elemento na Fila, colocaremos esse novo elemento no Final da Fila, ou seja, na posição  $\text{Último} + 1$ . Em seguida, o valor da variável  $\text{Último}$  precisa ser incrementado. A inserção de um elemento leva à situação da [Figura 3.8a](#) à situação da [Figura 3.8b](#).



Retirar um elemento de uma Fila implica, necessariamente, em retirar o Primeiro elemento. No exemplo da Figura 3.8, inicialmente o Primeiro da Fila é o elemento de valor A (Figuras 3.8a e 3.8b). Se retirarmos A, o primeiro da Fila

passará a ser o B. É assim que funciona uma Fila: atendimento por ordem de chegada — o primeiro que entra será sempre o primeiro a ser atendido e a sair da Fila.

Nessa primeira implementação de Fila estamos mantendo o primeiro elemento da Fila sempre na posição 1 do vetor. Assim, ao retirar A, teremos que mover o elemento de valor B para a posição 1. Analogamente, teremos que realocar cada um dos demais elementos da Fila para uma posição anterior à sua posição. Veja na [Figura 3.8c](#) como ficaria a Fila após a retirada do elemento de valor A. Note que o valor da variável `Último` precisou ser decrementado e que os elementos B, C, D e E tiveram que ser realocados devido à retirada de A. Por isso dizemos que essa primeira implementação de Fila é realizada com *realocação de elementos*.

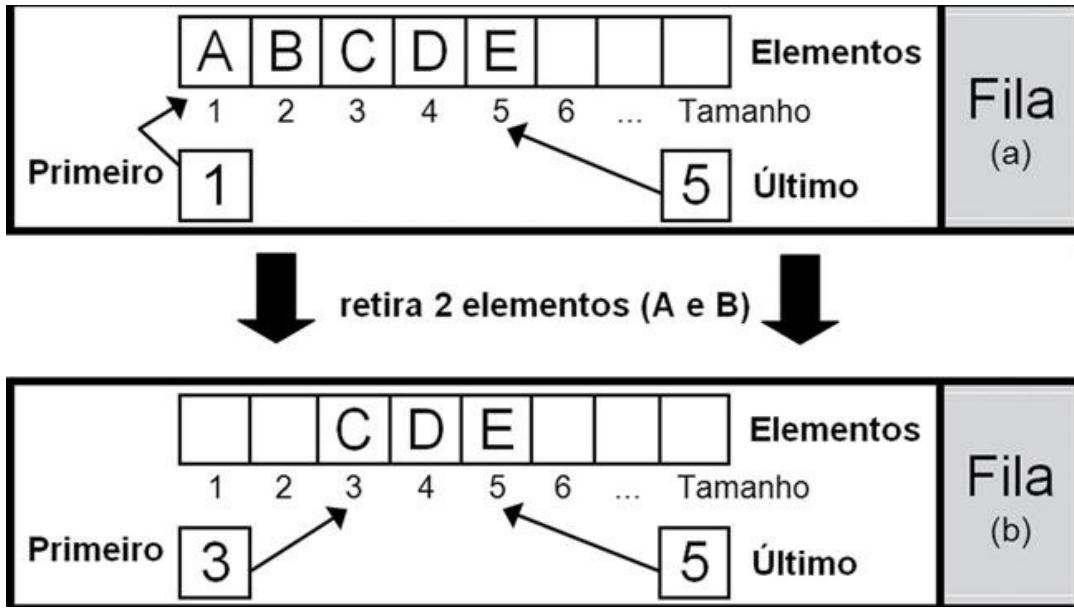
### Exercício 3.4 TAD Fila com realocação de elementos

Implemente o TAD Fila com realocação de elementos, segundo ilustrado pelas [Figuras 3.7](#) e [3.8](#). Implemente as operações Insere, Retira, Cria, Vazia e Cheia, conforme definidas na [Figura 3.4](#). Ao inserir, verifique se a Fila não está cheia; ao retirar, verifique se a Fila não está vazia.

### 3.4.2 Segunda solução: Fila sem realocação dos elementos

Para exercitar nossas habilidades, queremos desenvolver uma segunda implementação de Fila, também com Alocação Sequencial e Estática de Memória, mas sem realocação de elementos. Ou seja, queremos retirar um elemento da Fila, mas não queremos mover os demais elementos, como era necessário fazer em nossa primeira solução.

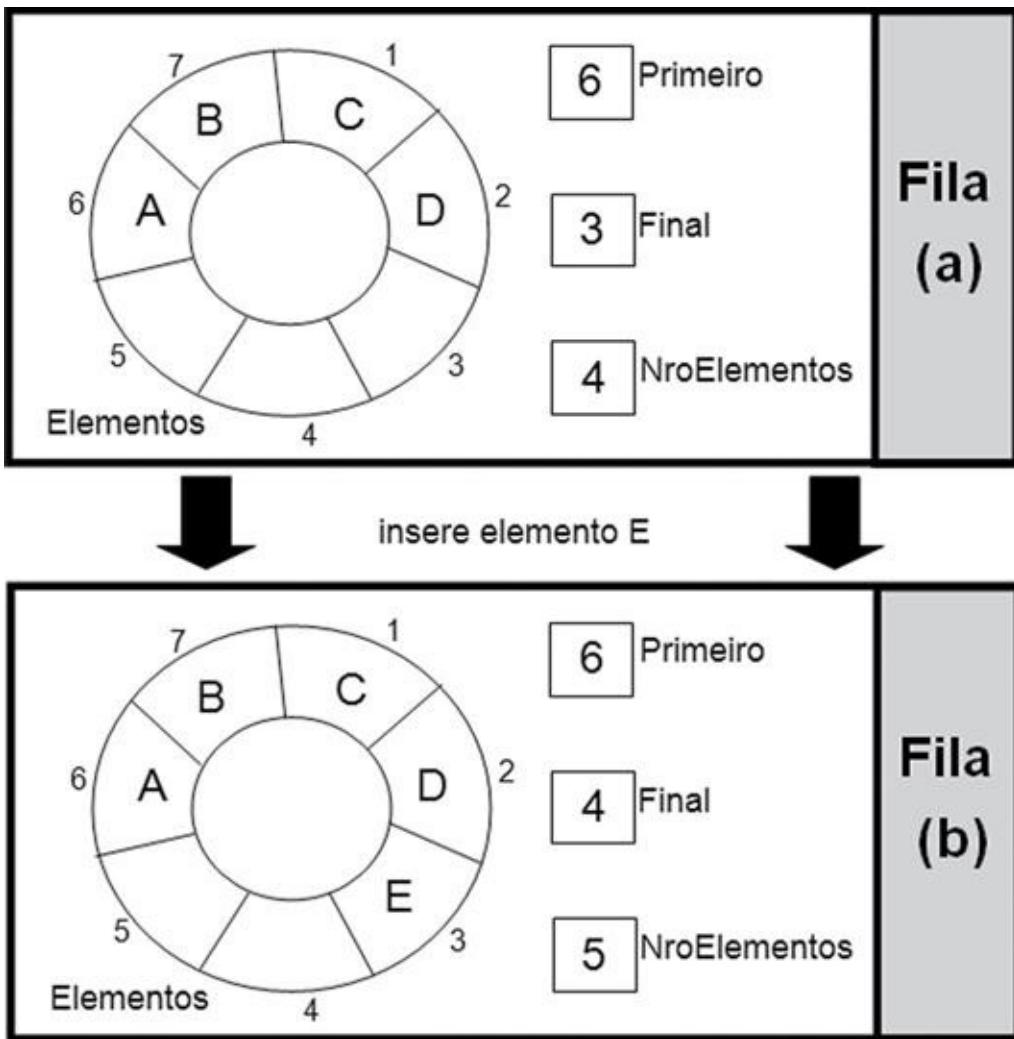
A realocação dos elementos era necessária porque fixamos o primeiro elemento da Fila na posição 1 do vetor. Se permitirmos que o primeiro elemento da Fila seja armazenado em qualquer posição do vetor, não precisaremos realocar os demais elementos a cada operação de retirada. A [Figura 3.9](#) mostra uma Fila que inicialmente contava com os elementos A, B, C, D e E. Após a retirada de A e B, os elementos C, D e E não foram realocados. Note que o primeiro elemento da Fila não está mais armazenado na posição 1 do vetor, mas na posição 3, conforme indicado pela variável `Primeiro`.



**FIGURA 3.9** Operação Retira, sem realocação de elementos.

Considerando que não temos a realocação de elementos, como podemos usar mais de uma vez os espaços do vetor Elementos? Por exemplo, na situação final da [Figura 3.9](#), as posições 1 e 2 já foram utilizadas para armazenar os elementos A e B, mas no momento estão vazias. Como poderemos utilizá-las novamente?

Que tal imaginar que o nosso vetor Elementos é circular? A [Figura 3.10](#) apresenta um esquema da implementação de uma Fila através de um vetor circular. Na prática, é o mesmo vetor que conhecemos, mas desenhado de modo mais conveniente para nossos objetivos do momento.



**FIGURA 3.10** Fila em vetor circular: Operação Insere.

Nessa implementação, a Fila é composta pelo vetor Elementos e por outras três variáveis: Primeiro, Final e NroElementos. A variável Primeiro indica a posição do primeiro elemento da Fila no vetor. No exemplo da [Figura 3.10](#), Primeiro tem o valor 6, indicando que o primeiro elemento da Fila está armazenado na posição 6 do vetor.

A variável Final indica o final da Fila. Note que Final não indica o último elemento da Fila, mas a primeira posição após o último. No exemplo da [Figura 3.10a](#), o último elemento da Fila está na posição 2 do vetor e a variável Final tem o valor 3, ou seja, uma posição após o último.

A variável NroElementos indica o número total de elementos na Fila em determinado momento. Na situação da [Figura 3.10a](#), NroElementos tem valor 4, indicando que, naquele momento, a Fila possui quatro elementos: A, B, C e D. Armazenar o número total de elementos na Fila nos ajudará a identificar a

situação de Fila Cheia e a situação de Fila Vazia. Essa solução foi adotada por [Celes, Cerqueira e Rangel \(2004\)](#) e também por [Pereira \(1996\)](#).

## Inserindo elementos na Fila

Para inserir um novo elemento na Fila esquematizada na [Figura 3.10a](#), basta atribuir o valor do novo elemento à posição 3 do vetor Elementos. Essa posição do vetor é indicada pela variável Final. Após isso é preciso atualizar o valor da variável Final e também o valor da variável NroElementos. Se partirmos da situação ilustrada na [Figura 3.10a](#), a inserção do elemento E resultará na situação ilustrada na [Figura 3.10b](#). A variável NroElementos passou a ter o valor 5, indicando que a Fila agora possui cinco elementos. A variável Final passou a ter o valor 4, indicando que, se quisermos inserir outro elemento, deveremos inseri-lo na posição 4.

## Exercício 3.5 Operação Insere na Fila

Conforme especificado na [Figura 3.4](#), a operação Insere recebe como parâmetros a Fila na qual queremos inserir um elemento e o valor do elemento que queremos inserir. O elemento só não será inserido se a Fila já estiver cheia. Você encontrará a seguir uma possível solução para este exercício, mas tente propor uma solução antes de consultar a resposta.

```
Insere (parâmetro por referência F do tipo Fila, parâmetro X do tipo Char, parâmetro por referência DeuCerto do tipo Boolean);
/* Insere o elemento X na Fila F. O parâmetro DeuCerto deve indicar se a operação foi bem-sucedida ou não. A operação
só não será bem-sucedida se tentarmos inserir um elemento em uma Fila cheia */
```

A [Figura 3.11](#) apresenta um algoritmo conceitual para a Operação Insere. Se a Fila F estiver cheia, sinalizamos através do parâmetro DeuCerto que o elemento X não foi inserido. Mas, se a Fila F não estiver cheia, X será inserido. Para isso, incrementamos o número de elementos ( $\text{NroElementos} = \text{NroElementos} + 1$ ) e armazenamos X no vetor Elementos, na posição indicada pela variável Final ( $\text{F.Elementos}[ \text{F.Final} ] = \text{X}$ ). Ou seja, o elemento X entra no Final da Fila F.

```

Insere (parâmetro por referência F do tipo Fila, parâmetro X do tipo Char, parâmetro por
referência DeuCerto do tipo Boolean) {
    /* Insere o elemento X na Fila F. O parâmetro DeuCerto deve indicar se a operação foi
bem-sucedida ou não. A operação só não será bem-sucedida se tentarmos inserir um
elemento em uma Fila cheia */

    Se (Cheia(F)== Verdadeiro)           // se a Fila F estiver cheia...
    Então  DeuCerto = Falso;             // ... não podemos inserir

    Senão { DeuCerto = Verdadeiro;      // inserindo X na Fila F... operação deu certo..
            F.NroElementos = F. NroElementos + 1; // aumenta o número de elementos
            F.Elementos[ F.Final ] = X;          // insere X no final da Fila F
            // avançando o apontador Final... Atenção: o vetor é circular
            Se (F.Final == Tamanho) // se Final estiver na última posição do vetor..
            Então F.Final = 1;        // ..avança para a primeira posição (ajustar para 0?)
            Senão F.Final = F.Final + 1; // senão avança para a próxima posição
        }; // fim do senão
    } // fim da operação Insere
}

```

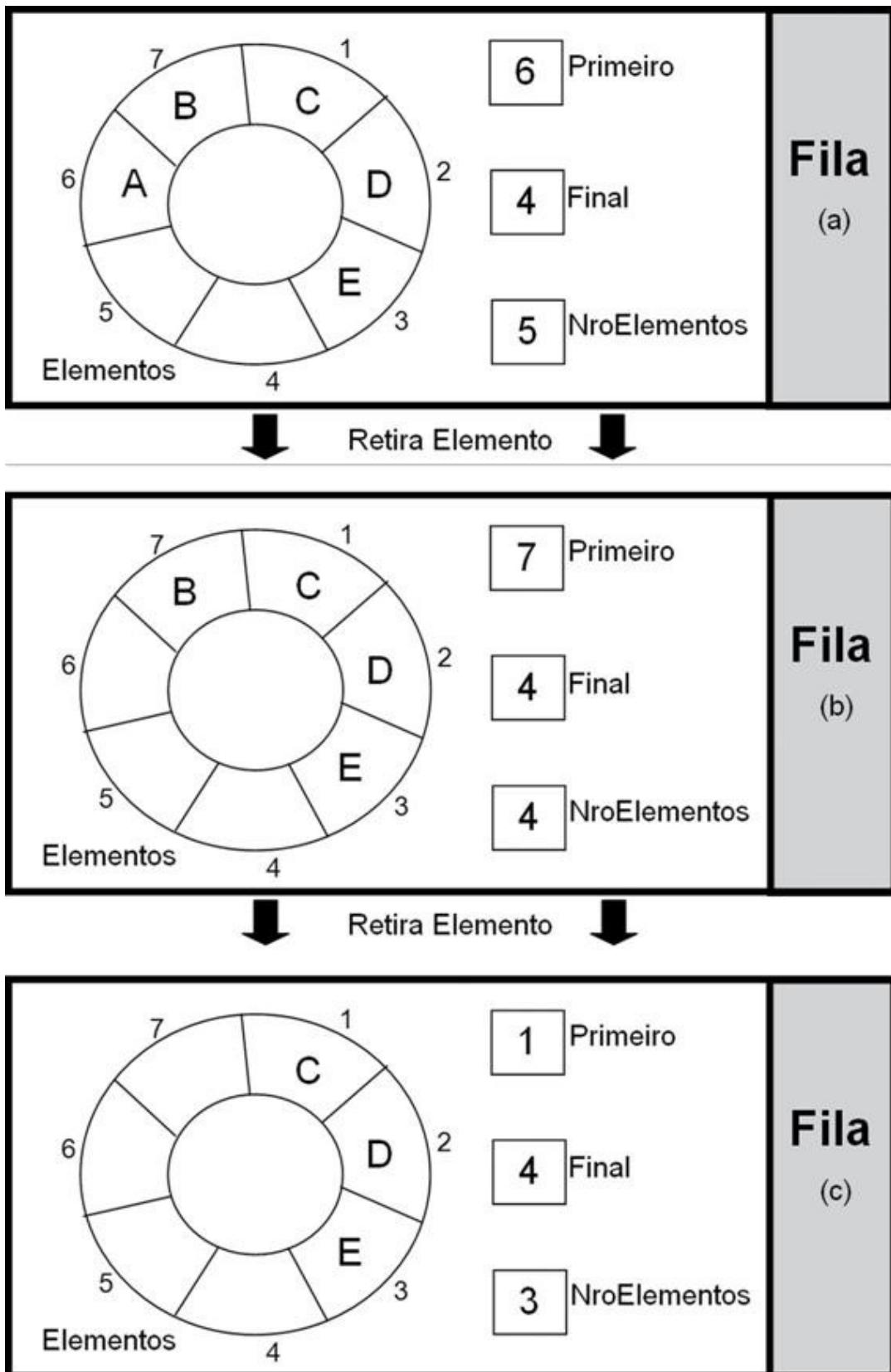
**FIGURA 3.11** Insere em uma Fila — vetor circular.

Após inserir X na posição indicada pela variável Final, precisamos atualizar o valor da variável Final. Se a variável Final não estiver apontando para a última posição do vetor, passará a apontar para a posição seguinte à sua posição atual. É o que acontece no exemplo da [Figura 3.10](#). Como estamos tratando o vetor como circular, se a variável Final estivesse apontando para a última posição do vetor deveria passar a apontar para a primeira posição do vetor. Na [Figura 3.10](#), a primeira posição do vetor é 1 e a última posição é 7. Pode ser conveniente ajustar esses valores para 0 e 6, respectivamente, para compatibilização com o padrão da linguagem de programação escolhida.

## Retirando elementos da Fila

A [Figura 3.12](#) exemplifica a operação que retira um elemento da Fila. A partir da situação inicial ([Figura 3.12a](#)), retiramos um elemento — o elemento de valor A — e chegamos à situação retratada na [Figura 3.12b](#). Ao retirar A, diminuímos o número de elementos da Fila de cinco para quatro e avançamos o apontador Primeiro da posição 6 para a posição 7. Se retirarmos mais um elemento — agora o elemento de valor B —, chegaremos à situação da [Figura 3.12c](#). O número de elementos passou de quatro para três, e o apontador Primeiro avançou de 7 para 1. Estamos tratando o vetor Elementos como um vetor circular. Se o apontador Primeiro ou o apontador Final estiver na última posição do vetor,

“avançar” significa retornar à primeira posição.



**FIGURA 3.12** Avançando os apontadores Primeiro e Final em um

**vetor circular: após a Última posição do vetor, retorna para a Primeira posição.**

## Exercício 3.6 Operação Retira da Fila

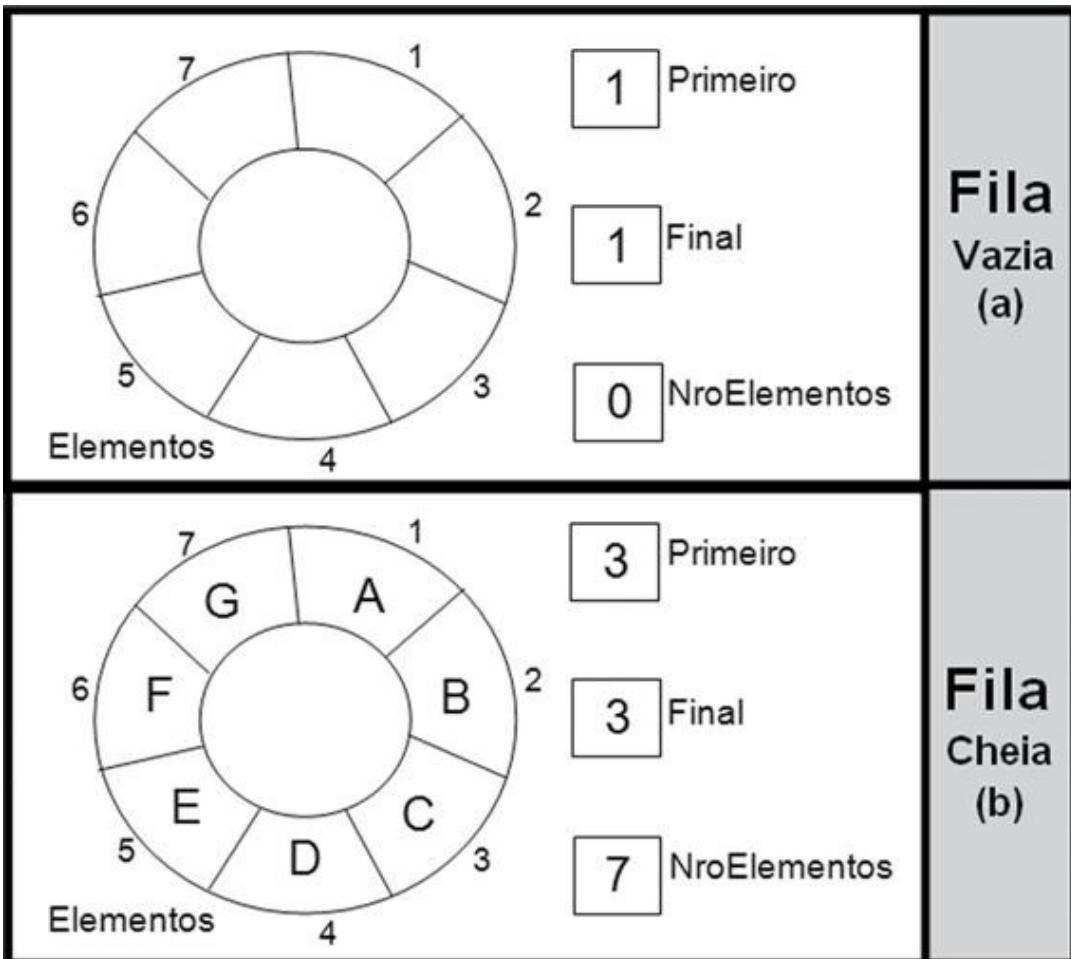
Conforme especificado na [Figura 3.4](#), a operação Retira recebe como parâmetro a Fila da qual queremos retirar um elemento. Caso a Fila não estiver vazia, a operação retorna o valor do elemento retirado. O elemento a ser retirado deve ser sempre o primeiro da Fila.

Retira (parâmetro por referência F do tipo Fila, parâmetro por referência X do tipo Char, parâmetro por referência DeuCerto do tipo Boolean);

/\* Caso a Fila F não estiver vazia, retira o primeiro elemento da Fila e retorna o seu valor no parâmetro X. Se a Fila F estiver vazia, o parâmetro DeuCerto retorna Falso \*/

## Operações Cria, Vazia e Cheia

A [Figura 3.13](#) ilustra a situação em que a Fila está vazia (2.13a) e a situação em que a Fila está cheia (2.13b). Em uma Fila vazia, o número de elementos é zero, e em uma Fila cheia, o número de elementos é igual ao Tamanho da Fila ou, ainda, igual ao tamanho do vetor. Na [Figura 3.13b](#), o tamanho do vetor é 7, e o número de elementos também é sete. Logo, a Fila está cheia.



**FIGURA 3.13** Fila Vazia e Fila Cheia em um vetor circular.

Conforme já mencionamos, armazenar o número total de elementos na Fila nos ajuda a identificar a situação de Fila Cheia e a situação de Fila Vazia. Essa solução foi adotada por [Celes, Cerqueira e Rangel \(2004\)](#) e por [Pereira \(1996\)](#).

### Exercício 3.7 Operação Cria a Fila

Conforme especificado na [Figura 3.4](#), criar a Fila significa inicializar os valores de modo a indicar que a Fila está vazia.

Cria (parâmetro por referência F do tipo Fila);

/\* Cria a Fila F, inicializando a Fila como vazia - sem nenhum elemento. \*/

### Exercício 3.8 Operação para testar se a Fila está vazia

Conforme especificado na [Figura 3.4](#), a operação Vazia testa se a Fila passada como parâmetro está vazia (sem elementos), retornando o valor Verdadeiro (Fila vazia) ou Falso (Fila não vazia).

```
Boolean Vazia (parâmetro por referência F do tipo Fila);  
/* Retorna Verdadeiro se a Fila F estiver vazia - sem nenhum elemento; retorna Falso caso contrário */
```

### **Exercício 3.9 Operação para testar se a Fila está cheia**

Conforme especificado na [Figura 3.4](#), a operação Cheia testa se a Fila passada como parâmetro está cheia. A Fila estará cheia se na estrutura de armazenamento não couber mais nenhum elemento.

```
Boolean Cheia (parâmetro por referência F do tipo Fila);  
/* Retorna Verdadeiro se a Fila F estiver cheia, ou seja, se na estrutura de armazenamento não couber mais nenhum elemento;  
Retorna Falso caso contrário */
```

### **Exercício 3.10 Fila em vetor circular sem o total de elementos**

Nos exercícios anteriores, implementamos uma Fila com um vetor circular e utilizamos a variável Total para indicar o total de elementos. Seria possível implementar uma Fila com vetor circular sem armazenar o total de elementos? Como você faria para diferenciar a situação de Fila Vazia e a situação de Fila Cheia?

### **Exercício 3.11 Número de elementos de uma Fila**

Desenvolva um algoritmo para calcular o número de elementos de uma Fila F. Considere que a Fila F já existe, ou seja, não precisa ser criada. Você pode criar Filas auxiliares, se necessário. Ao final da execução dessa Operação, F precisa estar exatamente como estava no início da Operação — os mesmos elementos, na mesma ordem. Considere que a Fila F possui elementos do tipo Char.

```
Int NúmeroDeElementos(parâmetro por referência F do tipo Fila);  
/* retorna o número de elementos da uma Fila F, passada como parâmetro */
```

## 3.5 Botão da televisão ou chave de fenda?

Ao propor uma solução para o Exercício 3.11, você usou o botão de volume ou uma chave de fenda?

A Figura 3.14 apresenta duas possíveis soluções para o Exercício 3.11. A primeira solução é bem simples, mas é dependente da implementação: só funciona na implementação da Fila com vetor circular, que fizemos na Seção 3.4.2. Não funciona, por exemplo, na implementação de Fila com realocação de elementos que fizemos na Seção 3.4.1, pois nessa implementação não temos a variável que armazena o total de elementos.

```
Int NúmeroDeElementos (parâmetro por referência F do tipo Fila) {  
    /* retorna o número de elementos de uma Fila F, passada como parâmetro */  
  
    /* solução 1: dependente da implementação - abre a televisão para aumentar o volume */  
  
    Retorne F.Total           // F.Total armazena o total de elementos da Fila  
  
    /* solução 2: independente da implementação - aumenta o volume pelo botão de volume*/  
  
    Variável X do tipo Char; // elemento da Fila  
    Variável Contador do tipo Int;  
    Variável FilaAuxiliar do tipo Fila;  
  
    Contador = 0;  
    Cria (FilaAuxiliar);  
  
    /* Contando... retira um a um os elementos de F, atualizando o Contador */  
    Enquanto (Vazia(F) == Falso) Faça { // enquanto F não for vazia  
        Retira(F, X, DeuCerto);          // retira X de F  
        Contador = Contador + 1;         // incrementa o Contador  
        Insere(FilaAuxiliar, X, DeuCerto); }; // insere X na FilaAuxiliar  
  
    /* Retornando os elementos da FilaAuxiliar para F */  
    Enquanto (Vazia(FilaAuxiliar) == Falso) Faça { // enquanto FilaAuxiliar tiver elemento  
        Retira(FilaAuxiliar, X, DeuCerto);          // retira X da FilaAuxiliar  
        Insere(F, X, DeuCerto); }; // insere X em F  
  
    /* Dando o resultado */  
    Retorne Contador;  
}  
/* fim do procedimento NúmeroDeElementos */
```

**FIGURA 3.14** Número de elementos de uma Fila: duas soluções.

A segunda solução é independente da implementação, pois manipula a Fila exclusivamente através de seus Operadores Primitivos, definidos na [Figura 3.4](#). Essa segunda solução funciona tanto na implementação de Fila com realocação de elementos quanto na implementação de Fila com vetor circular. Por ser independente da implementação, essa segunda solução proporciona maior portabilidade e reusabilidade de código.

O [Exercício 3.11](#) é um exemplo cruel, pois a solução dependente da implementação é realmente muito simples e óbvia, mas abre a televisão com uma chave de fenda para aumentar o volume e, por isso, a portabilidade e o potencial para reuso de software são menores.

Botão do volume ou chave de fenda? Reflita e escolha a melhor estratégia caso a caso, ao propor soluções para os exercícios seguintes.

No final deste capítulo são apresentadas respostas ou sugestões para alguns dos exercícios propostos, mas para desenvolver as habilidades pretendidas, proponha suas próprias soluções antes de consultar as respostas e sugestões.

## **Exercício 3.12 Mesmo número de elementos?**

Desenvolva um algoritmo para testar se uma Fila F1 possui o mesmo número de elementos que uma Fila F2. Considere que as Filas F1 e F2 já existem e são passadas como parâmetro. Considere que as Filas F1 e F2 possuem elementos do tipo Char.

```
Boolean MesmoNúmeroDeElementos (parâmetros por referência F1, F2 do tipo Fila);
/* retorna Verdadeiro se as Filas F1 e F2 tiverem o mesmo número de elementos */
```

## **Exercício 3.13 As Filas são iguais?**

Desenvolva um algoritmo para testar se duas Filas F1 e F2 são iguais. Duas Filas são iguais se possuem os mesmos elementos, exatamente na mesma ordem. Você pode utilizar Filas auxiliares, caso necessário. Considere que as Filas F1 e F2 já existem e são passadas como parâmetro. Considere que as Filas F1 e F2 possuem elementos do tipo Char.

```
Boolean Iguals (parâmetros por referência F1, F2 do tipo Fila);  
/* retorna Verdadeiro se a Fila F1 for igual à Fila F2, ou seja, se as Filas possuírem exatamente os mesmos elementos, na  
mesma ordem. Se ambas as Filas forem vazias, devem ser consideradas iguais */
```

## Exercício 3.14 Implementar e testar uma Fila

Implemente uma Fila em uma linguagem de programação como C ou C++. A Fila pode ser definida como uma Classe ou como um Tipo Estruturado. Os elementos da Fila devem ser do tipo Char. Implemente a Fila como um Tipo Abstrato de Dados. Em um arquivo separado, faça o programa principal bem simples, para testar o funcionamento das Operações Primitivas da Fila.

## Exercício 3.15 Identificar outras aplicações de Filas

Identifique e liste aplicações de Filas. Identifique e anote alguns jogos que podem ser implementados com o uso de uma Fila e também outras aplicações. Sugestão de uso acadêmico: faça uma discussão em grupo. Ao final, cada grupo apresenta a todos os estudantes as aplicações que identificou. Pode ser uma boa fonte de ideias para definir bons jogos que também sejam aplicações de Filas.

## 3.6 Projeto *Snake*: Cobra como uma Fila de cores

Na versão do *Snake* descrita no Desafio 2, ao comer um quadradinho colorido de cor diferente da cor da Cabeça da Cobra, o jogador marca pontos, e o quadradinho comido é inserido no final da Cobra. Se a Cobra come um quadradinho colorido da mesma cor de sua Cabeça, o quadradinho colorido que representa a Cabeça da Cobra é retirado. A Cobra funciona como uma Fila de quadradinhos coloridos; uma Fila de cores (veja novamente a [Figura 3.3](#)).

Agora que você já sabe como usar uma Fila e já sabe também como implementar uma Fila, pode dar prosseguimento ao seu Projeto *Snake*. Adapte as regras do jogo, mas observe as seguintes recomendações na implementação do seu *Snake*:

- Armazene as cores que compõem a Cobra em um TAD Fila.
  - A aplicação e o TAD Fila devem estar em arquivos separados.
  - A aplicação (e outros módulos) deve manipular o TAD Fila exclusivamente através dos Operadores Primitivos: Empilha, Desempilha, Vazia etc.
- Aumente o volume da televisão apenas pelo botão de volume.

- Inclua no código do TAD Fila somente ações pertinentes ao armazenamento e recuperação das informações sobre a Cobra (quadrinhos coloridos que a compõem); faça o possível para deixar o TAD Fila o mais independente possível dos demais módulos; na medida do possível, não inclua no TAD Fila ações referentes à interface, por exemplo.
- Procure fazer a interface do modo mais independente possível dos demais componentes de software.

### **Exercício 3.16 Avançar o Projeto Snake — propor uma arquitetura de software**

Proponha uma arquitetura de software para o seu projeto Snake, identificando os principais módulos do sistema e o relacionamento entre eles. Reveja a [Figura 2.18](#), que trata da arquitetura do jogo *FreeCell*, e adapte conforme necessário para o seu novo projeto. Sugestão para uso acadêmico: desenvolva o projeto em grupo. Promova uma discussão entre o grupo, defina a arquitetura do software e uma divisão de trabalho entre os componentes do grupo.

### **Exercício 3.17 Avançar o Projeto Snake — definindo o tipo do Elemento da Cobra e o tipo de Comida da Cobra**

Defina como você representará o Elemento da Cobra. Ele armazena, em essência, uma cor. Defina um conjunto de cores que você gostaria de utilizar (amarelo, vermelho, azul, verde, cinza etc.) e um modo de representar essas cores em seu programa. Lembre-se de que a Comida da Cobra será do mesmo tipo do Elemento da Cobra. Em algum momento, você terá que comparar a cor da Comida da Cobra com a cor da Cabeça da Cobra, ou seja, a cor do primeiro elemento da Fila que representa a Cobra. Não se preocupe com a interface em si ao definir o tipo do Elemento da Cobra. Procure implementar a interface como um módulo independente, o tanto quanto possível.

### **Exercício 3.18 Avançar o Projeto Snake — implemente uma Fila de Elementos Coloridos**

Defina e implemente uma Fila de Elementos Coloridos. Os elementos da Fila devem ser do tipo definido no [Exercício 3.18](#). Faça um programa para testar sua Fila sem se preocupar com a qualidade da interface neste momento.

### **Exercício 3.19 Avançar o Projeto Snake — defina as regras,**

## **escolha um nome e inicie o desenvolvimento do seu *Snake***

Defina as regras da sua variação do *Snake*. Altere as regras em algum aspecto. Dê personalidade própria ao seu *Snake*; escolha para o seu jogo um nome representativo e interessante. Escreva as regras da sua versão do *Snake* e coloque em um arquivo-texto que possa ser utilizado na documentação do jogo ou em uma opção de Ajuda/Como Jogar. Sugestão para uso acadêmico: desenvolva o Projeto *Snake* em grupo. Tome as principais decisões em conjunto e divida o trabalho entre os componentes do grupo, cada qual ficando responsável por parte das atividades.

## **Exercício 3.20 Avançar o Projeto *Snake* — projeto da**

### **interface**

Avance um pouco mais no estudo da biblioteca gráfica que você escolheu para implementar uma interface visual e intuitiva para sua versão do *Snake*. Você pode iniciar seu estudo pelo Tutorial de Programação Gráfica disponível nos Materiais Complementares. Fique livre para estudar outros materiais e para adotar outras ferramentas gráficas para o seu projeto. Projete a interface do modo mais independente possível dos demais módulos. Se estiver trabalhando em grupo, um dos membros pode se responsabilizar por estudar e desenvolver os aspectos referentes à interface e depois ajudar os demais componentes nesse aprendizado. Você pode optar também por implementar primeiramente uma interface bem simples, textual, para testar os demais componentes do jogo e depois substituir por uma interface mais sofisticada.

### **Comece a desenvolver o seu *Snake* agora!**

Faça um *Snake* legal! Um *Snake* com a sua cara! Distribua para os amigos!  
Aprender a programar pode ser divertido!



### **Consulte nos Materiais Complementares**

Tutorial de Programação Gráfica

Banco de Jogos: Adaptações do *Snake*

Banco de Jogos: Aplicações de Filas



<http://www.elsevier.com.br/edcomjogos>

## Exercícios de fixação

**Exercício 3.21** O que é uma Fila?

**Exercício 3.22** Para que serve uma Fila? Em que tipo de situações uma Fila pode ser utilizada?

**Exercício 3.23** Qual a diferença entre uma Pilha e uma Fila?

**Exercício 3.24** Compare as implementações de Fila com realocação de elementos e em vetor circular. Na sua opinião, qual é mais interessante e por quê?

**Exercício 3.25** Quando você propôs soluções para os [Exercícios 3.1 \(junção de duas Filas\)](#) e [3.2 \(troca dos elementos entre duas Filas\)](#), nem sabia como o TAD Fila seria implementado. Logo, você teve que propor uma solução que manipulasse a Fila exclusivamente por seus Operadores Primitivos. Suponha agora que você conhece a implementação da Fila (vetor circular ou vetor com realocação de elementos). Que estratégia você escolheria para implementar os [Exercícios 3.1](#) e [3.2](#)? Você manteria a estratégia de manipular a Fila exclusivamente por seus Operadores Primitivos? Ou adotaria uma solução dependente da implementação? Justifique sua escolha.

**Exercício 3.26** Na versão do *Snake* descrita no Desafio 2, no decorrer do jogo a Cobra comerá um quadradinho colorido e você precisará saber se esse quadradinho colorido é da cor da cabeça da Cobra ou não. Para isso, precisará saber a cor da cabeça da Cobra. Você tem duas alternativas para implementar uma operação que consulta o valor do primeiro elemento de uma Fila: pelo botão do volume (acionando as Operações Primitivas da Fila) ou abrindo a televisão (solução dependente da implementação). Qual dessas duas alternativas você considera mais interessante para o seu projeto e por quê?

## Soluções para alguns dos exercícios

**Exercício 3.3 Esquema da implementação de uma Fila**

Veja a [Figura 3.7](#) (solução com realocação de elementos – compare com a [Figura 2.11](#) – Pilha); e veja a [Figura 3.10](#) (solução sem realocação de elementos).

## Exercício 3.6 Operação Retira da Fila

```
Retira (parâmetro por referência F do tipo Fila, parâmetro por referência X do tipo Char, parâmetro por referência DeuCerto do tipo Boolean) {  
    /* Caso a Fila F não estiver vazia, retira o primeiro elemento da Fila e retorna o seu valor no parâmetro X. Se a Fila F estiver vazia, o parâmetro DeuCerto retorna Falso */  
    Se (Vazia(F)==Falso)  
        Então DeuCerto = Falso; // se a Fila F estiver vazia, não podemos Retirar  
        Senão { // Retira elemento da Fila F e retorna o valor em X  
            DeuCerto = Verdadeiro; // a operação deu certo..  
            F.NroElementos = F.NroElementos - 1; // diminui o número de elementos  
            X = F.Elementos[ F.Primeiro ] = X; // X recebe valor do Primeiro elemento da Fila F  
  
            // avançando o apontador Primeiro. Atenção: o vetor é circular  
            Se (F.Primeiro == Tamanho) // se Primeiro estiver na última posição do vetor  
                Então F.Primeiro = 1; // primeira posição pode ser 0 - linguagem C  
                Senão F.Primeiro = F.Primeiro + 1; // avança para a próxima posição  
            }  
        } // fim da operação Retira
```

## Exercício 3.7 Operação Cria a Fila

```
Cria (parâmetro por referência F do tipo Fila) {  
    /* Cria a Fila F, inicializando a Fila como vazia - sem nenhum elemento. */  
    F.Primeiro = 1;  
    F.Final = 1;  
    F.NroElementos = 0;  
} // fim da operação Cria
```

## Exercício 3.8 Operação para testar se a Fila está vazia

```
Boolean Vazia (parâmetro por referência F do tipo Fila) {  
/* Retorna Verdadeiro se a Fila F estiver vazia - sem nenhum elemento; retorna Falso caso  
contrário */  
Se (F.NroElementos == 0)  
Então Retorne Verdadeiro;  
Senão Retorne Falso;  
} // fim da operação Vazia
```

## Exercício 3.9 Operação para testar se a Fila está cheia

```
Boolean Cheia (parâmetro por referência F do tipo Fila) {  
/* Retorna Verdadeiro se a Fila F estiver Cheia, ou seja, se na estrutura de armazenamento não  
couber mais nenhum elemento; retorna Falso caso contrário */  
Se (F.Total == Tamanho) // ou Tamanho -1, se vetor começar em zero  
Então Retorne Verdadeiro;  
Senão Retorne Falso;  
} // fim da operação Cheia
```

## Exercício 3.12 Mesmo número de elementos?

Sugestão: calcule o número de elementos de cada Fila através da solução ao [Exercício 3.11](#). Depois é só comparar o tamanho das duas filas.

## Exercício 3.13 As Filas são iguais?

Sugestão: veja a solução proposta para o [Exercício 2.4](#), que verifica se duas Pilhas são iguais.

## Referências e leitura adicional

1. Celes W, Cerqueira R, Rangel JL. *Introdução a estruturas de dados*. Rio de Janeiro: Elsevier; 2004; 171- 175.
2. Pereira SL. *Estruturas de dados fundamentais: conceitos e aplicações*. São Paulo: Érica; 1996; 64-66.

---

## CAPÍTULO 4

---

# Listas Encadeadas

---

## Seus objetivos neste capítulo

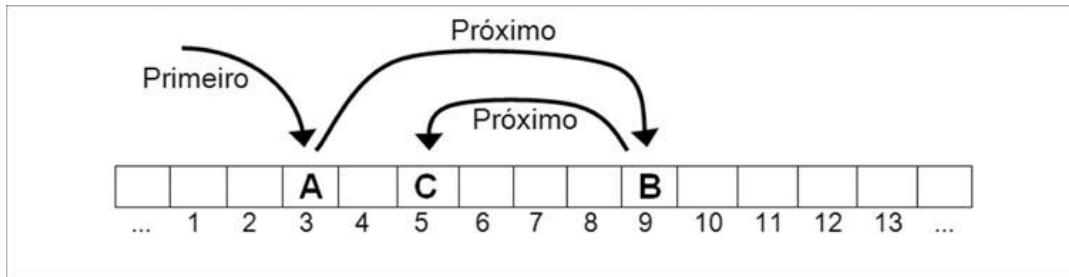
- Entender o que é Alocação Encadeada de Memória, no contexto do armazenamento temporário de conjuntos de elementos; conhecer o conceito de Listas Encadeadas e sua representação usual.
- Desenvolver habilidade para implementar Pilhas e Filas através do conceito de Listas Encadeadas.

### 4.1 Alocação Sequencial *versus* Alocação Encadeada

Na Alocação Sequencial, os elementos de um conjunto são armazenados em posições adjacentes de memória. A ordem dos elementos é definida implicitamente: se o primeiro elemento está na posição 1 do vetor, sabemos que o segundo elemento estará na posição 2 do vetor, o terceiro elemento na posição 3, e assim por diante.

Na Alocação Encadeada, os elementos de um conjunto não são armazenados necessariamente em posições adjacentes de memória. É até possível que o primeiro elemento do conjunto esteja armazenado bem ao lado do segundo elemento, mas também é possível que eles estejam armazenados em posições de memória bem distantes uma da outra. Se não podemos inferir que o segundo elemento do conjunto está armazenado bem ao lado do primeiro, a sequência entre os elementos precisa ser explicitamente indicada: na Alocação Encadeada, cada elemento do conjunto indica onde está armazenado o próximo na sequência.

Na [Figura 4.1](#), o primeiro elemento do conjunto é o elemento A, armazenado na posição 3 da memória. O elemento A indica que o segundo elemento é o B, e está armazenado na posição 9. Por sua vez, B indica que o terceiro elemento do conjunto é o C, armazenado na posição de memória 5.



**FIGURA 4.1** Indicação explícita da sequência dos elementos.

Note que os três elementos do conjunto — A, B e C — não estão armazenados em posições de memória adjacentes. Note também que a sequência entre os elementos é explicitamente indicada: o primeiro elemento indica qual é o segundo, que por sua vez indica qual é o terceiro.

**Definição: Alocação Encadeada de Memória para um conjunto de elementos**

- Os elementos não são armazenados, necessariamente, em posições de memória adjacentes.
- A ordem dos elementos precisa ser explicitamente indicada: cada elemento do conjunto aponta qual é o próximo na sequência.

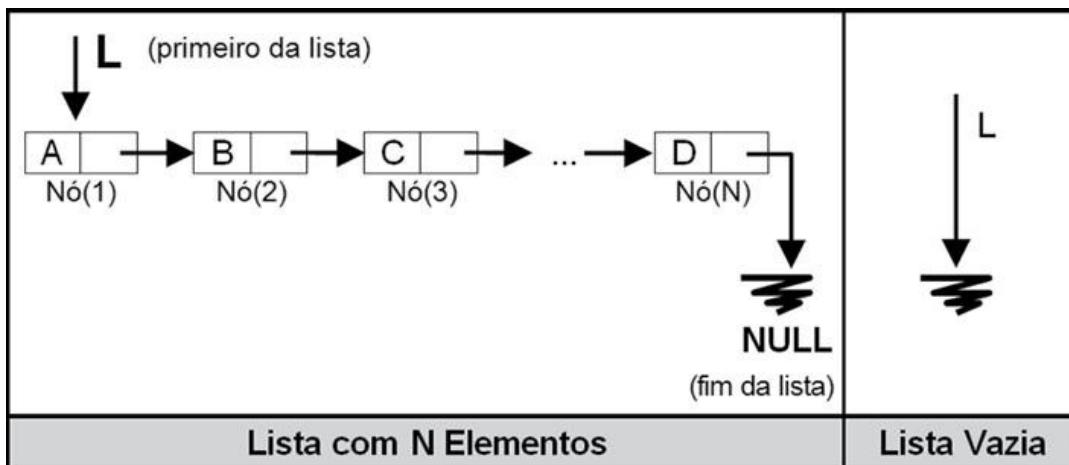
**FIGURA 4.2** Definição de Alocação Encadeada de Memória para um conjunto de elementos.

## 4.2 Listas Encadeadas: conceito e representação

Uma Lista Encadeada L, com N Nós — Nô(1), Nô(2), Nô(3)... Nô(N) —, é definida pelas seguintes características:

- Cada Nô N(i) é um conjunto composto por dois campos: o primeiro campo contém a informação a ser armazenada naquele Nô, e o segundo campo contém a indicação do Próximo elemento da Lista. Na [Figura 4.3](#), a informação armazenada no Nô é representada pelos caracteres A, B, C e D. A indicação do Próximo elemento é representada pelas setas horizontais.
- Os Nós da Lista não estão, necessariamente, em posições adjacentes de memória.

- O acesso aos elementos da Lista ocorre através de um ponteiro para o Primeiro elemento da Lista. Na [Figura 4.3](#), o Primeiro elemento da Lista é indicado pelo ponteiro L.
- O acesso aos demais elementos da Lista ocorre sempre a partir do Primeiro elemento e, a seguir, pela indicação de qual é o Próximo na sequência. No exemplo da [Figura 4.3](#), não é possível ter acesso direto ao elemento da Lista que contém o valor C. Para acessar o elemento que armazena o valor C, é preciso acessar o Primeiro elemento da Lista através do ponteiro L, depois seguir a indicação do Próximo elemento para chegar ao Nô(2) e então ao Nô(3).
- O último Nô da Lista — Nô(N) — aponta para um endereço de memória inválido chamado NULL; isso indica que Nô(N) guarda o último elemento do conjunto.
- Em uma Lista Encadeada vazia — sem elementos, o ponteiro para o Primeiro da Lista aponta para o endereço NULL.



**FIGURA 4.3** Representação usual de uma Lista Encadeada.

## 4.2.1 Notação para manipulação de Listas Encadeadas

A [Figura 4.4](#) apresenta uma notação conceitual para manipulação de Listas Encadeadas. Primeiramente são definidos os tipos `Node` e `NodePtr`, e declaradas as variáveis `L`, `P`, `P1` e `P2`. A [Figura 4.4](#) apresenta um conjunto de operações que podem ser aplicadas às variáveis definidas.

Definição de Tipos e Variáveis	
Defina o tipo Node = Registro { Info do tipo Char; // campo usado para armazenar informação Next do tipo ponteiro para Node; // indica o próximo elemento da Lista }	
Defina o tipo NodePtr = ponteiro para Node;	
Variáveis L, P, P1, P2 do tipo NodePtr; // variáveis do mesmo tipo que o campo Next Variável X do tipo Char; // X é do mesmo tipo que o campo Info	
Operações	
X = P→Info;	X recebe o valor do campo Info do nó apontado por P.
P→Info = X;	O campo Info do Nó apontado por P recebe o valor de X.
P1 = P2;	O ponteiro P1 passa a apontar para onde aponta P2.
P1 = P→Next;	P1 passa a apontar para onde aponta o campo Next do Nó apontado por P.
P→Next = P1;	O campo Next do Nó apontado por P passa a apontar para onde aponta P1.
P = NewNode;	Aloca um Nó e retorna o endereço em P.
DeleteNode(P);	Libera (desaloca) o Nó apontado por P.

**FIGURA 4.4** Notação conceitual para manipulação de Listas Encadeadas.

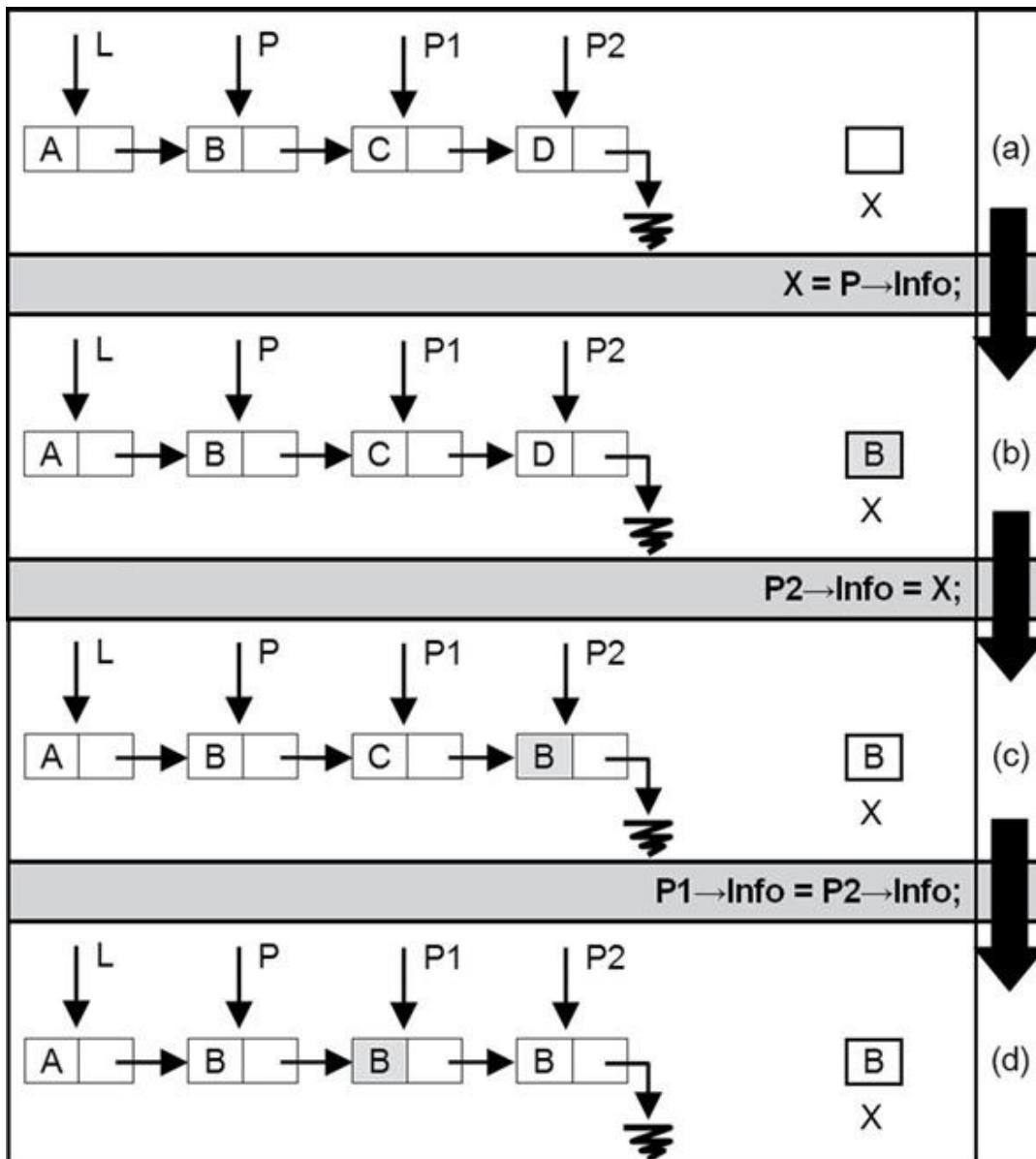
Node é o tipo definido para o Nό da Lista Encadeada. O tipo Node é um Registro com dois campos: o campo Info — utilizado para armazenar informação, e o campo Next — utilizado para indicar o próximo elemento da Lista. O campo Next e as variáveis L, P, P1 e P2 são do tipo NodePtr. O tipo NodePtr é um ponteiro para uma estrutura do tipo Node. Ou seja, variáveis do tipo NodePtr podem apontar para os Nós da Lista Encadeada.

Essa notação foi definida de modo a manter certa compatibilidade com as linguagens C e C++, e também com a nomenclatura adotada em parte da literatura sobre Estruturas de Dados. Por exemplo, [Drozdek \(2002\)](#) e também [Langsam, Augenstein e Tenenbaum \(1996\)](#), em suas versões não traduzidas, adotaram os termos Node, NodePtr, Info e Next. P→Info e P→Next são notações compatíveis com as linguagens C e C++. Os termos NewNode e DeleteNode fazem referência indireta aos comandos New e Delete da linguagem C++ (que serão abordados no [Capítulo 5](#)).

## 4.2.2 Entendendo o funcionamento das

## Operações

A Figura 4.5 ilustra o funcionamento das operações utilizadas para acessar e também para atualizar a informação armazenada nos Nós da Lista Encadeada.



**FIGURA 4.5** Operações para acessar e atualizar a informação armazenada.

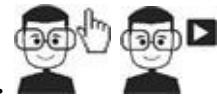
### Acessando e atualizando o campo Info de um Nô

Considerando como ponto de partida a situação da Figura 4.5a, aplicamos a

operação  $X = P \rightarrow \text{Info}$ . Aplicar essa operação significa que a variável X passará a ter o valor do campo Info do Nó apontado pelo Ponteiro P. Ou seja, a variável X passará a ter o valor B, conforme mostra a [Figura 4.5b](#).

À situação da [Figura 4.5b](#) aplicamos a operação  $P2 \rightarrow \text{Info} = X$ . Com essa operação, o campo Info do nó apontado por P2 recebe o valor da variável X. Veja na [Figura 4.5c](#) que a informação armazenada no Nó apontado por P2 passou a ter o valor B.

Com a aplicação da operação  $P1 \rightarrow \text{Info} = P2 \rightarrow \text{Info}$ , o campo Info do nó apontado por P1 recebe o valor do campo Info do Nó apontado por P2. Veja o resultado na [Figura 4.5d](#). Um modo alternativo de fazer a leitura do comando



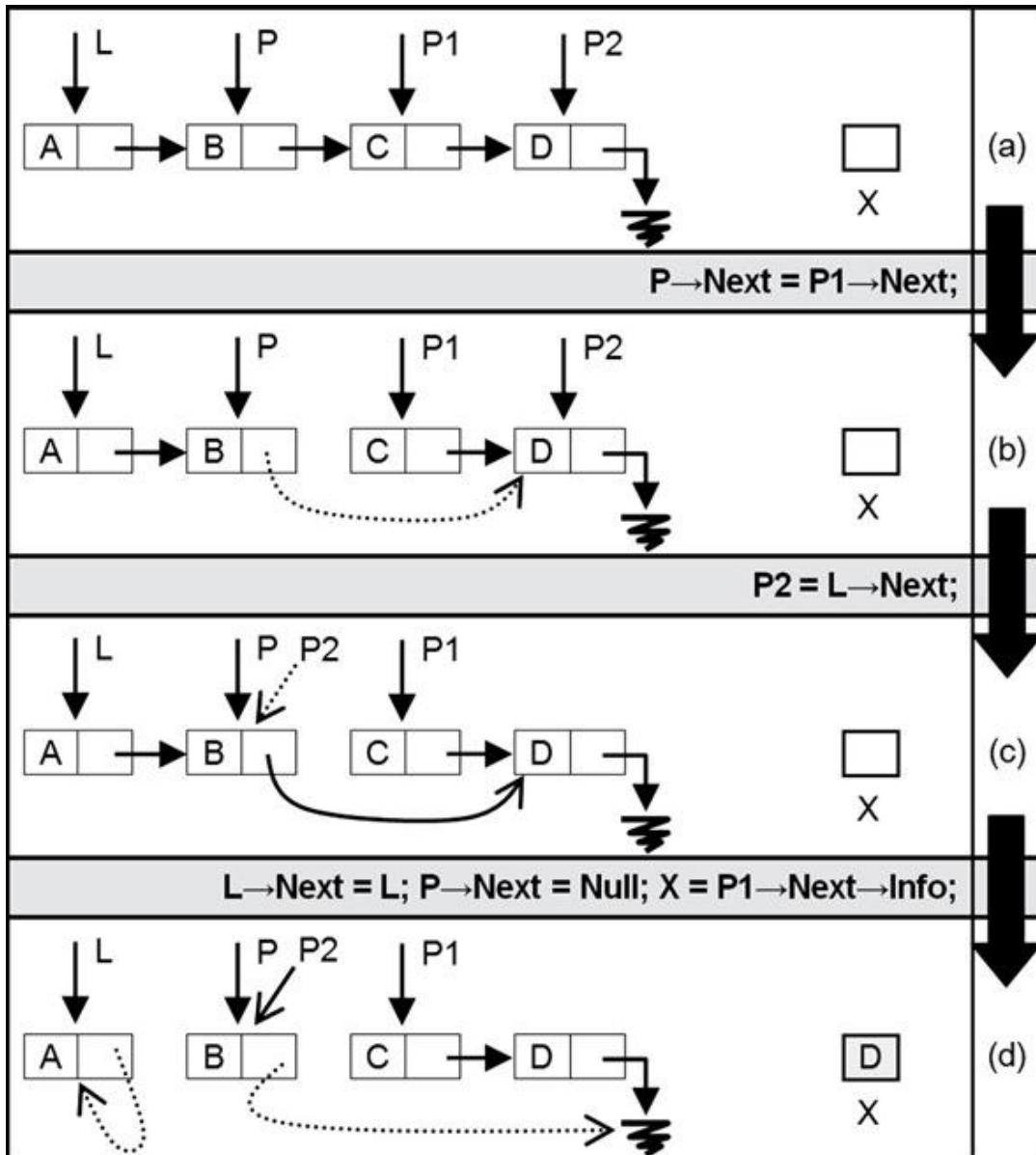
$P1 \rightarrow \text{Info} = P2 \rightarrow \text{Info}$  seria “Info de P1 recebe Info de P2”.

## Acessando e atualizando o campo Next de um Nó

As operações utilizadas para acessar e atualizar a indicação do Próximo elemento da Lista são exemplificadas na [Figura 4.6](#). Tendo como ponto de partida a situação inicial da [Figura 4.6a](#), aplicamos a operação  $P \rightarrow \text{Next} = P1 \rightarrow \text{Next}$  resultando na situação da [Figura 4.6b](#). O campo Next do nó apontado por P é atualizado com a informação do campo Next do nó apontado por P1. Note, na [Figura 4.6b](#), que o campo Next do Nó apontado por P passa a apontar para onde aponta P2. Se aplicássemos a operação  $P \rightarrow \text{Next} = P2$  à situação da



[Figura 4.6a](#), obteríamos o mesmo resultado.



**FIGURA 4.6** Operações para acessar e atualizar a indicação do próximo elemento da Lista.

A partir da situação ilustrada na [Figura 4.6b](#), aplicamos a operação **P2 = L → Next**. Com isso o apontador P2 passa a apontar para onde aponta o campo Next do Nό apontado por L, resultando na situação da [Figura 4.6c](#). Um modo alternativo de fazer a leitura do comando **P2 = L → Next** seria “P2 recebe Next de L”.

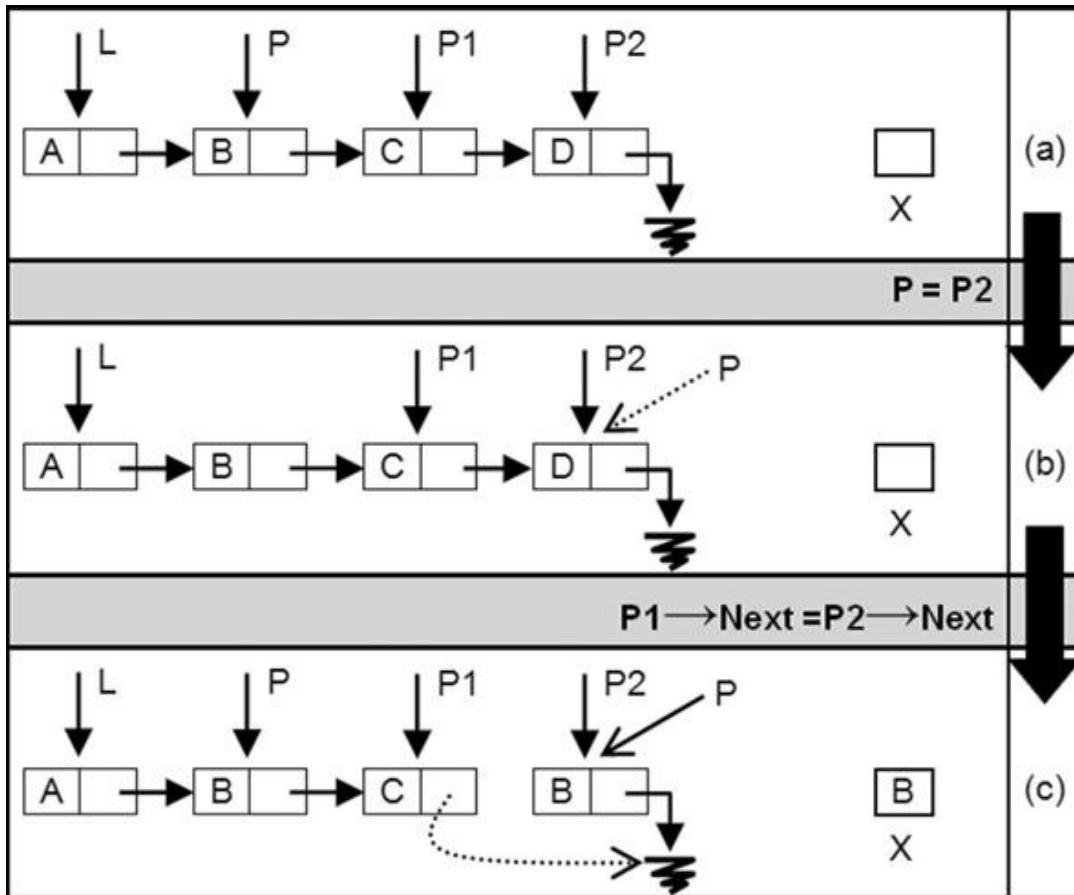
Para passar da situação da [Figura 4.6c](#) para a situação da [Figura 4.6d](#), aplicamos três operações. A primeira operação, **L → Next = L**, faz o campo Next do Nό apontado por L apontar para o próprio L. A segunda operação, **P → Next =**

**Null**, faz o campo Next do Nó apontado por P passar a apontar para o valor Null. Na terceira operação,  $X = P1 \rightarrow \text{Next} \rightarrow \text{Info}$ , a variável X recebe a informação armazenada no campo Info do Nó apontado por  $P1 \rightarrow \text{Next}$ . Utilizando parênteses para tirar a ambiguidade, o comando poderia ser expresso como  $X = (P1 \rightarrow \text{Next}) \rightarrow \text{Info}$ . Ou seja, X passa a ter o valor D. Um modo alternativo de fazer a leitura do comando  $X = P1 \rightarrow \text{Next} \rightarrow \text{Info}$  seria: “X recebe Info do Next de P1”.

## Movendo ponteiros

A situação inicial da [Figura 4.7a](#) é alterada pela operação  $P=P2$ . Com essa operação, o ponteiro P passa a apontar para onde aponta o ponteiro P2, resultando na situação da [Figura 4.7b](#). Note que passar a apontar para onde aponta  $P2 \rightarrow \text{Next}$  é diferente de passar a apontar para onde aponta P2. Para ilustrar essa diferença, a partir da situação da [Figura 4.7b](#) aplicamos a operação  $P1 \rightarrow \text{Next} = P2 \rightarrow \text{Next}$ . P1 → Next passará a apontar não para onde aponta P2, mas para onde aponta o campo Next do Nó apontado por P2. Ou seja, P1 → Next

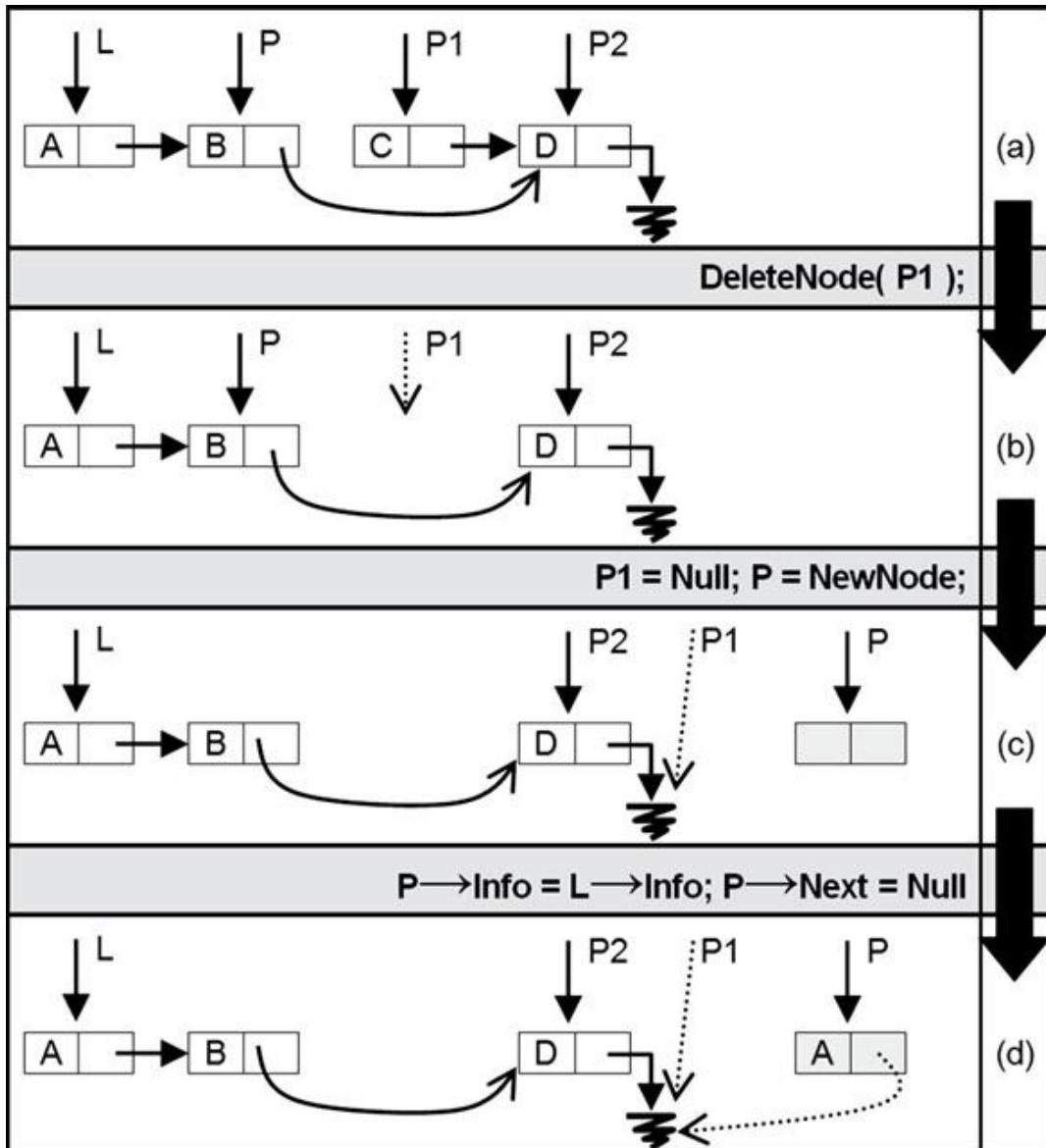
passará a apontar para Null ([Figura 4.7c](#)). 



**FIGURA 4.7** Operações para mover ponteiros.

## Alocando e desalocando Nós

As operações para alocar e desalocar Nós são ilustradas na Figura 4.8. Inicialmente, à situação inicial da Figura 4.8a aplicamos a operação **DeleteNode(P1)**. Essa operação desaloca o Nó apontado por P1, ou seja, libera a memória, para que possa ser utilizada para outras finalidades. Na prática, funciona como se o Nó apontado por P1 simplesmente deixasse de existir. Veja, na situação da Figura 4.8b, que o Nó que era apontado por P1 na Figura 4.8a simplesmente sumiu! Mas note na Figura 4.8b que P1 ainda continua a existir e parece apontar “para o nada”. A operação DeleteNode desalocou o Nó apontado por P1, mas não desalocou P1. Nessa situação, P1 está apontando para uma posição de memória que a princípio não está mais armazenando uma informação consistente e precisa ser atualizado para não causar erros na sequência do programa. Aplicamos então a operação **P1 = Null** para atualizar o valor de P1, que passa a apontar para Null (Figura 4.8c).



**FIGURA 4.8** Operações para alocar e desalocar Nós.

Após atualizar o valor de P1, aplicamos à [Figura 4.8b](#) a operação `P = NewNode`. Essa operação aloca um novo Nó. Na prática, um Nó que não existia passa a existir, e o acesso a esse Nó é feito a partir do ponteiro P. Com a operação `P = NewNode`, o ponteiro P deixa de apontar para o Nó para o qual estava apontando e passa a apontar para o novo Nó, recém-alocado ([Figura 4.8c](#)).

Note na [Figura 4.8c](#) que o Nó apontado por P, que acabou de ser alocado, não possui valor no campo Info nem no campo Next. Então, aplicamos duas operações: `P → Info = L → Info` e, em seguida, `P → Next = Null`. Na primeira dessas operações, o campo Info do Nó apontado por P passa a ter o valor do

campo Info do Nó apontado por L, ou seja, passa a ter o valor A. A segunda dessas operações atualiza o campo Next do Nó apontado por P, que passa a



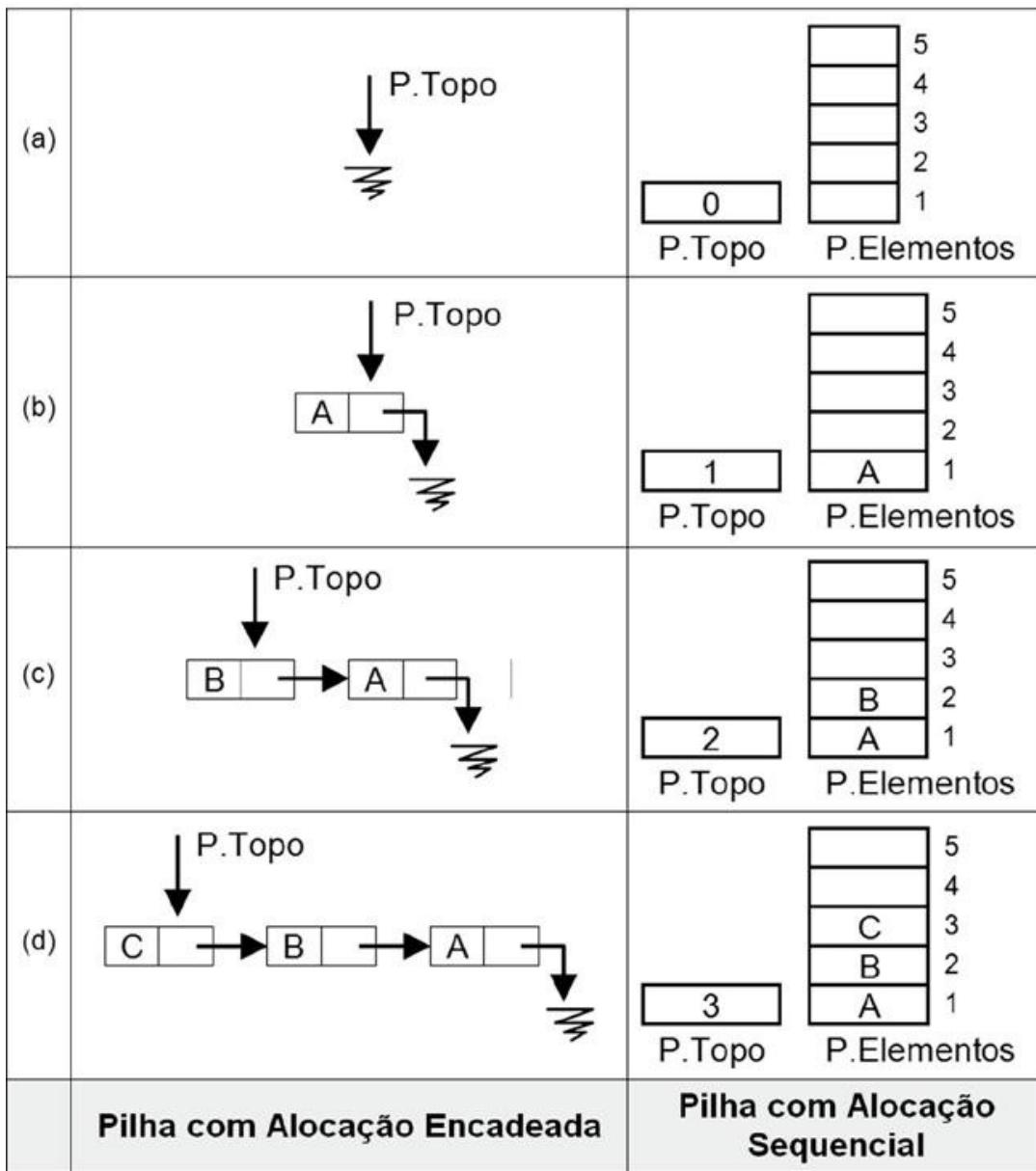
apontar para Null ([Figura 4.8d](#)) .

## 4.3 Implementando uma Pilha como uma Lista

### Encadeada

No [Capítulo 2](#), implementamos uma Pilha com Alocação Sequencial e Estática. Os elementos da Pilha ficavam armazenados em um vetor. Queremos agora implementar uma Pilha com Alocação Encadeada, ou seja, utilizaremos uma Lista Encadeada para implementar a Pilha.

A [Figura 4.9](#) apresenta diagramas comparando duas maneiras de implementar uma Pilha P. Os diagramas da coluna mais à direita mostram a Pilha P implementada com Alocação Sequencial, como fizemos no [Capítulo 2](#). Os diagramas da coluna à esquerda mostram a mesma Pilha P implementada como uma Lista Encadeada. Na [Figura 4.9a](#) está representada uma Pilha vazia, ou seja, sem nenhum elemento. Na [Figura 4.9b](#), a Pilha P contém um elemento, o elemento A. Na [Figura 4.9c](#), a Pilha P já conta com dois elementos, sendo que o elemento B está no topo da Pilha. Na [Figura 4.9d](#), a Pilha P está com três elementos, sendo que o elemento C está no topo da Pilha.



**FIGURA 4.9** Pilha como Lista Encadeada.

Na representação da Pilha como uma Lista Encadeada, quando a Pilha está vazia P.Topo aponta para Null. Considere que P.Topo é do tipo NodePtr ou ponteiro para Nó, conforme definido na [Figura 4.4](#). Quando a Pilha possui algum elemento, P.Topo aponta para o elemento que está no topo da Pilha. A ordem dos elementos na Pilha é explicitamente indicada pelo campo Next de cada um dos elementos. Ou seja, no caso em que a Pilha P conta com três elementos ([Figura 4.9d](#)), o elemento do Topo da Pilha está no Nó apontado por P.Topo; o elemento que vem em seguida está armazenado no Nó apontado pelo campo Next do Nó apontado por P.Topo, e assim por diante.

Queremos agora implementar as Operações Primitivas de uma Pilha — Empilha, Desempilha, Cria, Vazia e Cheia —, detalhadas na [Figura 2.6](#). Implementaremos a Pilha como uma Lista Encadeada, mas as operações precisam produzir exatamente o mesmo efeito que produzem aquelas que implementamos no [Capítulo 2](#), com Alocação Sequencial. Vista de fora, a caixa-preta Pilha precisa ser exatamente a mesma. Por dentro, a implementação será outra.

## Exercício 4.1 Operação Empilha

Conforme especificado na [Figura 2.6](#), a operação Empilha recebe como parâmetros a Pilha na qual queremos empilhar um elemento, e o valor do elemento que queremos empilhar. A Pilha deve ser implementada como uma Lista Encadeada. Você encontrará uma possível solução para este exercício a seguir. Mas tente propor uma solução antes de consultar a resposta.

```
Empilha (parâmetro por referência P do tipo Pilha, parâmetro X do tipo Char, parâmetro por referência DeuCerto do tipo Boolean);
/* Empilha o elemento X na Pilha P. O parâmetro DeuCerto deve indicar se a operação foi bem-sucedida ou não */
```

A [Figura 4.10](#) apresenta um algoritmo conceitual para a Operação Empilha. Se a Pilha P estiver cheia, sinalizamos através do parâmetro DeuCerto que o elemento X não foi inserido na Pilha P. Mas, se a Pilha P não estiver cheia, X será inserido. Para isso, alocamos um Nó (PAux = NewNode), armazenamos a informação X no campo Info do Nó apontado por PAux ( $Paux \rightarrow Info = X$ ), apontamos o campo Next do Nó apontado por PAux para o topo da Pilha ( $Paux \rightarrow Next = P.Topo$ ) e fazemos P.Topo apontar para PAux ( $P.Topo = PAux$ ). O Nó que acabou de ser inserido passará a ser o topo da Pilha.

```

Empilha (parâmetro por referência P do tipo Pilha, parâmetro X do tipo Char, parâmetro
por referência DeuCerto do tipo Boolean) {

/* Empilha o elemento X na Pilha P, ambos passados como parâmetro. O parâmetro
DeuCerto deve indicar se a operação foi bem-sucedida ou não */

Variável PAux do tipo NodePtr;
/* PAux é uma variável auxiliar do tipo NodePtr. O tipo Pilha, nesta implementação
encadeada, contém o campo P.Topo, que também é do tipo NodePtr, ou seja, um ponteiro
para Nô, conforme definido na Figura 4.6 */

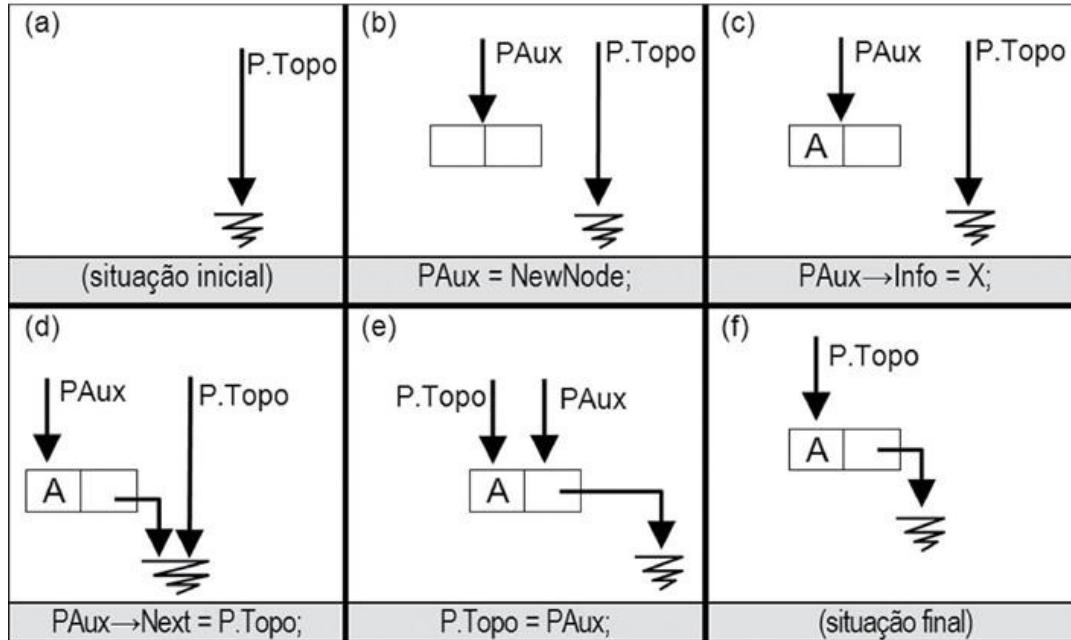
Se (Cheia(P)== Verdadeiro)           // se a Pilha P estiver cheia...
Então   DeuCerto = Falso;            // ... não podemos empilhar
Senão { PAux = NewNode;             // aloca um novo Nô
PAux→Info = X;                   // armazena o valor de X no novo Nô
PAux→Next = P.Topo;              // o próximo deste novo nô será o elemento do topo
P.Topo = PAux;                   // o topo da pilha passa a ser o novo Nô
DeuCerto = Verdadeiro;           // a operação deu certo
} // fim do procedimento Empilha

```

**FIGURA 4.10** Algoritmo conceitual — Empilha.

Vamos executar passo a passo esse algoritmo da operação Empilha? A [Figura 4.11](#) mostra a execução passo a passo da operação Empilha partindo da situação

em que a Pilha P está vazia ([Figura 4.11a](#)) .



**FIGURA 4.11** Execução da Operação Empilha partindo da situação inicial de Pilha Vazia.

A princípio, nessa implementação conceitual de uma Pilha Encadeada, vamos considerar que a Pilha nunca ficará cheia. Ou seja, vamos considerar que o parâmetro DeuCerto sempre retornará o valor verdadeiro. Nesse tipo de implementação, uma estrutura só ficará cheia se não houver memória disponível no computador ou no dispositivo em que o programa estiver sendo executado. Ao implementar em uma linguagem de programação, será possível verificar se a operação de alocar memória para um Nó da Pilha foi bem-sucedida ou não.



Considerando, então, que a Pilha não está cheia, executamos o comando PAux = NewNode, que aloca um novo Nó. A variável PAux estará apontando para esse novo Nó alocado, conforme mostra a [Figura 3.11b](#).

O comando seguinte, conforme o algoritmo descrito na [Figura 4.10](#), é PAux → Info = X. Esse comando atribui o valor do parâmetro X ao campo Info do Nó apontado por PAux. O parâmetro X contém o valor do elemento que queremos inserir na Pilha P. O resultado da execução desse comando é apresentado na [Figura 4.11c](#).

Na [Figura 4.11d](#) atualizamos o valor do campo Next do Nó apontado por PAux, que passa a apontar para onde P.Topo está apontando, ou seja, passa a apontar para Null.

A seguir temos o comando P.Topo = PAux, pelo qual atribuímos a P.Topo o valor de PAux, ou seja, P.Topo passa a apontar para onde aponta PAux ([Figura 4.11e](#)).

Considerando que PAux é uma variável local, ao final da execução da operação Empilha PAux deixa de existir, e a situação final fica conforme mostra a [Figura 4.11f](#). A Pilha P estava inicialmente vazia e, com a execução da operação Empilha, passou a ter um elemento.

## Exercício 4.2 Executar passo a passo a Operação Empilha partindo de situação inicial com um elemento na Pilha

A [Figura 4.11](#) mostrou a execução da operação Empilha tendo como situação inicial a Pilha P vazia. Com a execução da operação Empilha, a Pilha P passou a ter um elemento. Desenhe passo a passo a execução da operação Empilha



partindo agora da situação inicial em que a Pilha P contém um elemento e levando a Pilha P a ter dois elementos. O elemento a ser inserido deve passar a ser o topo da Pilha. Considere que a Pilha nunca estará cheia.

### Dica importante

Sempre que for elaborar um algoritmo sobre Listas Encadeadas, execute o algoritmo fazendo o desenho passo a passo. Confira se a execução realmente levará o diagrama à situação final desejada. Desenhando passo a passo, será bem mais fácil elaborar algoritmos corretos sobre Listas Encadeadas.

Implemente agora os algoritmos para as operações Desempilha, Cria, Vazia e Cheia, considerando uma Pilha como uma Lista Encadeada. Você encontrará soluções para estes exercícios a seguir. Mas é importante que proponha soluções antes de consultar as respostas.

## Exercício 4.3 Operação Desempilha

Conforme especificado na [Figura 2.6](#), a operação Desempilha recebe como parâmetro a Pilha da qual queremos retirar um elemento. Caso a Pilha não estiver vazia, a operação retorna o valor do elemento retirado. Implemente a operação Desempilha considerando a Pilha como uma Lista Encadeada.

```
Desempilha(parâmetro por referência P do tipo Pilha, parâmetro por referência X do tipo Char, parâmetro por referência DeuCerto do tipo Boolean);
/* Se a Pilha P estiver vazia, o parâmetro DeuCerto deve retornar Falso. Caso a Pilha P não estiver vazia, a operação Desempilha deve retornar o valor do elemento do topo da Pilha no parâmetro X. O Nó em que se encontra o elemento do topo deve ser desalocado, e o topo da Pilha deve ser atualizado para o próximo elemento*/
```

## Exercício 4.4 Operação Cria

Conforme especificado na [Figura 2.6](#), a operação Cria recebe como parâmetro a Pilha que deverá ser criada. Criar a Pilha significa inicializar os valores de modo a indicar que a pilha está vazia, ou seja, não contém nenhum elemento.

```
Cria (parâmetro por referência P do tipo Pilha);
/* Cria a Pilha P, inicializando a Pilha como vazia - sem nenhum elemento. */
```

## Exercício 4.5 Operação Vazia

Conforme especificado na [Figura 2.6](#), a operação Vazia testa se a Pilha passada como parâmetro está vazia (sem elementos), retornando o valor Verdadeiro (Pilha vazia) ou Falso (Pilha não vazia).

```
Boolean Vazia (parâmetro por referência P do tipo Pilha);  
/* Retorna Verdadeiro se a Pilha P estiver vazia - sem nenhum elemento; Falso caso contrário. */
```

## Exercício 4.6 Operação Cheia

Conforme especificado na [Figura 2.6](#), a operação Cheia testa se a Pilha passada como parâmetro está cheia. Nessa implementação conceitual, considere que a Pilha nunca estará cheia.

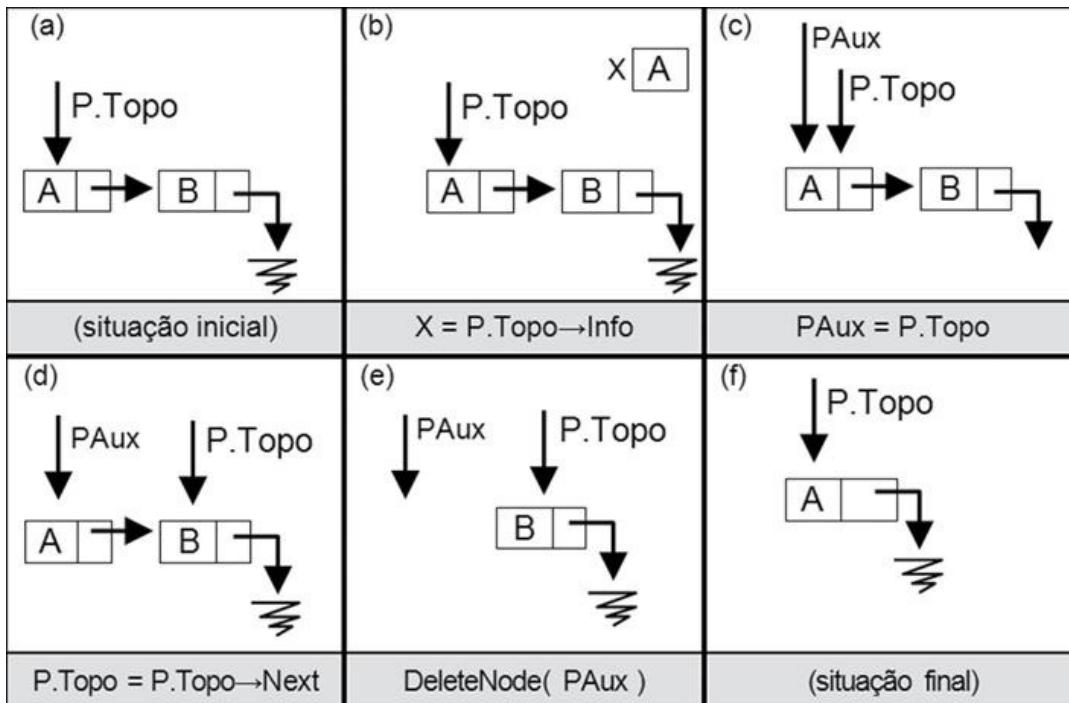
```
Boolean Cheia (parâmetro por referência P do tipo Pilha);  
/* Nessa implementação conceitual, a operação cheia retorna sempre o valor Falso. Ou seja, a Pilha nunca estará cheia */
```

A [Figura 4.12](#) apresenta soluções para os [Exercícios 4.3 a 4.6](#). Na operação Cria, basta apontar P para Null e estaremos criando uma Pilha vazia, conforme mostra o diagrama da [Figura 4.11a](#). A operação Vazia consiste em verificar se a Pilha está vazia ou não. Em uma Pilha vazia, nessa implementação encadeada, P.Topo aponta para Null ([Figura 4.11a](#)). Quando uma Pilha não é vazia, o valor de P.Topo é diferente de Null ([Figura 4.11f](#) ou coluna à esquerda das [Figuras 4.9b, 4.9c e 4.9d](#)). Conforme já mencionamos, nessa implementação conceitual de uma Pilha Encadeada a operação Cheia retorna sempre falso. Na operação Desempilha, o primeiro passo é verificar se a Pilha está vazia. Não é possível desempilhar um elemento de uma Pilha vazia. Caso a Pilha não estiver vazia, o parâmetro X retorna o valor do campo Info do nó apontado por P.Topo, ou seja, X retorna o valor do elemento que está no topo da Pilha ( $X = P.\text{Topo} \rightarrow \text{Info}$ ) — [Figura 4.13b](#). Em seguida colocamos a variável auxiliar PAux apontando para onde aponta P.Topo ( $\text{Paux} = P.\text{Topo}$ ) — [Figura 4.13c](#). Avançamos o Topo da Pilha para o próximo elemento ( $P.\text{Topo} = P.\text{Topo} \rightarrow \text{Next}$ ) — [Figura 4.13d](#) — e em seguida desalocamos o Nó que continha o elemento que estava no Topo da Pilha ( $\text{DeleteNode}(\text{PAux})$ ) — [Figura 4.13e](#). Para isso foi preciso guardar em PAux o endereço do Nó que queríamos desalocar; note na [Figura 4.13d](#) que

P.Topo já não está mais apontando para esse Nó, agora apontado apenas por PAux.

```
Cria (parâmetro por referência P do tipo Pilha) {  
    /* Cria a Pilha P, inicializando a Pilha como vazia - sem nenhum elemento. */  
    P.Topo = Null; // P.Topo passa a apontar para Null. Isso indica que a Pilha está vazia.  
} // fim do Cria  
  
Boolean Vazia (parâmetro por referência P do tipo Pilha) {  
    /* Retorna Verdadeiro se a Pilha P estiver sem nenhum elemento; Falso caso contrário */  
    Se (P.Topo == Null)  
        Então Retorne Verdadeiro; // a Pilha P está vazia  
    Senão Retorne Falso; // a Pilha P não está vazia  
} // fim do Vazia  
  
Boolean Cheia (parâmetro por referência P do tipo Pilha) {  
    /* Nessa implementação conceitual, a operação cheia retorna sempre Falso */  
    Retorne Falso; // nessa implementação conceitual, a Pilha nunca estará cheia.  
} // fim do Cheia  
  
Desempilha(parâmetro por referência P do tipo Pilha, parâmetro por referência X do tipo  
Char, parâmetro por referência DeuCerto do tipo Boolean) {  
    /* Se a Pilha P estiver vazia, o parâmetro DeuCerto deve retornar Falso. Caso a Pilha P  
    não esteja vazia, a operação Desempilha deve retornar o valor do elemento do Topo da  
    Pilha no parâmetro X. O Nó em que se encontra o elemento do Topo deve ser  
    desalocado, e o Topo da Pilha deve então ser atualizado para o próximo elemento */  
  
    Variável PAux do tipo NodePtr; // ponteiro para Nó, auxiliar  
  
    Se (Vazia( P ) == Verdadeiro)  
        Então DeuCerto = Falso;  
    Senão { DeuCerto = Verdadeiro;  
        X = P.Topo→Info;  
        PAux = P.Topo ; // aponta PAux para P.Topo, para guardar esse endereço  
                    // antes de mudar P.Topo de lugar  
        P.Topo = P.Topo→Next; // o topo da Pilha avança para o próximo elemento.  
        DeleteNode( PAux ); // desaloca o Nó que armazenava o elemento do topo  
    }  
} // fim do Desempilha
```

**FIGURA 4.12** Operações Cria, Vazia, Cheia e Desempilha.



**FIGURA 4.13** Execução da Operação Desempilha partindo da situação inicial de Pilha com dois elementos.

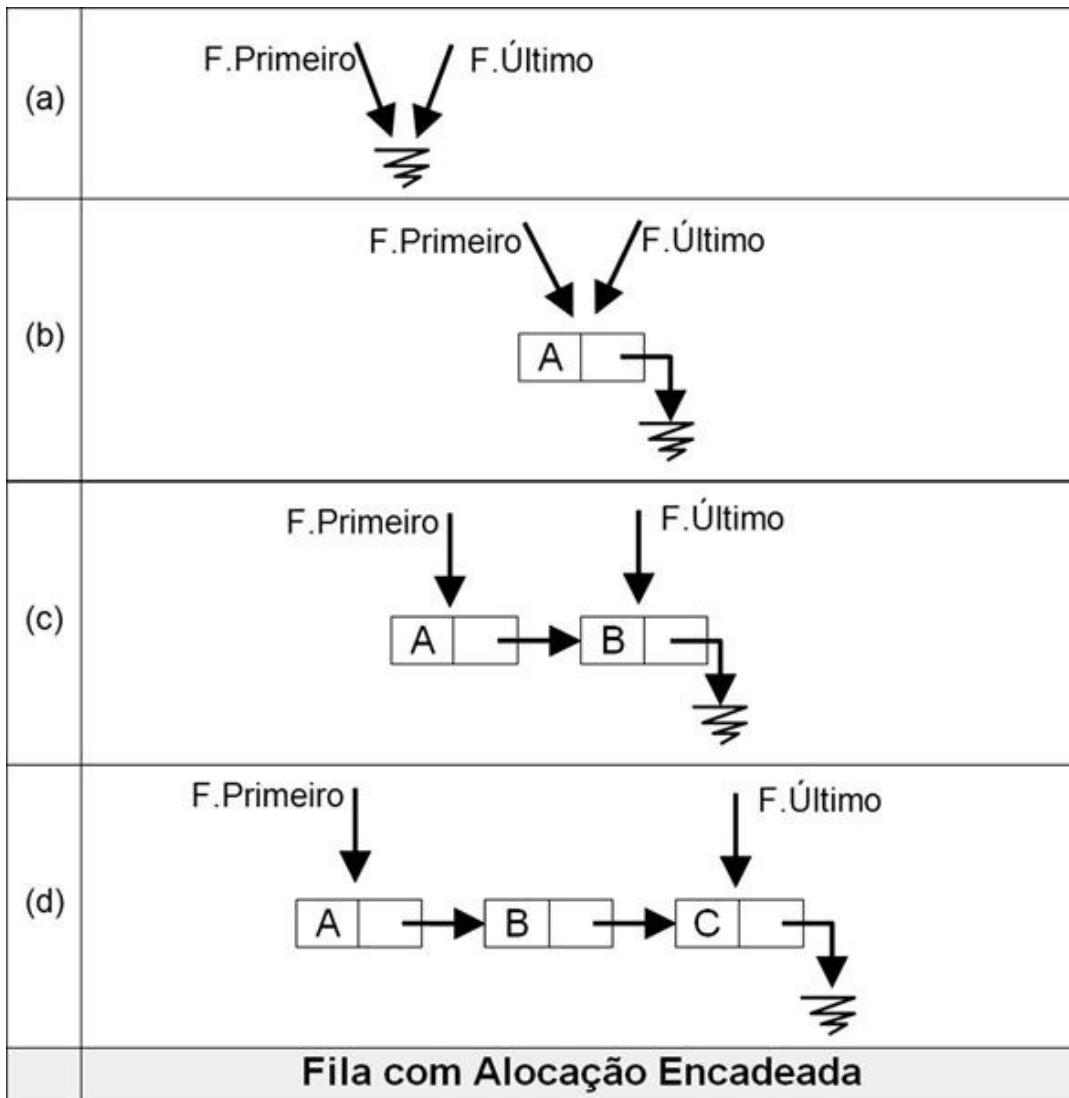
A Figura 4.13 mostra a execução passo a passo da operação Desempilha partindo de uma situação inicial com dois elementos na Pilha. Partindo de uma situação inicial diferente, a execução se altera.

### Exercício 4.7 Executar passo a passo a Operação Desempilha partindo de situação inicial com um único elemento na Pilha

Desenhe passo a passo a execução da operação Desempilha partindo da situação inicial em que a Pilha P contém um único elemento e levando a Pilha P a ficar vazia. Adapte os diagramas da Figura 4.13.

## 4.4 Implementando uma Fila como uma Lista Encadeada

A Figura 4.14 apresenta o esquema da implementação de uma Fila através de uma Lista Encadeada. A Fila F contém dois campos do tipo ponteiro para Nó: F.Primeiro e F.Ultimo, que apontam, respectivamente, para o Primeiro e para o Último elemento da Fila. Na situação em que a Fila é vazia, ambos os ponteiros — Primeiro e Último — apontam para Null (Figura 4.14a).



**FIGURA 4.14** Fila como Lista Encadeada.

Quando a Fila tem um único elemento, tanto F.Primeiro quanto F.Ultimo apontam para esse elemento (Figura 4.14b). Com dois, três ou mais elementos, F.Primeiro apontará sempre para o Primeiro elemento, e F.Ultimo, para o Último elemento da Fila (Figuras 4.14c e 4.14d).

### Exercício 4.8 Operação Cria a Fila

Criar a Fila significa inicializar os valores de modo a indicar que a Fila está vazia. Especificação do TAD Fila na [Figura 3.4](#).

Cria (parâmetro por referência F do tipo Fila);  
/\* Cria a Fila F, inicializando como vazia - sem nenhum elemento - Figura 4.14a \*/

## Exercício 4.9 Operação para testar se a Fila está vazia

Conforme especificado na [Figura 3.4](#), a operação Vazia testa se a Fila passada como parâmetro está vazia (sem elementos), retornando o valor Verdadeiro (Fila vazia) ou Falso (Fila não vazia).

Boolean Vazia (parâmetro por referência F do tipo Fila);  
/\* Retorna Verdadeiro se a Fila F estiver vazia - sem nenhum elemento; Falso caso contrário \*/

## Exercício 4.10 Operação para testar se a Fila está cheia

Conforme especificado na [Figura 3.4](#), a operação Cheia testa se a Fila passada como parâmetro está cheia. Nessa versão conceitual de uma Fila Encadeada, considere que a Fila nunca estará cheia.

Boolean Cheia (parâmetro por referência F do tipo Fila);  
/\* Nessa implementação conceitual, a operação cheia retorna sempre o valor Falso \*/

## Exercício 4.11 Operação Insere na Fila

Conforme especificado na [Figura 3.4](#), a operação Insere recebe como parâmetros a Fila na qual queremos inserir um elemento e o valor do elemento que queremos inserir. O elemento só não será inserido se a Fila já estiver cheia.

Insere (parâmetro por referência F do tipo Fila, parâmetro X do tipo Char, parâmetro por referência DeuCerto do tipo Boolean);  
/\* Insere o elemento X na Fila F. O parâmetro DeuCerto deve indicar se a operação foi bem-sucedida ou não. A operação só não será bem-sucedida se tentarmos inserir um elemento em uma Fila cheia \*/

## Exercício 4.12 Operação Retira da Fila

Conforme especificado na [Figura 3.4](#), a operação Retira recebe como parâmetro a Fila da qual queremos retirar um elemento. Caso a Fila não estiver vazia, a

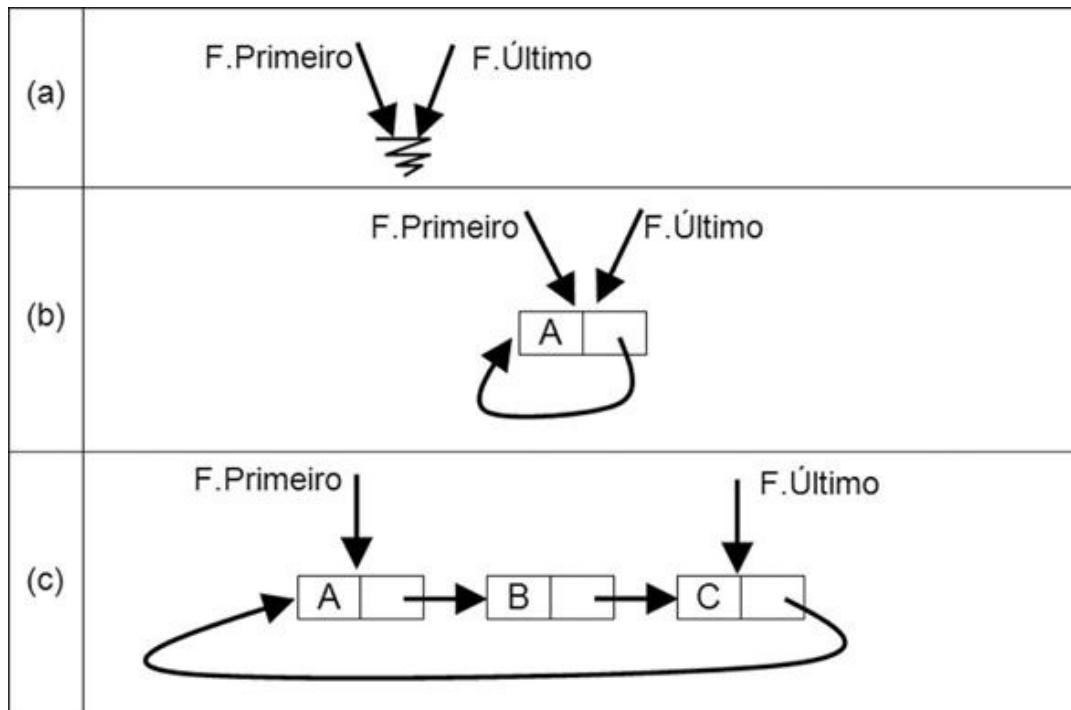
operação retorna o valor do elemento retirado. O elemento a ser retirado deve ser sempre o primeiro da Fila.

Retira (parâmetro por referência F do tipo Fila, parâmetro por referência X do tipo Char, parâmetro por referência DeuCerto do tipo Boolean);

/\* Caso a Fila F não estiver vazia, retira o primeiro elemento da Fila e retorna o seu valor no parâmetro X. Se a Fila F estiver vazia, o parâmetro DeuCerto retorna Falso \*/

### Exercício 4.13 Fila como Lista Encadeada Circular

Em uma Lista Encadeada Circular, o campo Next do Último elemento da Lista não aponta para Null, mas para o Primeiro elemento da Lista. Implemente as operações Cria, Vazia, Cheia, Insere e Retira, para uma Fila implementada como Lista Encadeada Circular, como ilustrado nos diagramas da [Figura 4.15](#).

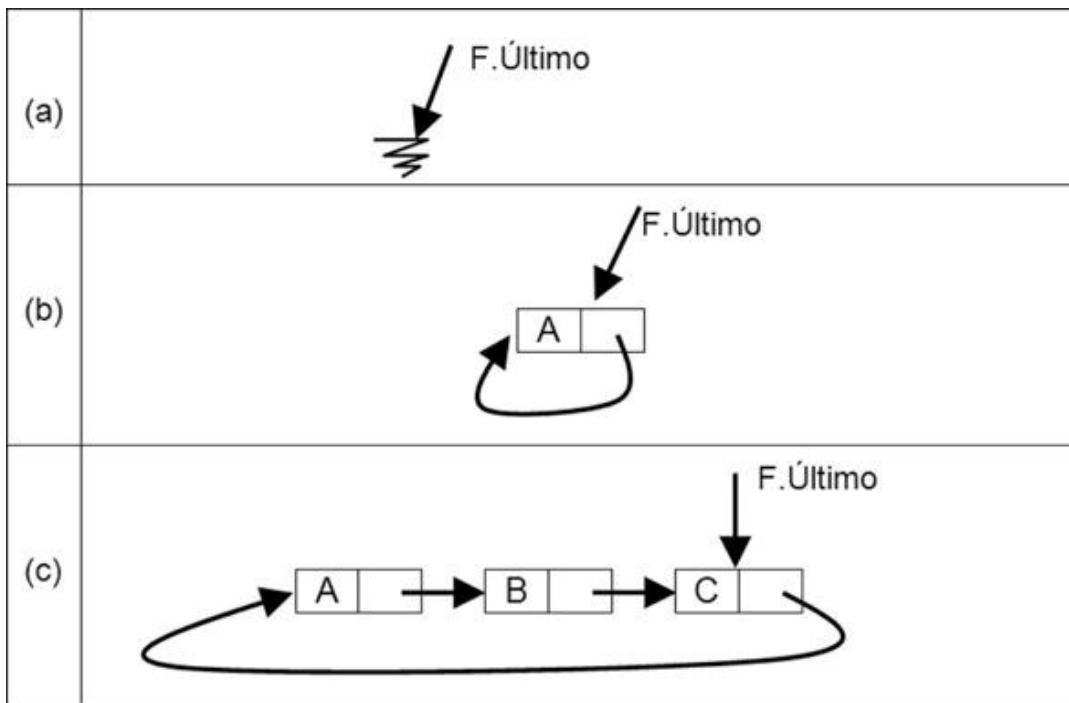


**FIGURA 4.15** Fila como Lista Encadeada Circular.

### Exercício 4.14 Fila Circular com Ponteiro só para o Último

Em uma Fila implementada como uma Lista Encadeada Circular, podemos deixar de armazenar o ponteiro para o Primeiro elemento, pois o campo Next do

Último elemento já estará indicando o Primeiro elemento da Fila. Implemente as operações Cria, Vazia, Cheia, Insere e Retira, para uma Fila implementada como Lista Encadeada Circular com ponteiro apenas para o Último elemento da Fila, como ilustrado nos diagramas da [Figura 4.16](#).



**FIGURA 4.16** Fila como Lista Encadeada Circular com o ponteiro apenas para o Último elemento da Fila.

### Exercício 4.15 Pilha como Lista Encadeada Circular

Implemente as operações Cria, Vazia, Cheia, Empilha e Desempilha, para uma Pilha implementada como Lista Encadeada Circular, como ilustrado nos diagramas da [Figura 4.17](#).

**Consulte nos Materiais Complementares**



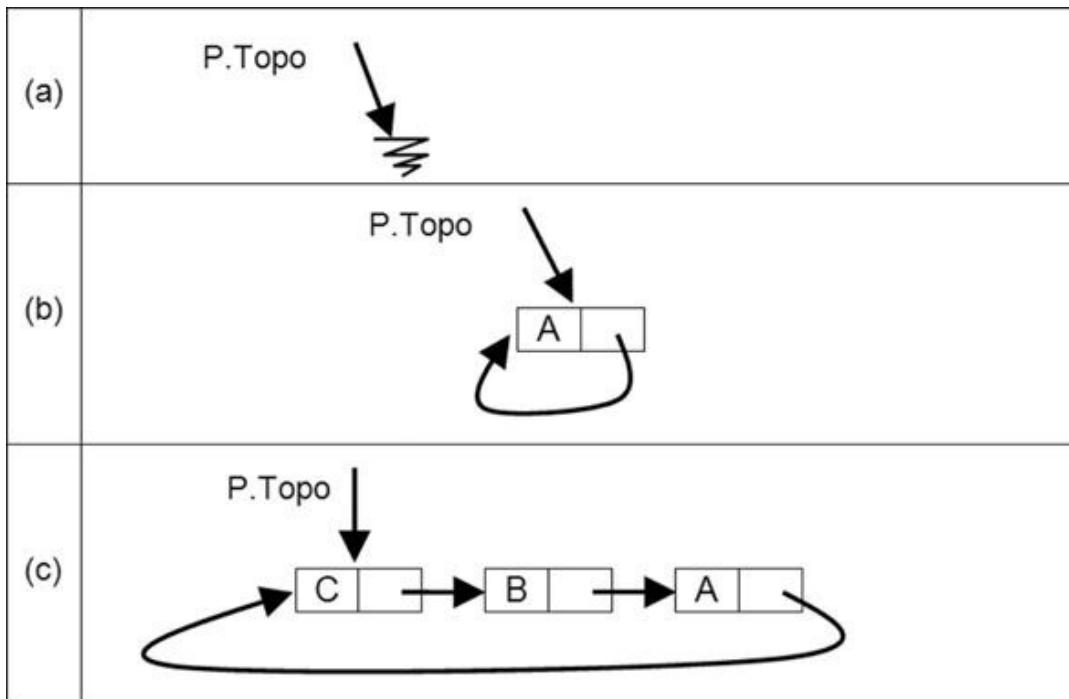
Vídeos sobre Listas Encadeadas



Animações sobre Listas Encadeadas



<http://www.elsevier.com.br/edcomjogos>



**FIGURA 4.17** Pilha como Lista Encadeada Circular.

## Exercícios de fixação

**Exercício 4.16** Qual a diferença entre Alocação Sequencial e Alocação Encadeada de Memória?

**Exercício 4.17** Na sua opinião, quais são as vantagens de utilizar Alocação Encadeada para um conjunto de elementos? Quais as possíveis desvantagens?

**Exercício 4.18 Avanço de Projeto.** Após fazer uma reflexão sobre as vantagens e desvantagens da Alocação Sequencial e da Alocação Encadeada, reflita sobre as características dos jogos que você está desenvolvendo. Então, procure responder: qual técnica de implementação parece ser mais adequada às características dos jogos que você está desenvolvendo no momento: Alocação Sequencial ou Alocação Encadeada de Memória?

**Exercício 4.19** Faça diagramas ilustrando a implementação de uma Pilha como uma Lista Encadeada; explique sucintamente o funcionamento.

**Exercício 4.20** Faça diagramas ilustrando a implementação de uma Fila como uma Lista Encadeada; explique sucintamente o funcionamento.

## Soluções para alguns dos exercícios

### Exercícios 4.8, 4.9, 4.10, 4.11 e 4.12

```
Defina o tipo Node = Registro {  
    Info do tipo Char; // campo usado para armazenar informação  
    Next do tipo ponteiro para Node; // indica o próximo elemento da Lista  
}
```

Defina o tipo NodePtr = ponteiro para Node;

```
Defina o tipo Fila = Registro {  
    Primeiro, Último do tipo NodePtr;  
}
```

```
Cria (parâmetro por referência F do tipo Fila) {  
/* Cria a Fila F, inicializando a Fila como vazia - sem nenhum elemento *  
    F.Primeiro = Null;  
    F.Ultimo = Null;  
} // fim do Cria
```

```
Boolean Vazia (parâmetro por referência F do tipo Fila) {  
/* Retorna Verdadeiro se a Fila F estiver vazia - sem nenhum elemento; retorna Falso caso  
contrário. Na implementação encadeada da Figura 4.16, verificar se a Fila está vazia significa  
verificar se F.Primeiro ou F.Ultimo aponta para Null */  
Se (F.Primeiro == Null) // ou F.Ultimo == Null  
Então Retorne Verdadeiro;  
Senão Retorne Falso;  
} // fim do Vazia
```

```
Boolean Cheia (parâmetro por referência F do tipo Fila) {  
/* Nessa implementação conceitual, a operação cheia retorna sempre o valor Falso */  
Retorne Falso;  
} // fim do Cheia
```

```
Insere (parâmetro por referência F do tipo Fila, parâmetro X do tipo Char, parâmetro por referência  
DeuCerto do tipo Boolean) {  
/* Insere o elemento X na Fila F. O parâmetro DeuCerto deve indicar se a operação foi bem-  
sucedida ou não. A operação só não será bem-sucedida se tentarmos inserir um elemento em uma  
Fila cheia */
```

Variável PAux do tipo NodePtr;

```
Se (Cheia( F)==Verdadeiro)  
Então DeuCerto = Falso;  
Senão { DeuCerto = Verdadeiro;  
    PAux = NewNode; // aloca o Nó  
    PAux→Info = X; // armazena a informação no novo Nó  
    PAux→Next = Null; // em uma fila, o elemento sempre entra no final  
    Se (Vazia( F)==Verdadeiro)  
        Então F.Primeiro = PAux; // entrando o primeiro elemento da Fila  
        Senão F.Ultimo→Next = PAux; // já há algum elemento na Fila  
        F.Ultimo = PAux; // o elemento que acabou de entrar passa a ser o Último  
    }; // fim do senão  
} // fim do Insere
```

```
Retira (parâmetro por referência F do tipo Fila, parâmetro por referência X do tipo Char, parâmetro por referência DeuCerto do tipo Boolean) {  
/* Caso a Fila F não estiver vazia, retira o Primeiro elemento e retorna o seu valor no parâmetro X.  
Se a Fila F estiver vazia, o parâmetro DeuCerto retorna Falso */
```

Variável PAux do tipo NodePtr;

```
Se (Vazia( F ) == Verdadeiro)  
Então      DeuCerto = Falso;  
Senão {  DeuCerto = Verdadeiro;  
        X = F.Primeiro→Info; // pega a informação do primeiro da Fila, retorna em X  
        PAux = F.Primeiro;   // salva o endereço do Nó, para ser liberado  
        F.Primeiro = F.Primeiro→Next; // avança o primeiro da Fila para o próximo  
        Se F.Primeiro = Null // se a Fila tinha um único elemento...  
        Então F.Último = Null; // então F.Primeiro e F.Último vão apontar para Null  
        DeleteNode ( PAux ); // libera o Nó  
    };  
} // fim do Retira
```

## Referências e leitura adicional

1. Drozdek A. *Estrutura de dados e algoritmos em C++*. São Paulo: Thomson; 2002.
2. Langsam Y, Augenstein MJ, Tenenbaum AM. *Data Structures Using C and C++*. 2nd ed. New Jersey: Prentice Hall; 1996; Upper Saddle River.

---

## CAPÍTULO 5

---

# Listas Encadeadas com Alocação Dinâmica

---

## Seus objetivos neste capítulo

- Entender o que é Alocação Dinâmica de Memória, no contexto do armazenamento temporário de conjuntos de elementos.
- Entender que a Alocação Encadeada e a Alocação Dinâmica são conceitos independentes que, quando combinados, formam uma técnica flexível e poderosa para armazenamento temporário de conjuntos de elementos.
- Desenvolver habilidade para implementar estruturas encadeadas com Alocação Dinâmica de Memória.
- Fazer uma reflexão visando a escolher a técnica de armazenamento mais adequada aos jogos que você está desenvolvendo.

## 5.1 Alocação Dinâmica de Memória para um conjunto de elementos

Conforme estudamos no [Capítulo 2](#), na Alocação Estática de Memória definimos previamente o tamanho máximo do conjunto e reservamos memória para todos os seus elementos. O espaço reservado não pode crescer ou diminuir ao longo da execução do programa. Mesmo se a quantidade de elementos no conjunto for menor que o seu tamanho máximo, o espaço para todos os elementos permanecerá reservado. Por exemplo, suponha que reservamos espaço para armazenar 1.000 elementos. No decorrer da execução do programa, apenas 150 elementos entram no conjunto. Mas o espaço para os outros 850 elementos permanecerá reservado durante toda a execução.

Na Alocação Dinâmica, o espaço de memória pode ser alocado *no decorrer* da execução do programa, quando for efetivamente necessário. Com a Alocação Dinâmica, podemos alocar memória para um elemento de cada vez: quando um novo elemento entrar no conjunto, reservamos memória para armazená-lo. Se

um único elemento entrou no conjunto, teremos alocado espaço para um único elemento; se entraram 150, teremos alocado espaço para 150 elementos.

<b>Definição: Alocação Dinâmica de Memória para um conjunto de elementos</b>
Na Alocação Dinâmica de Memória para um conjunto de elementos:
<ul style="list-style-type: none"><li>• Espaços de memória podem ser alocados no decorrer da execução do programa quando forem efetivamente necessários.</li><li>• É possível alocar espaço para um elemento de cada vez.</li><li>• Espaços de memória também podem ser liberados no decorrer da execução do programa quando não forem mais necessários.</li><li>• Também é possível liberar espaço de um elemento de cada vez.</li></ul>

**FIGURA 5.1** Alocação Dinâmica para um conjunto de elementos.

## 5.2 Alocação Dinâmica nas linguagens C e C++

A Figura 5.2 apresenta um conjunto de comandos das linguagens C e C++. Na linha 1 da Figura 5.2 estamos declarando duas variáveis do tipo Inteiro, denominadas X e Y. Na linha 2, declaramos duas variáveis do tipo Ponteiro para Inteiro, P1 e P2. Isso significa que P1 e P2 podem armazenar o endereço da variável X, o endereço da variável Y ou o endereço de outras variáveis do tipo Inteiro.

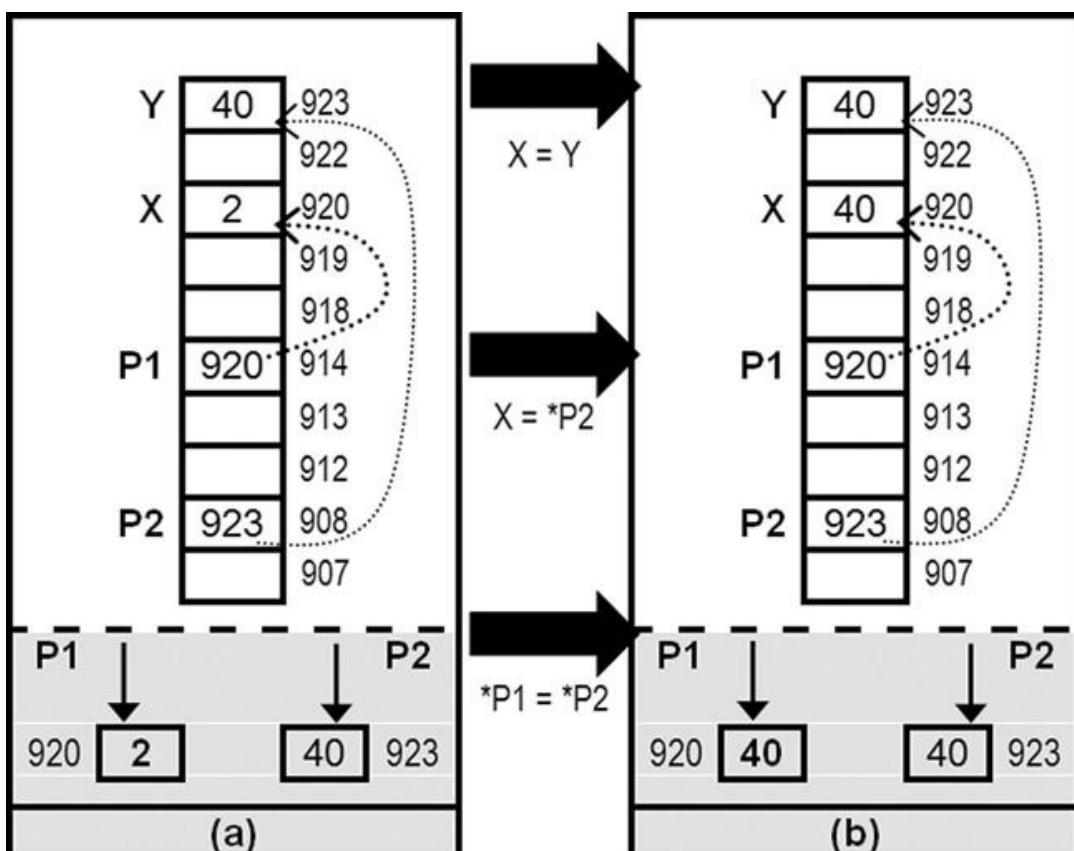
	<b>C++</b>	<b>C</b>
1	int X, Y;	int X, Y;
2	int *P1, *P2;	int *P1, *P2;
3	X = Y;	X = Y;
4	X = *P2;	X = *P2;
5	*P1 = *P2;	*P1 = *P2;
6	P1 = P2;	P1 = P2;
7	P1 = &X;	P1 = &X;
8	P1 = new int;	P1 = (int *) malloc( (unsigned) (sizeof(int)) );
9	delete P1;	free( ( char * ) P1 );

**FIGURA 5.2** Comandos nas linguagens C e C++.

Através dos comandos das linhas 1 e 2, espaços de memória para as variáveis

X, Y, P1 e P2 são reservados através de Alocação Estática. Ou seja, os espaços de memória alocados para X, Y, P1 e P2 permanecem reservados durante toda a execução do programa ou módulo em que esses comandos estiverem inseridos.

Os diagramas da [Figura 5.3](#) mostram uma representação de X, Y, P1 e P2 alocados na memória: a variável X está alocada na posição 920 da memória, Y na posição 923, P1 na posição 914 e P2 na posição 908. A representação da [Figura 5.3](#) considera que cada variável do tipo Inteiro ocupa duas unidades de memória, e cada variável do tipo Ponteiro para Inteiro ocupa quatro unidades de memória. Essa mesma consideração é adotada nas figuras seguintes.



**FIGURA 5.3** Representação de X, Y, P1 e P2 — execução dos



comandos  $X = Y$  ou  $X = *P2$  ou  $*P1 = *P2$ .

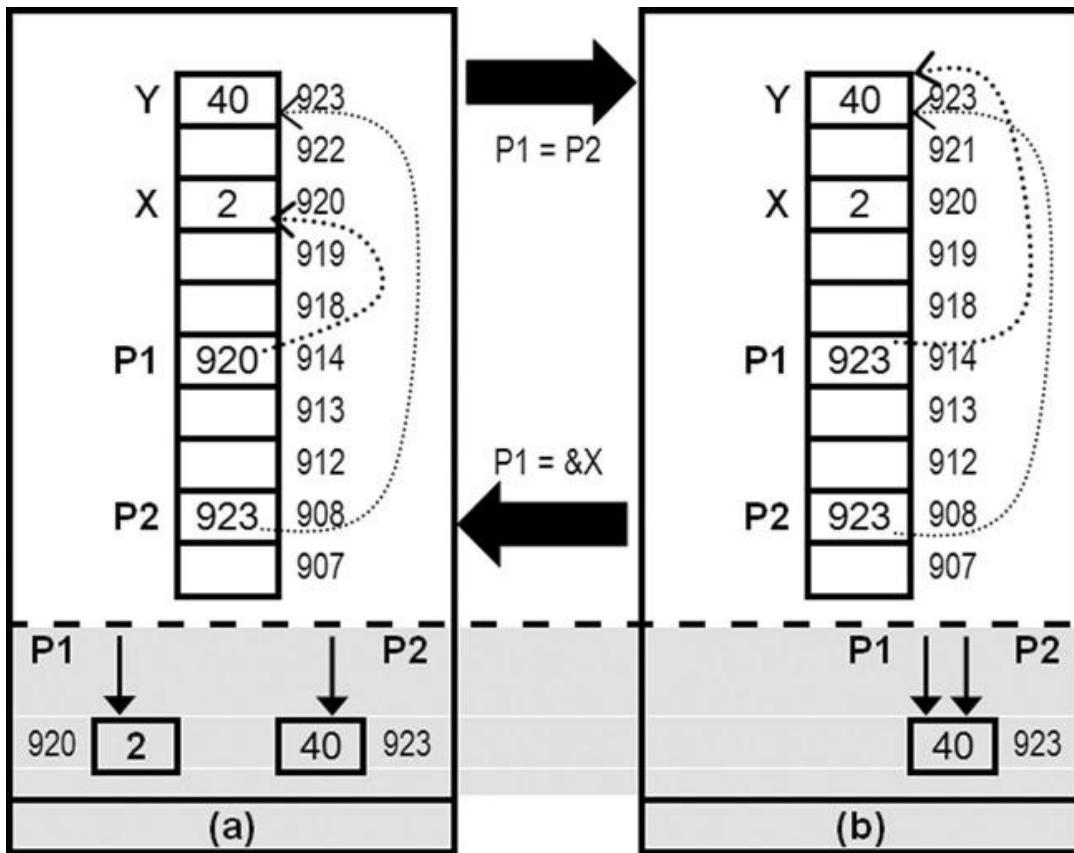
P1, que é do tipo Ponteiro para Inteiro, está armazenando o endereço de memória da variável X, que é 920; P2, também do tipo Ponteiro para Inteiro, está armazenando o endereço da variável Y, 923. Podemos interpretar que P1 e P2 “apontam” para os endereços de X e Y, respectivamente, conforme mostram

as setas pontilhadas. Na parte inferior da [Figura 5.3](#) há outra representação, destacada com o fundo cinza, que deixa essa interpretação ainda mais evidente.

A [Figura 5.3a](#) mostra uma situação inicial, e a [Figura 5.3b](#) mostra essa situação inicial modificada pela execução do comando da linha 3 da [Figura 5.2](#) ( $X = Y$ ). Com a execução desse comando, X passa a ter o valor da variável Y, que é 40. Note que o valor armazenado em P1 (920) não muda. O que muda é o valor armazenado no espaço de memória para onde P1 aponta, que armazenava o valor 2, e passa a armazenar o valor 40.

O comando da linha 4 da [Figura 5.2](#) ( $X = *P2$ ) deve ser lido como “X recebe o conteúdo apontado por P2”. Na [Figura 5.3a](#), o valor do conteúdo apontado por P2 é 40. Assim, se executado sobre a situação da [Figura 5.3a](#), o comando  $X = *P2$  resultaria na situação da [Figura 5.3b](#), onde o valor da variável X é 40. O comando da linha 5 da [Figura 5.2](#) ( $*P1 = *P2$ ) pode ser lido como “o conteúdo apontado por P1 recebe o conteúdo apontado por P2”. Na [Figura 5.3a](#), o conteúdo apontado por P1 é o conteúdo de X, e o conteúdo apontado por P2 é o conteúdo de Y. Logo, os comandos das linhas 3 ( $X = Y$ ), 4 ( $X = *P2$ ) e 5 ( $*P1 = *P2$ ) da [Figura 5.2](#) produzem exatamente o mesmo efeito na [Figura 5.3a](#), resultando na situação da [Figura 5.3b](#).

A [Figura 5.4](#) mostra a alteração de uma situação inicial ([Figura 5.4a](#)) pela execução do comando da linha 6 da [Figura 5.2](#) ( $P1 = P2$ ). A partir da execução dessa operação, o ponteiro P1 passa a ter o mesmo valor do ponteiro P2. Na [Figura 5.4a](#), P1 armazena o valor 920; já na [Figura 5.4b](#), P1 armazena o valor 522, que é o mesmo valor armazenado em P2. Em uma interpretação mais visual, P1 passa a apontar para o mesmo lugar para onde aponta P2. A representação na parte inferior da figura mostra que, com o comando  $P1 = P2$ , estamos “movendo” o ponteiro P1 para onde está o ponteiro P2.



**FIGURA 5.4** Representação de X, Y, P1 e P2 — execução dos



comandos  $P1 = P2$  e  $P1 = \&X$ .

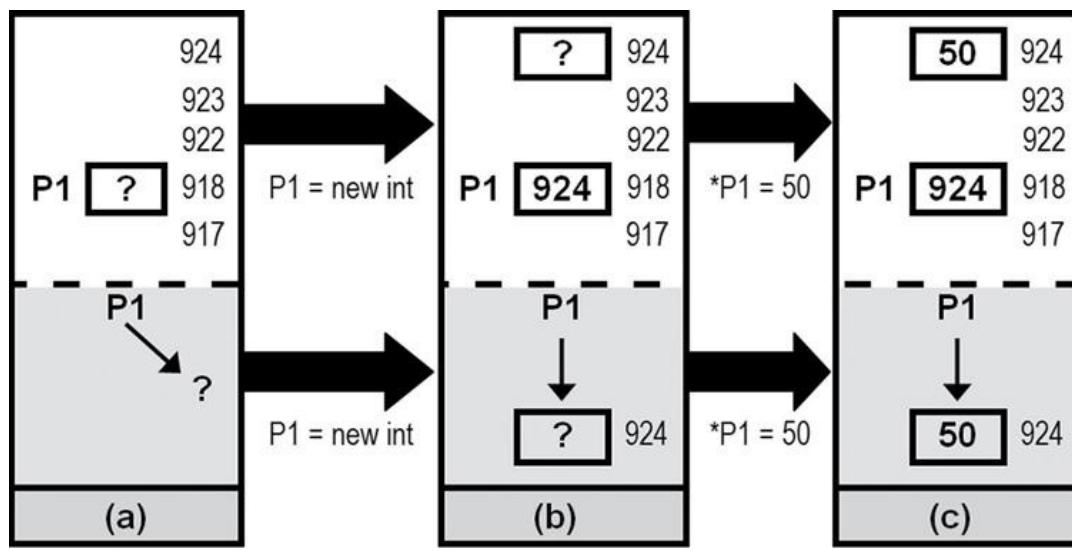
O comando da linha 7 da [Figura 5.2](#) ( $P1 = \&X$ ) pode ser lido como “P1 recebe o endereço da variável X”. Se tomarmos como ponto de partida a situação da [Figura 5.4b](#), a execução desse comando resultará na situação ilustrada na [Figura 5.4a](#). Ou seja, P1 voltará a armazenar o valor 920, que é o endereço de memória da variável X. Visualmente, estariámos movendo P1 de volta para a posição em que se encontrava anteriormente: apontando para a posição de memória 920.

Preste bastante atenção à diferença de significado entre o valor de P1 e o valor do conteúdo apontado por P1. Na situação da [Figura 5.4a](#), o valor de P1 é 920 e o valor do conteúdo apontado por P1 é 2. Interpretando essa situação: P1 está apontando para a posição de memória 920 e nessa posição apontada por P1 está sendo armazenado um Inteiro, de valor 2.

## Alocando Memória Dinamicamente

Através dos comandos das linhas 1 e 2 da [Figura 5.2](#), as variáveis X, Y, P1 e P2 foram alocadas estaticamente. A linha 8 da [Figura 5.2](#) traz comandos de Alocação Dinâmica de Memória. Em C++, o comando utilizado é **new**; em C, o comando é **malloc**.

Com a execução da linha 8 da [Figura 5.2](#) (em C++: **P1 = new int**), estamos alocando dinamicamente espaço de memória para armazenar uma variável do tipo Inteiro, e o endereço desse espaço de memória é atribuído à variável P1. Conforme mostra a [Figura 5.5](#), na situação inicial — [Figura 5.5a](#) — a variável P1 já está alocada e possui um valor qualquer, desconhecido. Ou seja, a variável P1 já existe e aponta para uma posição de memória indefinida. Após a execução do comando **P1 = new int**, um novo espaço de memória é alocado (no exemplo, posição 924), e P1 passa a apontar para essa posição de memória. Em uma representação mais visual, na parte de baixo da figura, em cinza, P1 deixou de apontar para uma posição qualquer, desconhecida, e passou a apontar para a posição 924 ([Figura 5.5b](#)).



**FIGURA 5.5** Alocando memória dinamicamente: execução dos



comandos **P1 = new int** e **\*P1 = 50**.

É possível verificar o sucesso de uma operação de alocação dinâmica de memória: se não houvesse memória disponível para ser alocada, em vez de apontar para a posição de memória recentemente alocada (924), P1 estaria apontando para NULL.

Na situação da [Figura 5.4b](#), o valor armazenado na posição 924 ainda é

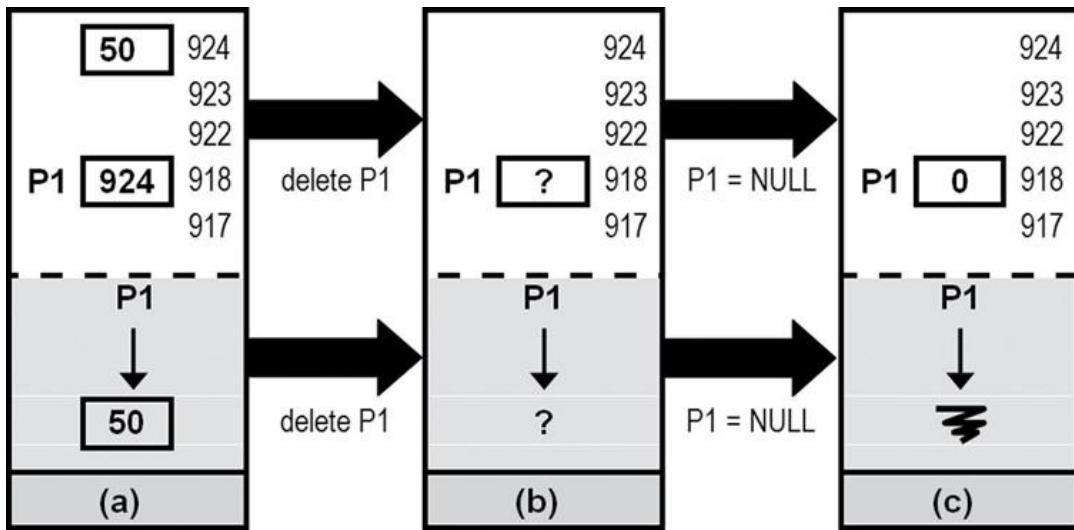
desconhecido. Para alterar essa situação podemos utilizar o comando **`*P1 = 50`**, por exemplo. A interpretação desse comando seria: “O conteúdo apontado por P1 recebe o valor 50.” A partir da execução desse comando, a posição de memória 924 passaria a armazenar o valor 50, como mostra a [Figura 5.5c](#).

Preste atenção ao seguinte ponto: P1 é uma variável do tipo Ponteiro para Inteiro. O comando `P1 = new int` aloca espaço de memória para armazenar valores do tipo Inteiro (ou seja, espaço equivalente a duas unidades de memória) e não valores do tipo Ponteiro para Inteiro (o que demandaria quatro unidades de memória).

Se olharmos para a parte superior da [Figura 5.5](#), será preciso certo grau de concentração para entender o que está acontecendo. Contudo, se observarmos a parte inferior da [Figura 5.5](#), com fundo cinza, a interpretação será muito simples: da situação *a* para a situação *b* simplesmente “aparece do nada” um espaço de memória, e P1 está apontando para ele; da situação *b* para a situação *c*, esse espaço de memória que “apareceu do nada” passa a ter um valor.

## Desalocando memória dinamicamente

Para desalocar memória dinamicamente, na linguagem C++ o comando utilizado é **`delete`**. Na linguagem C, o comando utilizado é **`free`**. Na linha 9 da [Figura 5.2](#) temos o comando (em C++) **`delete P1`**. Se tomarmos como ponto de partida a situação da [Figura 5.6a](#), o comando `delete P1` libera o espaço de memória apontado pela variável P1, resultando na situação da [Figura 5.6b](#). Note que P1 é uma variável do tipo Ponteiro para Inteiro. Ao executarmos o comando `delete P1`, estamos desalocando o espaço de memória referente a um Inteiro (duas unidades de memória).



**FIGURA 5.6** Desalocando memória dinamicamente: execução dos



comandos `delete P1` e `P1 = NULL`.

Na situação da [Figura 5.6b](#), conceitualmente não podemos mais manipular o conteúdo apontado por `P1`, pois o espaço de memória para o qual `P1` estava apontando foi desalocado. Precisamos então aplicar um comando como `P1 = NULL` (que equivale ao comando `P1 = 0`) para atribuir um valor bem definido e seguro para `P1`, resultando na situação da [Figura 5.6c](#).

Se você se concentrar na parte inferior da [Figura 5.6](#), com fundo cinza, a interpretação visual da situação será muito simples. Da situação *a* para a situação *b*, o espaço de armazenamento apontado por `P1` simplesmente “desaparece do nada”, deixando `P1` em uma situação conceitualmente confusa. Da situação *b* para a situação *c*, `P1` passa a apontar para uma posição bem definida, conhecida como `NULL`.

### 5.3 Nós de uma Lista Encadeada alocados dinamicamente

Até o momento utilizamos variáveis do tipo Inteiro e do tipo Ponteiro para Inteiro para ilustrar a Alocação Dinâmica de Memória e a manipulação de valores armazenados em espaços de memória alocados dinamicamente. Para alocar dinamicamente Nós de uma Lista Encadeada, em vez de Inteiros e Ponteiros para Inteiros, basta trabalharmos, analogamente, com Nós e Ponteiros para Nós.

Na [Figura 5.7](#) ilustramos na linguagem de programação C++ as definições

conceituais que fizemos no [Capítulo 4](#) ([Figura 4.4](#)). Na linha 1 definimos o tipo Node, e na linha 2, o tipo NodePtr.

1	struct Node { char Info; struct node *Next; };
2	typedef struct Node *NodePtr;
3	NodePtr P; PAux;
4	Int X;
5	P = new Node;
6	P->Info = 50;
7	P->Next = NULL;
8	X = P->Info;
9	PAux = P->Next;
10	Delete P;
11	P = NULL;

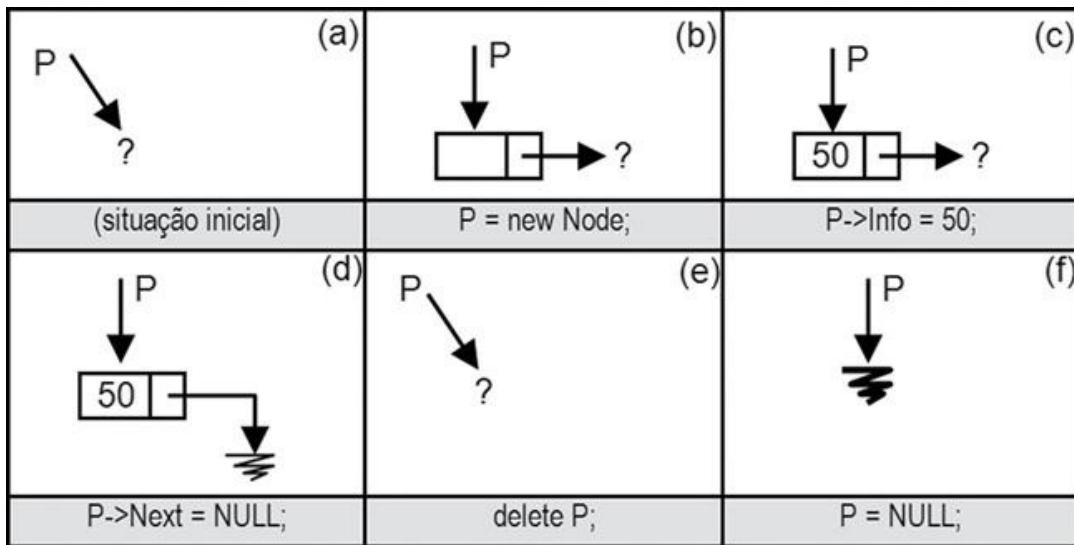
**FIGURA 5.7** Implementação de Listas Encadeadas com alocação



dinâmica: comandos em C++.

Cada Nó da Lista Encadeada possui os campos Info (utilizado para armazenar informação) e Next (utilizado para apontar o próximo elemento da lista). Conforme mencionamos no [Capítulo 4](#), essa nomenclatura foi definida de modo a manter certa compatibilidade com a adotada em parte da literatura sobre estruturas de dados (como [Drozdek \[2002\]](#) e também [Langsam, Augenstein e Tenenbaum \[1996\]](#)).

Na linha 3 declaramos as variáveis P e PAux do tipo Ponteiro para Nó, e na linha 4 declaramos X do tipo inteiro. Na linha 5 encontramos o comando **P = new Node**, que implementa a operação conceitual **P = NewNode** que utilizamos nos algoritmos do [Capítulo 4](#) (veja a [Figura 4.4](#)). Através do comando P = new Node estamos alocando dinamicamente espaço de memória para armazenar valores do tipo Node. Ou seja, estamos alocando um Nó da Lista Encadeada, e a variável P estará apontando para esse Nó. Note que, na situação da [Figura 5.8a](#), simplesmente “aparece do nada” um novo Nó, e a variável P passa a apontar para esse novo Nó ([Figura 5.8b](#)).



**FIGURA 5.8** Manipulando uma Lista Encadeada alocada dinamicamente: comandos em C++.

O comando da linha 6 da [Figura 5.7](#), **P- > Info = 50** pode ser lido da forma: “A porção Info do Nó apontado por P recebe o valor 50.” Um modo alternativo de realizar essa leitura seria: “Info de P recebe 50.” Na situação da [Figura 5.8b](#), o Nó apontado por P ainda não possui valor em seus campos Info e Next. Após executarmos o comando **P- > Info = 50**, teremos como resultado a situação da [Figura 5.8c](#), na qual o campo Info do Nó apontado por P possui valor 50. Analogamente, se executarmos em seguida o comando da linha 7, **P- > Next = NULL**, chegaremos à situação da [Figura 5.8d](#). O acesso aos campos Info e Next de um Nó apontado por P é exemplificado pelas linhas 8 e 9 da [Figura 5.7](#).

Podemos exemplificar a operação de desalocar um Nó da Lista Encadeada com o comando da linha 10 da [Figura 5.7](#): **delete P**, que levaria a situação da [Figura 5.8d](#) à situação da [Figura 5.8e](#). O Nó apontado por P simplesmente “desaparece”! O comando **delete P** implementa a operação conceitual **DeleteNode (P)**, que utilizamos nos algoritmos do [Capítulo 4](#) (veja a [Figura 4.4](#)).

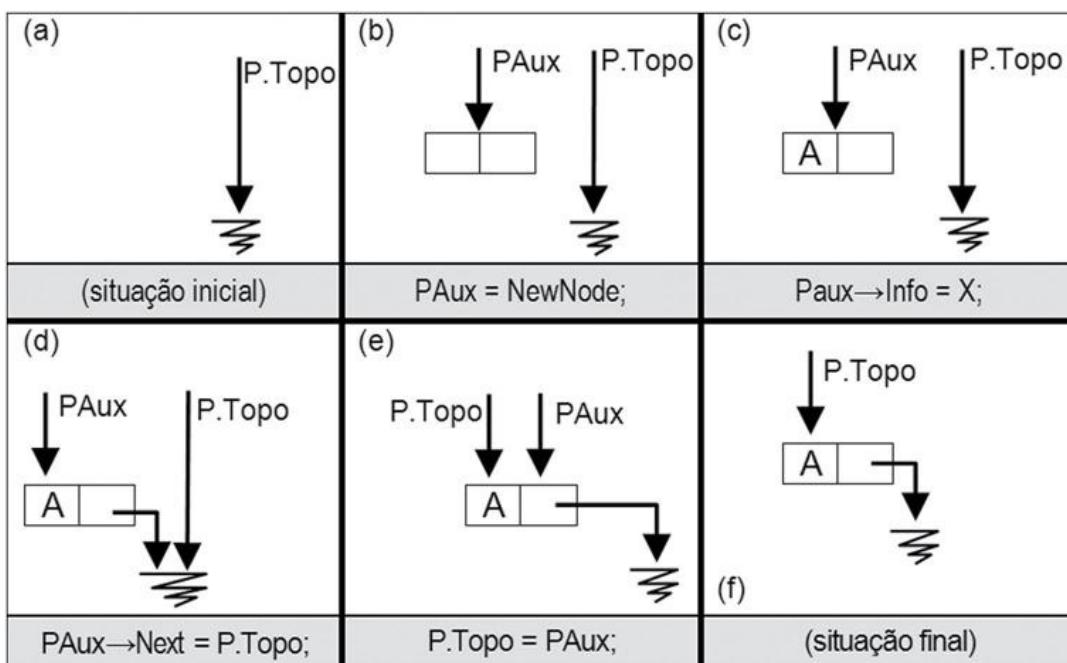
Finalmente, para não deixar o ponteiro P apontando para uma posição indefinida, aplicamos o comando **P = NULL** (linha 11 da [Figura 5.7](#)) e chegamos à situação da [Figura 5.8f](#).

### Dica importante: desenhe!

Ao elaborar e testar algoritmos sobre Listas Encadeadas, utilize sempre um conjunto de diagramas. Desenhe a execução do seu algoritmo! Desenhe passo a passo! A representação visual simplifica a compreensão e evita erros.

## Exercício 5.1 Revisar comandos da Operação Empilha

No [Capítulo 4](#) implementamos uma Pilha como uma Lista Encadeada e utilizamos uma notação conceitual para a elaboração dos algoritmos. A [Figura 5.9](#) é uma reprodução da [Figura 4.11](#), e suas imagens ilustram a execução passo a passo da operação Empilha, para uma situação inicial com a Pilha vazia. Implemente em C++ cada um dos comandos expressos em linguagem conceitual na parte cinza de cada diagrama. Consulte a [Figura 5.7](#) se tiver dúvidas.



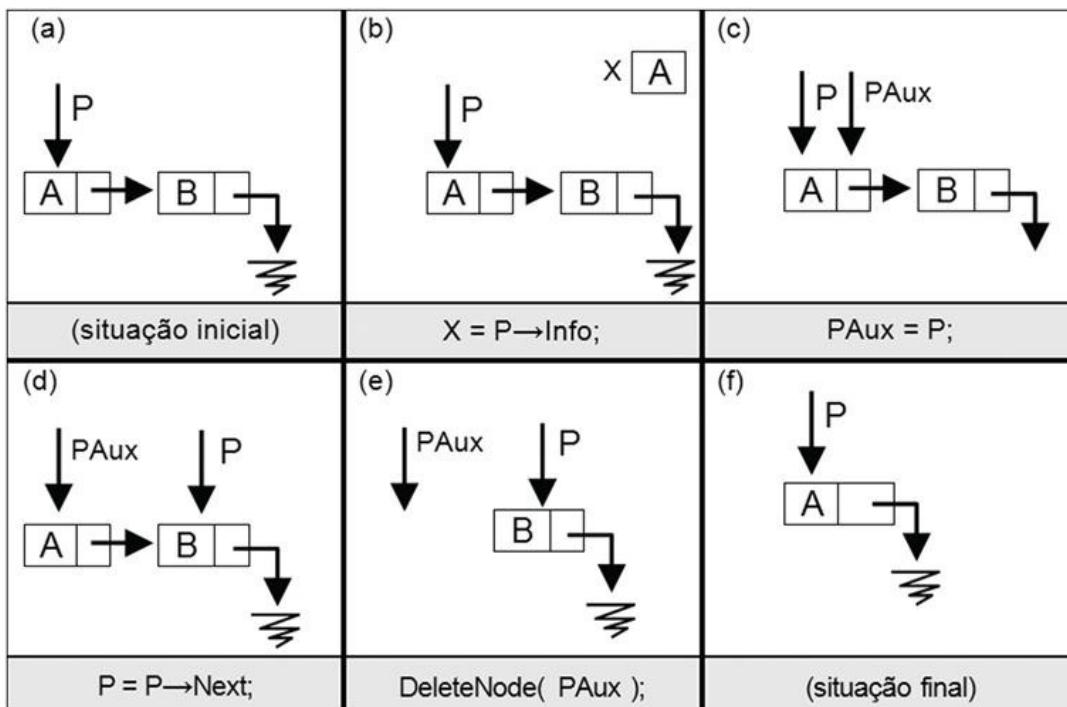
**FIGURA 5.9** Operação Empilha partindo de Pilha Vazia.

Na linguagem C++, a implementação do comando conceitual `PAux = NewNode` é `PAux = new Node` ([Figura 5.9b](#)). O comando conceitual da [Figura 5.9c](#), `Paux → Info = X` pode ser implementado por `PAux->Info = X`. `PAux → Next = P.Topo` ([Figura 5.9d](#)) pode ser implementado por `PAux->Next = P.Topo`. Na [Figura 5.9e](#), a notação conceitual é idêntica à implementação em C++, ou seja: `P.Topo = PAux`.

## Exercício 5.2 Revisar comandos da Operação Desempilha

No [Capítulo 4](#), implementamos uma Pilha como uma Lista Encadeada e utilizamos uma notação conceitual para a elaboração dos algoritmos. A [Figura](#)

5.10 é uma reprodução da Figura 4.13, e suas imagens ilustram a execução passo a passo da operação Desempilha, para uma situação inicial com a Pilha com dois elementos. Implemente em C++ cada um dos comandos expressos em linguagem conceitual na parte cinza de cada diagrama. Consulte a Figura 5.7 se tiver dúvidas.



**FIGURA 5.10** Operação Desempilha — situação inicial de Pilha com dois elementos.

### Exercício 5.3 Implemente uma Pilha em C++ com Alocação Encadeada e Dinâmica de Memória

No Capítulo 4, implementamos uma Pilha como uma Lista Encadeada e utilizamos uma notação conceitual para a elaboração dos algoritmos. Implemente, na linguagem de programação C++, uma Pilha como uma Lista Encadeada e com Alocação Dinâmica de Memória. Implemente as Operações Primitivas Empilha, Desempilha, Cria, Vazia e Cheia. Tome como base os algoritmos elaborados nos Exercícios 4.1 a 4.6 (Figuras 4.10 e 4.12). Em um arquivo separado, faça um programa para testar o funcionamento das operações da Pilha.

## **Exercício 5.4 Implemente uma Fila em C++ com Alocação Encadeada e Dinâmica de Memória**

No [Capítulo 4](#) implementamos uma Fila como uma Lista Encadeada e utilizamos uma notação conceitual para a elaboração dos algoritmos. Implemente, na linguagem de programação C++, uma Fila como uma Lista Encadeada e com Alocação Dinâmica de Memória. Implemente as Operações Primitivas Retira, Insere, Cria, Vazia e Cheia. Tome como base a especificação e os algoritmos elaborados nos [Exercícios 4.8](#) a [4.12](#). Em um arquivo separado, faça um programa para testar o funcionamento das operações da Fila.

## **5.4 Alocação Sequencial e Estática ou Encadeada e Dinâmica?**

Nos [Capítulos 2 e 3](#), utilizamos o conceito de Alocação Sequencial juntamente com o conceito de Alocação Estática para implementar estruturas do tipo Pilha e Fila. Alocação Sequencial é um conceito independente do conceito de Alocação Estática. Os conceitos foram combinados.

Analogamente, a Alocação Encadeada de Memória é um conceito independente do conceito de Alocação Dinâmica de Memória. Mas são conceitos que se encaixam perfeitamente conforme mostram as implementações discutidas nos [Capítulos 4 e 5](#).

## **Comparando Sequencial-Estática com Encadeada-Dinâmica**

Na técnica de Alocação Sequencial e Estática, utilizamos um vetor e indicamos previamente a quantidade máxima de elementos que podem entrar no conjunto. Reservamos memória para a quantidade máxima de elementos, mesmo que na prática o número de elementos seja menor.

Na técnica de Alocação Encadeada e Dinâmica, não é necessário indicar previamente quantos elementos poderão fazer parte do conjunto. Novos elementos podem ser agregados ao conjunto na medida da necessidade. Na prática, na Alocação Encadeada e Dinâmica, o limite para o crescimento de um conjunto é a disponibilidade de memória do computador ou dispositivo em que o programa está sendo executado. Assim, a Alocação Encadeada e Dinâmica é uma técnica bastante flexível em relação ao número de elementos e possibilita que a memória seja compartilhada com eficiência entre diversas estruturas de

armazenamento.

Outra característica da Alocação Encadeada e Dinâmica é a facilidade para modelar diferentes situações. É bastante simples ajustar uma lista encadeada para que seja circular ou não, para que tenha um indicador para o final da lista ou não, e assim por diante. Nos próximos capítulos implementaremos diversas outras variações das Listas Encadeadas, evidenciando ainda mais sua flexibilidade.

Para uma leitura complementar sobre alocação de memória no contexto de conjuntos de elementos, consulte [Pereira \(1996, p. 9-14, 83-93\)](#), [Langsam, Augenstein e Tenenbaum \(1996, p. 203-207\]\)](#), e [Celes, Cerqueira e Rangel \(2004, p. 64-72\)](#).

### **Exercício 5.5 Avanço de Projeto: refletir sobre a aplicação FreeCell e definir a técnica mais adequada para implementação da Pilha**

Você já implementou uma Pilha com duas técnicas diferentes: com Alocação Sequencial e Estática ([Capítulo 2](#)) e com Alocação Encadeada e Dinâmica ([Capítulos 4 e 5](#)). Considerando a aplicação em que será utilizada a Pilha, qual dessas duas técnicas de implementação você considera mais adequada?

### **Exercício 5.6 Avanço de Projeto: refletir sobre a aplicação Snake e definir a técnica mais adequada para implementação da Fila**

Você já implementou uma Fila com duas técnicas diferentes: com Alocação Sequencial e Estática ([Capítulo 3](#)) e com Alocação Encadeada e Dinâmica ([Capítulos 4 e 5](#)). Considerando a aplicação em que será utilizada a Fila, qual dessas duas técnicas de implementação você considera mais adequada?

## **Comparação entre Alocação Sequencial e Estática e Alocação Encadeada e Dinâmica**

A Alocação Sequencial e Estática é uma técnica simples e adequada a situações em que a quantidade de elementos que poderão entrar no conjunto é previsível, com pequena margem de variação. A Alocação Encadeada e Dinâmica é flexível com relação à quantidade de elementos e pode ser facilmente adaptada para modelar diferentes necessidades; é uma técnica poderosa e muito utilizada para o armazenamento temporário de conjuntos de

elementos.

## Consulte nos Materiais Complementares

Vídeos sobre Alocação Dinâmica nas Linguagens C e C++



<http://www.elsevier.com.br/edcomjogos>

## Exercícios de fixação

**Exercício 5.7 Diferença entre Alocação Estática e Alocação Dinâmica de Memória.** Qual a diferença entre Alocação Estática e Alocação Dinâmica de Memória, no contexto do armazenamento temporário de conjuntos de elementos?

**Exercício 5.8 Diferença entre Alocação Sequencial e Alocação Encadeada de Memória.** Qual a diferença entre Alocação Sequencial e Alocação Encadeada de Memória, no contexto do armazenamento temporário de conjuntos de elementos?

**Exercício 5.9 Conceitos independentes.** Explique por que a Alocação Encadeada e a Alocação Dinâmica são conceitos independentes que, quando combinados, formam uma técnica poderosa.

**Exercício 5.10 Diagrama da implementação de Pilha — Alocação Sequencial e Estática de Memória.** Como seria implementar uma Pilha com Alocação Sequencial e Estática de Memória? Faça um diagrama e explique resumidamente o funcionamento dessa técnica de implementação.

**Exercício 5.11 Diagrama da implementação de Pilha — Alocação Encadeada e Dinâmica de Memória.** Como seria implementar uma Pilha com Alocação Encadeada e Dinâmica de Memória? Faça um diagrama e explique resumidamente o funcionamento dessa técnica de implementação.

**Exercício 5.12 Vantagens e desvantagens.** Quais são as vantagens e desvantagens das técnicas de Alocação Sequencial e Estática e Alocação Encadeada e Dinâmica? Sugestão de uso acadêmico: discuta com os colegas as vantagens e desvantagens dessas técnicas.

**Exercício 5.13 Implemente uma classe Node em C++.** Revise a definição

dos tipos Node e NodePtr realizada na [Figura 5.7](#) e implemente uma classe Node. A classe Node deve definir um Nó com os campos Info e Next, e o tipo NodePtr. Defina e implemente as operações: NewNode (para alocar um Nó), DeleteNode (liberar um Nó), GetInfo (acesso ao valor da informação armazenada no Nó), SetInfo (atualização da informação armazenada no Nó), GetNext (acesso à indicação do próximo elemento da lista, armazenada no Nó) e SetNext (atualização da indicação do próximo elemento da lista, armazenada no Nó). Altere a solução do [Exercício 5.3](#) ou do [Exercício 5.4](#), para que utilize a classe Node. Realize alguns testes para se certificar do correto funcionamento.

**Exercício 5.14 Testar e aprimorar a portabilidade e a reusabilidade das soluções com Pilha.** No [Exercício 5.3](#), você implementou um TAD Pilha com Alocação Encadeada e Dinâmica e, em um arquivo separado, fez um Programa Teste, para verificar o funcionamento do TAD Pilha. Avalie o grau de portabilidade e o potencial de reusabilidade de sua solução da seguinte forma: pegue o mesmo Programa Teste que utilizou para avaliar o funcionamento do TAD Pilha implementado com Alocação Encadeada e Dinâmica, e utilize para testar o TAD Pilha que você implementou nos [Exercícios 2.12](#) ou [2.13](#), com Alocação Sequencial e Estática. Idealmente, o seu Programa Teste deve executar com ambas as versões do TAD Pilha, sem necessidade de qualquer alteração. Se, ao trocar a implementação da Pilha, você tiver que fazer alguma alteração no Programa Teste, as soluções devem ser aprimoradas até que o Programa Teste execute com ambas as implementações de Pilha, sem necessidade de qualquer alteração.

**Exercício 5.15 Testar e aprimorar a portabilidade e a reusabilidade das soluções com Fila.** Analogamente ao Exercício 5.14, teste e aprimore a portabilidade das soluções com Fila.

**Exercício 5.16 Avanço de Projeto: avaliar a portabilidade das soluções com Pilha e Fila de seus jogos.** Você desenvolveu ou está desenvolvendo jogos que utilizam Pilhas e Filas. Suas soluções estão adequadamente portáveis e reutilizáveis? Analogamente ao realizado nos Exercícios 5.14 e 5.15, avalie o grau de portabilidade e o potencial para reusabilidade das soluções com Pilhas e Filas em seus jogos. Aprimore as soluções onde necessário.

**Exercício 5.17 Conclusões sobre portabilidade e reusabilidade.** Que conclusões adicionais sobre portabilidade e reusabilidade de software você tirou após a realização dos Exercícios 5.14, 5.15 e 5.16?

## Soluções para alguns dos exercícios

### Exercício 5.2 Revisar comandos da Operação Desempilha

- b) X = P->Info;
- c) PAux = P;
- d) P = P->Next;
- e) delete PAux;

### Exercício 5.4 Fila Encadeada e Dinâmica em C++

```

/* arquivo FilaEncDinamc.h - implementa um TAD Fila */
#include<conio.h>
#include<stdio.h>

struct Node {
    char Info;
    Node *Next;
};

typedef struct Node *NodePtr;

struct Fila{
    NodePtr Primeiro;
    NodePtr Ultimo;
};

void Cria(Fila *F){
    F->Primeiro = NULL;
    F->Ultimo = NULL;
}

bool Vazia(Fila *F){
    if(F->Primeiro ==NULL) // ou F->Ultimo==NULL
        return true;
    else
        return false;
}

bool Cheia(Fila *F){
    return false; // veja teste na operação Insere, após alocação de memória
}

void Insere(Fila *F, char X, bool *DeuCerto){
    NodePtr PAux = new Node;
    if (PAux == NULL)
        *DeuCerto = false; // se PAux retornar NULL, não há mais memória - fila cheia
    else { *DeuCerto = true;
        PAux->Info = X;
        PAux->Next = NULL;
        if(Vazia(F))
            F->Primeiro = PAux;
        else F->Ultimo->Next = PAux;
        F->Ultimo = PAux;
    } // else
} // Insere

void Retira(Fila *F, char *X, bool *DeuCerto){
    NodePtr PAux;
    if(Vazia(F)) {
        DeuCerto=false;
    } else { *DeuCerto = true;
        *X = F->Primeiro->Info;
        PAux = F->Primeiro;
        F->Primeiro = F->Primeiro->Next;
        if(F->Primeiro==NULL){ // a fila ficara vazia
            F->Ultimo = NULL;
            delete(PAux);
        } // else
    } // retira
}

```



```

void Destroi(Fila *F){ // remove todos os nós da Fila
    char X;
    bool Ok;
    while (Vazia(F)==false) {
        Retira(F, &X, &Ok);
    } // while
} // Destroi

/* arquivo FilaEncDinamc.cpp - testa o TAD Fila */
#include "FilaEncDinamc.h"
#include<iostream>

using namespace std;

void Imprime(Fila *F){ // imprime sem abrir a TV
    char X;
    Fila *FAux = new Fila;
    bool Ok;
    Cria(FAux);
    while (Vazia(F)==false) {
        Retira(F, &X, &Ok);
        if (Ok) {
            Insere(FAux, X, &Ok);
        } // if
    } // while
    printf("\n F.Primeiro --> ");
    while (Vazia(FAux)==false) {
        Retira(FAux, &X, &Ok);
        if (Ok){
            printf("%c ", X);
            Insere(F, X, &Ok);
        } // if
    } // while
    printf(" <- F.Ultimo \n");
}

int main(){
    Fila *F = new Fila;
    Cria(F);
    bool Ok;
    char Valor;
    char Op = 't';
    while (Op != 's') {
        cout << "digite: (i)inserir,(r)retirar, (s)sair [enter]" << endl;
        cin >> Op;
        switch (Op) {
            case 'i' : cout << "digite um UNICO CARACTER para inserir [enter]" << endl;
                        cin >> Valor;
                        Insere(F,Valor,&Ok);
                        if (Ok==true) cout << "valor inserido" << endl;
                        else cout << "nao conseguiu inserir" << endl;
                        break;
            case 'r' : Retira (F,&Valor,&Ok);
                        if (Ok==true) cout << "valor retirado= " << Valor << endl;
                        else cout << "nao conseguiu retirar" << endl;
                        break;
            default : cout << "saindo... " << endl; Op = 's'; break;
        }; // case
    Imprime (F);
}

```

```
    } // while
    Destroi(F);      // retira todos os elementos da fila
    cout <<"a fila apos a operacao destroi... " << endl;
    Imprime(F);
    cout <<"pressione uma tecla... " << endl;
    getch();
    return(0);
}
```

## Referências e leitura adicional

1. Celes W, Cerqueira R, Rangel JL. *Introdução a estruturas de dados*. Rio de Janeiro: Elsevier; 2004.
2. Drozdek A. *Estrutura de dados e algoritmos em C++*. São Paulo: Thomson; 2002.
3. Langsam Y, Augenstein MJ, Tenenbaum AM. *Data Structures Using C and C++*. 2nd ed. New Jersey: Prentice Hall; 1996; Upper Saddle River.
4. Pereira SL. *Estruturas de dados fundamentais: conceitos e aplicações*. São Paulo: Érica; 1996.

---

# **PARTE III**

## Listas Cadastrais

### **OUTLINE**

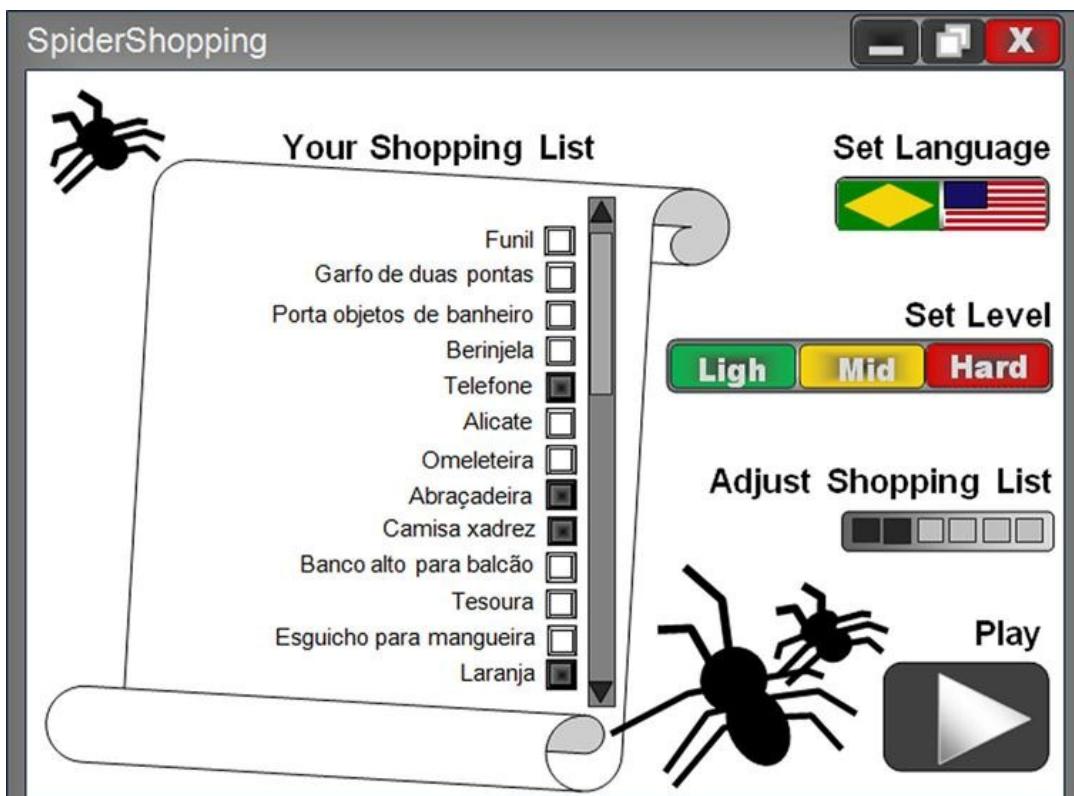
Desafio 3 Desenvolver um jogo que use listas de elementos

Capítulo 6 Listas Cadastrais

Capítulo 7 Generalização de Listas Encadeadas

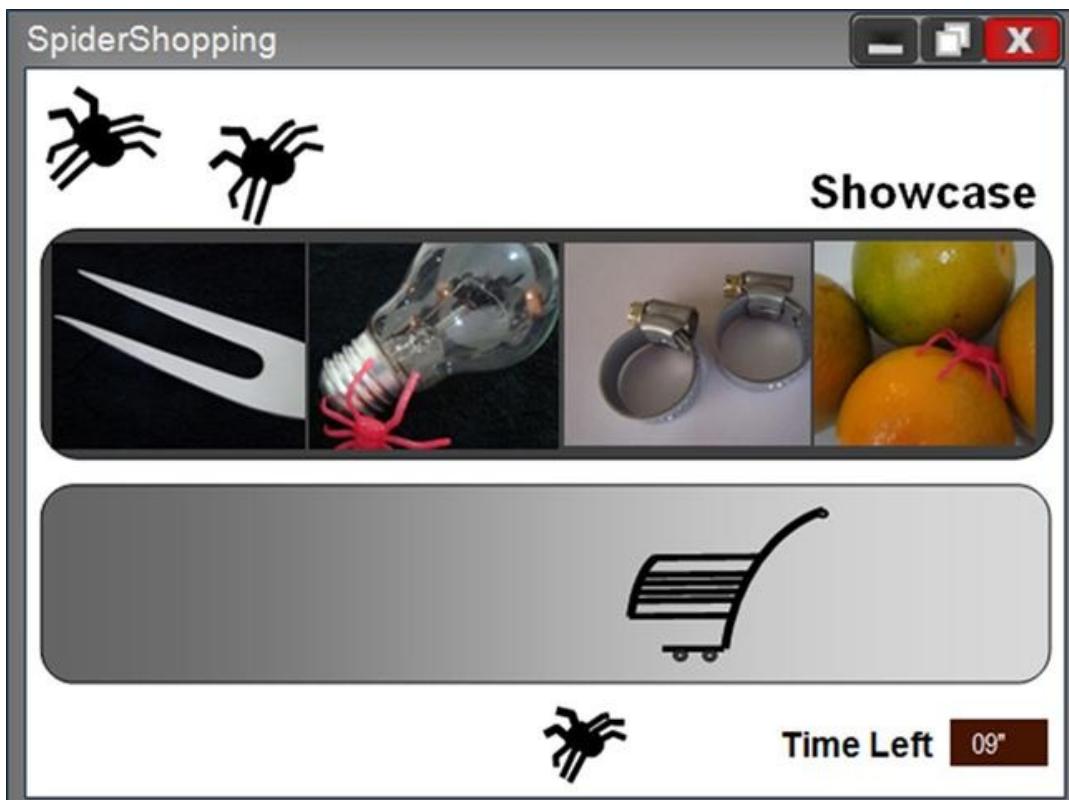
## DESAFIO 3

# Desenvolver um jogo que use listas de elementos



**FIGURA D3.1** Spider Shopping: o jogador recebe uma lista de compras e tenta identificar os produtos em uma vitrine.

O *Spider Shopping* é um game em que o jogador recebe uma lista de compras e precisa comprar, um a um, os itens da lista. As compras são feitas de modo visual, em uma vitrine. Para comprar o item certo, é preciso identificar a imagem do produto e lembrar que esse produto faz parte da lista de compras.



**FIGURA D3.2** Produtos que fazem parte da lista de compras devem ser identificados em uma vitrine.

A cada item comprado corretamente, o jogador ganha pontos. O jogador perde pontos se comprar algum item que não faz parte da lista ou se comprar um mesmo item mais de uma vez. Se o jogador escolher um nível de dificuldade mais alto, as imagens aparecerão na vitrine mais rapidamente. Para se dar bem no jogo, é preciso memorização e agilidade, para associar instantaneamente uma imagem a um item da lista.

Alguns dos itens da vitrine podem ser exóticos, e é possível que o jogador nem saiba o nome. Por isso é possível fazer alguns ajustes na lista antes de ir às compras, retirando ou acrescentando alguns dos elementos. Ao final da sessão de compras, o jogador poderá ver a lista de compras e também os produtos do carrinho de compras. Às vezes aparecem algumas aranhas para distrair o jogador.

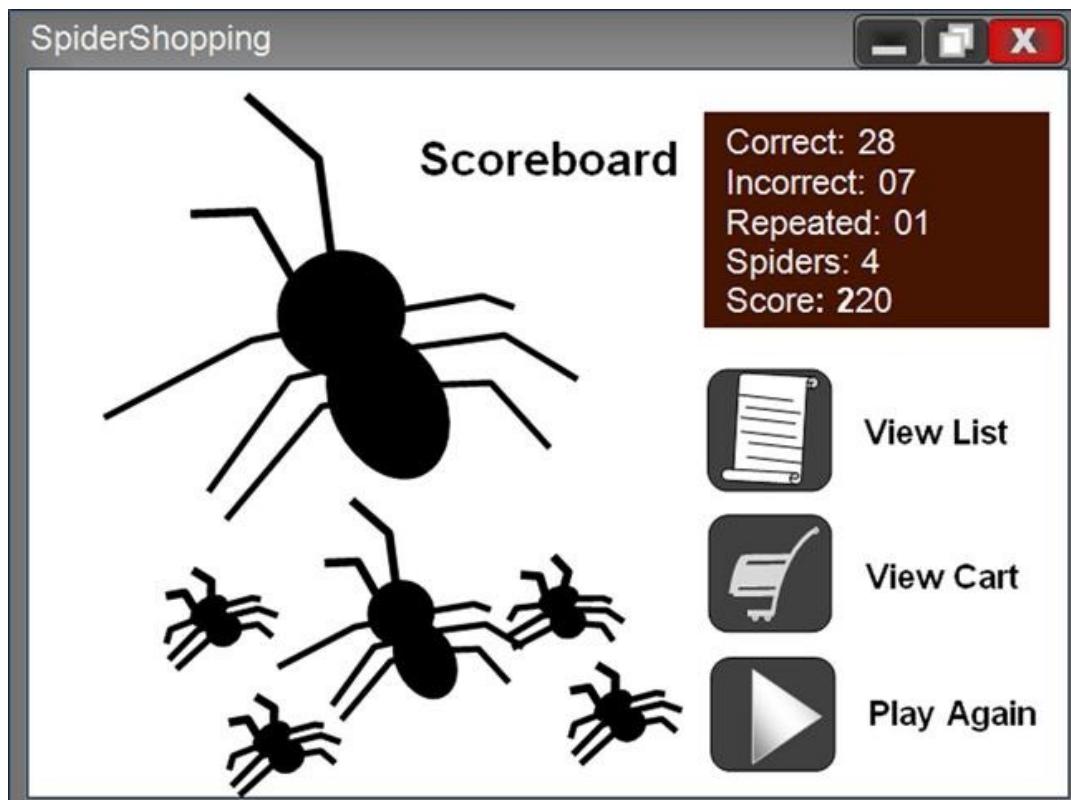
## Como implementar a Lista de Compras?

Se você fosse desenvolver um jogo como o *Spider Shopping*, como faria para implementar a Lista de Compras? Como faria para possibilitar ao usuário

acrescentar e retirar itens da lista inicial? Como você faria para saber se um produto comprado está ou não na Lista de Compras? Como você faria para saber se o usuário comprou um mesmo produto uma, duas, três ou mais vezes?

## Seu desafio: desenvolver um jogo que use listas

Seu desafio é desenvolver uma adaptação do *Spider Shopping*. Ajuste algumas de suas características e dê uma nova cara ao jogo! Você pode até inventar um jogo totalmente novo, mas mantenha as seguintes características: um jogo que utilize uma ou mais listas de elementos, como listas de compras, listas de pessoas, de animais ou de outros elementos. O jogo deve permitir que elementos sejam acrescentados e retirados das listas. Deve ser significativo para o jogo verificar se um elemento faz parte ou não de determinada lista.



**FIGURA D3.3** O jogador perde pontos se comprar um item que não faz parte da Lista de Compras.

## Por onde começar?

Os Capítulos 6 e 7 de *Estruturas de dados com jogos* o ajudarão a definir um modo interessante de armazenar e manipular listas de elementos. Após estudar esses capítulos, você terá uma boa orientação para conceber e implementar sua adaptação do *Spider Shopping* e de outros jogos com as características essenciais propostas para o Desafio 3.

Construa seu próprio jogo! Um jogo com a sua cara! Em vez de um *Spider Shopping*, crie um “*Sua Cara*” *Shopping*! Ou um jogo totalmente novo, com as características indicadas como essenciais. Use sua criatividade! Aprender a programar pode ser divertido!

### Consulte nos Materiais Complementares



Banco de Jogos: Aplicações de Listas



<http://www.elsevier.com.br/edcomjogos>

---

## CAPÍTULO 6

---

# Listas Cadastrais

---

## Seus objetivos neste capítulo

- Entender o que é e para que serve uma estrutura do tipo Lista Cadastral.
- Desenvolver habilidade para manipular Listas Cadastrais através de seus operadores primitivos.
- Ganhar experiência na elaboração de algoritmos sobre listas encadeadas, implementando Listas Cadastrais como listas encadeadas ordenadas, listas não ordenadas, listas circulares, listas com elementos repetidos e outras variações.
- Iniciar o desenvolvimento do seu jogo referente ao Desafio 3.

## 6.1 O que é uma Lista Cadastral?

Quando retiramos um elemento de uma Fila, retiramos sempre o primeiro elemento, independentemente do seu valor. O critério de retirada é exatamente a posição do elemento no conjunto: só podemos retirar o primeiro elemento da Fila. O mesmo ocorre com estruturas do tipo Pilha: só podemos retirar o elemento que está no topo da Pilha.

Essa lógica de retirada dos elementos de uma Fila e de uma Pilha não é adequada a situações nas quais precisamos retirar um elemento do conjunto, não por sua posição, mas pelo seu valor. Imagine que uma empresa mantenha um Cadastro de Funcionários — um grande arquivo de aço, contendo uma pastinha para cada funcionário. Então, um funcionário chamado *Moacir* decide trabalhar em outra empresa, e o Cadastro de Funcionários deve ser atualizado.

Não queremos retirar a primeira pastinha do Cadastro; queremos retirar, especificamente, a pastinha do *Moacir*. Não importa a posição da pastinha do *Moacir* no arquivo: esteja ela no começo, no meio ou no final, é a pasta do *Moacir* que queremos retirar do Cadastro. O critério de retirada não é a posição do elemento no conjunto, mas o valor do elemento.

Um exemplo do mundo dos games: no *Spider Shopping* — game do Desafio

3, temos uma Lista de Compras. Antes de ir às compras, o jogador pode retirar da Lista alguns dos itens que ele não conhece. Por exemplo, suponha que o jogador não saiba bem o que seja um *Esguicho* e decide retirar esse item da Lista de Compras.



**Atualizando um Cadastro de Funcionários**

Se o funcionário de nome *Moacir* deve ser desligado da empresa:

- Procuramos a pastinha do *Moacir*.
- Retiramos, especificamente, a pastinha que guarda os dados do *Moacir* — esteja ela onde estiver.

**FIGURA 6.1** Atualizando um Cadastro de Funcionários.

	<b>Atualizando uma Lista de Compras</b>
Ywz Xx Yz <input type="checkbox"/>	
Xxxxxx XX xxx <input type="checkbox"/>	
<b>Esguicho <input checked="" type="checkbox"/> ✓</b>	
xXwz Ywx xwz <input type="checkbox"/>	
ZZx XwX Zzz <input type="checkbox"/>	

Se queremos retirar o item *Esguicho* da Lista:

- Procuramos na Lista o item cujo valor é *Esguicho*.
- Retiramos da Lista, especificamente, o item cujo valor é *Esguicho* — esteja esse item no começo, no meio ou no final da Lista.

**FIGURA 6.2** Atualizando uma Lista de Compras.

Para atualizar a Lista de Compras, retiramos não o primeiro elemento da Lista (como faríamos em uma Fila). Retiramos, especificamente, o elemento cujo

valor é *Esguicho*. Não importa a posição do item na Lista: esteja ele no começo, no meio ou no fim, é o elemento de valor *Esguicho* que queremos retirar. Assim como no exemplo do Cadastro de Funcionários, o critério de retirada não é a posição do elemento no conjunto, mas o valor do elemento.

O Cadastro de Funcionários e a Lista de Compras exemplificam uma estrutura de armazenamento que denominamos Lista Cadastral. Em uma Lista Cadastral, o ingresso, a retirada e o acesso aos elementos do conjunto ocorrem em função do valor dos elementos, e não em função da posição dos elementos no conjunto. Por exemplo, a entrada de um novo elemento de valor X no conjunto pode ser rejeitada se no conjunto já houver um elemento cujo valor é X. Um segundo exemplo: a retirada de um elemento de valor Y pode não ser realizada caso não seja encontrado no conjunto um elemento de valor Y.

## 6.2 Operações de um TAD Lista Cadastral

A [Figura 6.4](#) especifica as Operações Primitivas do Tipo Abstrato de Dados (TAD) — Lista Cadastral sem elementos repetidos. A operação *EstáNaLista* verifica se o valor X faz parte da Lista L, tendo como resultados os valores Verdadeiro (indicando que o valor está na Lista) ou Falso (indicando que não está).

### Definição: Lista Cadastral

Em uma estrutura de armazenamento denominada Lista Cadastral, a inserção, a retirada e o acesso aos elementos do conjunto ocorrem em função do valor dos elementos, e não em função da posição dos elementos no conjunto.

### Lista Cadastral, Cadastro ou Lista?

Diversos livros utilizam o termo "*lista*" em referência a uma Lista Cadastral. Podemos, por simplificação, utilizar o termo Lista. Mas é importante não confundir a estrutura de armazenamento *Lista Cadastral* com as *Listas Encadeadas* – referência à técnica de alocação encadeada de memória, que estudamos no Capítulo 4. O termo "*cadastro*" também pode ser utilizado em referência a uma *Lista Cadastral*.



**FIGURA 6.3** Definição de Lista Cadastral.

Operações e parâmetros	Funcionamento
EstáNaLista (L,X)	Verifica se o elemento de valor X faz parte da Lista Cadastral L, retornando o valor Verdadeiro caso X estiver na Lista L e o valor Falso caso X não fizer parte de L.
Insere (L,X,Ok)	Insere o elemento de valor X na Lista L, caso a Lista L já não tiver um elemento de valor X. Caso a Lista L já tiver um elemento de valor X, nenhum elemento será inserido e, nesse caso, o parâmetro Ok deve retornar o valor Falso.
Retira(L,X,Ok)	Retira da Lista L o elemento de valor X, caso X estiver na Lista. Nesse caso, o parâmetro Ok deve retornar o valor Verdadeiro. Se X não estiver na Lista, nenhum elemento será retirado, e o parâmetro Ok retornará o valor Falso.
Vazia(L)	Verifica se a Lista Cadastral L está vazia, retornando o valor Verdadeiro para vazia e Falso caso contrário.
Cheia(L)	Verifica se a Lista Cadastral L está cheia. Uma Lista cheia é uma Lista em que não cabe mais nenhum elemento.
Cria(L)	Cria uma Lista Cadastral L, iniciando sua situação como vazia.
PegaOPrimeiro(L, X, TemElemento)	X retorna o valor do primeiro elemento da Lista L se esse primeiro elemento existir. Se não existir esse primeiro elemento (Lista vazia), o parâmetro TemElemento retornará o valor Falso.
PegaOPróximo(L, X, TemElemento)	X retorna o valor do próximo elemento da Lista, em relação à última chamada a uma das operações PegaOPrimeiro ou PegaOPróximo. Se não existir esse próximo elemento (final da Lista), o parâmetro TemElemento retornará o valor Falso.

**FIGURA 6.4** Operações do TAD Lista Cadastral sem elementos repetidos.

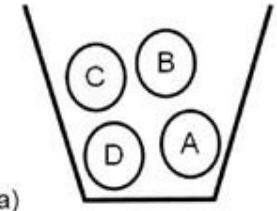
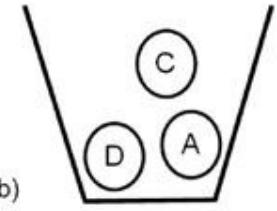
A operação Insere primeiramente verifica se o valor do elemento que está sendo inserido já está na Lista. Se aquele valor já fizer parte da Lista, não será permitida a inserção de um valor repetido.

A operação Retira também verifica se o valor X faz parte da Lista L. Se o elemento X for encontrado na Lista L, seja no começo, no meio ou no final da

lista, será retirado. Se X não for encontrado em L, nenhum elemento será retirado da Lista.

A operação Cria inicializa a Lista L como vazia. As operações Vazia e Cheia verificam se a Lista L está vazia (sem nenhum elemento) ou cheia (situação em que não cabe mais nenhum elemento na Lista).

Observe na [Figura 6.5](#) a execução de um conjunto de operações sobre uma Lista Cadastral sem elementos repetidos. A Lista L contém inicialmente quatro elementos: A, B, C e D. Não pense em como a Lista L é efetivamente implementada. Pense simplesmente em um conjunto com quatro elementos, A, B, C e D, como na [Figura 6.5a](#), e pense que sobre esse conjunto podemos aplicar as operações definidas na [Figura 6.4](#).

Lista L	Operação	Resultado
 (a)	Insere(L, 'A', Ok)	Não insere, pois a Lista L já contém o valor 'A'.
	EstáNaLista(L, 'F')	Resultado Falso, pois o valor 'F' não está na Lista L.
	EstáNaLista(L, 'B')	Resultado Verdadeiro, pois a Lista L contém elemento de valor 'B'.
	Retira(L, 'F', Ok)	Não retira, pois a Lista L não contém elemento de valor 'F'.
 (b)	Retira(L, 'B', Ok)	Retira o elemento de valor 'B' da Lista L (situação da Figura 6.5a), que ficará agora apenas com os elementos A, C e D (situação da Figura 6.5b).

**FIGURA 6.5** Ilustrando a execução das operações Insere, Retira e EstáNaLista.

Se aplicarmos a operação Insere(L, 'A', Ok) à situação da [Figura 6.5a](#), o parâmetro Ok retornará o valor Falso. O valor 'A' não será inserido na lista L, pois a Lista já contém um elemento de valor 'A'. A Lista Cadastral que estamos implementando no momento não permite elementos repetidos.

A operação EstáNaLista verifica se o valor passado como parâmetro faz parte da Lista ou não. Na primeira execução da operação EstáNaLista aplicada à situação da [Figura 6.5a](#), passamos como parâmetro o valor 'F' e recebemos

como resposta o valor Falso, indicando que a Lista L não contém o valor 'F'. Na segunda chamada, também aplicada à [Figura 6.5a](#), passamos como parâmetro o valor 'B'. Dessa vez, o resultado da função é o valor Verdadeiro, indicando que a Lista L contém um elemento de valor 'B'.

Com a operação Retira(L, 'F', Ok), estamos tentando retirar da Lista L o elemento de valor 'F'. A execução dessa operação sobre a situação da [Figura 6.5a](#) não retiraria nenhum elemento e o parâmetro Ok retornaria o valor Falso, pois a Lista L não contém elemento com valor 'F'. Se ainda sobre a situação da [Figura 6.5a](#) aplicarmos novamente a operação Remove, mas agora solicitando a retirada do valor 'B', o parâmetro ok retornará o valor Verdadeiro, indicando que foi retirado um elemento de valor 'B' da Lista L. A Lista, que continha quatro elementos (A, B, C, D), passará a ter apenas três elementos (A, C, D), conforme ilustra a [Figura 6.5b](#).

## Operações para percorrer uma Lista

No [Capítulo 2](#), utilizamos a operação Desempilha para retirar todos os elementos de uma Pilha, processar esses elementos para algum propósito e depois retornar os elementos à Pilha original. Com as operações Empilha e Desempilha, conseguimos transferir elementos de uma Pilha para outra ([Exercício 2.1](#)), pudemos verificar se uma Pilha P1 possui mais elementos do que uma segunda Pilha P2 ([Exercício 2.2](#)), pudemos verificar a presença de um elemento de valor X em uma Pilha P ([Exercício 2.3](#)) e se duas Pilhas P1 e P2 são iguais ([Exercício 2.4](#)). No [Capítulo 3](#), utilizando as operações Insere e Retira, pudemos percorrer todos os elementos de uma Fila e, com isso, pudemos, por exemplo, juntar ([Exercício 3.1](#)) ou trocar ([Exercício 3.2](#)) os elementos de duas Filas.

Quando lidamos com Pilhas e Filas conseguimos percorrer todos os elementos do conjunto acionando as operações de Retirar ou Desempilhar, pois essas operações simplesmente removem o primeiro do conjunto, independentemente do seu valor. Removendo repetidas vezes o primeiro elemento, o conjunto logo se tornará vazio e, assim, teremos percorrido todo o conjunto. Essa lógica de chamadas sucessivas à operação Retira não pode ser utilizada para percorrer uma Lista Cadastral, pois em uma Lista Cadastral a operação Retira não remove o primeiro elemento do conjunto, mas um elemento de valor específico. Para percorrer os elementos de uma Lista Cadastral precisamos de operações como PegaOPrimeiro e PegaOPróximo, especificadas na [Figura 6.4](#).

A [Figura 6.6](#) mostra o resultado da execução de uma sequência de chamadas às operações PegaOPrimeiro e PegaOPróximo, sobre uma Lista Cadastral sem

elementos repetidos. A Lista L contém quatro elementos: A, B, C e D. Não pense em como a Lista L é efetivamente implementada. Mas pense que, de alguma maneira, é possível estabelecer uma sequência entre os elementos do conjunto e que essa sequência é A, B, C e D.

Lista L	Operação	Resultado
	PegaOPrimeiro(L, X, TemElemento)	TemElemento: Verdadeiro X: A
	PegaOPróximo(L, X, TemElemento)	TemElemento: Verdadeiro X: B
	PegaOPróximo(L, X, TemElemento)	TemElemento: Verdadeiro X: C
	PegaOPróximo(L, X, TemElemento)	TemElemento: Verdadeiro X: D
	PegaOPróximo(L, X, TemElemento)	TemElemento: Falso X: ?
	PegaOPrimeiro(L, X, TemElemento)	TemElemento: Verdadeiro X: A

**FIGURA 6.6** Ilustrando a execução das operações PegaOPrimeiro e PegaOPróximo.

Ao executarmos a operação PegaOPrimeiro sobre a situação da [Figura 6.6](#), o parâmetro TemElemento retorna o valor Verdadeiro e o parâmetro X retorna o valor 'A'. Isso indica que existe um primeiro elemento na Lista L e seu valor é 'A'.

Executamos em seguida a operação PegaOPróximo e recebemos como resultado Verdadeiro (ou seja, temos um próximo elemento) e 'B' (o valor desse próximo elemento é 'B'). Executamos mais três vezes a operação PegaOPróximo e recebemos, respectivamente, Verdadeiro/'C', Verdadeiro/'D' e Falso/?.. Nesta última execução, o parâmetro TemElemento retornou o valor Falso, indicando que, em relação à última chamada (que retornou o valor 'D'), não há um próximo elemento. A Lista acabou!

Finalmente, chamamos outra vez a operação PegaOPrimeiro e recebemos novamente os valores Verdadeiro e 'A'. Se chamarmos 10 vezes a operação PegaOPrimeiro sobre a situação da [Figura 6.6](#), receberemos como resultado sempre esses mesmos valores. Se, em seguida, executarmos 10 vezes seguidas a operação PegaOPróximo, teremos como resultados Verdadeiro/'B', Verdadeiro/'C', Verdadeiro/'D', Falso/?., Falso/?., Falso/?., Falso/?., Falso/?., Falso/?., Falso/?., Falso/?.

## Exercício 6.1 Operação para imprimir todos os elementos de uma Lista

Utilize as operações especificadas na [Figura 6.4](#) — Insere, Retira, EstáNaLista, Vazia, Cheia, Cria, PegaOPrimeiro e PegaOPróximo — e desenvolva uma operação para imprimir o valor de todos os elementos de uma Lista.

```
/* Imprime todos os valores armazenados na Lista L */
/* Imprime todos os valores armazenados na Lista L */
/* Imprime todos os valores armazenados na Lista L */
/* Imprime todos os valores armazenados na Lista L */
/* Imprime todos os valores armazenados na Lista L */
/* Imprime todos os valores armazenados na Lista L */
/* Imprime todos os valores armazenados na Lista L */
ImprimeTodos (parâmetro por referência L do tipo Lista);
/* Imprime todos os valores armazenados na Lista L */
```

A [Figura 6.7](#) apresenta uma solução para o [Exercício 6.1](#). Chamamos primeiramente a operação PegaOPrimeiro e com isso damos início ao processo de percorrer todos os elementos da Lista. A seguir, chamamos repetidamente a operação PegaOPróximo, para pegar o valor de cada um dos elementos da Lista. A repetição se encerra quando a variável TemElemento assumir o valor Falso.

```
ImprimeTodos (parâmetro por referência L do tipo Lista) {
    /* Imprime todos os valores armazenados na Lista L */

    Variável X do tipo Char;
    Variável TemElemento do tipo Boolean;

    PegaOPrimeiro( L, X, TemElemento );      // pega o primeiro elemento da Lista, se existir
    Enquanto (TemElemento == Verdadeiro) Faça {
        { Imprime( X );
            PegaOPróximo( L, X, TemElemento ); }
    } // fim ImprimeTodos
```

**FIGURA 6.7** Algoritmo para imprimir os elementos de uma Lista.

## Exercício 6.2 Interseção de duas Listas L1 e L2

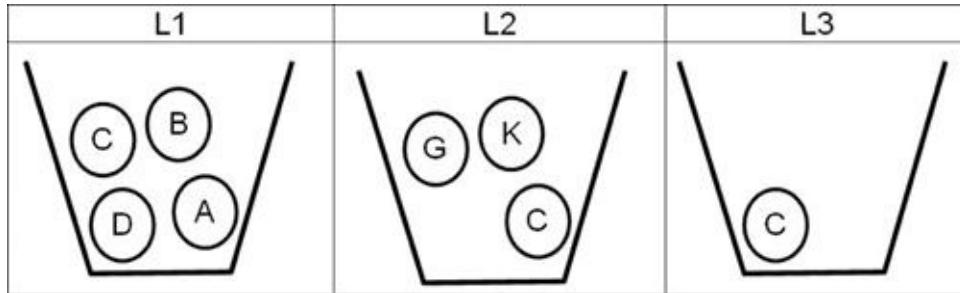
Utilize as operações especificadas na [Figura 6.4](#) — Insere, Retira, EstáNaLista, Vazia, Cheia, Cria, PegaOPrimeiro e PegaOPróximo — e desenvolva uma operação que recebe como parâmetros duas Listas L1 e L2, e cria uma terceira Lista L3, formada pela interseção de L1 e L2. Ou seja, L3 deverá conter todos os elementos que pertencem tanto a L1 quanto a L2.

```

Interseção (parâmetros por referência L1, L2, L3 do tipo Lista);
/* Recebe L1 e L2, e cria L3 contendo a interseção entre L1 e L2.
Ou seja, L3 terá todos os elementos que pertencem tanto a L1 quanto
a L2 */

```

A [Figura 6.8](#) apresenta um diagrama ilustrando a criação de uma Lista L3 a partir da interseção de L1 e L2. L1 contém os elementos A, B, C e D. Já a Lista L2 contém os elementos G, K e C. O único elemento que pertence tanto a L1 quanto a L2 é C. Assim, a Lista L3 deverá conter somente o elemento C.



**FIGURA 6.8 Criando L3 como a interseção de L1 e L2.**

Uma possível solução para o [Exercício 6.2](#) é apresentada na [Figura 6.9](#). Primeiramente, criamos a Lista L3 vazia. Em seguida percorremos a Lista L1: para cada elemento de L1, verificamos se ele pertence também à Lista L2. Caso o elemento de L1 fizer parte também de L2, será inserido em L3.

```

Interseção (parâmetros por referência L1, L2, L3 do tipo Lista) {
/* Recebe L1 e L2, e cria L3 contendo a interseção entre L1 e L2. Ou seja, L3 terá todos
os elementos que pertencem a L1 e também a L2 */

Variável X do tipo Char;
Variável TemElemento do tipo Boolean;
Variável Ok do tipo Boolean;

Cria(L3);           // cria a Lista L3, vazia

/* para cada elemento de L1, verifica se está também em L2; se estiver, insere em L3 */
PegaOPrimeiro( L1, X, TemElemento );           // pega o primeiro de L1
Enquanto (TemElemento == Verdadeiro) Faça {
    Se (EstáNaLista(L2, X)==Verdadeiro) // Elemento X de L1 está também em L2?
        Então Insere (L3, X, Ok); // Se estiver, insere X em L3
        PegaOPróximo( L1, X, TemElemento ); } // pega o próximo de L1
} // fim Interseção

```

**FIGURA 6.9** Algoritmo de interseção de duas Listas.

Para percorrer L1, executamos de início a operação PegaOPrimeiro e, em seguida, chamamos repetidamente a operação PegaOPróximo. A repetição se encerra quando a variável TemElemento assumir o valor Falso.

### Exercício 6.3 Posição do elemento no conjunto

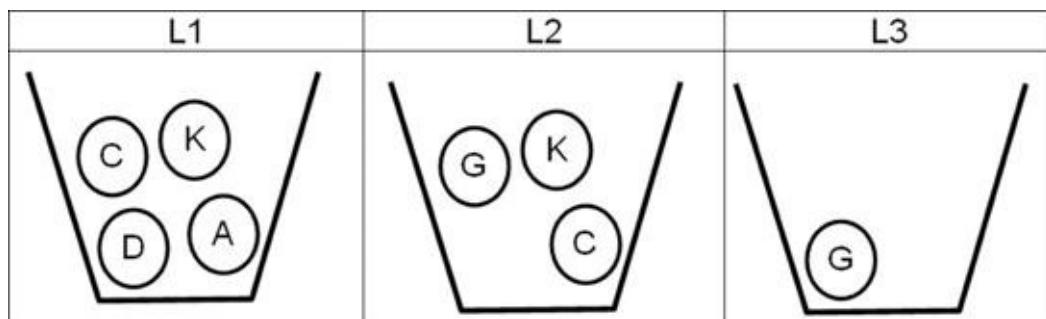
Utilize as operações especificadas na [Figura 6.4](#) — Insere, Retira, EstáNaLista, Vazia, Cheia, Cria, PegaOPrimeiro e PegaOPróximo — e desenvolva uma operação que determina a posição de um elemento X em uma Lista L. Caso X não estiver em L, a operação deve retornar o valor zero.

```
Inteiro PosiçãoNoConjunto (parâmetro por referência L do tipo Lista, parâmetro X do tipo char);
```

```
/* Retorna valor numérico que indica a posição de X na Lista L.  
Por exemplo, retorna 1 se X for o primeiro elemento da Lista L, 2  
se for o segundo, e assim por diante. Se X não estiver em L,  
retorna o valor zero */
```

### Exercício 6.4 Estão em L2 e não estão em L1

Utilize as operações especificadas na [Figura 6.4](#) — Insere, Retira, EstáNaLista, Vazia, Cheia, Cria, PegaOPrimeiro e PegaOPróximo — e desenvolva uma operação que recebe como parâmetros duas Listas L1 e L2, e cria uma terceira Lista L3, contendo todos os elementos que pertencem a L2 mas não pertencem a L1, conforme ilustrado na [Figura 6.10](#).



**FIGURA 6.10** L3 contém os que estão em L2 e não estão em L1.

```
EmL2MasNãoEmL1 (parâmetros por referência L1, L2, L3 do tipo Lista)
```

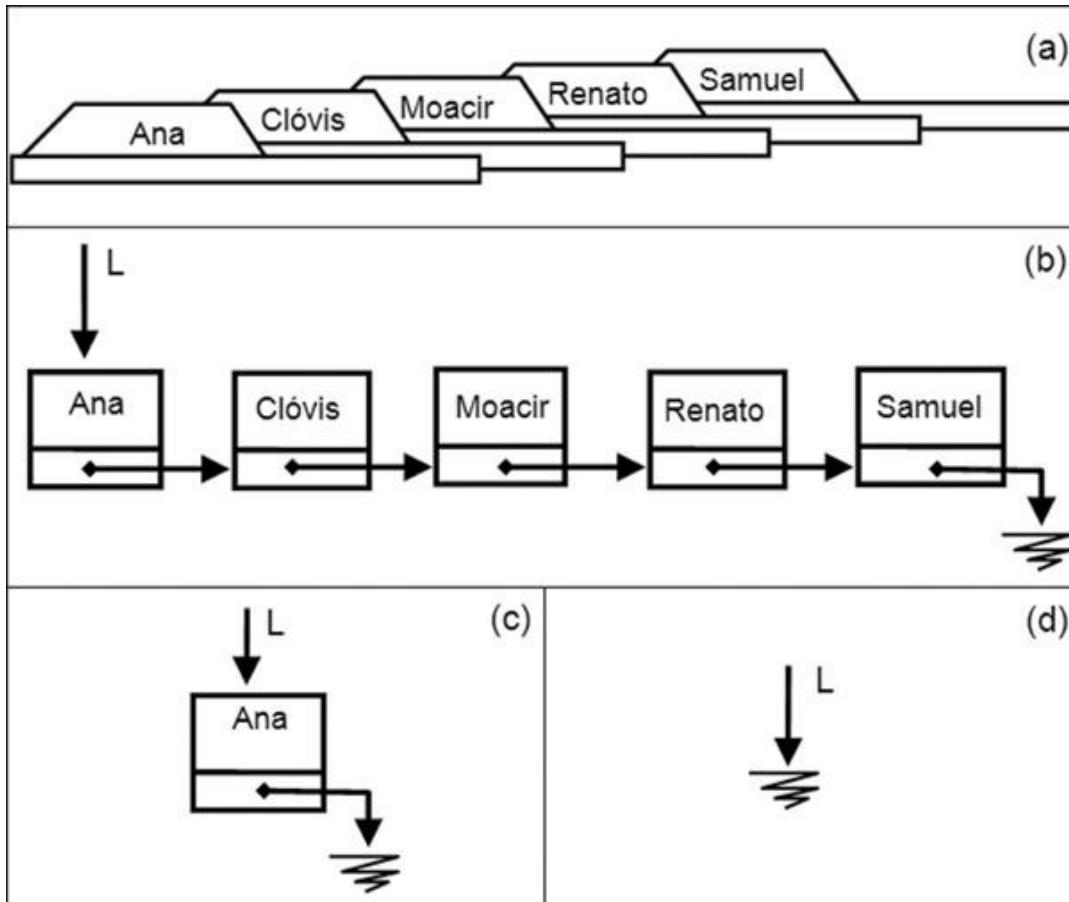
```
/* Recebe L1 e L2, e cria L3 contendo todos os elementos que
```

estão em L2 mas que não estão em L1\*/

## 6.3 Implementando uma Lista Cadastral sem elementos repetidos como uma Lista Encadeada Ordenada

No exemplo do Cadastro de Funcionários, as pastinhas contendo os dados sobre cada funcionário poderiam estar ordenadas alfabeticamente. Isso facilitaria a busca pela pastinha de um funcionário.

A [Figura 6.11a](#) ilustra a situação de um Cadastro com cinco pastinhas, ordenadas alfabeticamente. Na [Figura 6.11b](#), esse Cadastro com cinco elementos é representado como uma Lista Encadeada Ordenada. O ponteiro L aponta para o primeiro Nô da Lista, que armazena o valor Ana. Os demais elementos da Lista estão ordenados alfabeticamente; cada elemento aponta para o próximo elemento da Lista.



**FIGURA 6.11** Representando uma Lista Cadastral como uma Lista Encadeada Ordenada.

A Figura 6.11c mostra a representação de uma Lista Ordenada com um único elemento, de valor *Ana*. Finalmente, a Figura 6.11d ilustra a situação de uma Lista Ordenada vazia.

## Implementando a operação de retirar elemento da Lista

Em uma Lista Cadastral, retiramos do conjunto o elemento que possui um valor específico. A operação Retira pode ser resumida em dois passos: primeiro, procuramos o elemento; segundo, se o elemento for encontrado, o removemos da Lista. Considerando a implementação da Lista Cadastral como um TAD Lista Encadeada Ordenada sem elementos repetidos, conforme ilustrado nos diagramas da Figura 6.11, podemos descrever esses dois passos como apresentado na Figura 6.12.

### Algoritmo — Retira Elemento — primeira versão

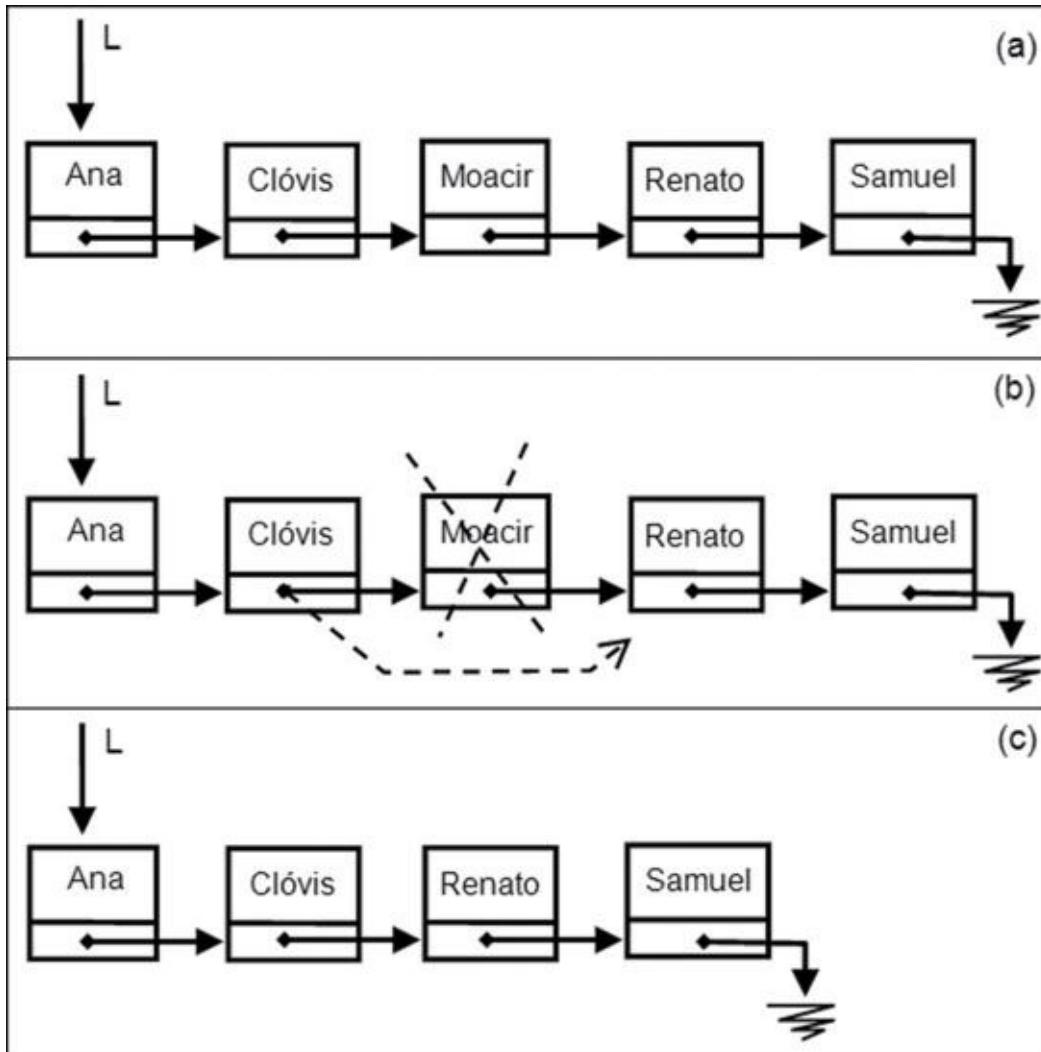
- Passo 1: Procuramos na Lista L o Nó que contém o valor X, sendo X o elemento a ser removido.
- Passo 2: Eliminamos da Lista L o Nó que guarda o valor X.

**FIGURA 6.12** Retirando elemento de uma Lista Cadastral



implementada como uma Lista Encadeada Ordenada.

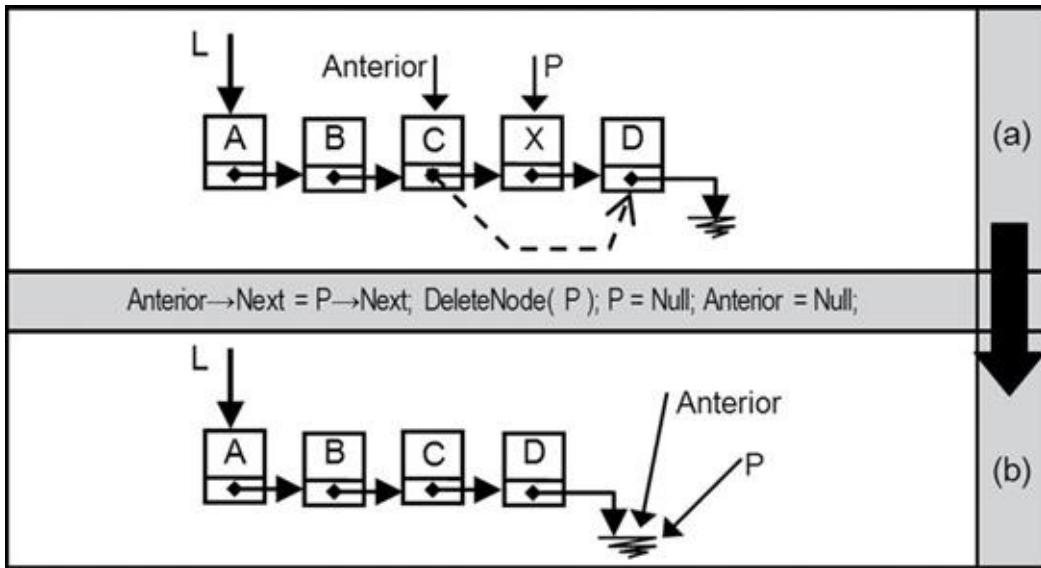
Por exemplo, se o funcionário Moacir pediu demissão da empresa, temos que retirar da Lista o Nó que armazena o valor Moacir. Precisamos, primeiramente, encontrar na Lista L o Nó que guarda o valor Moacir e então remover esse Nó, conforme ilustrado nas Figuras 6.13a, b e c.



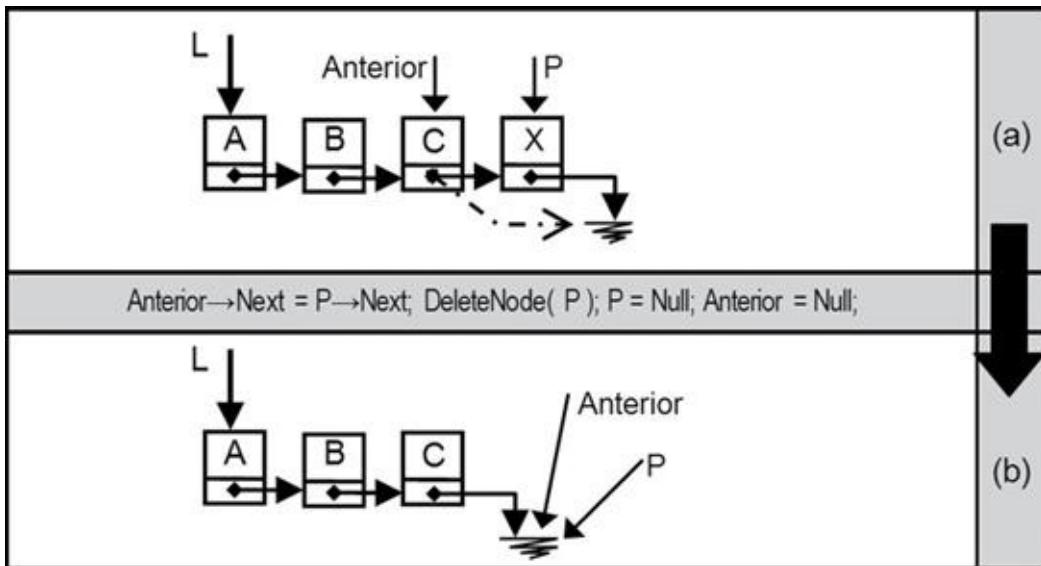
**FIGURA 6.13** Removendo o N o que cont m o valor **Moacir**.

Nem sempre o N o contendo o valor do elemento que queremos remover est r  “no meio” da Lista, como no exemplo da [Figura 6.13](#).  poss vel que o elemento a ser removido esteja no in cio ou no final da Lista; ali s,  poss vel que o elemento que queremos remover nem esteja na Lista. E a oper o que retira um elemento da Lista precisa tratar todos esses casos.

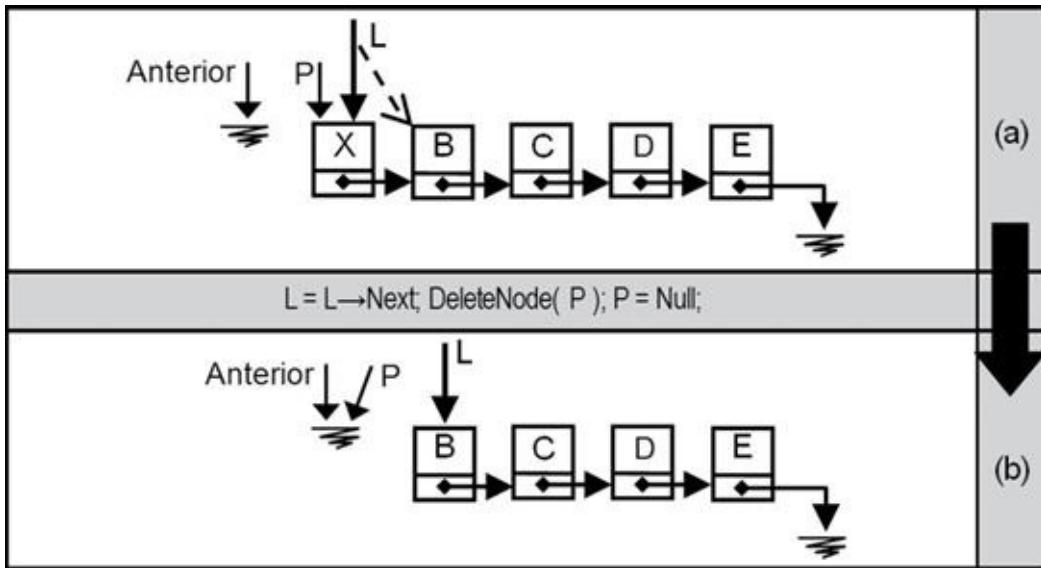
Os diagramas das [Figuras 6.14](#) a [6.19](#) ilustram diferentes situa es para a oper o Retira. Em todas essas situa es, L  o ponteiro para o in cio da Lista, e X  o elemento que queremos remover.



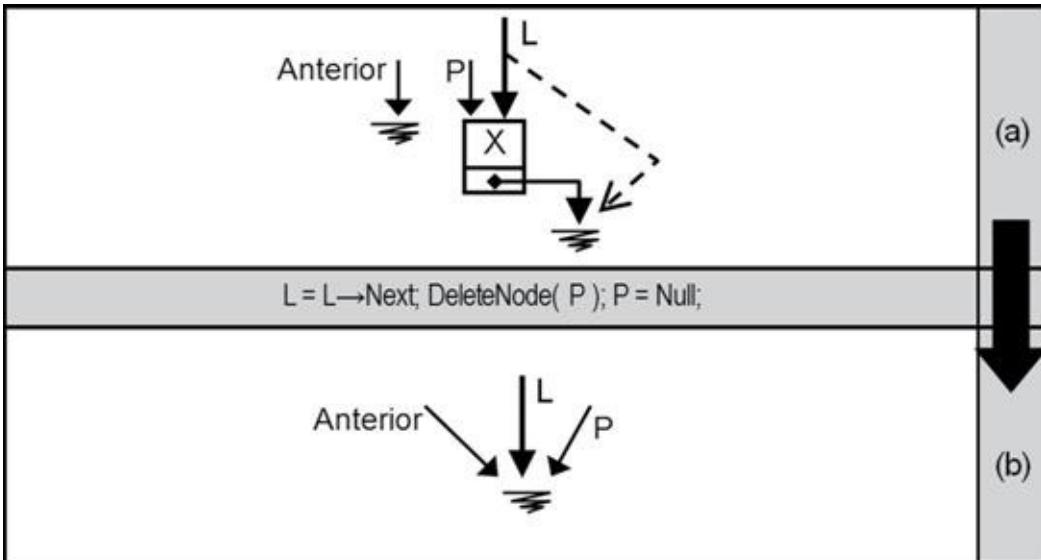
**FIGURA 6.14** Retirando elemento de uma Lista Ordenada. Caso 1: X no meio da Lista.



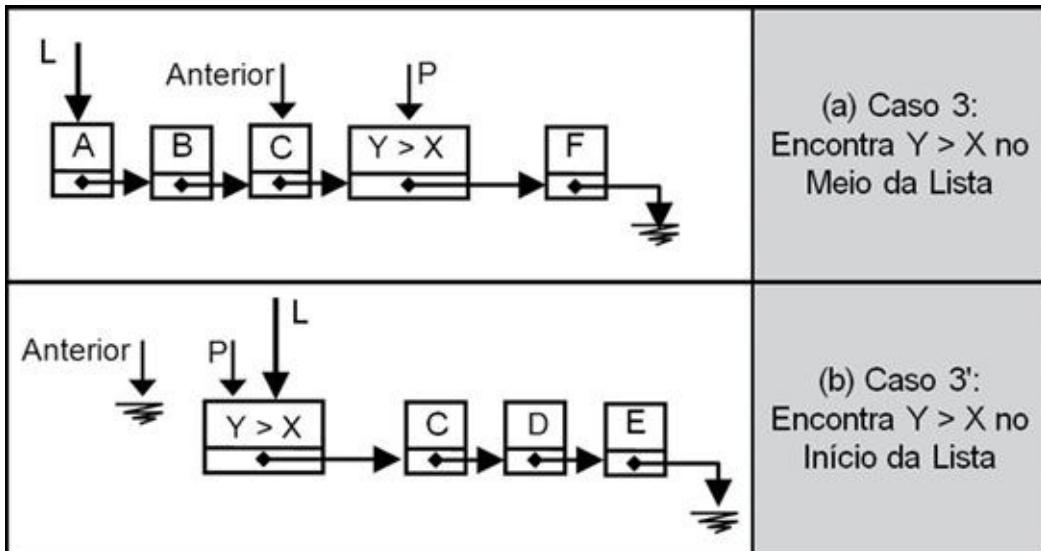
**FIGURA 6.15** Retirando elemento de uma Lista Ordenada. Caso 1': X no final da Lista.



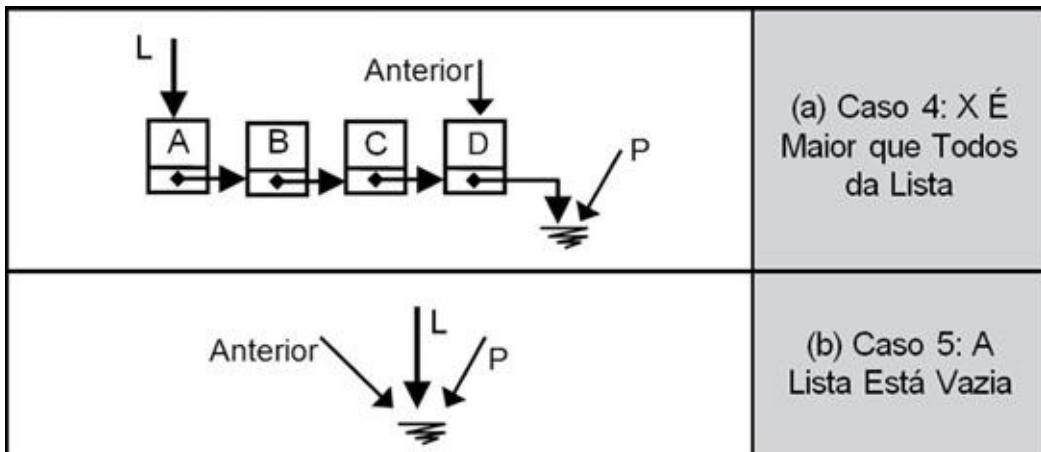
**FIGURA 6.16** Retira de Lista Ordenada. Caso 2: X no início da Lista.



**FIGURA 6.17** Retirando elemento de uma Lista Ordenada. Caso 2': X como único elemento da Lista



**FIGURA 6.18** Retira de Lista Ordenada. X não está na Lista. Caso 3: encontra  $Y > X$  no meio da Lista. Caso 3': encontra  $Y > X$  no início da Lista.



**FIGURA 6.19** Retira de Lista Ordenada.–X não está na Lista. Caso 4: X maior que todos da Lista. Caso 5: Lista vazia.

A [Figura 6.14](#) ilustra a remoção do elemento X da Lista L no caso 1, em que X é encontrado no meio da Lista. Na situação inicial, [Figura 6.14a](#), o ponteiro P, que é uma variável temporária, está apontando para o Nó que contém X. O ponteiro Anterior, que também é uma variável temporária, está apontando para o Nó anterior ao Nó para o qual P está apontando.

Para remover o Nó que contém o elemento de valor X, aplicamos primeiramente o comando  $\text{Anterior} \rightarrow \text{Next} = P \rightarrow \text{Next}$ . Através desse comando,

o campo Next do nó apontado por Anterior passa a apontar para onde aponta o campo Next do Nó apontado por P, ou seja, passa a apontar para o Nó que contém o valor D. A seta pontilhada indica a mudança resultante da execução de  $\text{Anterior} \rightarrow \text{Next} = \text{P} \rightarrow \text{Next}$ . Em seguida, aplicamos o comando `DeleteNode(P)`, que desaloca o Nó apontado por P. Com o `DeleteNode`, o Nó apontado por P simplesmente some do diagrama. Finalmente, atribuímos o valor Null a P e a Anterior, apenas para deixá-los apontando para uma posição bem definida, e chegamos à situação da [Figura 6.14b](#).

Na situação da [Figura 6.15a](#), X foi encontrado no último Nó da Lista. Nesse Caso 1' podemos aplicar exatamente os mesmos comandos aplicados no Caso 1. Tanto no Caso 1 quanto no Caso 1', o campo Next do Nó apontado pelo ponteiro Anterior passará a apontar para onde aponta o campo Next do Nó apontado por P. A diferença prática é que, no Caso 1  $\text{P} \rightarrow \text{Next}$  está apontando para um Nó, e no Caso 1' está apontando para Null.

Na situação da [Figura 6.16a](#), encontramos X no primeiro Nó da Lista. Nesse Caso 2 precisamos avançar L para o próximo Nó através do comando  $\text{L} = \text{L} \rightarrow \text{Next}$  e então liberar o Nó apontado por P com o comando `DeleteNode(P)`. Apenas para deixar P apontando para uma posição bem definida, apontamos P para Null e chegamos à situação da [Figura 6.16b](#).

Na [Figura 6.17a](#), tratamos o Caso 2', situação em que X está no primeiro Nó da Lista, assim como no Caso 2. Mas, no Caso 2', esse primeiro Nó da Lista é também o único Nó da Lista. Os comandos aplicados no Caso 2' são os mesmos aplicados no Caso 2. Porém, no Caso 2', ao avançar L com o comando  $\text{L} = \text{L} \rightarrow \text{Next}$ , o ponteiro L passará a apontar para Null (seta pontilhada e [Figura 6.17b](#)).

As [Figuras 6.18](#) e [6.19](#) ilustram situações em que X não é encontrado na Lista. Para identificar a situação em que X não está na Lista, procuramos X a partir do primeiro Nó e avançamos para o próximo, o próximo e o próximo, sucessivamente, até encontrar o final da Lista (Null) ou até encontrar um valor Y maior que X. Considerando que a Lista é ordenada, se encontrarmos um valor Y maior do que o valor X que procuramos, podemos concluir que X não está na Lista.

No Caso 3, encontramos um valor  $Y > X$  no meio da Lista — [Figura 6.18a](#). No Caso 3', encontramos  $Y > X$  logo no início da Lista — [Figura 6.18b](#).

No Caso 4, ilustrado na [Figura 6.19a](#), X é maior que todos os elementos da Lista. Após procurar em cada um dos nós, chegamos ao final da Lista sem ter encontrado X. No Caso 5 — [Figura 6.19b](#) — a Lista está vazia, e antes mesmo

de começarmos a procurar, chegamos ao final da Lista, indicado pelo valor Null.

Nesses casos em que X não é encontrado — Caso 3, Caso 3', Caso 4 e Caso 5 —, não eliminamos nenhum Nô da Lista. Tornando o algoritmo da [Figura 6.12](#) um pouco mais específico, chegamos ao algoritmo da [Figura 6.20](#).

#### Algoritmo – Retira Elemento – segunda versão

- Passo 1: definir variáveis temporárias P e Anterior. Anterior começa em Null, P começa em L. Avança Anterior e P até que P chegue ao Nô que contém X ou a um Nô que contém um valor Y > X ou ao final da Lista (Null). Anterior avança sempre uma posição atrás de P.
- Passo 2: com base na posição de Anterior e P, e com base no valor armazenado no Nô apontado por P (caso P for diferente de Null), identificar o caso (Caso 1, Caso 1', Caso 2, Caso 2', Caso 3, Caso 3', Caso 4 ou Caso 5) e executar as ações necessárias.

**FIGURA 6.20** Retira elemento de Lista Cadastral implementada como Lista Encadeada Ordenada — segunda versão.

### Exercício 6.5 Operação Retira Elemento de uma Lista Cadastral implementada como uma Lista Encadeada Ordenada

Conforme especificado na [Figura 6.4](#), a operação Retira recebe como parâmetro a Lista L, da qual queremos retirar o elemento de valor X. Caso X for encontrado na Lista L, deve ser removido. Se X não for encontrado na Lista L, nenhum elemento deve ser retirado da Lista.

Retira (parâmetro por referência L do tipo Lista, parâmetro X do tipo Char, parâmetro por referência Ok do tipo Boolean);

```
/* Caso X for encontrado na Lista L, retira X da Lista e Ok
retorna Verdadeiro. Se X não estiver na Lista L, não retira nenhum
elemento e Ok retorna o valor Falso */
```

O algoritmo da [Figura 6.21](#) implementa em uma linguagem conceitual a operação que Retira um elemento específico de uma Lista Cadastral implementada como uma Lista Encadeada Ordenada. Assim como nas versões preliminares das [Figuras 6.12](#) e [6.20](#), o algoritmo foi dividido em dois passos: (passo 1) procura X na Lista L e (passo 2) se encontrar, remove o Nô que contém

X.

```
Retira (parâmetro por referência L do tipo Lista, parâmetro X do tipo Char, parâmetro por  
referência Ok do tipo Boolean) {  
    /* Caso X for encontrado na Lista L, retira X da Lista e Ok retorna Verdadeiro. Se X não  
    estiver na Lista L, não retira nenhum elemento e Ok retorna o valor Falso */  
    Variáveis P, Anterior do tipo NodePtr;      // Tipo NodePtr = ponteiro para Nó  
    Variável AchouX do tipo Boolean;  
    ProcuraX (L, X, P, Anterior, AchouX);      /* ProcuraX executa o passo 1: encontrar X—  
    na Lista L. ProcuraX indica, através da variável AchouX, se X foi encontrado em L  
    (Verdadeiro) ou não (Falso). Se X for encontrado, ProcuraX colocará P apontando para o  
    Nó que armazena X, e Anterior apontando para o Nó anterior ao Nó que armazena X  
    Figuras 6.14 a 6.19. Nos casos em que X estiver no primeiro Nó da Lista (Figura 6.16 e  
    6.17), Anterior estará apontando para Null */  
    /* Passo 2: se encontrou X, remover da Lista o Nó que contém X */  
    Se (AchouX == Verdadeiro)                  // se X foi encontrado na Lista  
        Então { Se (P != L)                      // se X não estiver no primeiro Nó da Lista  
            Então { Anterior → Next = P → Next; // Casos 1 ou 1': X no meio ou no último  
                DeleteNode( P );               // Nó da Lista L— veja a Figura 6.14 e  
                P = Null;                   // a Figura 6.15  
                Anterior = Null }  
            Senão { L = L → Next;           // Casos 2 ou 2': X no primeiro Nó da Lista  
                DeleteNode( P );           // veja as Figuras 6.16 e 6.17  
                P = Null }  
            Ok = Verdadeiro;  
        }  
    Senão Ok = Falso; // X não foi encontrado— Casos 3, 3', 4 ou 5; não retira elemento  
}  
// fim Retira
```

**FIGURA 6.21** Retira elemento de Lista Cadastral Programada como



Lista Encadeada Ordenada — implementação do passo 2.

O procedimento ProcuraX implementa o passo 1. ProcuraX nos indica se X foi encontrado na Lista L (variável AchouX == Verdadeiro). Se X for encontrado em L, ProcuraX colocará o ponteiro P apontando para o Nó que contém o valor X, como nas [Figuras 6.14](#) a [6.17](#). ProcuraX também posicionará o ponteiro Anterior apontando para o Nó anterior ao Nó que contém o valor X ([Figuras 6.14](#) e [6.15](#)). Nos casos em que X for encontrado no primeiro Nó da Lista, ProcuraX colocará Anterior apontando para Null ([Figuras 6.16](#) e [6.17](#)).

Após a execução de ProcuraX, verificamos se X foi encontrado na Lista ou

não. Se X foi encontrado ( $AchouX = Verdadeiro$ ), precisamos identificar o caso a tratar: Caso 1 ([Figura 6.14](#)), Caso 1' ([Figura 6.15](#)), Caso 2 ([Caso 6.16](#)) ou Caso 2' ([Figura 6.17](#)).

Nos Casos 1 e 1', X não está no primeiro Nó da Lista, ou seja, o ponteiro P não estará apontando para onde aponta o ponteiro L. Assim, perguntando se P é diferente de L ( $P \neq L$ ) tratamos conjuntamente os casos em que X é encontrado no meio da Lista ou no último Nó da Lista (Casos 1 e 1' — [Figuras 6.14 e 6.15](#)). Nos casos em que X está no primeiro Nó da Lista, P é igual a L. Tratamos, então, conjuntamente os casos 2 e 2' — [Figuras 6.15 e 6.16](#).

Em todos os casos em que X é encontrado na Lista — Casos 1, 1', 2 e 2' — a variável Ok recebe o valor Verdadeiro. Nos demais casos — 3, 3', 4 e 5 — X não é encontrado na lista, não removemos nenhum Nó e a variável Ok recebe o valor Falso.

## Implementação do primeiro passo: o procedimento ProcuraX

A [Figura 6.22](#) apresenta uma implementação do procedimento ProcuraX. Para implementar ProcuraX, usamos como roteiro o passo 1 do algoritmo da [Figura 6.20](#).

```
ProcuraX (parâmetros por referência L do tipo Lista, parâmetro por referência X do tipo Char, parâmetros por referência P, Anterior do tipo NodePtr, parâmetro por referência AchouX do tipo Boolean) {  
    /* ProcuraX executa o passo 1: procura X na Lista L e indica através da variável AchouX se X foi encontrado (Verdadeiro) ou não (Falso). Se X for encontrado, ProcuraX coloca P apontando para o Nó que armazena X e Anterior apontando para o Nó anterior ao Nó que armazena X — Figuras 6.14 a 6.19. Se X estiver no primeiro Nó da Lista (Figuras 6.16 e 6.17), coloca Anterior apontando para Null */  
    P = L;          // P começa em L  
    Anterior = Null; // Anterior começa em Null  
    /* avança P e Anterior até encontrar X ou Y > X, ou Null... Anterior corre atrás de P */  
    Enquanto ((P != Null) E (P → Info < X)) Faça  
        { Anterior = P;  
         P = P → Next; }  
    Se ( (P != Null) E (P → Info == X) )  
        Então AchouX = Verdadeiro;  
    Senão AchouX = Falso;  
} fim ProcuraX
```

**FIGURA 6.22 Retira elemento de Lista Cadastral Programada como Lista Encadeada Ordenada — implementação do passo 1.**

Inicializamos P em L e Anterior em Null. A seguir, avançamos P e Anterior até encontrar X ou até encontrar um valor Y maior do que X, ou até encontrar o final da Lista. Para expressar essa lógica, utilizamos o comando Enquanto ((P != Null) E (Info(P) < X)). Ou seja, enquanto não encontrarmos o final da lista, enquanto não encontrarmos X e enquanto não encontrarmos um Y maior do que X, continuamos avançando. Se encontrarmos Null ou X, ou Y > X, o comando de repetição será interrompido.

Do modo como a lógica foi expressa, é imprescindível o uso do operador lógico “E” no comando de repetição. Se em vez de “E” utilizarmos “OU”, P e Anterior continuarão avançando mesmo se X for encontrado, o que ocasionará um erro.

Ao final do procedimento ProcuraX, o comando de repetição pode ter sido interrompido por termos achado X, por termos achado um Y maior do que X, ou ainda por termos encontrado o final da Lista. Precisamos então verificar se X foi encontrado ou não, e para isso perguntamos se ((P! = Null) E (P → Info == X)). Não basta perguntar se P → Info == X porque, no caso em que o comando de repetição foi interrompido por termos chegado ao final da Lista, P estará apontando para Null. E se P estiver apontando para Null, P → Info não existe. O resultado da execução de P → Info quando P está apontando para Null não é previsível e poderá resultar em um erro. Evitamos esse erro ao perguntarmos se ((P! = Null) e (P → Info == X)). Se essa condição for verdadeira, saberemos que encontramos X na Lista L.

## **Exercício 6.6 Operação Insere elemento em uma Lista Cadastral implementada como uma Lista Encadeada Ordenada**

Conforme especificado na [Figura 6.4](#), a operação deve inserir o elemento de valor X na Lista L, caso a Lista L já não tenha um elemento de valor X. Caso a Lista L já tiver um elemento de valor X, nenhum elemento deverá ser inserido e, nesse caso, o parâmetro Ok deve retornar o valor Falso. **Importante: para implementar a operação Insere, primeiramente identifique os casos de inserção e faça diagramas, como os das Figuras 6.14 a 6.19.** Só então desenvolva o algoritmo. Como sugestão, faça um algoritmo em dois passos análogos aos da [Figura 6.20](#). Verifique se o procedimento ProcuraX que

implementa o passo 1 da operação Retira pode ser utilizado também na implementação da operação Insere.

```
Insere (parâmetro por referência L do tipo Lista, parâmetro X do tipo Char, parâmetro por referência Ok do tipo Boolean);
```

```
/* Caso o valor X já não estiver na Lista L, insere X e Ok retorna Verdadeiro. Se X já estiver na Lista L, não insere nenhum elemento e Ok retorna o valor Falso */
```

## **Exercício 6.7 Operação que verifica se um elemento faz parte de uma Lista Cadastral, implementada como uma Lista Encadeada Ordenada**

Conforme especificado na [Figura 6.4](#), a operação EstáNaLista verifica se o elemento de valor X faz parte da Lista Cadastral L, retornando o valor Verdadeiro caso X estiver na Lista L, e o valor Falso caso X não fizer parte da Lista L.

```
Boolean EstáNáLista (parâmetro por referência L do tipo Lista, parâmetro X do tipo Char);
```

```
/* Caso X for encontrado na Lista L, retorna Verdadeiro. Retorna Falso caso X não estiver na Lista L */
```

## **Exercício 6.8 Operação que cria a Lista Cadastral**

Criar a Lista Cadastral significa inicializar os valores de modo a indicar que a Lista está vazia, ou seja, sem nenhum elemento.

```
Cria (parâmetro por referência L do tipo Lista);
```

```
/* Cria a Lista Cadastral L, inicializando a Lista como vazia — sem nenhum elemento */
```

## **Exercício 6.9 Operação para testar se a Lista está vazia**

Conforme especificado na [Figura 6.4](#), a operação Vazia testa se a Lista Cadastral L passada como parâmetro está vazia (sem elementos), retornando o valor Verdadeiro (Lista vazia) ou Falso (Lista não vazia).

```
Boolean Vazia (parâmetro por referência L do tipo Lista);
```

```
/* Retorna Verdadeiro se a Lista L estiver vazia — sem nenhum elemento; retorna Falso caso contrário */
```

## **Exercício 6.10 Operação para testar se a Lista está cheia**

Conforme especificado na [Figura 6.4](#), a operação Cheia testa se a Lista Cadastral passada como parâmetro está cheia. A Lista estará cheia se na estrutura de armazenamento não couber mais nenhum elemento. Na implementação com alocação encadeada e dinâmica de memória, podemos considerar que a estrutura nunca ficará cheia, e testar o sucesso da operação que aloca memória dinamicamente para inserir um elemento na Lista.

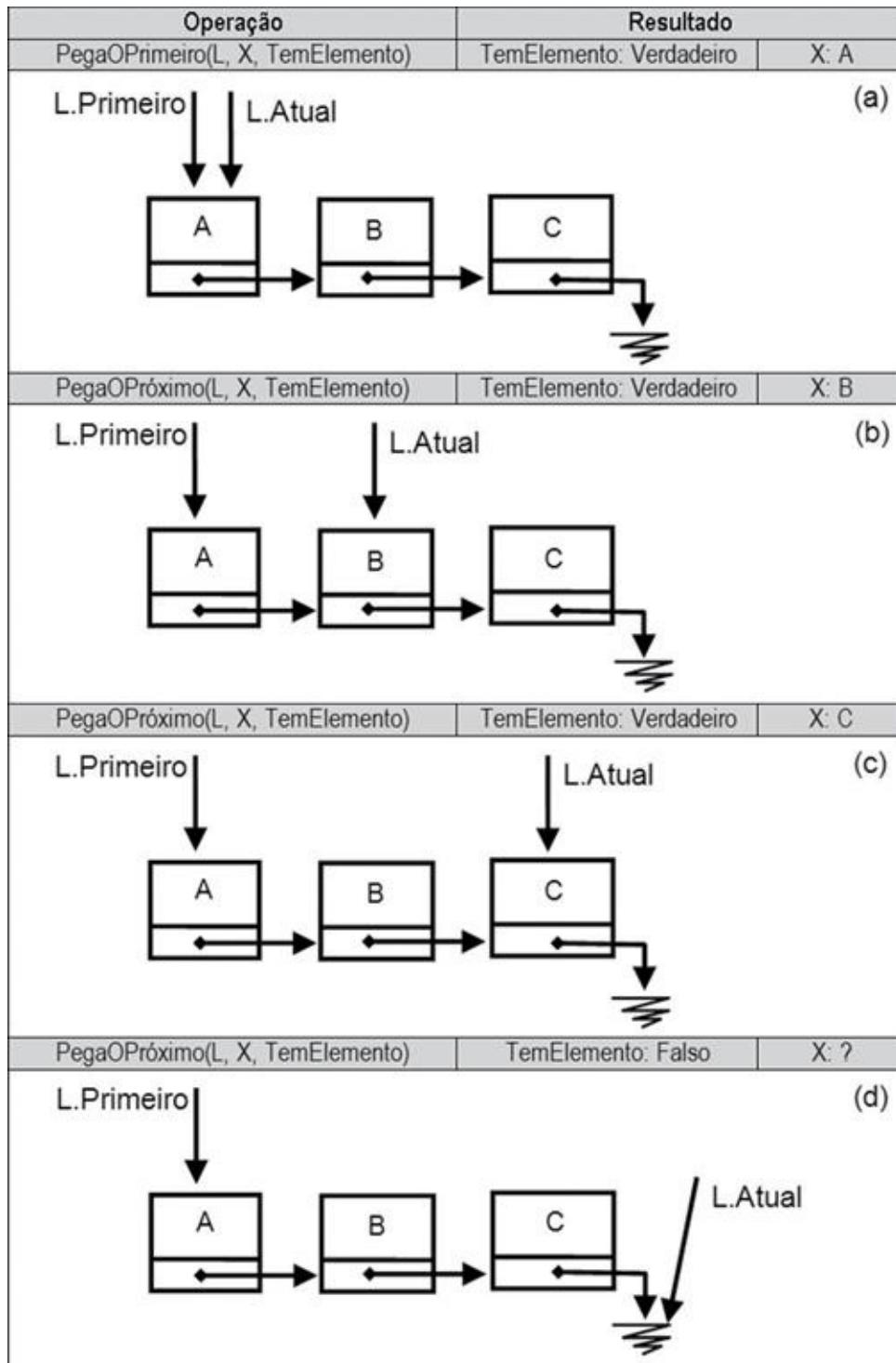
```
Boolean Cheia (parâmetro por referência L do tipo Lista);
/* Retorna Verdadeiro se a Lista Cadastral L estiver cheia, ou
seja, se na estrutura de armazenamento não couber mais nenhum
elemento; Retorna Falso caso contrário. Na implementação com
alocação encadeada e dinâmica de memória, podemos considerar que a
estrutura de armazenamento nunca ficará cheia e testar o sucesso da
operação que aloca memória dinamicamente para inserir um elemento
na Lista */
```

## Implementando operações para percorrer a Lista Cadastral

Conforme especificado na [Figura 6.4](#) e exemplificado nas [Figuras 6.6, 6.7](#) e [6.9](#), para percorrer uma Lista Cadastral precisamos acionar as operações PegaOPrimeiro e PegaOPróximo repetidas vezes. O resultado da execução da operação PegaOPróximo é dependente do resultado da chamada anterior a uma das operações — PegaOPrimeiro ou PegaOPróximo.

Para agregar a funcionalidade de percorrer a Lista, além de um ponteiro para o início da estrutura, precisamos de um segundo ponteiro permanente. Assim, a Lista Cadastral L será composta por dois ponteiros: L.Primeiro e L.Atual. O ponteiro L.Primeiro apontará sempre o primeiro elemento da Lista Cadastral. O ponteiro L.Atual será utilizado nas operações de percorrer a Lista.

A [Figura 6.23](#) mostra o resultado da execução de uma sequência de chamadas às operações PegaOPrimeiro e PegaOPróximo, sobre uma Lista Cadastral sem elementos repetidos implementada como uma Lista Encadeada Ordenada. A Lista L contém três elementos: A, B e C. O Ponteiro L.Primeiro sempre ficará apontando o primeiro elemento da Lista. Ao executarmos a operação PegaOPrimeiro, o ponteiro L.Atual passará a apontar para onde aponta L.Primeiro. No exemplo da Lista com três elementos da [Figura 6.23](#), a chamada à operação PegaOPrimeiro resultaria na situação da [Figura 6.23a](#); o parâmetro TemElemento retornaria o valor Verdadeiro, e o parâmetro X retornaria o valor 'A'. Isso indica que existe um primeiro elemento na Lista L e seu valor é 'A'.



**FIGURA 6.23** Implementação das operações PegaOPrimeiro e PegaOPróximo de Lista Cadastral implementada como Lista Encadeada Ordenada.

Executamos em seguida a operação PegaOPróximo e recebemos como resultado Verdadeiro (ou seja, temos um próximo elemento) e 'B' (o valor desse

próximo elemento é 'B'). A situação dos ponteiros L.Primeiro e L.Atual ficaria conforme mostra a [Figura 6.23b](#). Executamos mais uma vez a operação PegaOPróximo e temos como resultado Verdadeiro/'C' — [Figura 6.23c](#). Executamos ainda mais uma vez a operação PegaOPróximo. Dessa vez, o parâmetro TemElemento retornará o valor Falso, indicando que, em relação à última chamada (que retornou o valor 'C'), não há um próximo elemento. O ponteiro L.Atual estará apontando para o valor Null — [Figura 6.23d](#) —, indicando que a Lista acabou de ser percorrida.

Se chamarmos novamente a operação PegaOPrimeiro, o ponteiro L.Atual passará a apontar para onde aponta o ponteiro L.Primeiro. Esteja onde estiver o ponteiro L.Atual, a operação PegaOPrimeiro o levará para a posição do ponteiro L.Primeiro.

Considerando como situação inicial a Lista com três elementos da [Figura 6.23](#), se chamarmos 10 vezes, repetidamente, a operação PegaOPrimeiro, receberemos como resultado sempre esses mesmos valores: verdadeiro/'A', e ambos os ponteiros — L.Primeiro e L.Atual — estarão sempre apontando para o primeiro elemento da Lista. Se, em seguida, executarmos 10 vezes, repetidamente, a operação PegaOPróximo, teremos como resultados Verdadeiro/'B' (L.Atual apontando para o segundo elemento), Verdadeiro/'C' (L.Atual apontando para o terceiro elemento) e, nas próximas sete chamadas, teremos como resultado Falso/? (com L.Atual apontando para Null). O ponteiro L.Primeiro estará sempre apontando para o primeiro elemento da Lista.

Na situação em que a Lista L estiver vazia, ambos os ponteiros — L.Primeiro e L.Atual — apontarão para Null, e em qualquer chamada às operações PegaOPrimeiro ou PegaOPróximo o parâmetro TemElemento retornará o valor Falso.

## **Exercício 6.11 Operação que retorna o valor do primeiro elemento de uma Lista Cadastral, implementada como uma Lista Encadeada Ordenada**

Conforme especificado na [Figura 6.4](#), a operação PegaOPrimeiro retorna o valor do primeiro elemento da Lista L no parâmetro X, se esse primeiro elemento existir. Se não existir esse primeiro elemento (Lista vazia), o parâmetro TemElemento retornará o valor Falso.

PegaOPrimeiro (parâmetro por referência L do tipo Lista, parâmetro por referência X do tipo Char, parâmetro por referência TemElemento do tipo Boolean);

```
/* Caso a lista estiver vazia, TemElemento retorna o valor Falso.  
Caso a lista não estiver vazia, TemElemento retornará Verdadeiro, e  
o valor do primeiro elemento da Lista retornará no parâmetro X */
```

Uma possível solução para o [Exercício 6.11](#) é apresentada na [Figura 6.24](#). O primeiro passo é mover o ponteiro L.Atual para a posição do ponteiro L.Primeiro (`L.Atual = L.Primeiro`). O próximo passo é verificar se existe o primeiro elemento da Lista, ou seja, se `L.Atual != Null`. Se existir o primeiro elemento da Lista, o ponteiro L.Atual será diferente de Null e estará apontando para esse primeiro elemento. Caso não existir um primeiro elemento na Lista, o ponteiro L.Atual estará apontando para Null.

```
PegaOPrimeiro (parâmetro por referência L do tipo Lista, parâmetro por referência X do  
tipo Char, parâmetro por referência TemElemento do tipo Boolean) {  
  
/* Caso a lista estiver vazia, TemElemento retorna o valor Falso. Caso a lista não estiver  
vazia, TemElemento retornará Verdadeiro, e o valor do primeiro elemento da Lista  
retornará no parâmetro X */  
  
L.Atual = L.Primeiro;      // L.Atual passa a apontar para onde aponta L.Primeiro  
Se (L.Atual != Null)        // verifica se existe um primeiro elemento.... se existir,  
Então { TemElemento = Verdadeiro;    // ... TemElemento retornará Verdadeiro;;;  
        X = L.Atual→Info; }    //... e X retornará o valor do primeiro elemento  
Senão  Tem Elemento = Falso;  
} // fim PegaOPrimeiro
```

**FIGURA 6.24** Algoritmo da operação PegaOPrimeiro, de uma Lista Cadastral implementada como Lista Encadeada Ordenada.

## Exercício 6.12 Operação que retorna o valor do próximo elemento de uma Lista Cadastral, implementada como uma Lista Encadeada Ordenada

Conforme especificado na [Figura 6.4](#), a operação PegaOPróximo retorna o valor do próximo elemento da Lista, em relação à última chamada a uma das operações PegaOPrimeiro ou PegaOPróximo. Se não existir esse próximo elemento (lista vazia ou final da Lista), o parâmetro TemElemento retornará o valor Falso.

PegaOPróximo (parâmetro por referência L do tipo Lista, parâmetro por referência X do tipo Char, parâmetro por referência TemElemento do tipo Boolean);

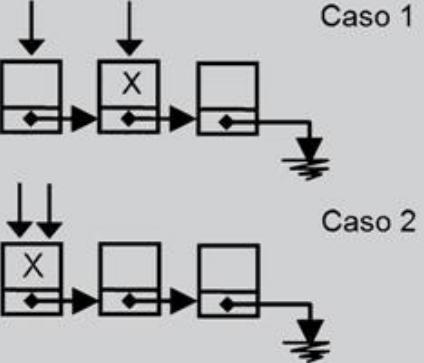
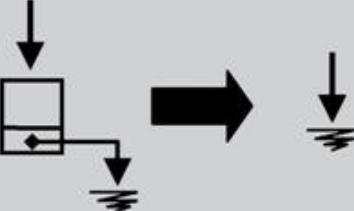
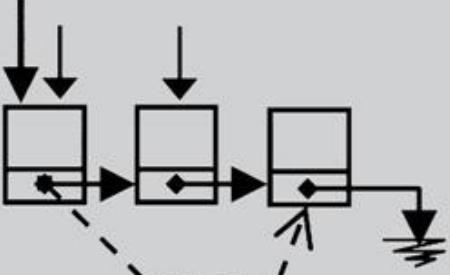
```
/* Caso a lista não estiver vazia e caso houver um próximo elemento em relação à última chamada de PegaOPrimeiro ou PegaOProximo, TemElemento retornará Verdadeiro, e o valor do próximo elemento da Lista retornará no parâmetro X. Caso a lista estiver vazia ou caso não houver um próximo elemento em relação à última chamada, o parâmetro TemElemento retornará o valor Falso */
```

## Exercício 6.13 Revisar os Exercícios 6.5 a 6.10 adaptando as soluções para uma Lista com dois ponteiros

Na implementação dos Exercícios 6.5 a 6.10, utilizamos uma Lista L com um único ponteiro permanente — o ponteiro L. Contudo, para possibilitar a implementação das operações para percorrer a Lista, utilizamos a Lista L com dois ponteiros: L.Primeiro e L.Atual. Ajuste as soluções propostas para os Exercícios 6.5 a 6.10 adotando agora uma Lista L com dois ponteiros permanentes — L.Primeiro e L.Atual.

## 6.4 Outras implementações de Lista Cadastral

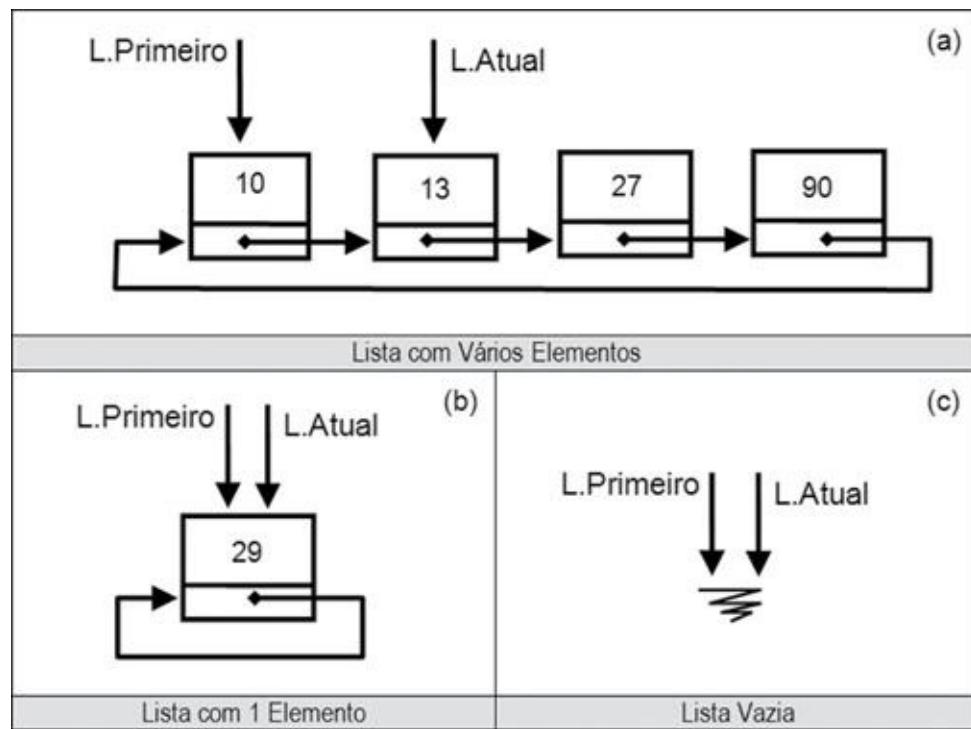
Na Seção 6.3, programamos uma Lista Cadastral como uma Lista Encadeada Ordenada. Lista Encadeada Ordenada foi a *técnica de implementação* utilizada. Podemos implementar a mesma Lista Cadastral com outra técnica — por exemplo, com Alocação Sequencial e Estática de Memória. Mas como estamos interessados em ganhar experiência nas implementações encadeadas, vamos propor a implementação de uma Lista Cadastral com técnicas ligeiramente diferentes das adotadas nos exercícios anteriores: como uma Lista Encadeada Circular Ordenada ou como uma Lista Encadeada Circular não ordenada. Também iremos propor a implementação de uma Lista Cadastral com elementos repetidos. Ao desenvolver as soluções para os exercícios, siga as orientações da Figura 6.25.

Passos para construir uma boa solução	
<p><b>Passo 1: identificar casos e desenhar a situação inicial.</b> Identifique todos os casos, enumere como Caso 1, Caso 2, Caso 3 etc. Faça um desenho da situação inicial de cada caso; pense sempre nas situações de lista vazia, lista com um único elemento, inserir ou eliminar no começo, no meio e no final da lista, encontrar ou não encontrar o elemento na lista, inserir o primeiro, retirar o único, e assim por diante.</p>	
<p><b>Passo 2: desenho da situação final.</b> Para cada caso identificado, faça um desenho também da situação final desejada, ou seja, de como o desenho deverá ficar após a execução da operação que você está projetando.</p>	
<p><b>Passo 3: algoritmo para tratar cada caso separadamente.</b> Identifique, para cada caso, o trecho de algoritmo necessário para levar o desenho da situação inicial para a situação final.</p>	<p>Para tratar o Caso 2:</p> <ul style="list-style-type: none"> <li>• DeleteNode(P);</li> <li>• L=NULL;</li> </ul>
<p><b>Passo 4: faça um algoritmo geral do modo mais simples.</b> Só faça o algoritmo geral após identificar todos os casos, desenhar a situação inicial e final, e após identificar os trechos de algoritmo necessários para tratar cada caso, individualmente. Não pense em fazer o algoritmo do modo "mais curto"; pense em fazer do modo "mais simples", tratando separadamente cada caso.</p>	<p>Algoritmo geral</p> <p>Se Caso 1 Então [tratar Caso 1] Senão Se Caso 2 Então [tratar Caso 2] Senão Se Caso 3 Então [tratar Caso 3] Senão...</p>
<p><b>Passo 5: testar cada caso alterando o desenho passo a passo.</b> Após elaborar o algoritmo completo, teste passo a passo, fazendo desenhos. Teste para todos os casos identificados. Para cada caso, parte da situação inicial e execute o algoritmo. A cada comando, altere o desenho. Ao final da execução, verifique se o desenho chegou à situação final pretendida.</p>	

**FIGURA 6.25** Passos para construir uma boa solução.

### Exercício 6.14 Lista Cadastral sem elementos repetidos implementada como uma Lista Encadeada Ordenada Circular

Implemente uma Lista Cadastral sem elementos repetidos através de uma Lista Encadeada Ordenada Circular. Conforme mostram os diagramas da Figura 6.26, em uma Lista Circular o campo Next do último Nó da Lista deve apontar para o primeiro Nó da Lista, em vez de apontar para Null (Figuras 6.26a e 6.26b). Quando a Lista estiver vazia, os ponteiros L.Primeiro e L.Atual devem apontar para Null (Figura 6.26c). Implemente as operações Cria, Vazia, Cheia, EstáNaLista, Insere, Retira, PegaOPrimeiro e PegaOPróximo.



**FIGURA 6.26** Lista Cadastral implementada como Lista Encadeada Ordenada Circular.

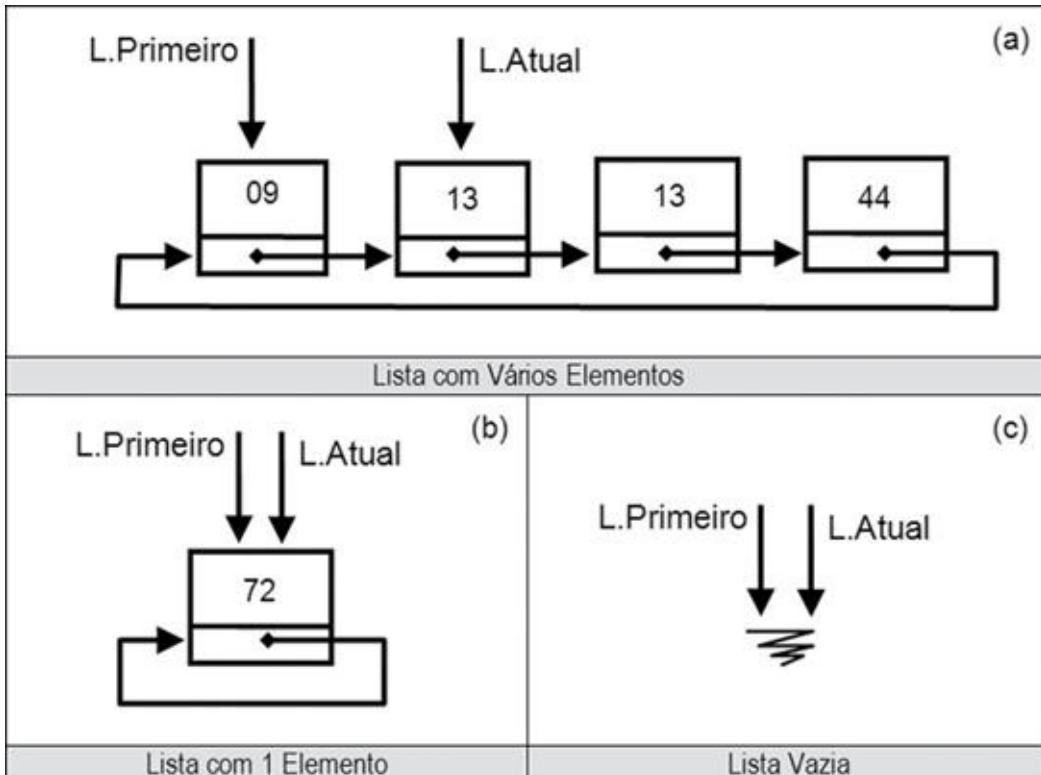
### Exercício 6.15 Operação Destrói

Considerando uma Lista Cadastral implementada como uma Lista Encadeada Ordenada e Circular, conforme os diagramas da [Figura 6.26](#), desenvolva a operação Destroi, que remove todos os nós da Lista, independentemente do seu valor. Implemente da forma mais apropriada para proporcionar portabilidade e reusabilidade.

```
Destroi (parâmetro por referência L do tipo Lista);  
/* Remove (desaloca) todos os Nós da Lista L */
```

### **Exercício 6.16 Lista Cadastral com elementos repetidos implementada como uma Lista Encadeada Ordenada Circular**

Implemente uma Lista Cadastral com elementos repetidos, através de uma Lista Encadeada Ordenada Circular. Conforme mostra o diagrama da [Figura 6.27a](#), a lista deve permitir a entrada de elementos repetidos. Considerando que a lista é ordenada, os elementos repetidos devem estar juntos. Implemente as operações Cria, Vazia, Cheia, EstáNaLista, Insere, Retira, PegaOPrimeiro e PegaOPróximo. A operação Retira deve remover um único elemento, ainda que a Lista contenha outros elementos com o mesmo valor.



**FIGURA 6.27** Lista Cadastral com elementos repetidos.

### Exercício 6.17 Operação RetiraTodosDeValorX de uma Lista Cadastral com elementos repetidos

No [Exercício 6.16](#) desenvolvemos a operação Retira, que remove um único elemento da Lista. Mas como a Lista permite elementos repetidos, implemente agora a operação RetiraTodosComValorX, que elimina não apenas um, mas todos os elementos da Lista que tiverem valor igual a um valor fornecido. Implemente da forma mais apropriada para proporcionar portabilidade e reusabilidade.

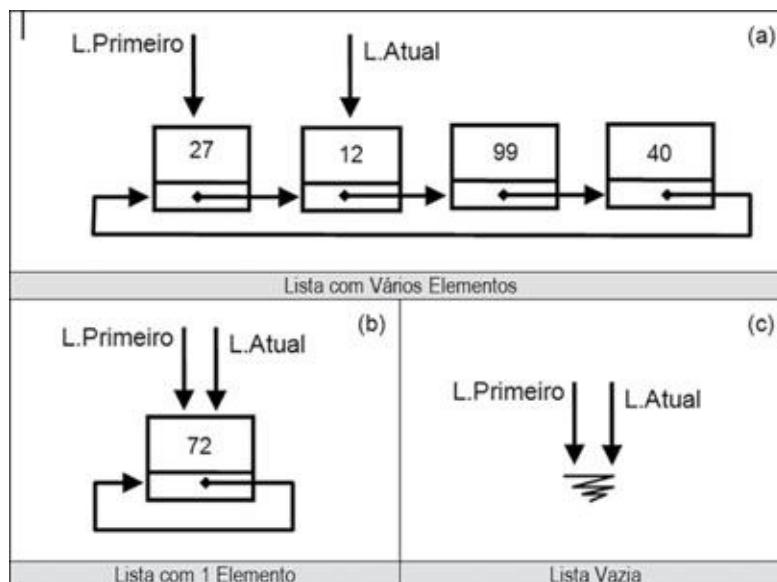
RetiraTodosComValorX (parâmetro por referência L do tipo Lista, parâmetro X do tipo Char, parâmetro por referência Ok do tipo Boolean);

```
/* Retira todos os elementos de valor X que forem encontrados na
Lista L. Se algum elemento de valor X for encontrado e removido, Ok
retorna Verdadeiro. Se nenhum elemento de valor X for encontrado na
Lista L, não retira nenhum elemento e Ok retorna o valor Falso */
```

### Exercício 6.18 Lista Cadastral sem elementos repetidos,

## implementada como uma Lista Encadeada Circular Não Ordenada

Implemente uma Lista Cadastral sem elementos repetidos, através de uma Lista Encadeada Circular Não Ordenada, conforme ilustram os diagramas da [Figura 6.28](#). Note, na [Figura 2.28a](#), que os elementos não estão ordenados. Implemente as operações Cria, Vazia, Cheia, EstáNaLista, Insere, Retira, PegaOPrimeiro e PegaOPróximo.



**FIGURA 6.28** Lista Cadastral implementada como uma Lista Encadeada Circular Não Ordenada.

## Exercício 6.19 Implementar e testar uma Lista Cadastral em uma linguagem de programação

Implemente uma Lista Cadastral em uma linguagem de programação, como C ou C++. Os elementos da Lista devem ser do tipo inteiro. Você pode escolher implementar sua Lista Cadastral como uma Lista Encadeada Circular ou Não Circular, Ordenada ou Não Ordenada, com ou sem elementos repetidos. Implemente a Lista Cadastral como uma unidade independente. Em um arquivo separado, faça o programa principal bem simples, para testar o funcionamento da Lista. O programa principal deve manipular a Lista exclusivamente pelos operadores primitivos: Cria, Vazia, Cheia, EstáNaLista, Insere, Retira, PegaOPrimeiro e PegaOPróximo.

## 6.5 Avançando o Projeto *Spider Shopping*

No game proposto no Desafio 3 — *Spider Shopping*, inicialmente o jogador recebe uma Lista de Compras. O jogador pode então ajustar essa Lista, retirando itens que desconhece e acrescentando itens que pode reconhecer facilmente. Por exemplo, se o jogador não sabe o que é um *Esguicho*, pode remover esse item da Lista de Compras.

A Lista de Compras funciona como uma Lista Cadastral: quando o jogador solicita a remoção de um elemento X, precisamos retirar precisamente esse elemento X. Na Lista de Compras, não temos elementos repetidos. Assim, podemos implementar a Lista de Compras como um Tipo Abstrato de Dados (TAD) Lista Cadastral sem elementos repetidos.

Na sequência do jogo, uma vitrine mostrará ao jogador imagens de uma série de produtos. O jogador deverá “comprar” apenas os produtos que fazem parte da Lista de Compras. Se adicionar ao seu Carrinho de Compras um produto que consta da Lista, o jogador ganha pontos; se adicionar ao Carrinho um produto que não faz parte da Lista de Compras, o jogador perde pontos.

O jogador também perde pontos se comprar mais de uma vez um produto que faz parte da Lista de Compras. Ou seja, o Carrinho de Compras pode ter elementos repetidos. Assim, podemos implementar o Carrinho de Compras como um TAD Lista Cadastral com elementos repetidos

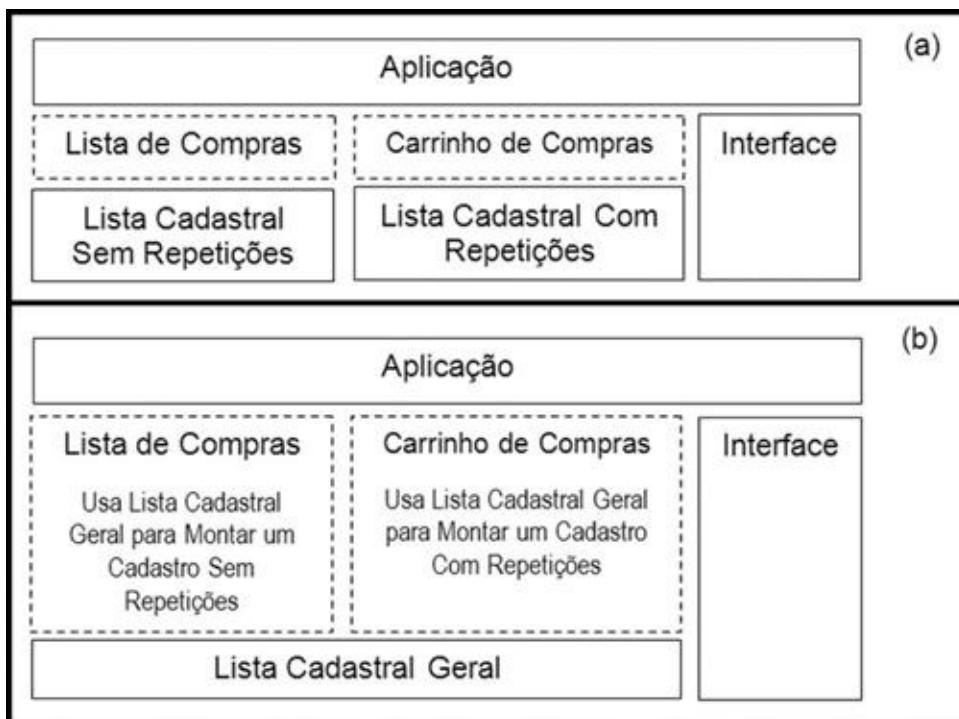
<p>Ywz Xx Yz <input type="checkbox"/></p> <p>Xxxxxx XX xxx <input type="checkbox"/></p> <p>yYY yyy YY <input checked="" type="checkbox"/> ✓</p> <p>xXwz Ywx xXwz <input type="checkbox"/></p> <p>ZZx XwX Zzz <input type="checkbox"/></p>	
<p><b>Lista de Compras: Lista Cadastral Sem Elementos Repetidos</b></p>	<p><b>Carrinho de Compras: Lista Cadastral Com Elementos Repetidos</b></p>

**FIGURA 6.29** Lista de Compras e Carrinho de Compras implementados como Listas Cadastrais.

Nos exercícios anteriores já implementamos uma Lista Cadastral com elementos repetidos e também sem elementos repetidos. As implementações são diferentes apenas em alguns dos aspectos.

Uma decisão a ser tomada no projeto do *Spider Shopping* refere-se à

arquitetura do software. Podemos ter implementações totalmente independentes para as Listas (uma sem repetições e outra com repetições), como na [Figura 6.30a](#). Poderíamos também ter uma Lista Cadastral Geral, que pode ser utilizada tanto como uma Lista sem repetições quanto como uma Lista com repetições, como na [Figura 6.30b](#). Qual dessas duas alternativas — A ou B — você considera mais adequada ao seu projeto? Você pode ainda pensar em uma outra alternativa.



**FIGURA 6.30** Projeto *Spider Shopping* — soluções alternativas quanto à arquitetura do software.

## Exercício 6.20 Avançar o Projeto *Spider Shopping* — propor uma arquitetura de software

Proponha uma arquitetura de software para o seu projeto do Desafio 3, identificando os principais módulos do sistema e o relacionamento entre eles. Dentre outros módulos, defina Tipos Abstratos de Dados para implementar a Lista de Compras e o Carrinho de Compras. Sugestão para uso acadêmico: desenvolva o projeto em grupo. Promova uma discussão, defina a arquitetura do software e uma divisão de trabalho entre os componentes do grupo.

Seja qual for a arquitetura de software que você adotar, visando a agregar

portabilidade e reusabilidade, observe as seguintes recomendações na implementação:

- Implemente a Lista de Compras e o Carrinho de compras como um Tipo Abstrato de Dados (TAD), ou seja, a aplicação (e outros módulos) deve manipular a Lista de Compras e o Carrinho de Compras exclusivamente através dos Operadores Primitivos — Insere, Retira, EstáNaLista, PegaOPrimeiro, e assim por diante. Aumente o volume da televisão apenas pelo botão de volume.
- Inclua no código dos TADs (Lista de Compras e Carrinho de Compras) exclusivamente ações pertinentes ao armazenamento e recuperação das informações sobre os itens que o jogador deve comprar, e sobre os itens que foram efetivamente comprados. Faça o possível para não incluir nos TADs ações relativas à interface ou a qualquer outro aspecto.
- Os TADs que implementam a Lista de Compras e o Carrinho de Compras devem estar em unidades de software independentes e em arquivos separados.
- Procure implementar a interface também em um módulo independente.

## **Exercício 6.21 Avançar o Projeto *Spider Shopping* — calcular o número de itens da Lista de Compras que foram efetivamente comprados pelo jogador**

Após a etapa de compras, para calcular a pontuação obtida pelo jogador é preciso comparar o Carrinho de Compras com a Lista de Compras para verificar o número de itens da Lista que foram efetivamente comprados. Ou seja, é preciso calcular quantos itens da Lista de Compras aparecem pelo menos uma vez no Carrinho de Compras. Implemente essa operação da forma mais apropriada para proporcionar portabilidade e reusabilidade. Antes de iniciar o desenvolvimento, consulte a solução do [Exercício 6.2](#), que calcula a interseção entre duas listas L1 e L2.

```
Inteiro ItensDaListaQueForamComprados (parâmetros por referência
ListaDeCompras, CarrinhoDeCompras do tipo Lista);
/* Recebe a Lista de Compras e o Carrinho de Compras e conta
quantos itens da Lista de Compras aparecem pelo menos uma vez no
Carrinho de Compras. Produz resultado do tipo inteiro. */
```

## **Exercício 6.22 Avançar o Projeto *Spider Shopping* — calcular o número de itens comprados incorretamente**

Implemente da forma mais apropriada para proporcionar portabilidade e reusabilidade uma operação que calcula o número de itens da Lista de Compras que foram comprados incorretamente, ou seja, o número de itens que estão no Carrinho de Compras, mas não estão na Lista de Compras. Antes de iniciar o desenvolvimento, consulte a solução do [Exercício 6.4](#), que seleciona os elementos de uma Lista L2 que não estão em uma Lista L1.

```
Inteiro ItensCompradosIncorretamente (parâmetros por referência  
ListaDeCompras, CarrinhoDeCompras do tipo Lista);  
/* Recebe a Lista de Compras e o Carrinho de Compras e conta  
quantos itens do Carrinho de Compras não fazem parte da Lista de  
Compras. Produz resultado do tipo inteiro */
```

### **Exercício 6.23 Avançar o Projeto *Spider Shopping* — calcular o número de itens comprados em excesso**

O jogador perderá pontos a cada item da Lista de Compras que foi comprado mais de uma vez. Implemente da forma mais apropriada para proporcionar portabilidade e reusabilidade uma operação que calcula o número de itens que foram comprados em excesso, ou seja, o número de itens da Lista de Compras que aparecem no Carrinho de Compras mais de uma vez.

```
Inteiro ItensCompradosEmExcesso (parâmetros por referência  
ListaDeCompras, CarrinhoDeCompras do tipo Lista);  
/* Recebe a Lista de Compras e o Carrinho de Compras e conta  
quantos itens foram comprados em Excesso, ou seja, número de itens  
da Lista de Compras que aparecem mais de uma vez no Carrinho de  
Compras. Produz resultado do tipo inteiro */
```

### **Exercício 6.24 Identificar outras aplicações de Listas Cadastrais**

Identifique alguns jogos que podem ser implementados com o uso de uma ou mais Listas Cadastrais. Identifique também outras aplicações fora do mundo dos games. Sugestão de uso acadêmico: faça uma discussão em grupo. Ao final, cada grupo apresenta a todos os estudantes um novo projeto de jogo que ilustre bem a estrutura Lista Cadastral.

### **Exercício 6.25 Avançar o Projeto — defina as regras, escolha um nome e inicie o desenvolvimento do seu jogo**

Altere o funcionamento do *Spider Shopping* em algum aspecto. Dê personalidade própria ao seu jogo e escolha um nome que reflita essa

personalidade própria. Você pode até criar um jogo totalmente novo. Mas, para cumprir os propósitos acadêmicos pretendidos no Desafio 3, mantenha a característica fundamental: um jogo que utilize uma ou mais Listas Cadastrais. Sugestão para uso acadêmico: desenvolva o projeto em grupo. Tome as principais decisões em conjunto e divida o trabalho entre os componentes do grupo, cada qual ficando responsável por parte das atividades.

Inicie agora o desenvolvimento do seu jogo referente ao Desafio 3!

## Consulte nos Materiais Complementares



Vídeos sobre Listas Cadastrais



Animações sobre Listas Cadastrais

Banco de Jogos: Aplicações de Listas



<http://www.elsevier.com.br/edcomjogos>

## Exercícios de fixação

**Exercício 6.26** Qual a diferença fundamental entre uma Lista Cadastral e uma Fila?

**Exercício 6.27** Uma Lista Cadastral pode ser implementada como uma Lista Encadeada, mas pode ser implementada também com outra técnica. Cite um exemplo de como isso pode acontecer. Explique a diferença entre os nomes Lista Cadastral e Lista Encadeada.

**Exercício 6.28** Faça um diagrama ilustrando a implementação de uma Lista Cadastral como uma Lista Encadeada Ordenada. Depois faça um segundo diagrama ilustrando a implementação de uma Lista Cadastral como uma Lista Encadeada Circular Não Ordenada. Os diagramas devem mostrar a Lista nas situações vazia, com um único elemento e com vários elementos.

**Exercício 6.29** Como seria a implementação de uma Lista Cadastral com Alocação Sequencial e Estática de Memória? Faça um diagrama e explique como seria o funcionamento.

**Exercício 6.30** Compare a implementação de uma Lista Cadastral através de uma Lista Encadeada com a implementação de uma Lista Cadastral com Alocação Sequencial e Estática de Memória. Qual técnica de implementação você acha mais interessante e por quê?

## Soluções para alguns dos exercícios

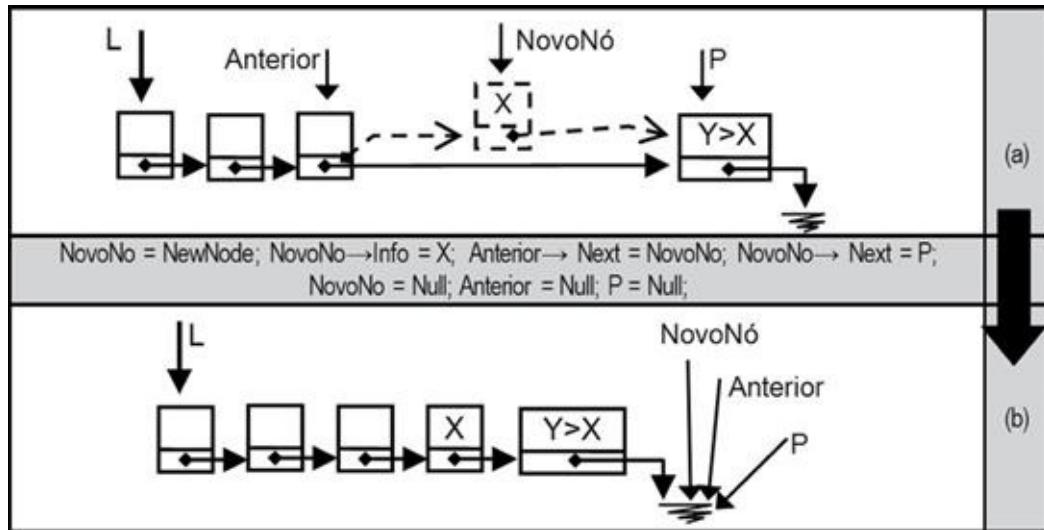
### **Exercício 6.6 Operação Insere Elemento em uma Lista Cadastral implementada como uma Lista Encadeada Ordenada**

É possível utilizar o mesmo procedimento “ProcuraX” utilizado na operação que Retira elemento da Lista. Se não for encontrado elemento na Lista, o ProcuraX terá posicionado o parâmetro P em uma posição à frente daquela em que o novo elemento deve ser inserido, e o ponteiro Anterior uma posição atrás.

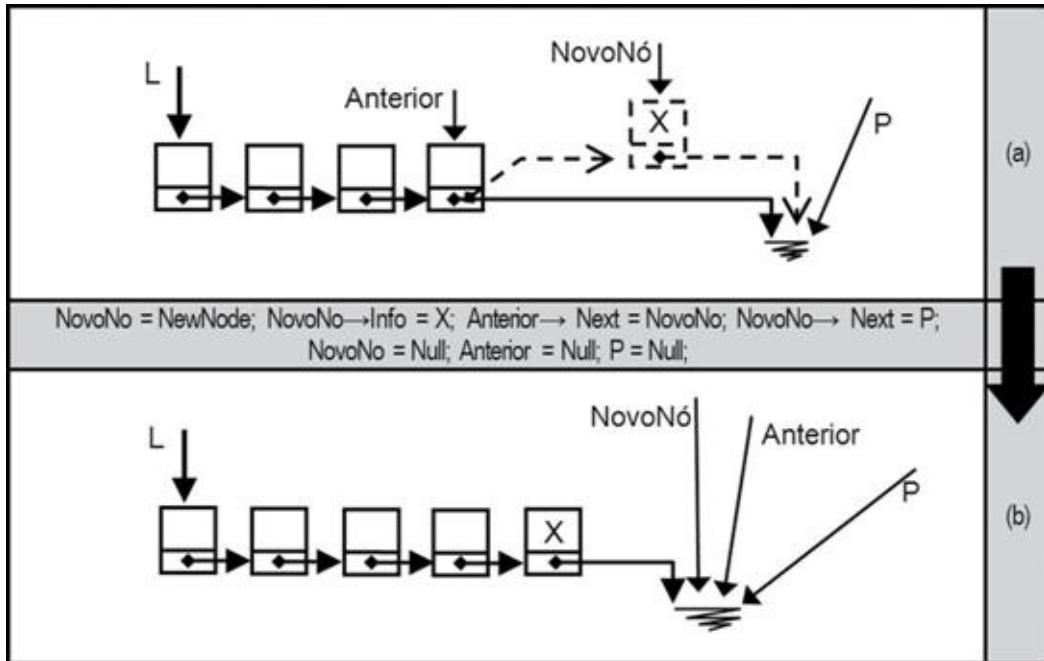
Algoritmo geral em dois passos — Insere Elemento

- Passo 1: definir variáveis temporárias P e Anterior. Anterior começa em Null, P começa em L. Avança Anterior e P até que P chegue ao Nó que contém X ou a um Nó que contém um valor  $Y > X$ , ou ao final da Lista (Null). Anterior avança sempre uma posição atrás de P.
- Passo 2: com base na posição de Anterior e P, e com base no valor armazenado no Nó apontado por P (caso P for diferente de Null), identificar o caso (Caso 1, Caso 1', Caso 2, Caso 3 ou Caso 4) e executar as ações necessárias.

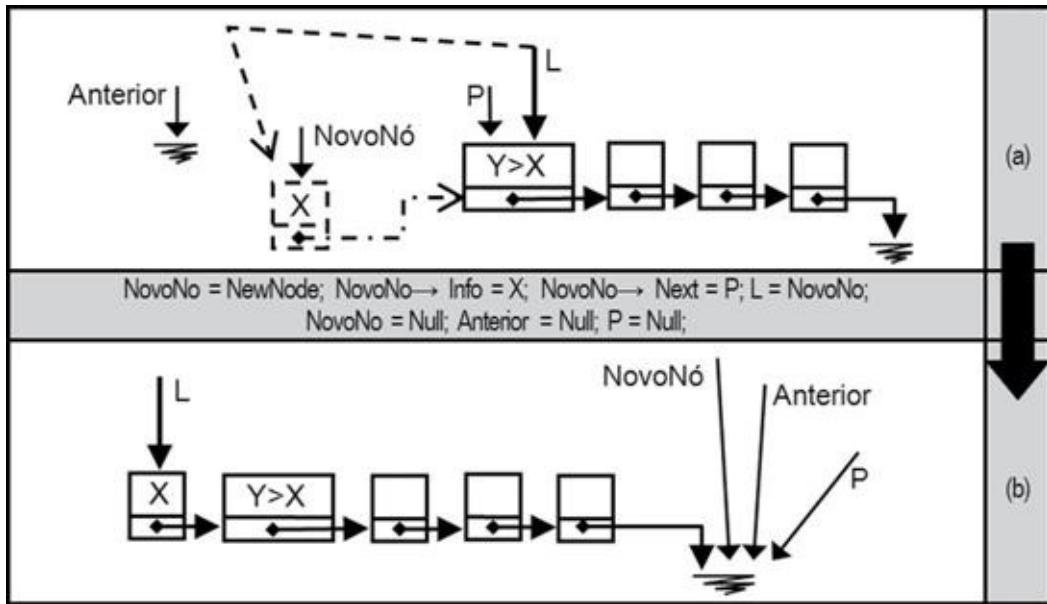
Caso 1: X não está na Lista e deve ser inserido no meio da Lista



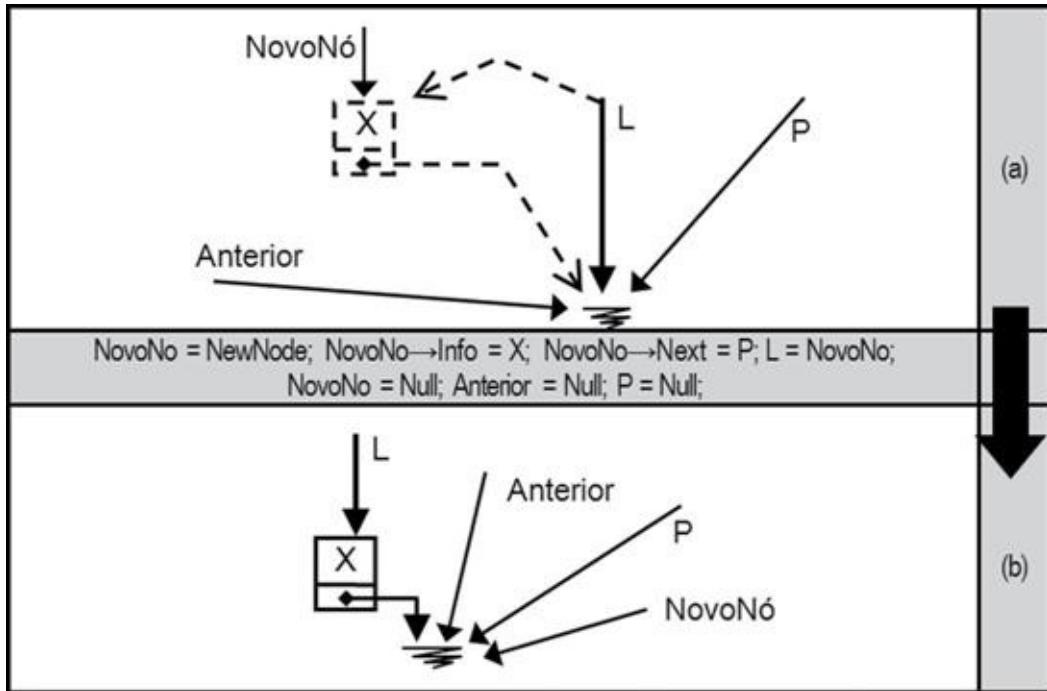
Caso 1': X não está na Lista e deve ser inserido no final da Lista



Caso 2: X Não está na Lista e deve ser inserido no início da Lista;



Caso 3: A lista está vazia



### Principais casos:

- Caso 1: X não está na Lista e deve ser inserido no meio da Lista.
- Caso 1': X não está na Lista e deve ser inserido no final da Lista.
- Caso 2: X Não está na Lista e deve ser inserido no início da Lista.

- Caso 3: A lista está vazia.
  - Caso 4: X já está na Lista.
- Algoritmo conceitual — Insere Elemento

```

Insere (parâmetro por referência L do tipo Lista, parâmetro X do tipo Char, parâmetro por referência Ok do tipo Boolean) {
/* Caso o valor X já não estiver na Lista L, insere X e Ok retorna Verdadeiro. Se X já estiver na Lista
L, não insere nenhum elemento e Ok retorna o valor Falso */

Variáveis P, Anterior, NovoNo do tipo NodePtr; // Tipo NodePtr = ponteiro para Nó
Variável AchouX do tipo Boolean;

ProcuraX (L, X, P, Anterior, AchouX) ;
/* ProcuraX executa o passo 1: encontrar X na Lista L. ProcuraX indica, através da variável
AchouX, se X foi encontrado em L (Verdadeiro) ou não (Falso). Se X não for encontrado, ProcuraX
colocará P apontando para a posição imediatamente posterior ao local onde X deve ser inserido, e
Anterior apontando a posição imediatamente anterior, conforme os diagramas dos Casos 1, 1', 2 e
3 */

/* Passo 2: se não encontrou X na Lista, insere o valor X */
Se (AchouX == Falso)      // se X não foi encontrado na Lista, então insere

/* No Caso 1 P é diferente de L, e P é diferente de Null, pois foi encontrado um valor Y > X. No
Caso 1', P é diferente de L, e P é igual a Null, pois X é maior que todos os valores da Lista. Em
ambos os casos, P é diferente de L. O algoritmo é o mesmo para tratar os Casos 1 e 1'. */

Então {  Se (P != L)      // Casos 1 ou 1': insere X no meio ou no último Nó da Lista
        Então {  NovoNo = Newnode;
                  NovoNo → Info = X;
                  Anterior → Next = NovoNo;
                  NovoNo → Next = P;
                  NovoNo = Null;
                  Anterior = Null;
                  P = Null;
        }
/* no Caso 2 P é igual a L e ambos são diferentes de Null. No Caso 3, P é igual a L e ambos são
iguais a Null. Em ambos os casos, P é igual a L. O algoritmo é o mesmo para tratar os Casos 2 e 3
*/
        Senão {  NovoNo = NewNode;      // Casos 2 ou 3: insere X no início...
                  NovoNo → Info = X; // ...ou como único elemento da lista
                  NovoNo → Next = P;
                  L = NovoNo;
                  NovoNo = Null;
                  Anterior = Null;
                  P = Null;
        }
        Ok = Verdadeiro;    // X foi inserido nessa operação
}
Senão  Ok = Falso; // X já estava na Lista, então nenhum valor é inserido nessa operação
} // fim Insere

```

## Exercício 6.7 Operação que verifica se um elemento faz parte de uma Lista Cadastral, implementada como uma

## **Lista Encadeada Ordenada**

```
Boolean EstáNaLista (parâmetro por referência L do tipo Lista, parâmetro X do tipo Char) {  
    /* Caso X for encontrado na Lista L, retorna Verdadeiro. Retorna Falso caso X não estiver na Lista  
    L */  
  
    Variáveis P, Anterior do tipo NodePtr; // Tipo NodePtr = ponteiro para Nó  
    Variável AchouX do tipo Boolean;  
  
    ProcuraX (L, X, P, Anterior, AchouX); /* o mesmo ProcuraX utilizado no Retira e no Insere */  
  
    Se AchouX = Verdadeiro  
        Então Retorne Verdadeiro;  
        Senão Retorne Falso;  
    } // fim EstaNaLista
```

## **Exercício 6.8 Operação que cria a Lista Cadastral**

```
Cria (parâmetro por referência L do tipo Lista) {  
    /* Cria a Lista Cadastral L, inicializando a Lista como vazia —  
    sem nenhum elemento. */  
    L=NULL;  
} // fim Cria
```

## **Exercício 6.9 Operação para testar se a Lista está vazia**

```
Boolean Vazia (parâmetro por referência L do tipo Lista) {  
    /* Retorna Verdadeiro se a Lista L estiver vazia — sem nenhum  
    elemento; retorna Falso caso contrário */  
    Se (L == NULL)  
        Então Retorne Verdadeiro;  
        Senão Retorne Falso;  
    } // fim Vazia
```

## **Exercício 6.10 Operação para testar se a Lista está cheia**

```
Boolean Cheia (parâmetro por referência L do tipo Lista) {  
    /* Retorna Verdadeiro se a Lista Cadastral L estiver cheia, ou  
    seja, se na estrutura de armazenamento não couber mais nenhum  
    elemento; retorna Falso caso contrário. Na implementação com
```

alocação encadeada e dinâmica de memória, podemos considerar que a estrutura de armazenamento nunca ficará cheia e testar o sucesso da alocação dinâmica de memória na operação insere \*/  
Retorne Falso; } // fim Cheia

## **Exercício 6.12 Operação que retorna o valor do próximo elemento de uma Lista Cadastral, implementada como uma**

### **Lista Encadeada Ordenada**



```
PegaOPróximo (parâmetro por referência L do tipo Lista, parâmetro por referência X do tipo Char,  
parâmetro por referência TemElemento do tipo Boolean) {  
/* Caso a lista não estiver vazia e caso houver um próximo elemento em relação à última chamada de  
PegaOPrimeiro ou PegaOProximo, TemElemento retornará Verdadeiro, e o valor do próximo  
elemento da Lista retornará no parâmetro X. Caso a lista estiver vazia ou caso não houver um próximo  
elemento em relação à última chamada (final da Lista), o parâmetro TemElemento retornará o valor  
Falso. */  
  
// tenta avançar para o próximo elemento  
Se (L.Atual != Null)  
Então L.Atual = L.Atual→ Next;  
  
// verifica se tem elemento. Se tiver, retorna o valor em X  
Se (L.Atual == Null)  
Então TemElemento = Falso;  
Senão { TemElemento = Verdadeiro;  
        X = L.Atual→ Info;  
    }  
} // fim PegaOProximo
```

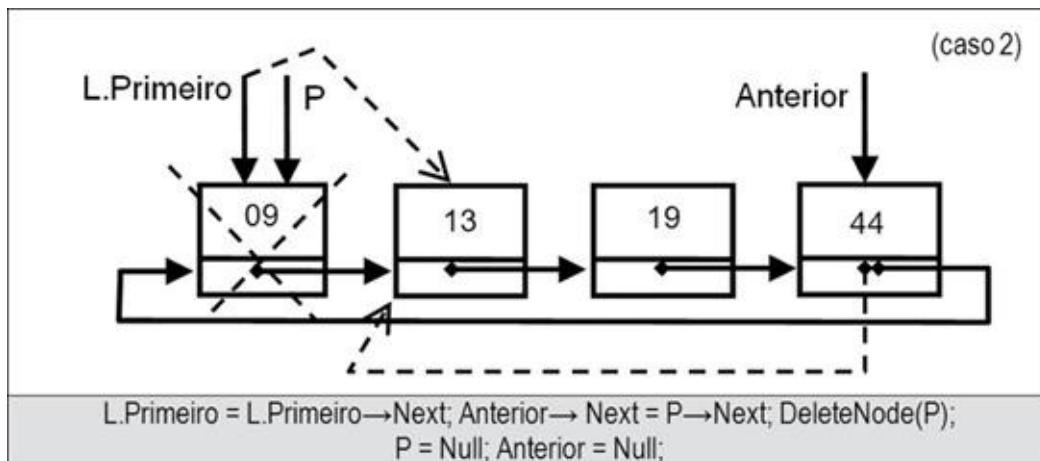
## **Exercício 6.13 Revisar os Exercícios 6.7 a 6.12 adaptando as soluções para uma Lista com dois ponteiros**

Basta substituir L por L.Primeiro, em todas as ocorrências. Na operação Cria, inicializar ambos os ponteiros — L.Primeiro e L.Atual com o valor Null.

## **Exercício 6.14 Lista Cadastral sem elementos repetidos implementada como uma Lista Encadeada Ordenada Circular**

Operação para retirar elemento — casos a tratar:

- Caso 1: Remover do meio da lista.
- Caso 1': Remover do final da lista.
- Caso 2: Remover o elemento que está no primeiro nó da lista, sendo que a lista possui vários elementos (veja o diagrama).
- Caso 3: Remover o elemento que está no primeiro nó da lista sendo que a lista contém um único elemento.
- Caso 4: Não achar X na lista.
- Caso 5: Lista vazia.



Comentários — Operação Retira:

- Neste exercício, um erro muito comum é o desenvolvedor esquecer de tratar o Caso 2 ou o Caso 3. São casos diferentes. No Caso 3, o ponteiro L precisa apontar para Null. No Caso 2, o ponteiro L precisa apontar para o próximo elemento.
- Também é comum, no Caso 2, o desenvolvedor esquecer de atualizar o campo Next do último nó da lista, que precisa apontar para o novo valor de L. A lista é circular. Faça o desenho passo a passo para não errar.
- Se estiver sendo retirado o elemento apontado por L.Atual, esse ponteiro também precisará ser atualizado.
- Na operação para inserir elementos, os casos e cuidados devem ser análogos: não esqueça de tratar o caso de inserir no início de uma Lista com vários elementos, e o caso de inserir em uma Lista vazia.

```

Retira (parâmetro por referência L do tipo Lista, parâmetro X do tipo Char, parâmetro por referência
Ok do tipo Boolean) {

/* Caso X for encontrado na Lista L, retira X da Lista e Ok retorna Verdadeiro. Se X não estiver na
Lista L, não retira nenhum elemento e Ok retorna o valor Falso */

Variáveis P, Anterior do tipo NodePtr; // Tipo NodePtr = ponteiro para Nó
Variável AchouX do tipo Boolean;
ProcuraX (L, X, P, Anterior, AchouX);

/* Passo 2: se encontrou X, remover da Lista o Nó que contém X */
Se (AchouX == Verdadeiro) // se X foi encontrado na Lista

Então { Se L.Primeiro = L.Primeiro→ Next) // Caso 3 - X for o único elemento na Lista
    Então { DeleteNode( P );
        Se (L.Atual == L.Primeiro)
        Então L.Atual = Null;
        L.Primeiro = Null; }

    Senão { Se (P == L.Primeiro)
        Então L.Primeiro = L.Primeiro→ Next; // Caso 2: X é o primeiro
        Anterior→ Next = P→ Next; // aplicado nos casos 1, 1' e 2
        L.Atual = Anterior;
        DeleteNode( P ); } // aplicado nos casos 1, 1' e 2

    // comandos aplicados nos casos 1, 1', 2 e 3
    Ok = Verdadeiro;
    P = Null;
    Anterior = Null;
}

Senão Ok = Falso; // X não foi encontrado - casos 4 ou 5; não retira elemento
} // fim Retira

```

Comentário:

- O procedimento ProcuraX precisa ser adaptado em relação ao desenvolvido para uma lista não circular. É preciso tomar cuidado para não deixar o algoritmo entrar em loop infinito.

```

ProcuraX (parâmetros por referência L do tipo Lista, parâmetro por referência X do tipo Char,
parâmetros por referência P, Anterior do tipo NodePtr, parâmetro por referência AchouX do tipo
Boolean) {

/* Executa o passo 1: procura X na Lista L e indica através da variável AchouX se X foi encontrado
(Verdadeiro) ou não (Falso). Se X for encontrado, coloca P apontando para o Nó que armazena X e
Anterior apontando para o Nó anterior ao Nó que armazena X. */

Se (L.Primeiro != Null)
Então  Se (L.Primeiro→ Info == X)    // se achou X logo no primeiro Nó...
        Então  { // posiciona P no primeiro e Anterior no último nó da Lista
                  P = L.Primeiro;
                  Anterior = P→ Next;
                  Enquanto (Anterior→ Next != P) Faça {Anterior = Anterior→ Next; }
                  } // fim então
        Senão { P = L.Primeiro→ Next;    // começa P no próximo de L.Primeiro
                  Anterior = L.Primeiro;    // Anterior começa em L.Primeiro
                  /* a lista é circular. Procurar até achar X ou Y > X ou até "dar uma volta"
                  (ou seja, até que P==L.Primeiro) */
                  Enquanto ( P != L.Primeiro) e (P→ Info < X) Faça {
                      Anterior = P;
                      P = P→ Next; }
                  Se (P→ Info != X)
                  Então AchouX = Falso;
                  Senão AchouX = Verdadeiro;
                  } // senão
        Senão AchouX = Falso;
} // fim ProcuraX

```

## Exercício 6.15 Operação Destrói de uma Lista Cadastral

Comentário:

- A forma mais apropriada para proporcionar portabilidade e reusabilidade é implementar o Destrói através das operações primitivas.

```

Destrói (parâmetro por referência L do tipo Lista) {
    /* desaloca todos os nós da Lista */
    Variável X do tipo Char;
    Variáveis TemElemento, Ok do tipo Boolean;

    PegaOPrimeiro( L, X, TemElemento ); // pega o primeiro elemento da Lista, se existir
    Enquanto (TemElemento == Verdadeiro) Faça
        { Retira( L, X, Ok );           // Retira da Lista o elemento de valor X
          PegaOPróximo( L, X, TemElemento );} // pega o próximo, se existir
    } // fim Destrói

```

## **Exercício 6.18 Lista Cadastral com elementos repetidos implementada como uma Lista Encadeada Ordenada Circular**

Basta adaptar a operação que insere elementos, permitindo que sejam inseridos elementos repetidos.

## **Exercício 6.17 Operação RetiraTodosDeValorX de uma Lista Cadastral com elementos repetidos**

Comentário:

- A forma mais apropriada para proporcionar portabilidade e reusabilidade é implementar essa operação através de chamadas sucessivas à operação que retira um único elemento.

```
RetiraTodosDeValorX (parâmetro por referência L do tipo Lista, parâmetro X do tipo Char,  
parâmetro por referência RetirouPeloMenos1 do tipo Boolean) {  
    /* Retira todos os elementos de valor X que forem encontrados na Lista L. Se algum elemento de  
    valor X for encontrado e removido, Ok retorna Verdadeiro. Se nenhum elemento de valor X for  
    encontrado na Lista L, não retira nenhum elemento e Ok retorna o valor Falso */  
  
    Variável Ok do tipo Boolean;  
    RetirouPeloMenos1 = Falso;  
    Repita  
        Retira(L, X, Ok);  
        Se (Ok == Verdadeiro)  
            Então RetirouPeloMenos1 = Verdadeiro;  
    Até que (Ok == Falso);  
    Ok = RetirouPeloMenos1;  
} // fim RetiraTodosDeValorX
```

## **Exercício 6.18 Lista Cadastral sem elementos repetidos, implementada como uma Lista Encadeada Circular Não Ordenada**



As operações de busca precisam ser ajustadas em relação à implementação de uma lista ordenada. Em vez de “Enquanto  $P \rightarrow \text{Info} < X$ ”, utilize “Enquanto  $P \rightarrow \text{Info} != X$ ”.

## **Exercício 6.21 Avançar o Projeto Spider Shopping — calcular o número de itens da Lista de Compras que foram efetivamente comprados pelo jogador**

Comentário:

- A solução desse exercício é muito parecida com a solução do [Exercício 6.2](#), que calcula a interseção entre duas Listas.

```
Inteiro ItensDaListaQueForamComprados (parâmetros por referência ListaDeCompras,
CarrinhoDeCompras do tipo Lista);
/* Recebe a Lista de Compras e o Carrinho de Compras e conta quantos itens da Lista de Compras
aparecem pelo menos uma vez no Carrinho de Compras. Produz resultado do tipo inteiro */
Variável X do tipo Char;
Variável TemElemento do tipo Boolean;
Variável Contador do tipo Inteiro;
Contador = 0;
/* para cada elemento da Lista de Compras, verifica se está também no Carrinho de Compras */
PegaOPrimeiro( ListaDeCompras, X, TemElemento ); // pega o primeiro da Lista (X)
Enquanto (TemElemento == Verdadeiro) Faça {
    Se (EstáNaLista(CarrinhoDeCompras, X)) // se X estiver também no Carrinho..
    Então Contador = Contador + 1; // ...conta mais 1 elemento
    PegaOPróximo( ListaDeCompras, X, TemElemento ); } // pega próximo da Lista..
    //.. de Compras, até acabarem os elementos.
Retorne Contador;
} // fim ItensDaListaQueForamComprados
```

## **Exercício 6.22 Avançar o Projeto Spider Shopping – Calcular o Número de Itens Comprados Incorretamente**

Solução análoga à do [Exercício 6.21](#). Basta contar quantos itens do Carrinho de Compras não fazem parte da Lista de Compras.

## **Exercício 6.23 Avançar o Projeto Spider Shopping — calcular o número de itens comprados em excesso**

Sugestão para solução — passos:

- Passo 1: crie uma nova lista chamada ItensCompradosCorretamente, contendo todos os elementos que aparecem tanto na Lista de Compras quanto no Carrinho de Compras (interseção entre as duas listas).
- Passo 2: desenvolva uma operação para verificar se um elemento X aparece

mais de uma vez em uma Lista L.

- Passo 3: percorra a Lista ItensCompradosCorretamente e conte quantos de seus elementos aparecem mais de uma vez no Carrinho de Compras.

- Passo 4: destrua a Lista ItensCompradosCorretamente.

Refinamento do passo 2:

Para verificar se um elemento X aparece mais de uma vez em uma Lista L:

- 2.1: retire X da Lista L.

- 2.2: verifique se (após retirar uma ocorrência de X) X ainda aparece na Lista L; caso aparecer, o resultado é Verdadeiro (X aparece mais de uma vez em L).

- 2.3: insira X na Lista L (devolvendo a ocorrência de X retirada no passo 2.1).

---

## CAPÍTULO 7

---

# Generalização de Listas Encadeadas

---

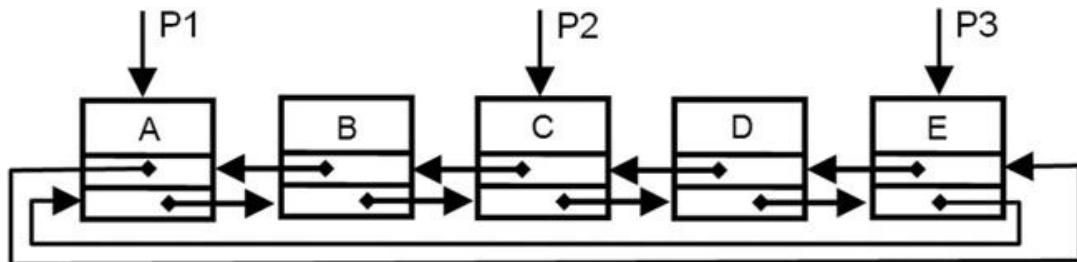
## Seus objetivos neste capítulo

- Estudar técnicas complementares para a implementação de Listas Encadeadas: Encadeamento Duplo, Nó Header e Primitivas de Baixo Nível.
- Ganhar experiência na elaboração de algoritmos sobre Listas Encadeadas, implementando Pilhas, Filas, Listas Cadastrais e Filas de Prioridades, utilizando combinações das técnicas complementares estudadas.
- Conhecer conceitos relativos à generalização de Listas Encadeadas: Listas Multilineares, Listas de Listas e Listas Genéricas quanto ao tipo de elemento.
- Entender que é possível conceber sua própria estrutura encadeada, para atender a necessidades específicas de determinada aplicação.

## 7.1 Listas Duplamente Encadeadas

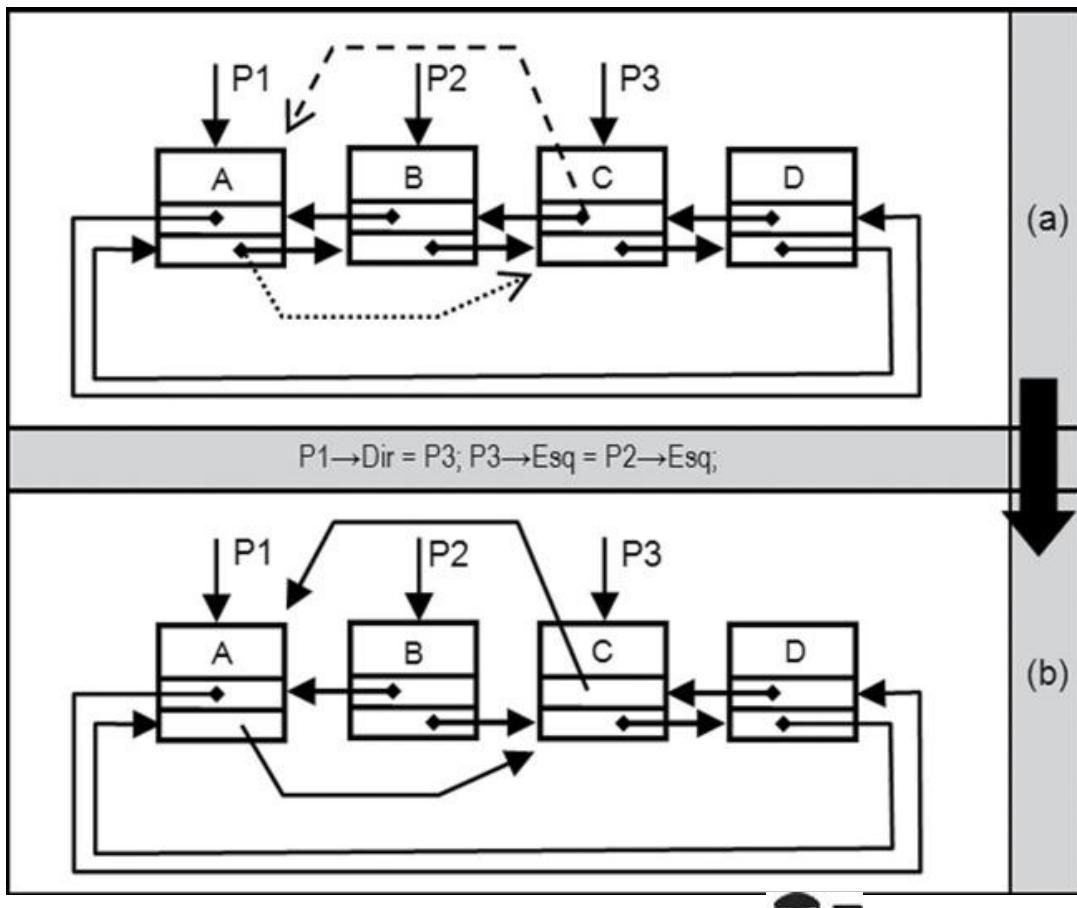
Em cada Nó das estruturas encadeadas que implementamos até o momento, temos um campo para armazenar a informação — que chamamos de Info — e um campo para indicar o próximo elemento — que chamamos de Next. Nessas implementações, podemos percorrer a lista em um único sentido: partindo do primeiro elemento avançamos para o próximo, depois para o próximo do próximo, e assim por diante.

Em uma Lista Duplamente Encadeada, além do campo para armazenar informação, cada Nó possui dois outros campos: um desses campos indica o próximo elemento da lista; o outro indica o elemento anterior. Para evitar uma interpretação incerta, vamos chamar esses campos de Dir e Esq, para indicar, respectivamente, o elemento à direita e o elemento à esquerda de um Nó. Na [Figura 7.1](#), o campo Dir do nó apontado por P2 está apontando para o Nó que contém o valor ‘D’; o campo Esq do Nó apontado por P2 está apontando para o Nó que armazena o valor ‘B’.



**FIGURA 7.1** Lista Circular Duplamente Encadeada.

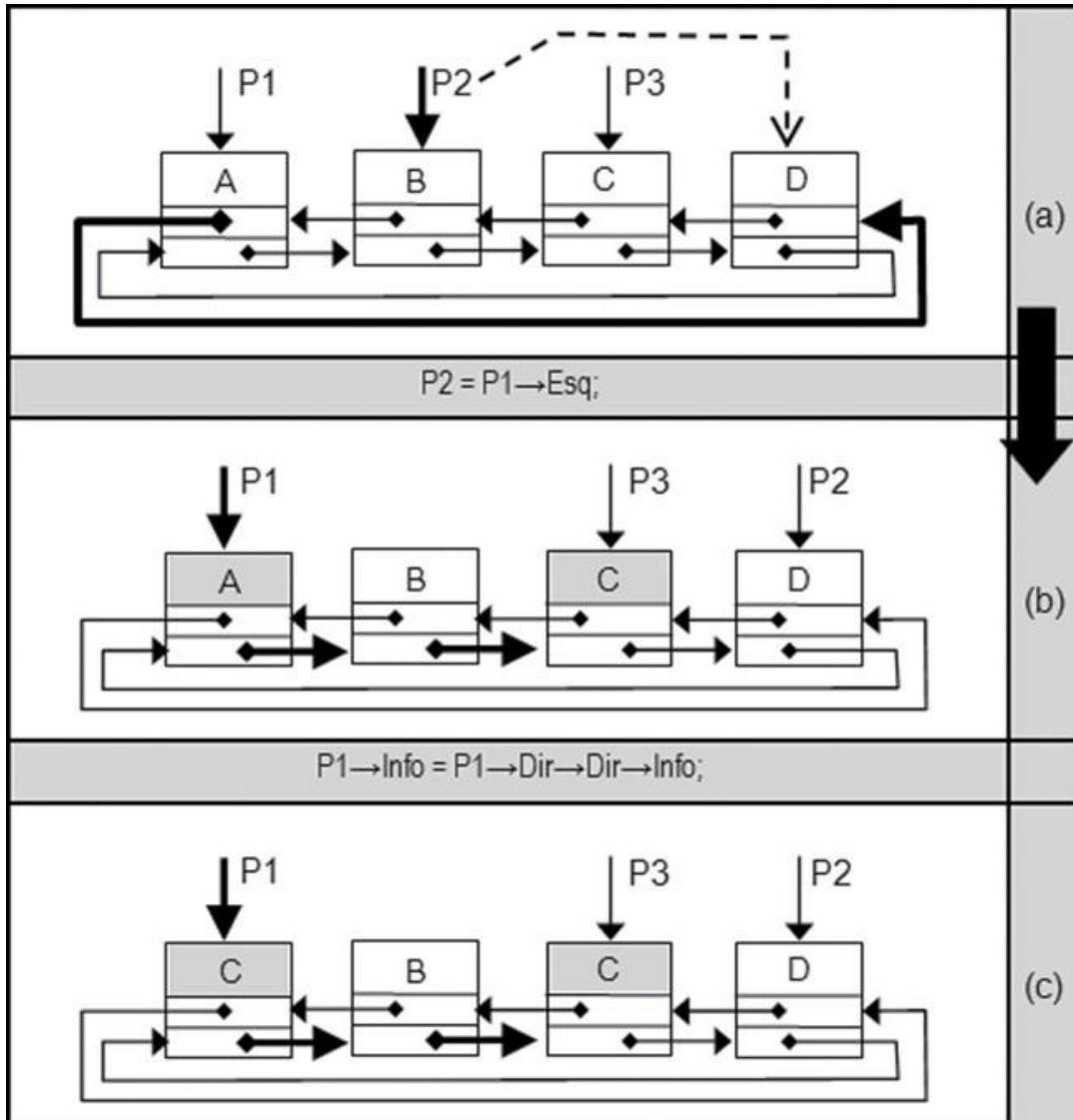
Partindo da situação ilustrada pela [Figura 7.2a](#), se aplicarmos o comando  $P1 \rightarrow Dir = P3$ , o campo Dir do Nó apontado por P1 passará a apontar para o Nó apontado por P3, conforme sugere a seta pontilhada. Se, em seguida, executarmos o comando  $P3 \rightarrow Esq = P2 \rightarrow Esq$ , o campo Esq do Nó apontado por P3 passará a apontar para onde aponta o campo Esq do Nó apontado por P2 — o Nó que contém o valor ‘A’ (seta tracejada).



**FIGURA 7.2** Campos Dir e Esq: direita e esquerda.



Como estamos manipulando uma Lista Circular, note na [Figura 7.3a](#) que o campo Esq do Nó apontado por P1 aponta para o Nó que contém o valor ‘D’. Se, na situação da [Figura 7.3a](#), aplicarmos o comando  $P2 = P1 \rightarrow \text{Esq}$ , P2 passará a apontar para o Nó que contém o valor ‘D’ ([Figura 7.3b](#)).



**FIGURA 7.3** Exemplificando a manipulação de uma Lista

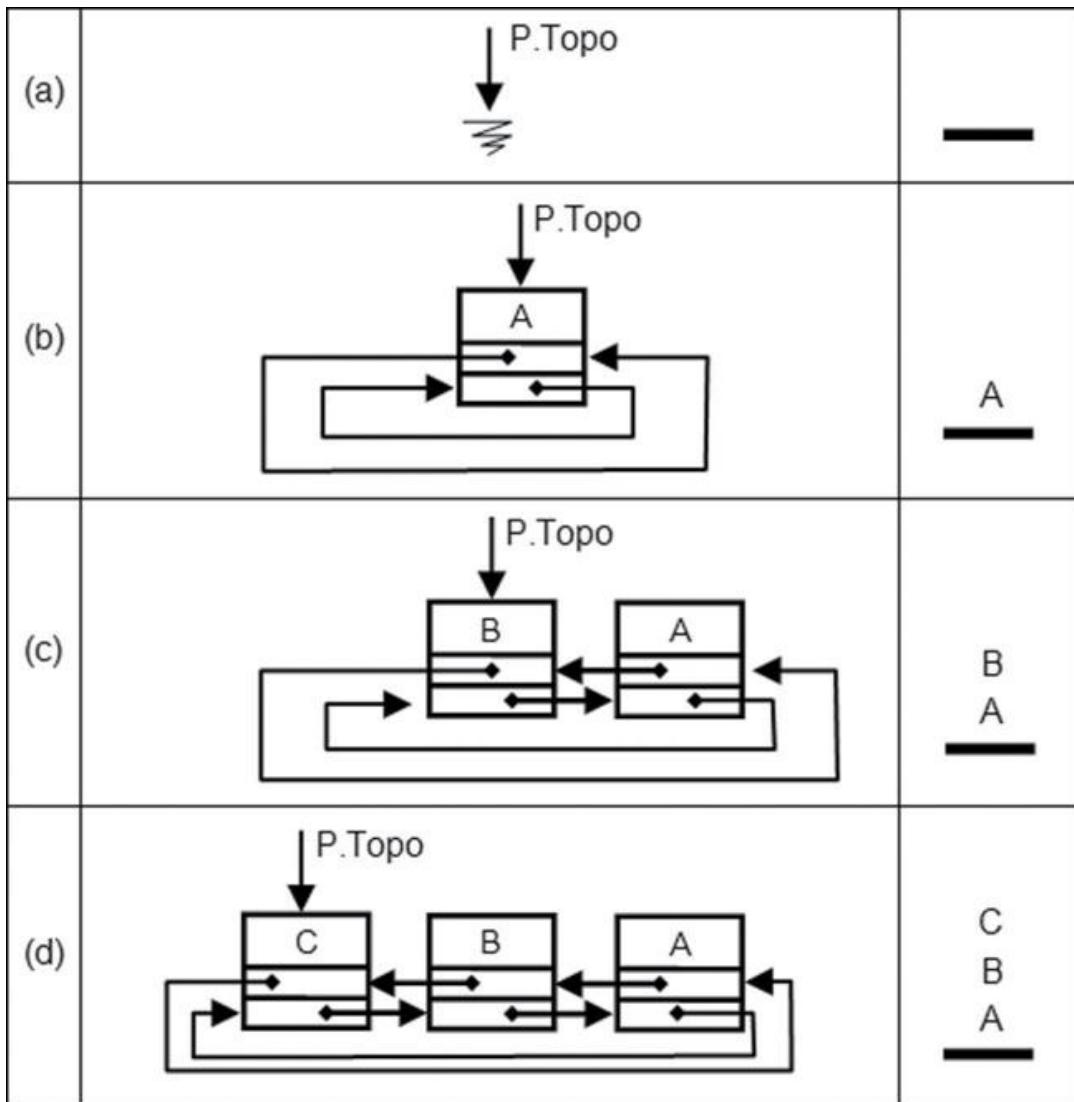


O comando aninhado  $P1 \rightarrow \text{Info} = P1 \rightarrow \text{Dir} \rightarrow \text{Dir} \rightarrow \text{Info}$  pode ser lido: “Info de P1 recebe Info do Dir do Dir de P1”. Ou seja, o campo Info do nó apontado por P1 recebe o valor do campo Info do nó apontado por  $P1 \rightarrow \text{Dir} \rightarrow \text{Dir}$ .  $P1 \rightarrow \text{Dir}$

aponta para o nó que armazena o valor B.  $P1 \rightarrow Dir \rightarrow Dir$  aponta para o nó que armazena o valor C. Assim, se aplicarmos o comando aninhado  $P1 \rightarrow Info = P1 \rightarrow Dir \rightarrow Dir \rightarrow Info$  à situação da [Figura 7.3b](#), o campo Info do nó apontado por  $P1$  receberá o valor C, como mostra a [Figura 7.3c](#).

## Implementando uma Pilha com Encadeamento Duplo

A [Figura 7.4](#) mostra uma Pilha P implementada através de uma Lista Circular Duplamente Encadeada. A [Figura 7.4a](#) mostra a situação de Pilha vazia. Se nessa Pilha vazia empilharmos o elemento ‘A’, teremos a situação da [Figura 7.4b](#). Se empilharmos mais um elemento — o elemento de valor ‘B’, chegaremos à situação da [Figura 7.4c](#), na qual a Pilha possui dois elementos, sendo que o elemento ‘B’ está no Topo. Se, a partir da situação da [Figura 7.4c](#), empilharmos mais um elemento, agora de valor ‘C’, chegaremos a uma Pilha com três elementos, com o elemento ‘C’ no Topo, conforme mostra a [Figura 7.4d](#).



**FIGURA 7.4** Pilha implementada como uma Lista Circular

Duplamente Encadeada.

Queremos agora implementar as Operações Primitivas de uma Pilha — Empilha, Desempilha, Cria, Vazia e Cheia —, detalhadas na [Figura 2.6](#). Implementaremos a Pilha como uma Lista Circular Duplamente Encadeada, conforme os diagramas da [Figura 7.4](#), mas as operações precisam produzir exatamente o mesmo efeito que produzem aquelas que implementamos no [Capítulo 2](#), com Alocação Sequencial, e no [Capítulo 4](#), com uma Lista Encadeada que não era circular e não possuía encadeamento duplo.

## Exercício 7.1 Empilha — Lista Circular Duplamente

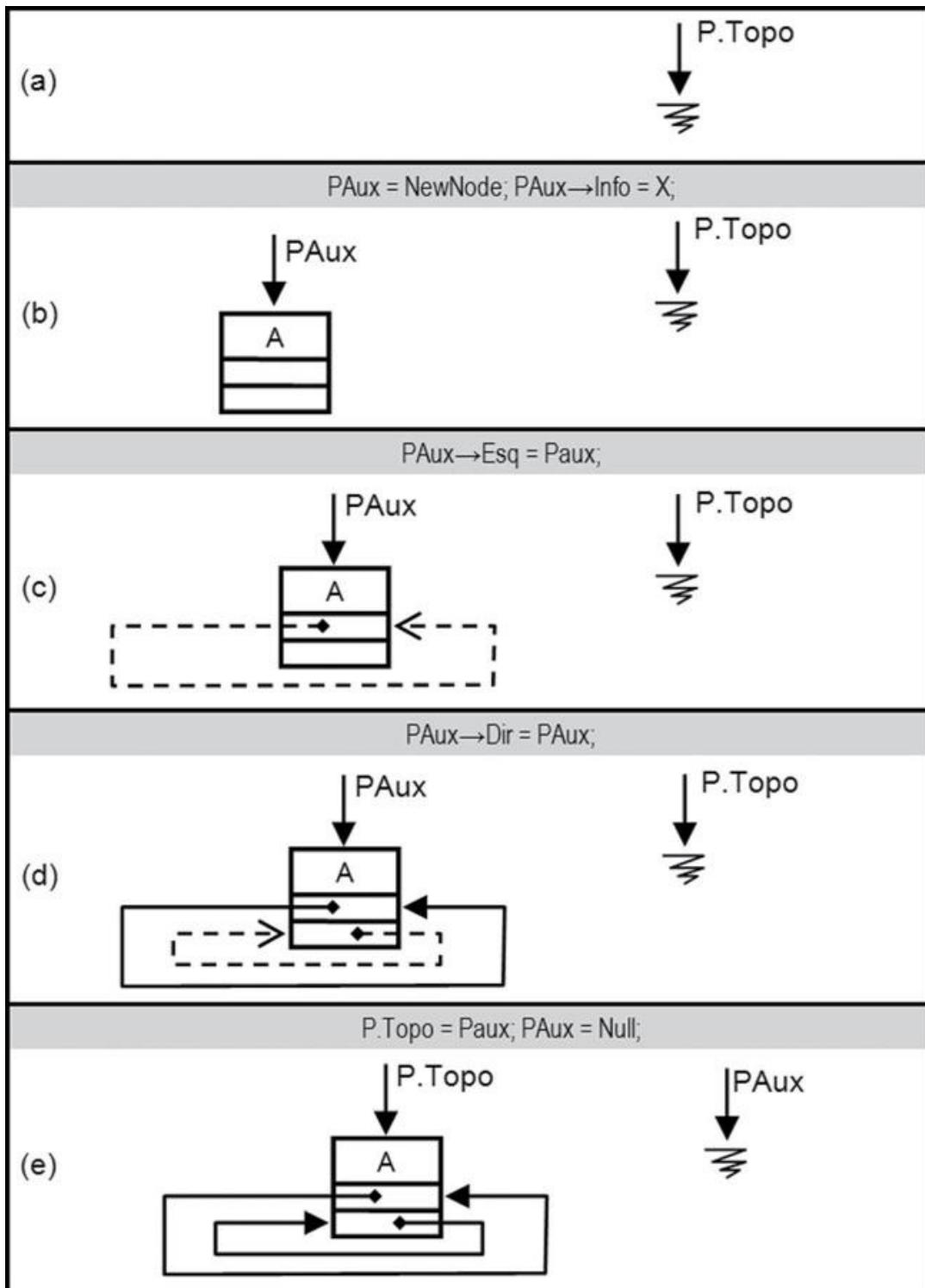
## Encadeada

A operação Empilha recebe como parâmetros a Pilha na qual queremos empilhar um elemento e o valor do elemento que queremos empilhar.

```
Empilha (parâmetro por referência P do tipo Pilha, parâmetro X do tipo Char, parâmetro por referência DeuCerto do tipo Boolean);
```

```
/* Empilha o elemento X na Pilha P. O parâmetro DeuCerto deve indicar se a operação foi bem-sucedida ou não */
```

A princípio, vamos tratar diferenciadamente dois casos: Caso 1 — Pilha vazia (ilustrada na [Figura 7.4a](#)) — e Caso 2 — Pilha não vazia ([Figuras 7.4b, 7.4c e 7.4d](#)). A [Figura 7.5](#) ilustra passo a passo o funcionamento da operação Empilha para a situação inicial de uma Pilha vazia.



**FIGURA 7.5** Empilha elemento em uma Pilha vazia.



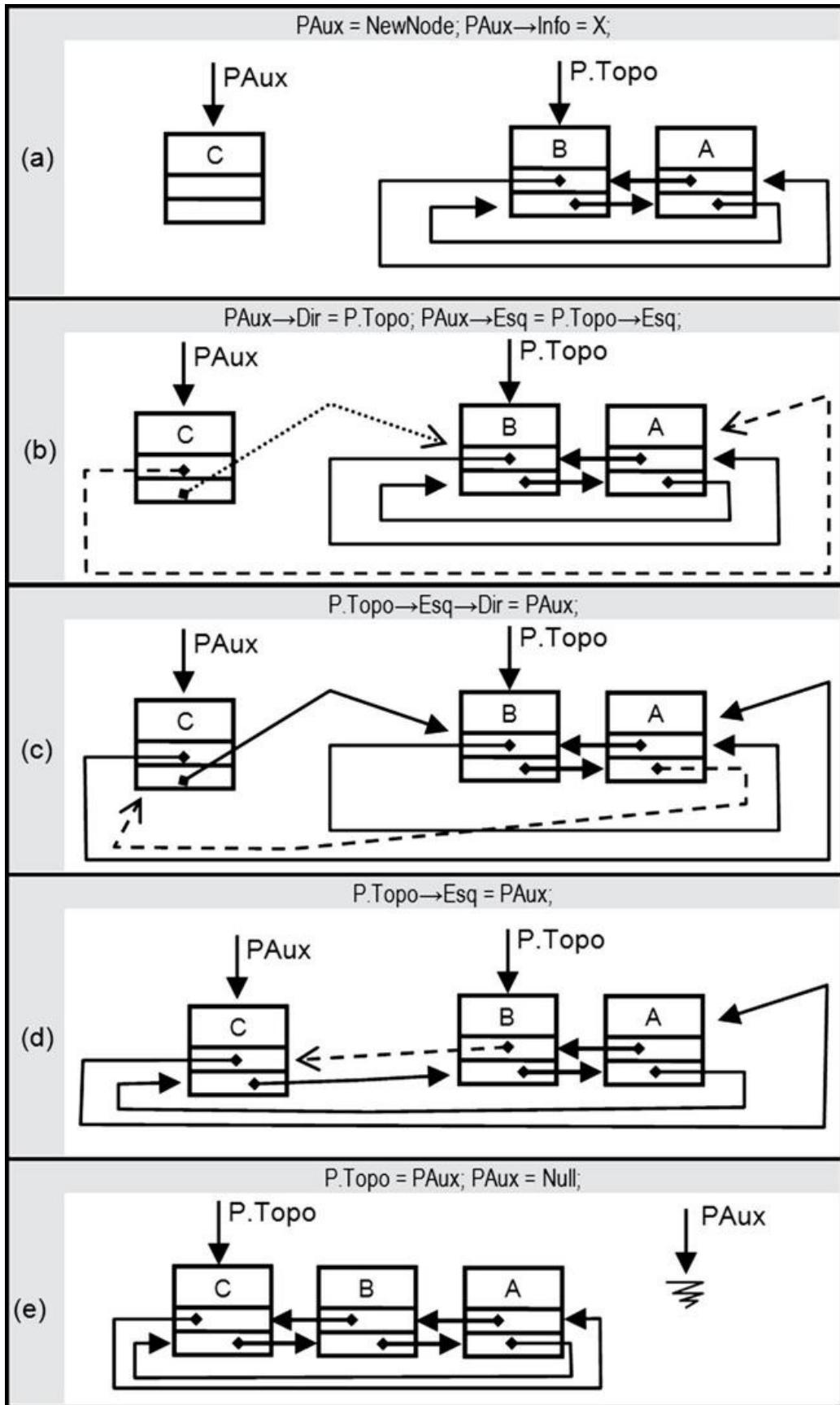
Com uma variável auxiliar chamada  $PAux$ , alocamos um Nó através do

comando conceitual PAux = NewNode. Em seguida armazenamos no N o apontado por PAux a informa o que queremos empilhar. Fazemos isso atrav s do comando PAux → Info = X. Com esses dois comandos, a situ o da [Figura 7.5a](#) passa para a situ o ilustrada na [Figura 7.5b](#).

A partir da situ o da [Figura 7.5b](#), executamos o comando PAux → Esq = Paux. Atrav s desse comando, o campo Esq do N o apontado por PAux passa a apontar para o pr prio PAux. A seta pontilhada na [Figura 7.5c](#) destaca o efeito da execu o de PAux → Esq = Paux. Em seguida executamos o comando PAux → Dir = PAux. Atrav s desse comando, o campo Dir do N o apontado por PAux passa a apontar para o pr prio PAux, conforme ilustra a seta pontilhada da [Figura 7.5d](#).

Atrav s do comando P.Topo = PAUx, o ponteiro P.Topo passa a apontar para onde aponta PAUx, ou seja, para o N o que acabou de ser alocado. Movemos PAux para Null, haja visto que PAux    uma vari vel tempor ria. Ao final da execu o dessa sequ ncia de comandos, a Pilha que estava vazia passou a ter um elemento ([Figura 7.5e](#)).

A [Figura 7.6](#) ilustra passo a passo o funcionamento da oper o Empilha para a situ o inicial de uma Pilha com dois elementos. A situ o da [Figura 7.6a](#) mostra a Pilha com dois elementos, sendo que o elemento B est  no Topo. A [Figura 7.6a](#) tamb m mostra o resultado da execu o dos comandos PAux = NewNode e PAux → Info = X. Um novo N o est  alocado, e o campo Info desse novo N o armazena a informa o que queremos empilhar.





**FIGURA 7.6** Empilha Elemento em uma Pilha não vazia.

A partir da situação da [Figura 7.6a](#), executamos o comando  $\text{PAux} \rightarrow \text{Dir} = \text{P.Topo}$ . Através desse comando, o campo Dir do Nó apontado por PAux passa a apontar para onde aponta P.Topo, ou seja, para o Nó que contém o valor ‘B’ (seta pontilhada da [Figura 7.6b](#)). Analogamente, com o comando  $\text{PAux} \rightarrow \text{Esq} = \text{P.Topo} \rightarrow \text{Esq}$ , o campo Esq do Nó apontado por PAux passa a apontar para onde aponta o campo Esq do Nó apontado por P.Topo, ou seja, passa a apontar para o Nó que contém o valor ‘A’ (seta tracejada da [Figura 7.6b](#)).

Em seguida, executamos o comando  $\text{P.Topo} \rightarrow \text{Esq} \rightarrow \text{Dir} = \text{PAux}$ , que produz o efeito indicado pela seta tracejada da [Figura 7.6c](#). Note que  $\text{P.Topo} \rightarrow \text{Esq}$  aponta para o Nó que contém o valor ‘A’. Portanto, com o comando aninhado  $\text{P.Topo} \rightarrow \text{Esq} \rightarrow \text{Dir} = \text{PAux}$  estamos apontando o campo Dir do Nó que contém o valor ‘A’ para PAux.

Caso você tenha alguma dificuldade para visualizar a execução do comando aninhado  $\text{P.Topo} \rightarrow \text{Esq} \rightarrow \text{Dir} = \text{PAux}$ , é possível dividir a execução desse comando em dois passos. Primeiramente, coloque um novo ponteiro auxiliar apontando para o Nó que contém o valor ‘A’ através do comando  $\text{NovoPonteiroAuxiliar} = \text{P.Topo} \rightarrow \text{Esq}$ . Em seguida você poderá executar  $\text{NovoPonteiroAuxiliar} \rightarrow \text{Dir} = \text{PAux}$ . O resultado da execução desses dois comandos será o mesmo da execução do comando aninhado  $\text{P.Topo} \rightarrow \text{Esq} \rightarrow \text{Dir} = \text{PAux}$ , destacado pela seta tracejada da [Figura 7.6c](#).

Em seguida, a partir da situação da [Figura 7.6c](#), executamos o comando  $\text{P.Topo} \rightarrow \text{Esq} = \text{PAux}$ . Através desse comando, o campo Esq do Nó apontado por P.Topo passa a apontar para PAux. A seta tracejada da [Figura 7.6d](#) destaca o efeito da execução de  $\text{P.Topo} \rightarrow \text{Esq} = \text{PAux}$ .

Através do comando  $\text{P.Topo} = \text{PAux}$ , o ponteiro P.Topo passa a apontar para onde aponta PAux, ou seja, para o Nó que acabou de ser alocado. Movemos PAux para Null, haja visto que PAux é uma variável temporária. Ao final da execução dessa sequência de comandos, a Pilha que continha dois elementos passou a ter três elementos. O elemento que acabou de ser inserido — o elemento ‘C’ — está no Topo da Pilha, conforme mostra a [Figura 7.6e](#).

A [Figura 7.7](#) apresenta um algoritmo conceitual para a operação Empilha, para uma Pilha implementada como uma Lista Circular Duplamente Encadeada. Se a Pilha estiver vazia, executamos a sequência de comandos cuja execução foi

ilustrada na [Figura 7.5](#). Caso a Pilha não esteja vazia, é executada a sequência de comandos ilustrada na [Figura 7.6](#).

```
Empilha (parâmetro por referência P do tipo Pilha, parâmetro X do tipo Char, parâmetro por referência DeuCerto do tipo Boolean) {  
  
    /* Empilha o elemento X, passado como parâmetro, na Pilha P também passada como parâmetro. O parâmetro DeuCerto deve indicar se a operação foi bem-sucedida ou não */  
  
    Variável PAux do tipo NodePtr;  
  
    Se (Cheia(P)==Verdadeiro) // se a Pilha P estiver cheia... não podemos empilhar  
    Então     DeuCerto = Falso;  
    Senão { DeuCerto = Verdadeiro;  
            Se (Vazia(P)==Verdadeiro)  
                Então { /* trata o Caso 1 — Insere em uma Pilha Vazia */  
                        PAux = NewNode;  
                        PAux→Info = X;  
                        PAux→Dir = PAux;  
                        PAux→Esq = PAux;  
                        P.Topo = PAux;  
                        PAux = Null; }  
  
            Senão { /* trata o Caso 2 — Insere em uma Pilha Não Vazia */  
                    PAux = NewNode;  
                    PAux→Info = X;  
                    PAux→Dir = P.Topo;  
                    PAux→Esq = P.Topo→Esq;  
                    P.Topo→Esq→Dir = PAux;  
                    P.Topo→Esq = PAux;  
                    P.Topo = PAux;  
                    PAux = Null; }  
        } // senão  
    } fim Empilha
```

**FIGURA 7.7** Algoritmo conceitual — Empilha.

## Exercício 7.2 Executar Empilha em novas situações

As [Figuras 7.5](#) e [7.6](#) mostraram a execução passo a passo da operação Empilha tendo como situação inicial a Pilha P vazia e a Pilha P com dois elementos, respectivamente. Execute passo a passo o algoritmo Empilha da [Figura 7.7](#) em

pelo menos duas novas situações iniciais. Por exemplo, na situação inicial de uma Pilha com um único elemento e na situação inicial de uma Pilha com quatro elementos. Desenhe passo a passo a execução do algoritmo.

### **Exercício 7.3 TAD Pilha — Operações Desempilha, Cria e Vazia**

Implemente as demais Operações Primitivas de uma Pilha especificadas na Figura 2.6: Desempilha, Cria, Vazia e Cheia. A Pilha deve ser implementada como uma Lista Circular Duplamente Encadeada, conforme os diagramas das Figuras 7.4 a 7.6.

### **Exercício 7.4 TAD Fila implementada como Lista Circular Duplamente Encadeada**

Assim como fizemos para implementar uma Pilha, implemente agora uma Fila através de uma Lista Circular Duplamente Encadeada. Primeiramente faça diagramas para uma Fila vazia, para uma Fila com um único elemento e para uma Fila com vários elementos. A seguir, implemente as Operações Primitivas de uma Fila, conforme especificado na Figura 3.4. Ao desenvolver os algoritmos, siga os passos sugeridos na Figura 6.25: identifique e desenhe cada caso, trate cada caso separadamente etc.

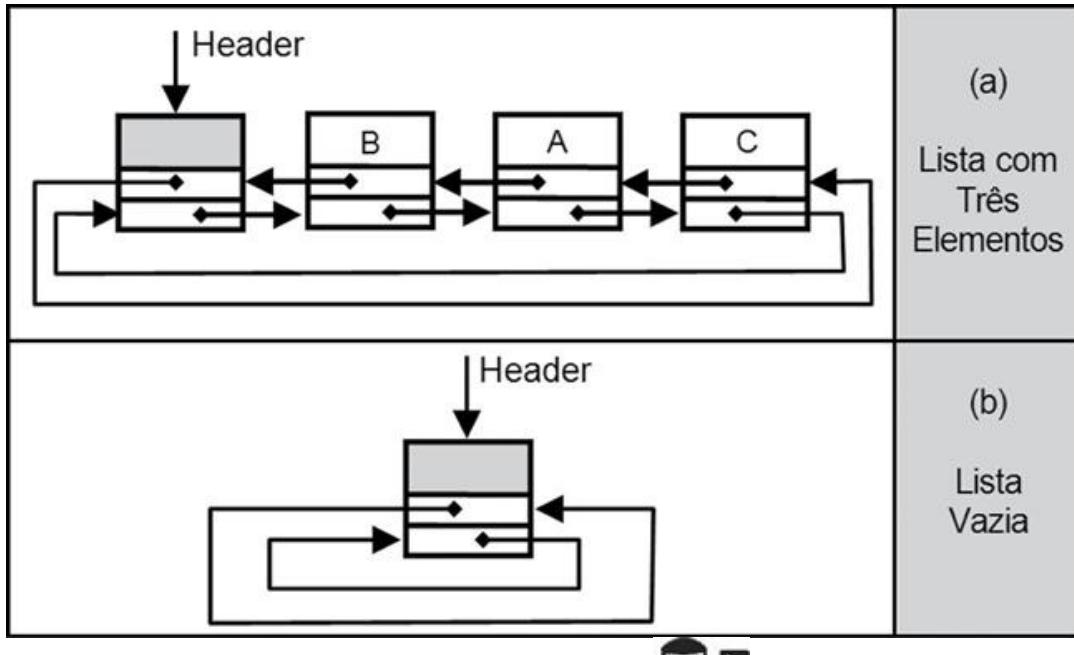
### **Exercício 7.5 TAD Lista Cadastral implementada como Lista Circular Duplamente Encadeada Não Ordenada**

Implemente uma Lista Cadastral através de uma Lista Circular Duplamente Encadeada Não Ordenada. Faça diagramas para uma Lista vazia, para uma Lista com um único elemento e para uma Lista com vários elementos. A seguir implemente as Operações Primitivas de uma Lista Cadastral, conforme especificado na Figura 6.4. Ao desenvolver os algoritmos, siga os passos sugeridos na Figura 6.25: identifique e desenhe cada caso, trate cada caso separadamente etc.

## **7.2 Listas com Nô Header**

Um “Nô Header” ou “Nô Cabeça” é um Nô que não é utilizado para armazenar elementos, mas apenas como estrutura de suporte, para marcar o início de uma Lista Encadeada. A Figura 7.8 mostra a representação de uma Lista Circular

Duplamente Encadeada com quatro Nós. Três desses Nós estão armazenando os elementos B, A e C. O quarto Nó é o Nó Header, que não armazena nenhum elemento, e apenas marca o início da Lista.



**FIGURA 7.8** Lista circular com Nό Header.

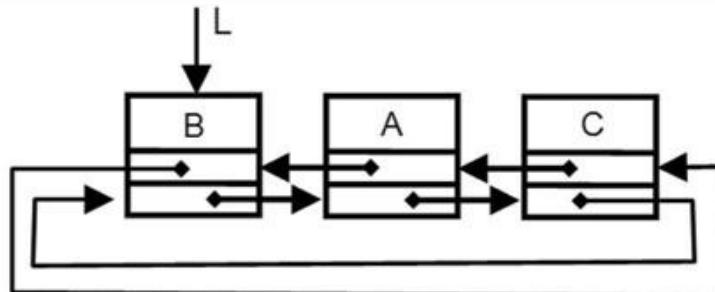


Note, na [Figura 7.8b](#), que em uma implementação com Header, a Lista vazia não é mais composta por um ponteiro apontando para Null. A Lista vazia é composta por um ponteiro apontando para o Nό Header.

Quando utilizamos o Null para identificar a situação de Lista vazia, os algoritmos precisam tratar separadamente os casos de inserir um elemento em uma Lista vazia e de retirar o único elemento de uma Lista, deixando-a vazia. Quando utilizamos um Nό Header, as operações para inserir ou retirar o único elemento de uma Lista não precisam ser tratadas como exceções, e os algoritmos tendem a ser mais simples.

O Nό Header também pode ser utilizado para auxiliar na busca de um elemento na Lista. Por exemplo, para encontrar um elemento X em uma Lista Circular Não Ordenada, Duplamente Encadeada, com três elementos, conforme ilustrado na [Figura 7.9a](#), precisamos de um comando de repetição com duas condições: a primeira condição ( $P \rightarrow \text{Info} ! = X$ ) interrompe a repetição quando achamos o elemento X; a segunda condição ( $P! = L$ ) interrompe a busca quando

já tivermos percorrido toda a Lista, ou seja, quando já tivermos “dado uma volta” na Lista Circular. Esta segunda condição é necessária para evitar que o algoritmo entre em situação de *loop infinito*, no caso de X não estar na Lista.



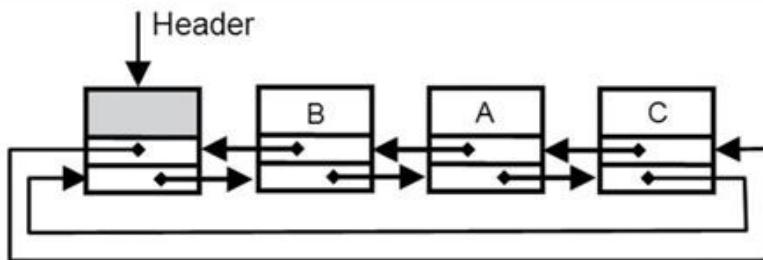
```

Se (L == Null)      // Lista Vazia
Então AchouX = Falso;
Senão { P = L→Dir // P é um ponteiro auxiliar que percorre a Lista

    /* note as duas condições de parada no comando de repetição: enquanto (não
       achou X) E (não deu a volta na Lista), continua avançando P */
    Enquanto (( P→Info != X ) E ( P != L ))
        P = P→Dir;
        Se (P→Dir == X)
            Então AchouX = Verdadeiro;
            Senão AchouX = Falso;
        } // senão
    }
}

```

#### (a) Busca em uma Lista sem Header



```

P = Header→Dir;      // P é um ponteiro auxiliar que percorre a Lista
Header→Info = X;     // atribuição de X ao Nô Header, apenas para auxiliar na busca

```

```

/* podemos suprimir a segunda condição de parada do comando de interrupção, (P != L),
   pois se X não estiver na lista a repetição será interrompida quando acharmos o X
   atribuído ao Header no início da operação */
Enquanto (P→Info != X)
    P = P→Dir;

```

```

/* se o X que achamos está no Header, na verdade X não está armazenado na lista. Se
   encontramos um X em um Nô que não seja o Header, aí sim podemos concluir que X
   está na Lista */
Se (P != Header)
    Então AchouX = Verdadeiro;    // achamos um X armazenado na Lista
    Senão AchouX = Falso;

```

#### (b) Busca em uma Lista com Header

**FIGURA 7.9 Comparando a busca com o Nó Header à Busca sem o  
Nó Header.**

Quando temos uma implementação com Header, antes de iniciar a busca podemos atribuir o valor de X ao campo Info do Nó Header. Então podemos suprimir a segunda condição de parada, ( $P \neq L$ ), pois se X não for encontrado em um dos Nós da Lista que armazenam elementos, será encontrado no Nó Header, interrompendo o comando de repetição.

No exemplo da [Figura 7.9b](#), a Lista contém os elementos ‘A’, ‘B’ e ‘C’. Se procurarmos um elemento de valor ‘D’, que não está na Lista, antes de iniciar o comando de repetição o algoritmo atribuirá o valor ‘D’ ao campo Info do Nó Header. Logo, a repetição será interrompida quando esse valor ‘D’ for localizado no Header.

Se estivéssemos procurando o valor ‘A’, que está armazenado na Lista, antes de iniciar o comando de repetição o algoritmo iria atribuir o valor ‘A’ ao campo Info do Nó Header. Mas, como vamos verificar todos os demais Nós da Lista antes de verificar o Nº Header, o comando de repetição será interrompido quando encontrarmos o valor ‘A’ que não está no Header. Um comando condicional no final do algoritmo (se  $P \neq \text{Header}$ ) diferencia a situação em que o comando de repetição foi interrompido ao chegar no Header, da situação em que a busca é interrompida ao encontrarmos uma informação que está realmente armazenada na Lista.

**Exercício 7.6 TAD Lista Cadastral implementada como  
Lista Circular Duplamente Encadeada Não Ordenada sem  
elementos repetidos e com Nº Header**

Implemente uma Lista Cadastral através de uma Lista Circular Duplamente Encadeada Não Ordenada sem Elementos repetidos e com Nº Header. Primeiramente desenhe uma Lista vazia, uma Lista com um único elemento e uma Lista com vários elementos. A seguir implemente as Operações Primitivas de uma Lista Cadastral, conforme especificado na [Figura 6.4](#) (Cria, Vazia, Cheia, Insere, Retira, EstáNaLista, PegaOPrimeiro e PegaOPróximo). Ao implementar a operação Cria, não esqueça que a Lista vazia deve conter o Nº Header. Ao implementar a operação EstáNaLista e a operação Retira, utilize a otimização resultante de atribuir o valor de X ao campo Info do Nº Header, antes de iniciar a busca, conforme exemplificado na [Figura 7.9b](#). Ao desenvolver os algoritmos,

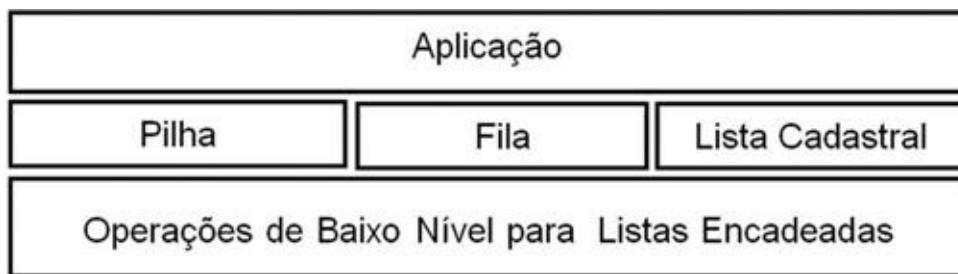
siga os passos sugeridos na [Figura 6.25](#): identifique e desenhe cada caso, trate cada caso separadamente etc.

### Exercício 7.7 TAD Fila com Nô Header

Implemente uma Fila através de uma Lista Circular Duplamente Encadeada com Nô Header. Primeiramente faça diagramas para uma Fila vazia, para uma Fila com um único elemento e para uma Fila com vários elementos. A seguir implemente as Operações Primitivas de uma Fila, conforme especificado na [Figura 3.4](#). Ao desenvolver os algoritmos, siga os passos sugeridos na [Figura 6.25](#): identifique e desenhe cada caso, trate cada caso separadamente etc.

## 7.3 Operações de Baixo Nível para Listas Encadeadas

Já implementamos Pilhas, Filas e Listas Cadastrais utilizando diversas variações das Listas Encadeadas: Listas Circulares, Listas Duplamente Encadeadas, Listas com Nô Header e combinações dessas técnicas. Queremos agora definir e implementar Operações de Baixo Nível, ou seja, operações mais básicas, para manipulação de Listas Encadeadas, que possibilitem uma implementação mais ágil e mais abstrata de Pilhas, Filas, Listas Cadastrais e outras estruturas de armazenamento, como ilustra a [Figura 7.10](#).



**FIGURA 7.10** Pilha, Fila e Lista Cadastral implementadas através de Operações de Baixo Nível para Listas Encadeadas.

A [Figura 7.11](#) apresenta um conjunto de operações básicas, de baixo nível, para manipulação de Listas Encadeadas. A operação Remove\_P remove da Lista Básica LB um Nô apontado pelo ponteiro P, passado como parâmetro. A operação InsereADireitaDeP insere um valor X na Lista Básica LB, posicionando o novo elemento imediatamente à direita do Nô apontado por P.

Operações	Funcionamento
Remove_P (LB, P, Ok)	Remove da Lista Básica LB o Nó apontado pelo ponteiro P passado como parâmetro. O parâmetro Ok retornará Verdadeiro se a operação deu certo ou Falso, caso contrário.
InsereADireitaDeP (LB, P, X,Ok)	Insere o elemento X na Lista Básica LB, na posição imediatamente à direita do Nó apontado por P. O parâmetro Ok retornará Verdadeiro se a operação deu certo ou Falso, caso contrário.
EstáNaLista (LB, X, P)	Se o elemento X fizer parte da Lista Básica LB, retorna Verdadeiro. Nesse caso, o ponteiro P retornará o endereço do Nó que contém X. Se X não for encontrado na Lista, retorna Falso.
Char Info_de_P (LB, P, Ok)	Retorna o valor do campo Info do Nó apontado por P. O parâmetro Ok retornará Verdadeiro se a operação deu certo, ou Falso, caso contrário.
Vazia (LB)	Funcionamento tradicional — Figura 6.4.
Cheia(LB)	Funcionamento tradicional — Figura 6.4.
Cria (LB)	Funcionamento tradicional — Figura 6.4.
PegaOPrimeiro(LB, X, TemElemento)	Funcionamento tradicional — Figura 6.4.
PegaOPróximo(LB, X, TemElemento)	Funcionamento tradicional — Figura 6.4.

**FIGURA 7.11** Operações de Baixo Nível para Listas Encadeadas.

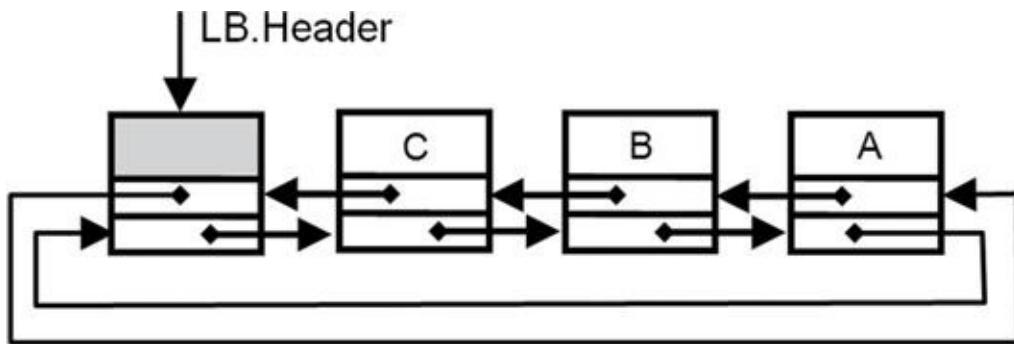
A operação EstáNaLista verifica se X está na Lista Básica LB. Se X estiver em LB, o parâmetro P estará apontando para o Nó que contém X. A operação Info\_de\_P retorna a informação armazenada no Nó apontado por P, dado que P aponte para um Nó válido de LB. As demais operações — Cria, Vazia, Cheia, PegaOPrimeiro e PegaOPróximo — têm os mesmos parâmetros e o mesmo funcionamento das operações de uma Lista Cadastral, conforme especificado na Figura 6.4.

## Exercício 7.8 Pilha implementada a partir das Operações de Baixo Nível para Manipulação de Listas Encadeadas

Implemente uma Pilha utilizando as Operações de Baixo Nível para Manipulação de Listas Encadeadas, especificadas na [Figura 7.11](#). Considere uma Lista Básica implementada como uma Lista Circular, Duplamente Encadeada e

com Nô Header. Primeiramente escolha uma posição da Lista Genérica na qual ficará armazenado o topo da Pilha, e implemente as operações Empilha, Desempilha, Vazia, Cheia e Cria.

Considerando a implementação da Lista Básica como uma Lista Circular, Duplamente Encadeada e com Nô Header, para implementar uma Pilha podemos convencionar, por exemplo, que o topo da Pilha ficará sempre à direita do Nô Header, apontado pelo ponteiro LB.Header. No exemplo da [Figura 7.12](#), a ordem de entrada dos elementos na Pilha foi ‘A’, depois ‘B’ e depois ‘C’. Assim, o elemento ‘C’ está no topo. A Lista Básica LB possui ainda um segundo ponteiro — LB.Atual, utilizado nas operações de percorrer a Lista, que não está representado no Diagrama da [Figura 7.12](#).



**FIGURA 7.12** Pilha implementada a partir de Operações de Baixo



Nível para Manipulação de Listas Encadeadas.

A [Figura 7.13](#) apresenta algoritmos para as operações Empilha e Desempilha. Se quisermos empilhar mais um elemento, de valor ‘D’, na Pilha ilustrada na [Figura 7.12](#), esse novo elemento deverá ser inserido à direita do Nô Header e ficará posicionado entre o Nô de valor ‘C’ e o Nô Header, apontado pelo ponteiro LB.Header. Assim, para implementar a operação Empilha a partir das Operações de Baixo Nível para Manipulação de Listas Encadeadas, descritas na [Figura 7.11](#), utilizamos a operação e os parâmetros: InsereADireitaDeP (LB, LB.Header, X, DeuCerto). Conforme especificado na [Figura 7.11](#), o primeiro parâmetro é a Lista Básica, LB. O segundo parâmetro indica o Nô à direita do qual queremos inserir o novo Nô, com o valor X. Como queremos inserir o novo elemento da Pilha à direita do Nô Header, passamos como segundo parâmetro o ponteiro LB.Header. O terceiro parâmetro, X, é o valor a ser inserido. DeuCerto retornará Verdadeiro ou Falso, indicando sucesso ou insucesso na operação.

```

Empilha (parâmetro por referência LB do tipo ListaBásica, parâmetro X do tipo Char,
parâmetro por referência DeuCerto do tipo Boolean) {
    /* Empilha o elemento X na Pilha implementada na Lista Básica LB */
    InsereADireitaDeP(LB, LB.Header, X, DeuCerto); /* insere X à direita do Nó apontado
    pelo segundo parâmetro, ou seja, à direita de LB.Header. O parâmetro DeuCerto é
    atualizado pela operação InsereADireitaDeP */
} // fim Empilha

Desempilha(parâmetro por referência LB do tipo ListaBásica, parâmetro por referência X
do tipo Char, parâmetro por referência DeuCerto do tipo Boolean) {
    /* Retira o elemento do Topo, retornando seu valor no parâmetro X */
    X = Info_de_P( LB, LB.Header→Dir, DeuCerto );
    /* pega o valor do Nó apontado por LB.Header→Dir */
    Remove_P ( LB, LB.Header→Dir, DeuCerto );
    /* remove o Nó apontado por LB.Header→Dir */
    /* O parâmetro DeuCerto é atualizado pela operação Remove_P */
} // fim Desempilha

```

**FIGURA 7.13** Empilha e Desempilha implementadas a partir de Operações de Baixo Nível de uma Lista Encadeada.

Na operação Desempilha, atribuímos a X a informação do nó que está no topo da Pilha com o comando `X = Info_de_P(LB, LB.Header→Dir, DeuCerto)` e removemos o nó que contém o elemento do topo da Pilha com o comando `Remove_P (LB, LB.Header→Dir, DeuCerto)`. Em ambos os comandos, o segundo parâmetro `LB.Header→Dir` indica o elemento do topo da Pilha.

As demais operações de uma Pilha — Cria, Vazia e Cheia — podem ser implementadas pelo acionamento direto das operações de mesmo nome da Lista Encadeada Genérica.

## Exercício 7.9 Fila a partir de Operações de Baixo Nível

Implemente uma Fila utilizando as Operações de Baixo Nível para Manipulação de Listas Encadeadas, especificadas na [Figura 7.11](#). Considere uma Lista Básica implementada como uma Lista Circular, Duplamente Encadeada e com Nó Header. Primeiramente escolha a posição da Lista Básica na qual ficará armazenado o primeiro elemento da Fila, a posição na qual ficará armazenado o último elemento da Fila e depois implemente as operações Insere e Retira.

Para implementar uma Fila a partir das operações da Lista Básica, basta convencionar, por exemplo, que o primeiro elemento da Fila ficará armazenado à

direita de LB.Header e o último elemento da Fila ficará à esquerda de LB.Header. Como queremos inserir novos elementos sempre no final da Fila, chamamos a operação InsereADireitaDeP tendo como segundo parâmetro o ponteiro LB.Header → Esq — veja o algoritmo na [Figura 7.14](#). Ou seja, estamos inserindo o novo Nó à direita do Nó apontado por LB.Header → Esq. Observe na [Figura 7.12](#) que LB.Header → Esq aponta para o Nó que contém o valor ‘A’. Ao solicitar a inserção à direita desse Nó, estamos colocando o novo elemento no final da Fila.

```
Insere (parâmetro por referência LB do tipo ListaBásica, parâmetro X do tipo Char,  
parâmetro por referência DeuCerto do tipo Boolean) {  
    /* Insere o elemento X no final da Fila implementada na Lista Básica LB */  
    InsereADireitaDeP(LB, LB.Header→Esq, X, DeuCerto); /* insere X à direita do Nó  
apontado pelo segundo parâmetro, ou seja, à direita de LB.Header→Esq. O parâmetro  
DeuCerto é atualizado pela operação InsereADireitaDeP */  
} // fim Insere na Fila  
  
Retira(parâmetro por referência LB do tipo ListaBásica, parâmetro por referência X do tipo  
Char, parâmetro por referência DeuCerto do tipo Boolean) {  
    /* Retira o primeiro elemento da Fila, retornando seu valor no parâmetro X */  
    X = Info_de_P( LB, LB.Header→Dir, DeuCerto );  
    /* pega valor do Nó apontado por LB.Header→Dir */  
    Remove_P ( LB, LB.Header→Dir, DeuCerto );  
    /* remove o Nó apontado por LB.Header→Dir */  
    /* O parâmetro DeuCerto é atualizado pela operação Remove_P */  
} // fim Retira da Fila
```

**FIGURA 7.14** Operações para inserir e retirar elementos de uma Fila implementadas com primitivas de baixo nível

Na operação Retira, chamamos Remove\_P, tendo como segundo parâmetro LB.Header → Dir. Ou seja, estamos solicitando a remoção do Nó apontado por LB.Header → Dir. Veja na [Figura 7.12](#) que LB.Header → Dir aponta para o Nó que contém o valor ‘C’, ou seja, o primeiro elemento da Fila, conforme convencionamos.

Utilizando as Operações de Baixo Nível, ficou bem fácil implementar uma Pilha e uma Fila, não ficou? Também é possível implementar uma Lista Cadastral com bastante agilidade.

## **Exercício 7.10 Lista Cadastral sem repetições a partir de Operações de Baixo Nível para Listas Encadeadas**

Implemente uma Lista Cadastral utilizando as Operações de Baixo Nível para Manipulação de Listas Encadeadas, especificadas na [Figura 7.11](#). Considere uma Lista Básica implementada como uma Lista Circular, Duplamente Encadeada e com Nó Header. Implemente as operações Insere e Retira.

### **Implementando as Operações de Baixo Nível**

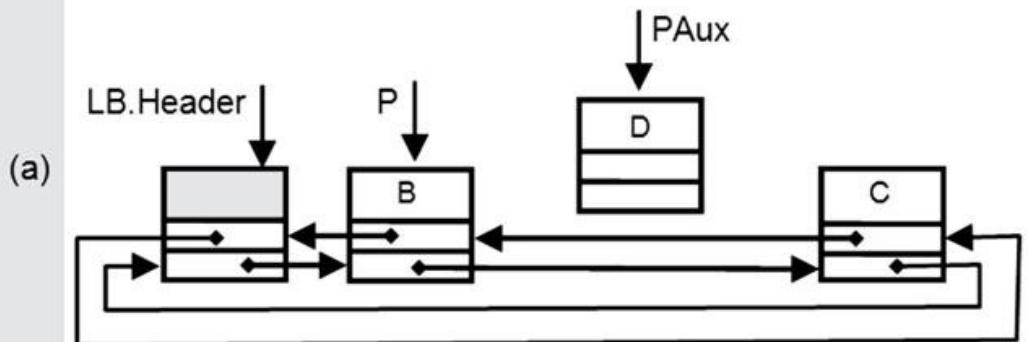
Já implementamos uma Pilha e uma Fila a partir das Operações de Baixo Nível especificadas na [Figura 7.11](#). Queremos agora implementar essas Operações de Baixo Nível.

## **Exercício 7.11 Operação InsereADireitaDeP**

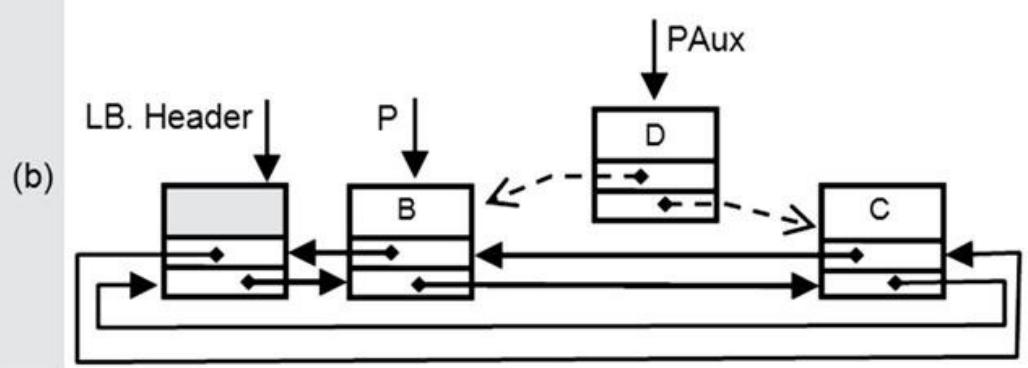
Implemente a operação InsereADireitaDeP de uma Lista Básica, implementada como uma Lista Circular, Duplamente Encadeada, com Nó Header, conforme ilustrado na [Figura 7.12](#).

A [Figura 7.15](#) apresenta, passo a passo, comandos e diagramas que implementam a operação InsereADireitaDeP. Na situação inicial, a lista conta com dois elementos, ‘A’ e ‘B’, além do Nó Header. Alocamos um novo Nó (PAux = NewNode) e colocamos a informação que queremos armazenar nesse novo Nó (PAux → Info = X) — [Figura 7.15a](#). Como queremos inserir esse novo Nó à direita do Nó apontado por P, o campo Dir do Nó apontado por PAux apontará para onde aponta o campo Dir do Nó apontado por P. O campo Esq do Nó apontado por PAux passa a apontar para P. O efeito dessas operações é destacado pelas setas pontilhadas da [Figura 7.15b](#).

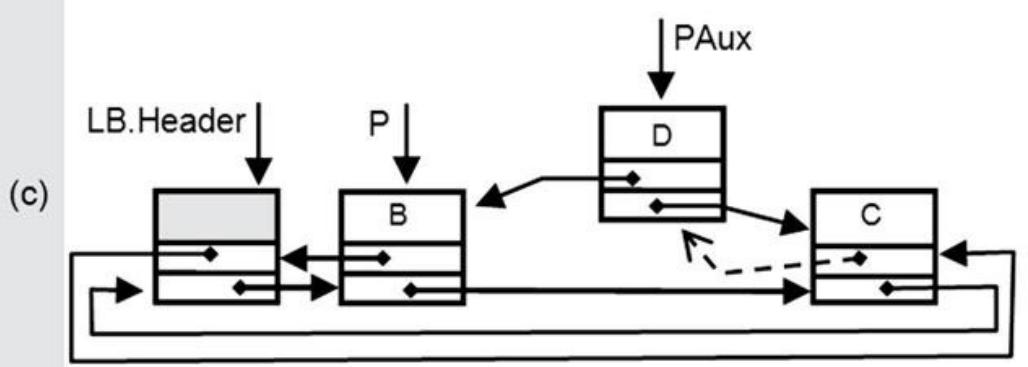
$\text{PAux} = \text{NewNode}; \text{PAux} \rightarrow \text{Info} = X;$



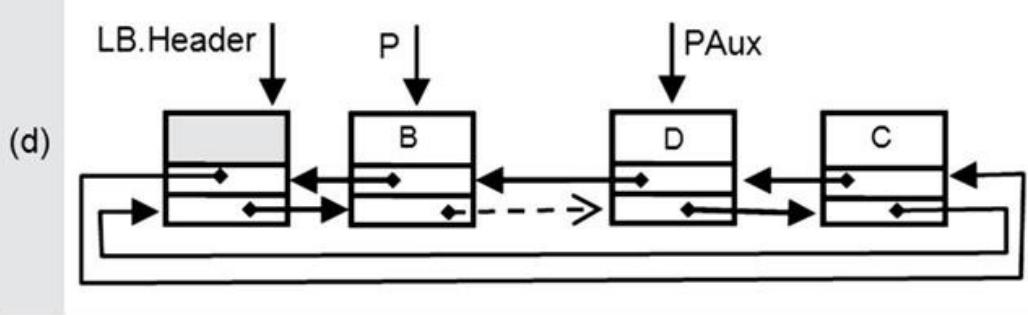
$\text{PAux} \rightarrow \text{Dir} = P \rightarrow \text{Dir}; \text{PAux} \rightarrow \text{Esq} = P;$



$P \rightarrow \text{Dir} \rightarrow \text{Esq} = \text{PAux};$



$P \rightarrow \text{Dir} = \text{PAux};$



**FIGURA 7.15 Execução — Operação InsereADireitaDeP.**

Em seguida, conforme ilustra a seta pontilhada da [Figura 7.15c](#), o campo Esq do Nô apontado por P → Dir passa a apontar para PAux. Finalmente, na [Figura 7.15d](#), o campo Dir do Nô apontado por P passa a apontar para PAux. PAux é uma variável temporária, que deixa de existir ao final da operação. A [Figura 7.16](#) apresenta um algoritmo conceitual que implementa a operação InsereADireitaDeP.

```
InsereADireitaDeP (parâmetro por referência LB do tipo ListaBásica, parâmetro P do tipo
NodePtr, parâmetro por referência DeuCerto do tipo Boolean) {
    /* Insere um novo Nô imediatamente à direita do Nô apontado por P, passado como
       parâmetro */

    Variável auxiliar NovoNô do tipo NodePtr; // NodePtr = ponteiro para Nô

    Se (Cheia(LB)==Verdadeiro)
        Então DeuCerto = Falso;
    Senão {
        DeuCerto = Verdadeiro;
        PAux = NewNode;
        PAux→Info = X;
        PAux→Dir = P→Dir;
        PAux→Esq = P;
        P→Dir→Esq = PAux;
        P→Dir = PAux;
        Aux = Null;
    } // fim Senão
} // fim Insere_à_Direita_de_P
```

**FIGURA 7.16 Implementação da Lista Genérica: Operações Remove\_P e Insere\_à\_Direita\_de\_P.**

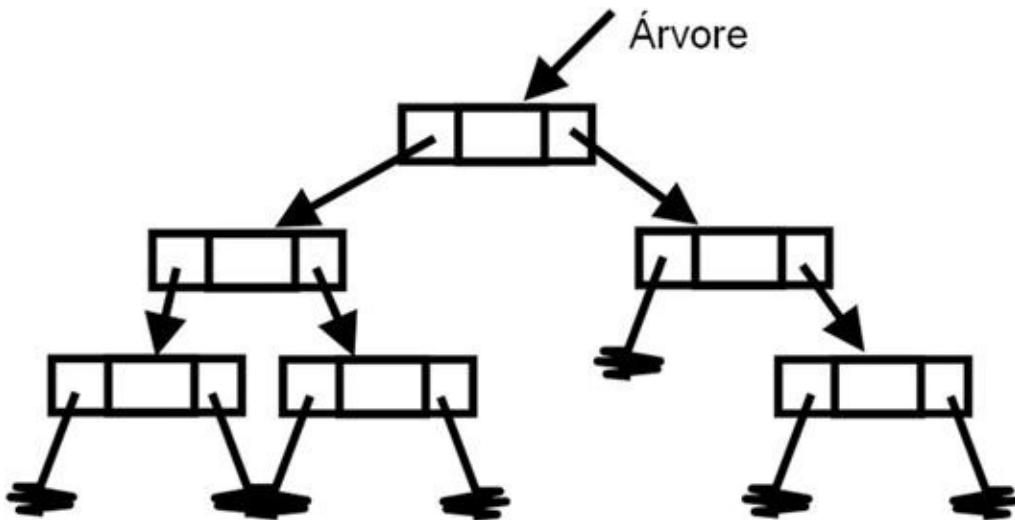
## **Exercício 7.12 Remove\_P, EstáNaLista e Info\_de\_P**

Implemente as seguintes Operações de Baixo Nível, especificadas na [Figura 7.11](#): Remove\_P, EstáNaLista e Info\_de\_P. A Lista Básica deve ser implementada como uma Lista Duplamente Encadeada, Circular e com Nô Header. Apresente também diagramas com a execução passo a passo das operações, semelhante ao realizado na [Figura 7.15](#).

## 7.4 Generalização de Listas Encadeadas

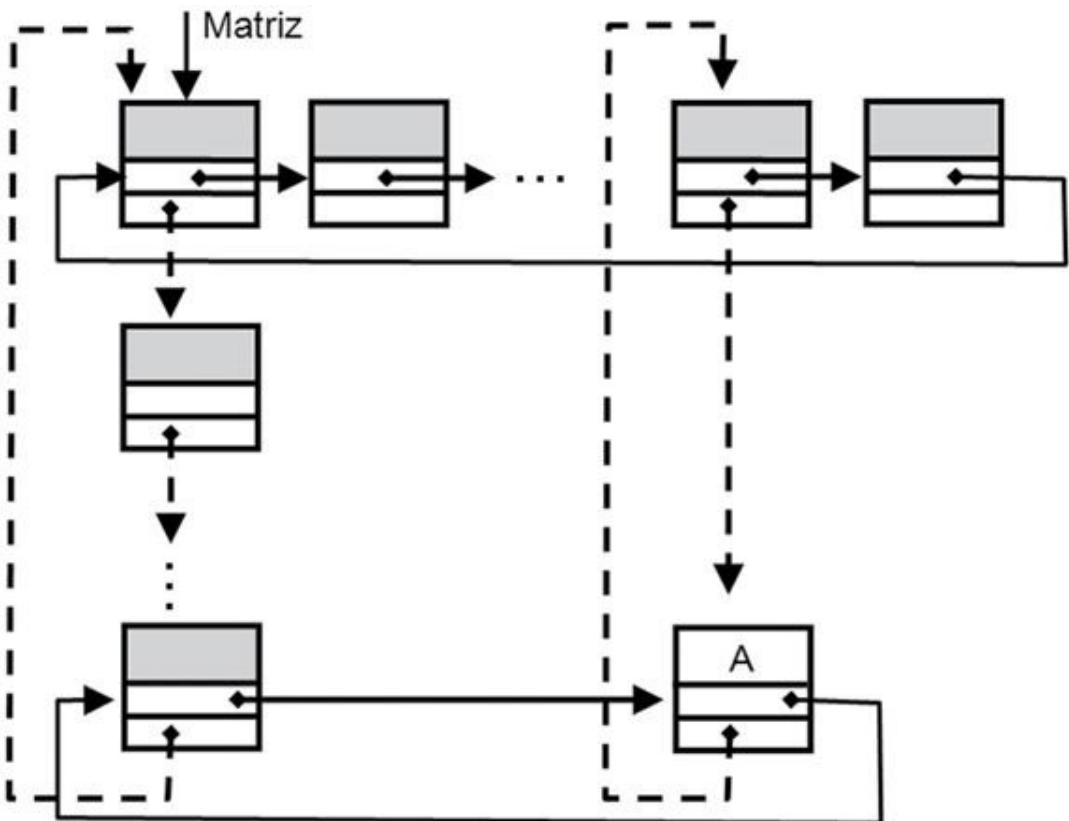
Até o momento estudamos Estruturas Encadeadas em que cada elemento possui um único elemento que é o seu próximo na sequência. No caso das Listas Duplamente Encadeadas, cada elemento possui um próximo e um anterior, os quais designamos como elemento da direita e elemento da esquerda.

É possível projetarmos Estruturas Encadeadas nas quais, para cada elemento, pode existir mais de um “próximo”. Um exemplo desse tipo de estrutura é a Árvore ([Figura 7.17](#)), que estudaremos mais detalhadamente nos capítulos seguintes.



**FIGURA 7.17** Exemplo: Árvores.

Um outro exemplo em que cada elemento pode possuir “mais que um próximo” seria uma Estrutura Encadeada para armazenamento e manipulação de Matrizes Esparsas — [Figura 7.18](#). Matrizes Esparsas são matrizes de grandes dimensões, em que a maioria dos elementos possui o valor zero. Os elementos não nulos estão “esparsos” na matriz, daí o nome Matrizes Esparsas.



**FIGURA 7.18** Exemplo: Matrizes Esparsas.

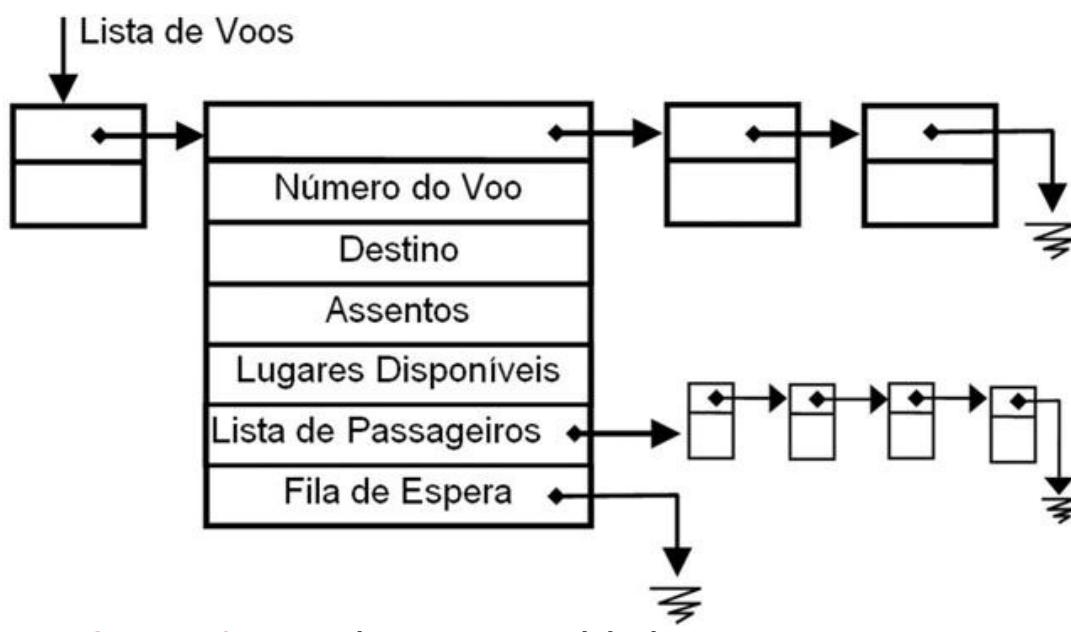
Na implementação encadeada da [Figura 7.18](#), cada elemento não nulo da Matriz está armazenado em um Nό. Cada Nό da estrutura é encadeado em duas Listas: uma Lista encadeando todos os elementos não nulos da mesma linha da matriz e uma Lista conectando todos os elementos não nulos na mesma coluna da Matriz.

As Listas de linhas e de colunas são implementadas como Listas Encadeadas Circulares, com Nό Header. Assim, temos um Nό Header para cada Linha e um Nό Header para cada Coluna. No exemplo da [Figura 7.18](#), está representado um único elemento não nulo, de valor ‘A’. Esse elemento está encadeado na Lista da Linha pelas setas contínuas e na Lista da Coluna pelas setas tracejadas.

## Estruturas aninhadas

Também é possível implementar estruturas encadeadas aninhadas. Imagine que, em um sistema de reserva de passagens aéreas, as informações são armazenadas em uma Lista de Voos. Cada Nό da Lista de Voos armazena o número do Voo, o destino do Voo, o total de assentos, o número de assentos ainda disponíveis, a Lista de Passageiros daquele Voo e a Fila de Espera para aquele Voo.

Como podemos armazenar uma Lista de Passageiros e uma Fila de Espera “dentro” de um Nó da Lista de Voos? Na prática, a Lista de Passageiros e a Fila de Espera não ficariam armazenadas “dentro” do Nó. O Nó armazenaria um ponteiro para a estrutura, como mostra a [Figura 7.19](#). Ao manipular a Lista de Voos, devemos considerar a Lista de Passageiros e a Fila de Espera como Tipos Abstratos de Dados e manipular essas estruturas exclusivamente através dos seus respectivos operadores — Insere, Elimina, e assim por diante. Usando a analogia que estudamos no [Capítulo 1](#), a Lista de Voos deve manipular a Lista Passageiros e a Fila de Espera apenas “pelos botões da televisão”.



**FIGURA 7.19** Exemplo: estruturas aninhadas.

## Estruturas genéricas quanto ao tipo do elemento: Templates

Implementamos Pilhas, Filas e Listas Cadastrais com elementos de um tipo específico. Por exemplo, uma Pilha de elementos do tipo Carta ou de elementos do tipo Inteiro. O funcionamento de uma Pilha de Cartas é idêntico ao funcionamento de uma Pilha de Inteiros. Se já desenvolvemos uma Pilha de Cartas e queremos implementar uma Pilha de Inteiros, temos que implementar tudo novamente e repetir boa parte do código?

A linguagem C++ oferece um mecanismo específico para a implementação de estruturas genéricas quanto ao tipo de elemento: os Templates. Ao implementar

uma Pilha, por exemplo, indicamos através de um Template que o tipo do elemento será definido posteriormente. Então, no momento de criar a Pilha, indicamos o tipo do elemento. Isso permite que um mesmo código seja utilizado para criar Pilhas de elementos de tipos diferentes.

No exemplo da [Figura 7.20](#), apresentamos trechos da definição de uma Classe Nó e de uma Classe Pilha. Indicamos, através do Template denominado “TipoDoElemento”, que o tipo dos elementos que serão armazenados na Pilha será definido posteriormente. Então criamos uma Pilha de elementos do tipo Carta e outra Pilha de elementos do tipo Inteiro.

```
/* definição da Classe Node */
template<class TipoDoElemento> class Node {
    //class Node usa template TipoDoElemento
private:
    TipoDoElemento Info; // o tipo do elemento será definido posteriormente
    Node<TipoDoElemento>* Next;
    ...
/* definição da Classe Pilha */
template<class TipoDoElemento> class Pilha {

private:
    Node<TipoDoElemento> * Topo;
    ...
/* no momento de criar as Pilhas, indicamos o tipo dos elementos */
Pilha<int> * p1 = new Pilha<int>();           // cria uma Pilha de Inteiros - P1
Pilha<Carta> * p2 = new Pilha<Carta>();        // cria uma Pilha de Cartas - P2
```

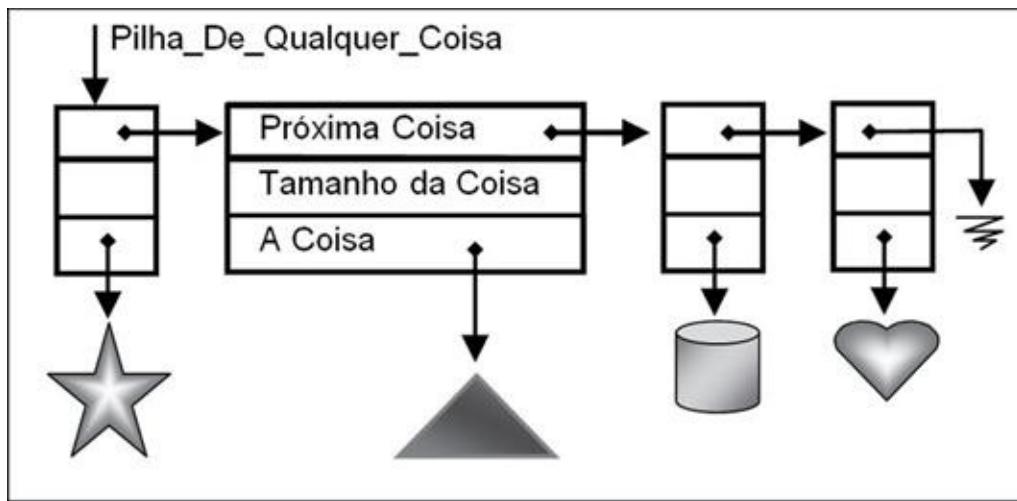
**FIGURA 7.20** Templates: Pilha de Inteiros e Pilha de Cartas.

## Elementos de tipos diferentes em uma mesma estrutura

O uso de Templates permite elaborar um código genérico quanto ao tipo de elemento, mas no momento de criar a estrutura o tipo precisará ser definido. Uma vez definido o tipo do elemento, a estrutura só poderá receber elementos daquele tipo. Por exemplo, se criarmos uma Pilha de Inteiros, não poderemos inserir um elemento do tipo Carta na Pilha de Inteiros.

E se quisermos inserir, em uma mesma Pilha, elementos de qualquer tipo? Como poderíamos inserir em uma mesma Pilha elementos do tipo Inteiro, do tipo Carta, do tipo Pessoa e do tipo Automóvel?

Em vez de armazenar o elemento “dentro” do Nó, podemos armazenar no Nó um ponteiro para o elemento. Além disso, podemos armazenar no Nó uma indicação do tamanho do elemento que será inserido, como esquematizado na Figura 7.21.



**FIGURA 7.21** Inserindo elementos de tipos diferentes em uma única Pilha.

## Funcionalidade versus técnica de implementação

Desenvolvemos Pilhas, Filas e Listas Cadastrais utilizando como técnicas de implementação as Listas Encadeadas e técnicas complementares como encadeamento duplo, listas circulares, nó header e primitivas de baixo nível. Considere, por exemplo, que implementamos a funcionalidade de uma Pilha através de uma lista duplamente encadeada e também através de uma lista com header. A funcionalidade da Pilha é exatamente a mesma, seja qual for a técnica de implementação utilizada. A lista duplamente encadeada e a lista com header foram apenas exemplos de técnicas de implementação para implementar a funcionalidade de uma Pilha.

As Listas Encadeadas, com suas diversas variações, constituem técnicas de implementação poderosas e flexíveis. É possível adaptar ou combinar algumas das técnicas estudadas para projetar sua própria estrutura encadeada, para implementar da melhor maneira possível a funcionalidade requerida por sua aplicação.

## **Exercício 7.13 Fila com atendimento prioritário a idosos**

Decidiu-se que uma Fila de espera em determinada organização deve respeitar dois critérios: (1) a idade do indivíduo que está na Fila, em anos; (2) a ordem de chegada. O primeiro a ser atendido será sempre o indivíduo com idade mais alta. Se houver mais de um indivíduo com a mesma idade, será respeitado o segundo critério, que é a ordem de chegada. Projete uma estrutura para implementar essa Fila com atendimento prioritário a idosos. Implemente a funcionalidade definida com a técnica de implementação que considerar mais adequada.

## **Exercício 7.14 Lista Cadastral em C++**

Implemente uma Lista Cadastral em C++. Escolha uma técnica de implementação para que você possa exercitar os conceitos estudados no Capítulo 7.

### **Projete sua própria estrutura!**

As Listas Encadeadas e suas variações constituem uma técnica poderosa e flexível para armazenamento temporário de conjuntos de elementos. É possível adaptar ou combinar diversas técnicas para projetar a estrutura encadeada que melhor atenda às peculiaridades de sua aplicação.

### **Consulte nos Materiais Complementares**

Vídeos sobre Listas Encadeadas — técnicas complementares



Animações sobre Listas Encadeadas — técnicas complementares



<http://www.elsevier.com.br/edcomjogos>

## **Exercícios de fixação**

**Exercício 7.15** Que vantagens você pode apontar quanto ao uso de um Nô Header em uma estrutura de armazenamento?

**Exercício 7.16** Em sua opinião, quais são as vantagens e as desvantagens de implementar Pilhas, Filas e Listas Cadastrais a partir de primitivas de baixo nível para manipulação de Listas Encadeadas, como as da [Figura 7.11](#)?

**Exercício 7.17** Explique a seguinte frase: “Pilhas, Filas e Listas Cadastrais possuem uma funcionalidade bem definida e podem ser implementadas através de Listas Encadeadas ou através de outras técnicas.” Nesse contexto, qual a diferença entre uma Lista Cadastral e uma Lista Encadeada?

## Soluções para alguns dos exercícios

### **Exercício 7.3 Pilha — Lista Circular Duplamente Encadeada — Desempilha, Cria e Vazia**

```

Cria (parâmetro por referência P do tipo Pilha) {
/* Cria a Pilha P, inicializando a Pilha como vazia - sem nenhum elemento. */
P.Topo = Null;
} // fim Cria Pilha

Boolean Vazia (parâmetro por referência P do tipo Pilha) {
/* Retorna Verdadeiro se a Pilha P estiver vazia; Falso caso contrário */
Se (P.Topo == Null)
Então Retorne Verdadeiro; // a Pilha P está vazia
Senão Retorne Falso; // a Pilha P não está vazia
} // fim Vazia

Desempilha(parâmetro por referência P do tipo Pilha, parâmetro por referência X do tipo Char,
parâmetro por referência DeuCerto do tipo Boolean) {
/* Se a Pilha P estiver vazia o parâmetro DeuCerto deve retornar Falso. Caso a Pilha P não estiver
vazia, a operação Desempilha deve retornar o valor do elemento do topo da Pilha no parâmetro X.
O Nó em que se encontra o elemento do topo deve ser desalocado, e o topo da Pilha deve
ser atualizado */

Variável PAux do tipo NodePtr;
Se (Vazia(P)==Verdadeiro)
Então DeuCerto = Falso; // se a Pilha P estiver vazia, não desempilha
Senão { DeuCerto = Verdadeiro;
        Se (P.Topo→Dir == P.Topo)
        Então { /* trata o Caso 1 - Pilha com um único elemento... passará a ser vazia */
                PAux = P.Topo;
                P.Topo = Null;
                DeleteNode(PAux);
                PAux = Null; // só pode ir para Null após o FreeNode
        }
        Senão { /* trata o Caso 2 - Pilha com vários elementos */
                PAux = P.Topo;
                P.Topo = P.Topo→Dir; // atualiza o Topo da Pilha
                P.Topo→Esq = PAux→Esq;
                PAux→Esq→Dir = P.Topo;
                DeleteNode(PAux);
                PAux = Null; // só pode ir para Null após o DeleteNode
        }
    }
} // fim Desempilha

```

## Exercício 7.4 TAD Fila implementada como Lista Circular Duplamente Encadeada

Sugestão: faça uma fila com dois ponteiros: F.Primeiro e F.Último, considere o elemento mais à esquerda como primeiro elemento da Fila e o mais à direita como o último.

**Exercício 7.5 TAD Lista Cadastral implementada como  
Lista Circular Duplamente Encadeada Não Ordenada**



Solução parcial – Operação Retira:

```
Retira (parâmetro por referência L do tipo Lista, parâmetro X do tipo Char, parâmetro por referência
Ok do tipo Boolean) {
/* Caso X for encontrado na Lista L, retira X da Lista e Ok retorna Verdadeiro. Se X não estiver na
Lista L, não retira nenhum elemento e Ok retorna o valor Falso */

Variável P do tipo NodePtr;           // Tipo NodePtr = ponteiro para Nó
Variável AchouX do tipo Boolean;

/* Passo 1: tenta encontrar X na Lista */
Se (L.Primeiro == Null)      // Lista Vazia
Então AchouX = Falso;
Senão { P = L.Primeiro→Dir; // P é um ponteiro auxiliar que percorre a Lista. A Lista não é..
        // ... ordenada; logo, podemos iniciar a busca em L.Primeiro→Dir

        /* enquanto (não achou X) E (não deu a volta na Lista), continua avançando P */
        Enquanto (( P→Info != X ) E ( P != L.Primeiro ))
            P = P→Dir;
        Se (P→Info == X)
            Então AchouX = Verdadeiro;
            Senão AchouX = Falso;
        }

/* Passo 2: se encontrou X, remover da Lista o Nó que contém X */
Se (AchouX == Verdadeiro)          // se X foi encontrado na Lista
Então {   Se (P != L.Primeiro)      // se X não estiver no primeiro Nó da Lista

        Então { /* aqui estamos removendo um Nó que não é apontado por L */
            Se (P == L.Atual) então L.Atual = L.Atual→Esq ;
            // L.Atual é usado nas operações de percorrer a Lista
            P→Dir→Esq = P→Esq;
            P→Esq→Dir = P→Dir;
            DeleteNode(P);
            P = Null;
        } // fim então

        Senão { /* aqui estamos removendo o Nó apontado por L.Primeiro */
            Se (P == L.Atual) então L.Atual = L.Atual→Esq ;
            // L.Atual é usado nas operações de percorrer a Lista

            Se (L.Primeiro→Dir == L.Primeiro) // se a Lista tem um único Nó...
                Então { L.Primeiro = Null; // ... a Lista passará a ser vazia...
                        L.Atual = Null;
                }
            Senão L.Primeiro = L.Primeiro→Dir; // .. L. .Primeiro passa a apontar para um
            outro Nó qualquer
                P→Dir→Esq = P→Esq;
                P→Esq→Dir = P→Dir;
                DeleteNode(P);
                P = Null;
            } // fim senão
        } // fim Então
} // fim Retira
```

## **Exercício 7.6 TAD Lista Cadastral implementada como Lista Circular Duplamente Encadeada Não Ordenada sem elementos repetidos e com Nó Header**

Solução parcial — Operações Cria e EstáNaLista:

```
Cria (parâmetro por referência L do tipo Lista) {  
    /* Cria a Lista sem nenhum elemento mas com o Nó Header. O ponteiro H aponta para o Nó  
    Header. */  
    L.Header = NewNode;  
    L.Header → Dir = L.Header;  
    L.Header → Esq = L.Header;  
    L.Atual = L.Header;  
} // fim Cria  
  
Boolean EstáNaLista (parâmetro por referência L do tipo Lista, parâmetro X do tipo Char) {  
    /* Caso X seja encontrado na Lista L, retorna Verdadeiro. Retorna Falso caso contrário */  
  
    Variável P do tipo NodePtr;          // Tipo NodePtr = ponteiro para Nó  
    Variável AchouX do tipo Boolean;  
  
    P = L.Header → Dir;      // P é um ponteiro auxiliar que percorre a Lista  
    L.Header → Info = X;      // atribuição de X ao Nó Header, apenas para auxiliar na busca  
  
    Enquanto (P → Info != X)  
        P = P → Dir;  
  
    Se (P != L.Header)  
        Então AchouX = Verdadeiro;    // achamos um X armazenado na Lista  
    Senão AchouX = Falso;  
    Retorne AchouX;  
} // fim EstáNaLista
```

## **Exercício 7.10 Implemente uma Lista Cadastral sem repetições a partir das Operações de Baixo Nível para Manipulação de Listas Encadeadas**

```

Insere (parâmetro por referência LB do tipo ListaBásica, parâmetro X do tipo Char, parâmetro por
referência DeuCerto do tipo Boolean) {
    /* Insere o elemento X na Lista Cadastral Sem Repetições implementada na Lista Básica LB */
    Variável P do tipo NodePtr; // NodePtr = Ponteiro para Nó
    Se (EstáNaLista(LB, X, P)==Verdadeiro)
        Então DeuCerto = Falso; // a Lista Cadastral é sem repetições
        Senão InsereADireitaDeP(LB, LB.Header, X, DeuCerto); /* insere X à direita do Nó apontado pelo
        segundo parâmetro. Não precisamos que a Lista seja ordenada. Assim, inserimos em qualquer
        posição - por exemplo, à direita de LB.Header. O parâmetro DeuCerto é atualizado pela operação
        InsereADireitaDeP */
    } // fim Insere

    Retira(parâmetro por referência LB do tipo ListaBásica, parâmetro por referência X do tipo Char,
    parâmetro por referência DeuCerto do tipo Boolean) {
        /* Retira o X da Lista */
        Variável P do tipo NodePtr; // NodePtr = Ponteiro para Nó
        Se (EstáNaLista(LB, X, P)==Verdadeiro) // P estará apontando para o Nó onde está X
            Então Remove_P (LB, P, DeuCerto); // remove o Nó apontado por P
            Senão DeuCerto = Falso; // X não está na Lista e não pode ser retirado
        } // fim Retira

```

## Exercício 7.12 Remove\_P, EstáNaLista e Info\_de\_P

Solução parcial:

```

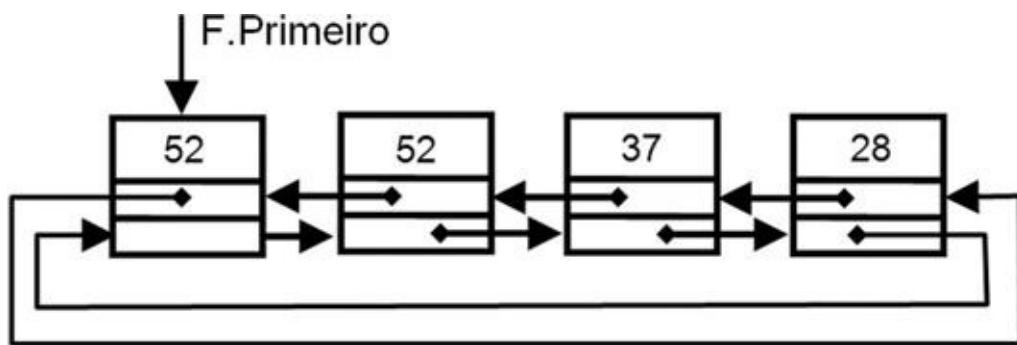
Remove_P (parâmetro por referência LB do tipo ListaBásica, parâmetro por referência P do tipo
NodePtr, parâmetro por referência DeuCerto do tipo Boolean) {
    /* Remove o Nó apontado pelo ponteiro P, passado como parâmetro. A operação só não será bem-
    sucedida se houver uma tentativa de remover o Nó Header, ou seja, se P = LB.Header */

    Se (P == LB.Header) // ou seja, se alguém está tentando remover o Nó Header...
        Então DeuCerto = Falso; // ... não permitimos remover o Nó Header
        Senão {
            DeuCerto = Verdadeiro;
            Se (P == LB.Atual) então LB.Atual = LB.Atual→Esq ; // usado nas operações de percorrer a
            lista
            P→Dir→Esq = P→Esq;
            P→Esq→Dir = P→Dir;
            DeleteNode(P);
            P = Null;
        }
    } // fim Remove_P

```

## Exercício 7.13 Fila com atendimento prioritário a idosos

- Implemente, por exemplo, através de uma lista ordenada pela idade.
- Na operação Insere, um indivíduo de idade X deve ser inserido APÓS todos os demais indivíduos com idade Y < X e APÓS todos os indivíduos com a mesma idade X que já estão na Fila.
- Retirar sempre o primeiro da Fila.



### Exercício 7.14 Lista Cadastral em C++

```

/* arquivo testedup2.h - implementa uma Lista Cadastral sem elementos repetidos através de uma
Lista Circular Duplamente Encadeada Não Ordenada e com Header */
#include<conio.h>
#include<stdio.h>
#include<iostream>

using namespace std;

struct Node {
    char Info;
    Node *Dir;
    Node *Esq;
};

typedef struct Node * NodePtr;

struct Lista{
    Node *Header;
    Node *Atual;
};

void Cria(Lista *L){
    L->Header = new Node();
    L->Header->Dir = L->Header;
    L->Header->Esq = L->Header;
    L->Atual = L->Header;
}

bool Vazia(Lista *L){
    if(L->Header->Dir == L->Header) // ou L->Header->Esq = L->Header
        return true;
    else
        return false;
}

bool EstaNaLista(Lista *L, char X) {
    NodePtr P; // Tipo NodePtr = ponteiro para Nó
    bool AchouX;

    P = L->Header->Dir; // P é um ponteiro auxiliar que percorre a Lista
    L->Header->Info = X; // atribuição de X ao Nó Header, apenas para auxiliar na busca
    while (P->Info != X)
        P = P->Dir;
    if (P != L->Header)
        AchouX = true; // achamos um X armazenado na Lista
    else
        AchouX = false;
    return AchouX;
} // fim EstaNaLista

void Insere(Lista *L, char X, bool *DeuCerto){
    if (EstaNaLista(L, X))
        *DeuCerto = false;
    else { NodePtr PAux = new Node;
        if (PAux == NULL) // se PAux retornar NULL, não há mais memória
            *DeuCerto = false; // a operação não deu certo - equivale a fila cheia
        else { *DeuCerto = true;
            PAux->Info = X;
            PAux->Dir = L->Header->Dir;
            PAux->Esq = L->Header;
            L->Header->Dir->Esq = PAux;
            L->Header->Dir = PAux;
        } // else
    } // else
} // Insere

```



```

void Retira(Lista *L, char *X, bool *DeuCerto){
    NodePtr P;
    if(Vazia(L)) {
        *DeuCerto=false;}
    else { P = L->Header->Dir; // P percorre a Lista
        L->Header->Info = *X; // apenas para auxiliar na busca
        while (P->Info != *X)
            P = P->Dir;
        if (P == L->Header)
            *DeuCerto = false; // achamos um X armazenado na Lista
        else { *DeuCerto = true;
            *X = P->Info; // na verdade é o mesmo valor...
            P->Dir->Esq = P->Esq;
            P->Esq->Dir = P->Dir;
            if (L->Atual == P) L->Atual = L->Atual->Esq;
            delete(P);
            P = NULL;
        } // else
    } // else
} // retira

void PegaOPrimeiro(Lista *L, char *X, bool *TemElemento) {
    L->Atual = L->Header->Dir;
    if (L->Atual != L->Header)
    {
        *TemElemento = true;
        *X = L->Atual->Info; }
    else *TemElemento = false;
} // fim PegaOPrimeiro

void PegaOProximo (Lista *L, char *X, bool *TemElemento) {
    L->Atual = L->Atual->Dir; // tenta avançar para o próximo elemento
    if (L->Atual == L->Header) // Se tiver elemento, retorna o valor em X
        *TemElemento = false;
    else { *TemElemento = true;
        *X = L->Atual->Info;
    }
} // fim PegaOProximo

void Destroi(Lista *L){ // remove todos os nós da Fila
    bool TemElemento, Ok;
    char X;
    PegaOPrimeiro(L, &X, &TemElemento ); // pega o primeiro, se existir
    while (TemElemento) {
        Retira(L, &X, &Ok); // ao retirar elemento, Atual passa para Atual->Esq
        PegaOProximo(L, &X, &TemElemento); }
    delete (L->Header);
    L->Header = NULL;
    L->Atual = NULL;
} // fim Destroi */

/* arquivo testedup2.cpp - testa o TAD arquivo testedup2.h
que implementa uma Lista Cadastral sem elementos repetidos através de uma
Lista Circular Duplamente Encadeada Não Ordenada e com Header */

#include "testedup2.h"

```



```

void Imprime(Lista *L){ // imprime sem abrir a TV
    bool TemElemento, Ok;
    char X;
    cout << "imprimindo a lista: ";
    if (Vazia(L)==false) {
        PegaOPrimeiro(L, &X, &TemElemento ); // pega o primeiro, se existir
        while (TemElemento) {
            cout << " " << X ;
            PegaOProximo(L, &X, &TemElemento); }
        }
        cout << " " << endl ;
    } // fim Imprime

int main(){
    Lista *L = new Lista;
    Cria(L);
    bool Ok;
    char Valor;
    char Op = 't';
    while (Op != 's') {
        cout << "digite: (i)inserir,(r)retirar,(t) testar,(s)sair [enter]" << endl;
        cin >> Op;
        switch (Op) {
            case 'i' : cout << "digite um UNICO CARACTER para inserir [enter]" << endl;
            cin >> Valor;
            Insere(L,Valor,&Ok);
            if (Ok==true) cout << " valor inserido" << endl;
            else cout << " nao conseguiu inserir" << endl;
            break;
            case 'r' : cout << "digite o valor a ser retirado enter]" << endl;
            cin >> Valor;
            Retira (L, &Valor,&Ok);
            if (Ok==true) cout << "valor retirado= " << endl;
            else cout << "nao conseguiu retirar" << endl;
            break;
            default : cout << "saindo... " << endl; Op = 's'; break;
        }; // case
        Imprime (L);
    } // while
    Destroi(L); // retira todos os elementos da fila
    cout << " pressione uma tecla... " << endl;
    getch();
    return(0);
}

```

---

# **PARTE IV**

## **Árvores**

### **OUTLINE**

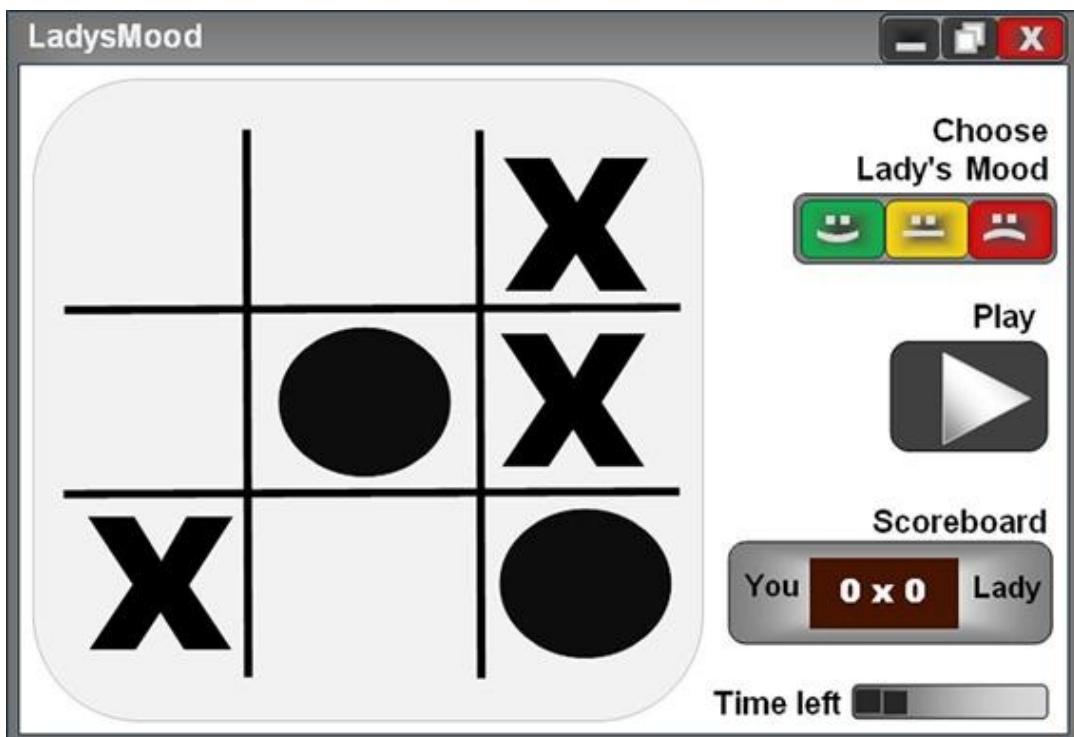
Desafio 4 Desenvolver um game com previsão de jogadas

Capítulo 8 Árvores

Capítulo 9 Árvores Balanceadas

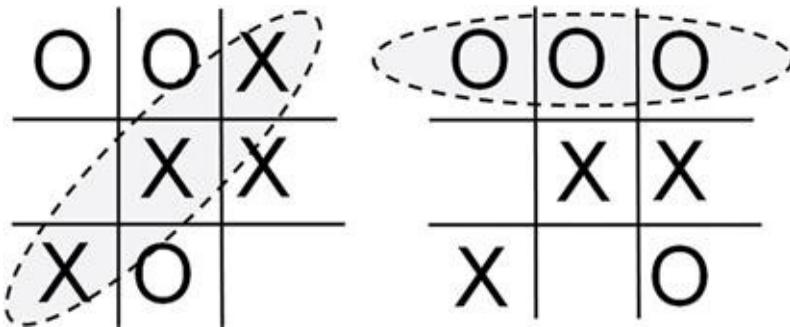
## DESAFIO 4

# Desenvolver um game com previsão de jogadas



**FIGURA D4.1** *Lady's Mood*: prevendo as próximas jogadas e escolhendo a melhor opção.

O *jogo da velha* é uma brincadeira muito conhecida, em que dois jogadores marcam alternadamente um dos nove espaços em um “tabuleiro”. O modo mais tradicional de jogar é desenhar esse tabuleiro em um papel: um dos jogadores marca um “X” e o outro jogador marca um “O”, tudo com uma caneta ou um lápis mesmo. Para vencer, é preciso fazer sua marca em três posições adjacentes, na horizontal, na vertical ou na diagonal. Se você ainda não conhece o *jogo da velha*, jogue algumas partidas em um site de jogos, como o OJogos ou o Papa Jogos (links 1 e 2). Isso o ajudará a conhecer as regras.

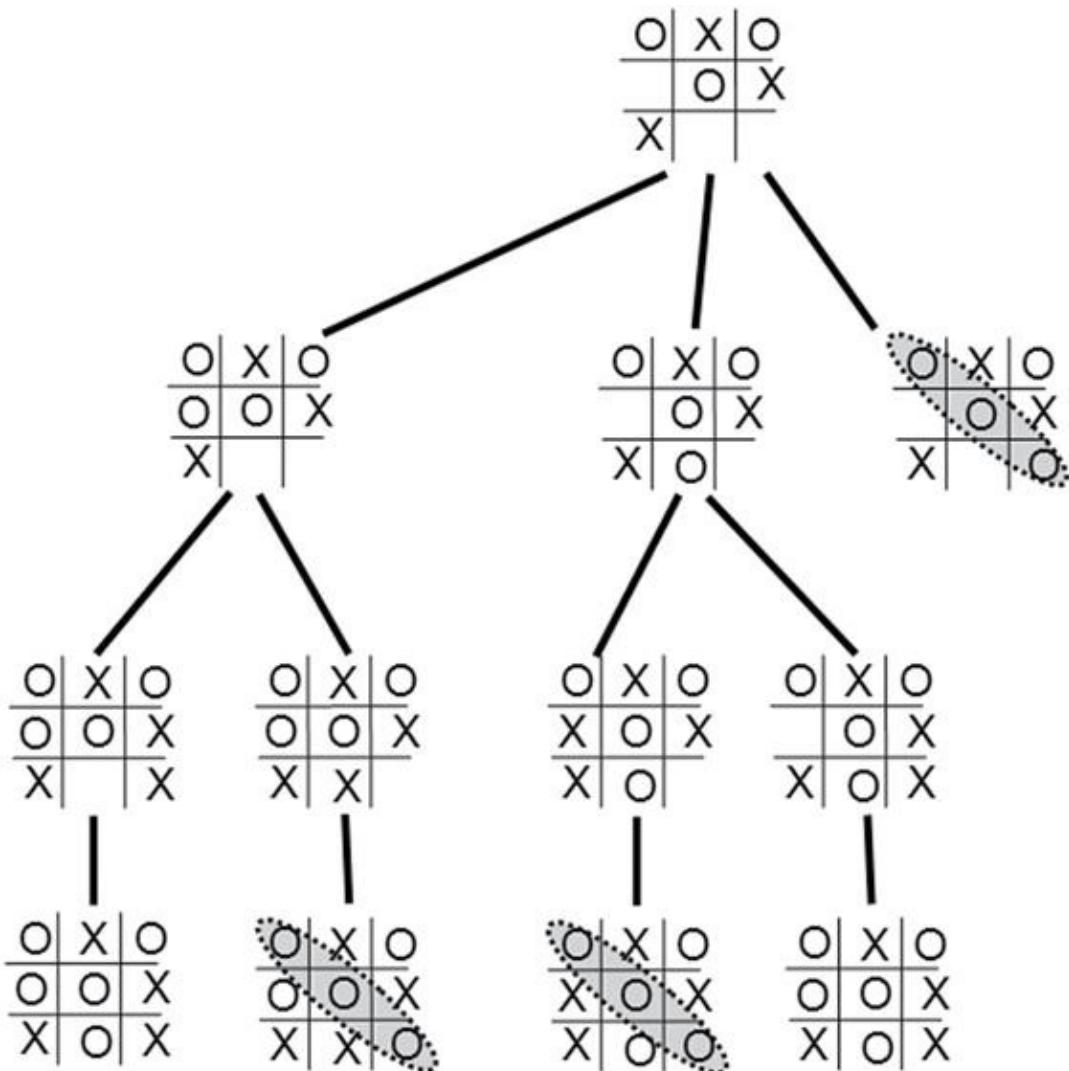


**FIGURA D4.2** “X” vencendo na diagonal e “O” vencendo na horizontal.

Nas versões em software do *jogo da velha*, é possível disputar partidas no modo jogador contra jogador e também no modo jogador contra computador. Nas partidas do tipo jogador contra computador, a cada rodada o game precisa identificar as possíveis jogadas e escolher a opção que proporcione maiores chances de vitória.

## Como escolher a melhor opção de jogada?

Se você fosse desenvolver um game como o *jogo da velha*, como faria para, a cada momento, identificar as próximas jogadas? Como faria para escolher a melhor opção?



**FIGURA D4.3** Trecho de Árvore de Decisão para o *jogo da velha*.

Um dos modos de dar esse tipo de “inteligência” ao jogo é com uma Árvore de Decisão: a partir de determinada situação de jogo, identificamos todas as opções de jogada. Para cada uma dessas opções de jogada, identificamos as próximas jogadas possíveis, e assim por diante, até chegar a uma situação conclusiva, de vitória ou derrota, por exemplo.

Mas como poderíamos armazenar uma Árvore de Decisão para o *jogo da velha*? Como poderíamos classificar as situações de jogo, e saber as opções de jogada para cada situação de jogo?

**Seu desafio: desenvolver um game com previsão de jogadas**

Seu desafio é desenvolver uma adaptação do *jogo da velha*. Use sua criatividade e desenvolva um *jogo da velha* com personalidade própria. Use níveis de dificuldade, ajuste as regras, inclua imagens e sons divertidos! Na verdade, você pode até projetar um game totalmente novo, mas mantendo uma característica essencial: um game que faça previsão de jogadas através de uma Árvore de Decisão. Ou, ainda, você pode desenvolver um game que utilize uma Árvore de Decisão para alguma outra finalidade.

## Por onde começar?

Os [Capítulos 8 e 9](#) de *Estruturas de dados com jogos* o ajudarão a definir um modo interessante de armazenar e manipular Árvores de Decisão, que podem ser utilizadas para identificar as jogadas disponíveis a cada momento da partida, para escolher a melhor opção de jogada ou para alguma outra decisão de jogo. Após estudar esses capítulos, você terá uma boa orientação para implementar sua adaptação do *jogo da velha*, e outros jogos com as características propostas para o Desafio 4.

Construa seu próprio *jogo da velha*! Um jogo da velha diferente, interessante e divertido! Ou crie um jogo totalmente novo, com as características indicadas para o Desafio 4. Aprender a programar pode ser divertido!

### Consulte nos Materiais Complementares

Banco de Jogos: Aplicações de Árvores



<http://www.elsevier.com.br/edcomjogos>

## Links

1. OJogos. Disponível em: <http://www.ojogos.com.br/jogo/tic-tac-toe.html>. (acesso em: outubro de 2013).
2. Papa Jogos. Disponível em: <http://www.papajogos.com.br/jogo/tic-tac-toe.html>. (acesso em: outubro de 2013).

---

## CAPÍTULO 8

# Árvores

---

### Seus objetivos neste capítulo

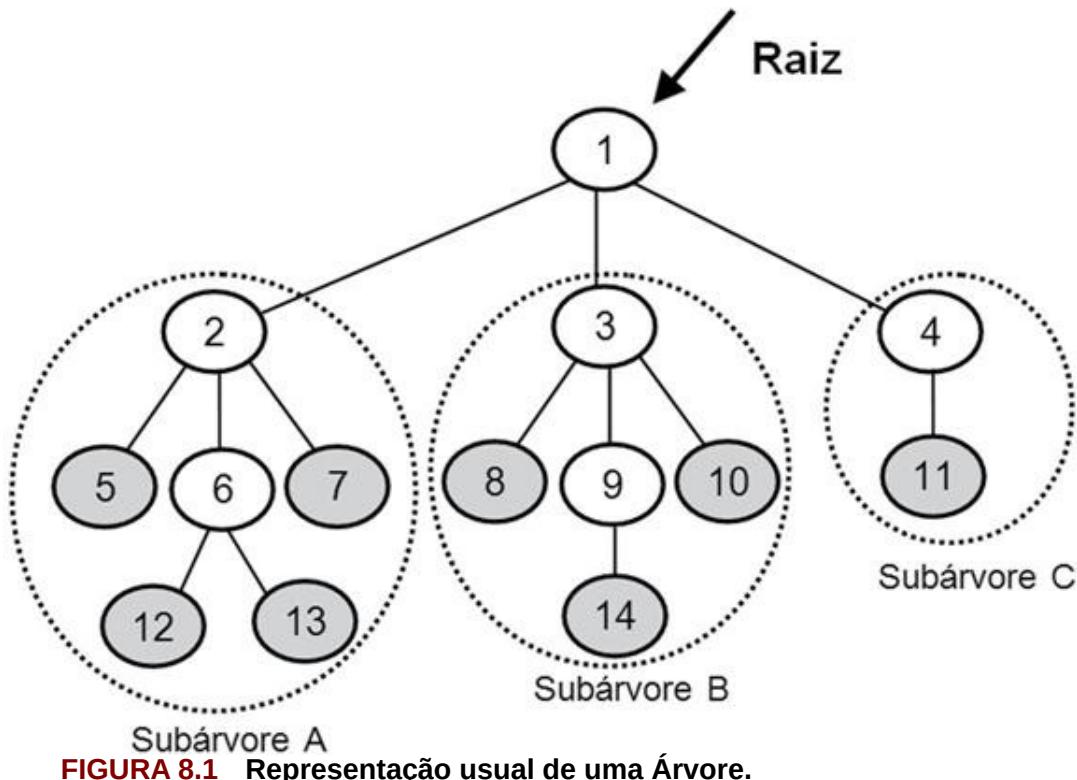
- Entender o conceito, a nomenclatura e a representação usual da estrutura de armazenamento denominada Árvore, e de um tipo especial de Árvore denominada Árvore Binária de Busca — ABB.
- Desenvolver habilidade para elaborar algoritmos sobre Árvores Binárias de Busca e sobre Árvores em geral.
- Conhecer aplicações e a motivação para o uso de Árvores, entender as situações em que o seu uso é pertinente.
- Iniciar o desenvolvimento do seu jogo referente ao Desafio 4.

### 8.1 Árvores: conceito e representação

A [Figura 8.1](#) mostra a representação usual de uma estrutura de armazenamento denominada Árvore, com as seguintes características:

- Cada Árvore possui um **Nó Raiz** e possivelmente várias **Subárvores**. Na [Figura 8.1](#), um ponteiro denominado Raiz está apontando para o Nó Raiz, que contém o valor ‘1’. As Subárvores do Nó Raiz estão destacadas com círculos pontilhados. A Subárvore C, mais à direita, é composta pelos Nós de valores ‘4’ e ‘11’.
- Cada Subárvore também pode ser considerada uma Árvore (e, nesse ponto, a definição passa a ser **recursiva**).
- Os Nós destacados com fundo cinza (‘5’, ‘7’, ‘8’, ‘10’, ‘11’, ‘12’, ‘13’ e ‘14’) são chamados de **Folhas**, ou **Nós Terminais**, pois não possuem Subárvores.
- Podemos dizer que os Nós com valores ‘2’, ‘3’ e ‘4’ são **Filhos** do Nó com valor ‘1’; podemos também dizer que o Nó de valor ‘1’ é **Pai** dos Nós de valor ‘2’, ‘3’ e ‘4’.
- A **Altura** de uma Árvore é definida pelo número de níveis em que os seus Nós estão distribuídos. No exemplo da [Figura 8.1](#), a Altura da Árvore como um todo é 4. No Nível 1 temos o Nó de valor ‘1’; no Nível 2 temos ‘2’, ‘3’ e

‘4’; no Nível 3 temos os Nós de valores ‘5’ a ‘11’; no Nível 4 da Árvore, temos os Nós de valores ‘12’, ‘13’ e ‘14’.



**FIGURA 8.1** Representação usual de uma Árvore.

Note que a Raiz da Árvore é representada na parte de cima do diagrama, e as Folhas da Árvore são representadas na parte de baixo. Ou seja, nessa representação, a Árvore está de cabeça para baixo! Note também que a Árvore é uma **estrutura hierárquica**: um Nό Pai é hierarquicamente “superior” aos seus Nόs Filhos.

## 8.2 Árvores Binárias e Árvores Binárias de Busca

Uma propriedade importante de uma Árvore é o **número máximo de Filhos**, considerados todos os Nós. No exemplo da [Figura 8.1](#), na Árvore como um todo, o número máximo de Filhos é três. O Nό de valor ‘1’, por exemplo, possui três Filhos. Nenhum nό da Árvore possui quatro ou mais filhos.

Nas **Árvores Binárias**, cada Nό da Árvore possui, no máximo, dois Filhos ou, ainda, duas Subárvore.

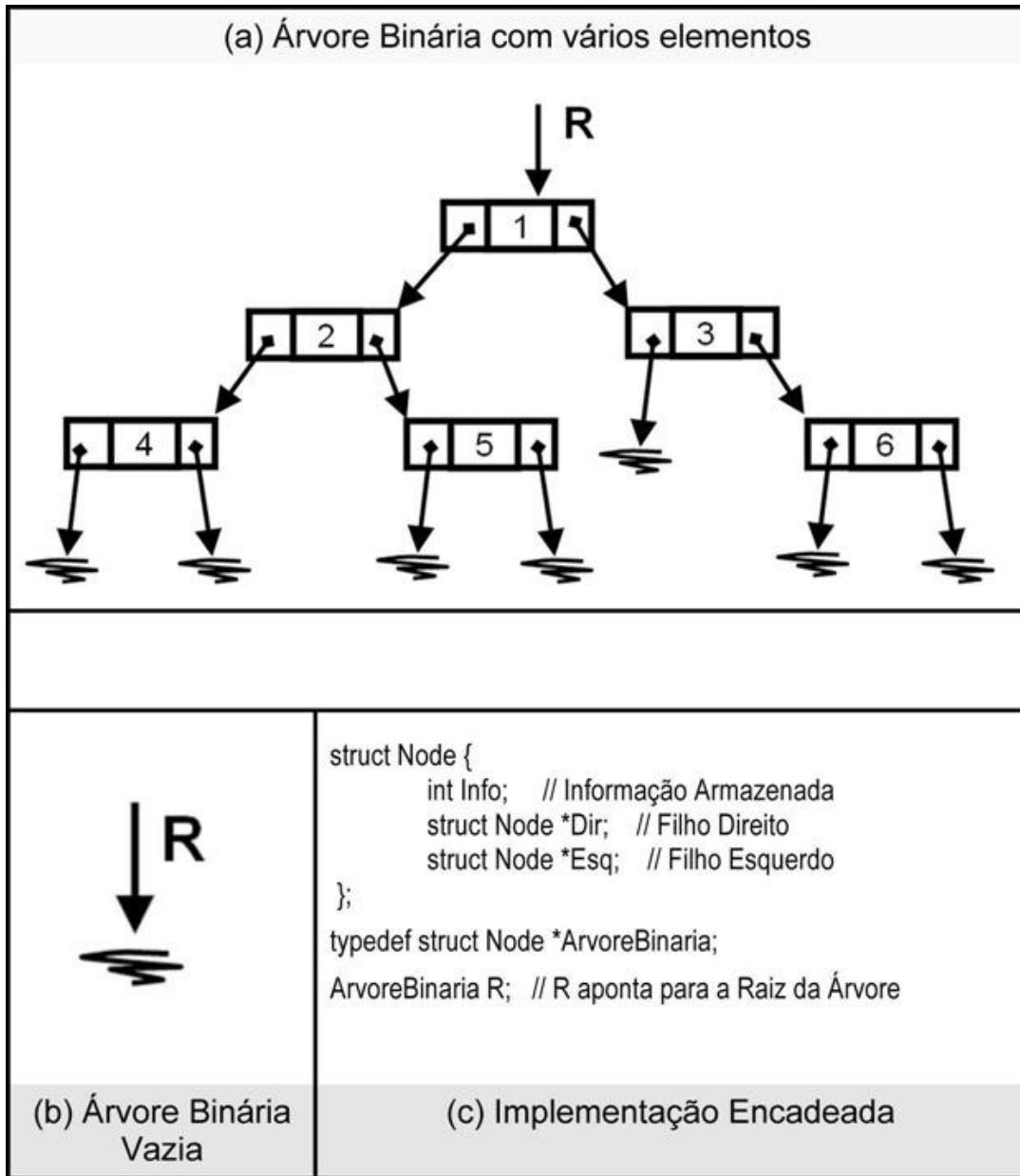
**Definição: Árvore Binária**

Nas Árvores Binárias, cada Nó da Árvore possui, no máximo, dois Filhos.

**FIGURA 8.2** Definição de Árvore Binária.

Nas Árvores Binárias, cada Nó da Árvore possui, no máximo, dois Filhos.

Podemos utilizar uma Árvore Binária para armazenamento temporário de conjuntos de elementos. A [Figura 8.3](#) mostra a representação de uma Árvore Binária, de Raiz R, implementada com Alocação Encadeada. Em cada Nó da Árvore Binária temos a **Informação** a ser armazenada (com valores ‘1’ a ‘6’), um ponteiro para o **Filho Direito** e um ponteiro para o **Filho Esquerdo**. Nos Nós Terminais, os campos que indicam os Filhos Esquerdo e Direito estão apontando para Null.



**FIGURA 8.3** Implementando uma Árvore Binária com Alocação Encadeada e Dinâmica.

Conforme a definição de [Drozdek \(2002; p. 198\)](#), em uma **Árvore Binária de Busca**, para cada Nó as informações armazenadas na Subárvore Esquerda são menores do que a informação armazenada no Nó Raiz, e as informações armazenadas na Subárvore Direita são maiores do que a informação armazenada no Nó Raiz. Para [Pereira \(1996, p. 176\)](#), uma Árvore Binária de Raiz R pode ser considerada uma Árvore Binária de Busca se atender a três critérios: (1) a Informação de cada Nó da Subárvore Esquerda do Nó apontado por R é menor do que a Informação armazenada no Nó apontado por R; (2) nenhum Nó da

Subárvore Direita do Nó apontado por R possui informação menor do que a Informação armazenada no Nó apontado por R; (3) as Subárvores Esquerda e Direita do Nó apontado por R também são ABBs. Note que, pelo critério 3, a definição de Pereira é recursiva.

Na definição de Pereira, uma ABB pode ter elementos repetidos (veja o critério 2). Isso não ocorre na definição de Drozdek. Adaptando a definição de Pereira de modo a não permitir repetição de informações em uma Árvore, chegamos à definição da [Figura 8.4](#).

#### Definição: Árvore Binária de Busca — ABB

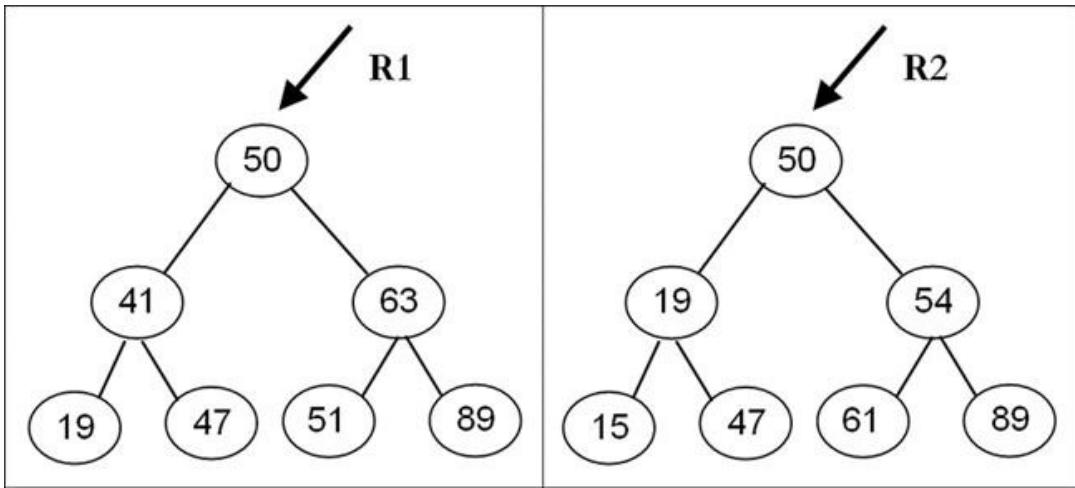
Uma Árvore Binária com Raiz R pode ser considerada uma Árvore Binária de Busca se atender aos três seguintes critérios:

- (1) A Informação de cada Nó da Subárvore Esquerda de R é menor do que a Informação armazenada no Nó apontado por R.
- (2) A Informação de cada Nó da Subárvore Direita de R é maior do que a Informação armazenada no Nó apontado por R.
- (3) As Subárvores Esquerda e Direita do Nó apontado por R também são ABBs.

**FIGURA 8.4** Definição de Árvore Binária de Busca – adaptada de Pereira (1996, p. 176), de modo a não permitir elementos repetidos.

### Exercício 8.1 São ABBs?

Verifique se as Árvores R1 e R2, da [Figura 8.5](#), são Árvores Binárias de Busca segundo a definição da [Figura 8.4](#).

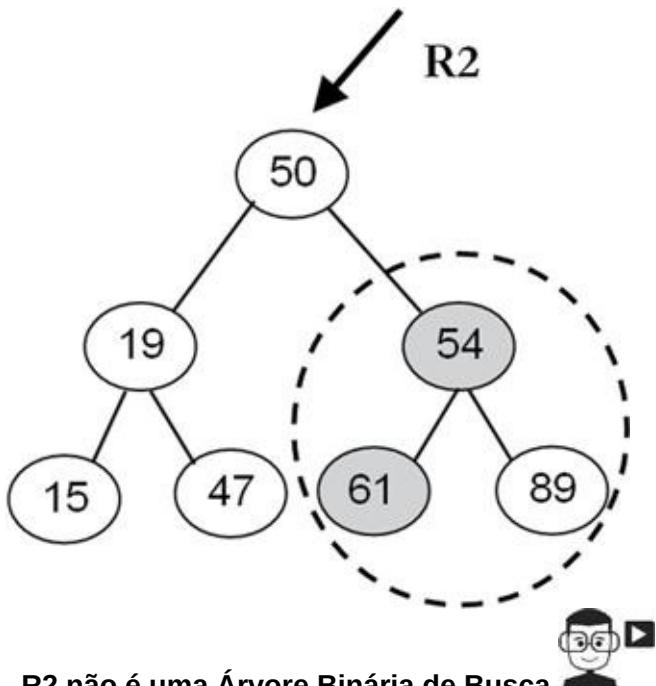


**FIGURA 8.5** R1 e R2 são Árvores Binárias de Busca?

A Árvore R1 é uma Árvore Binária de Busca, pois atende aos três critérios da definição da [Figura 8.4](#). As Informações armazenadas em todos os Nós da Subárvore Esquerda de R1 (19, 41 e 47) são menores do que a Informação armazenada no Nós Raiz (50); (2) as Informações armazenadas em todos os Nós da Subárvore Direita de R1 (51, 63 e 89) são maiores do que a Informação armazenada no Nós Raiz (50); (3) se aplicarmos essa mesma análise, recursivamente, para as Subárvores Esquerda e Direita de R1, os critérios continuam sendo respeitados.

A Árvore R2 não é uma ABB. Você saberia dizer a razão? Preste atenção ao critério 3 da definição de Árvore Binária de Busca e compare as Árvores R1 e R2. Por que R1 é uma ABB e R2 não é?

Vamos considerar isoladamente a Subárvore de R2 composta pelos Nós 54, 61 e 89 (destacada com o círculo pontilhado na [Figura 8.6](#)). Note que o Nós que armazena o valor 61 está na Subárvore Esquerda do Nós que armazena o valor 54. Como 61 é maior do que 54, essa situação quebra o critério 1 da definição de Árvores Binárias de Busca. Se a Subárvore composta pelos Nós 54, 61 e 89 não é uma Árvore Binária de Busca, pelo critério 3 a Árvore R2 como um todo também não será.



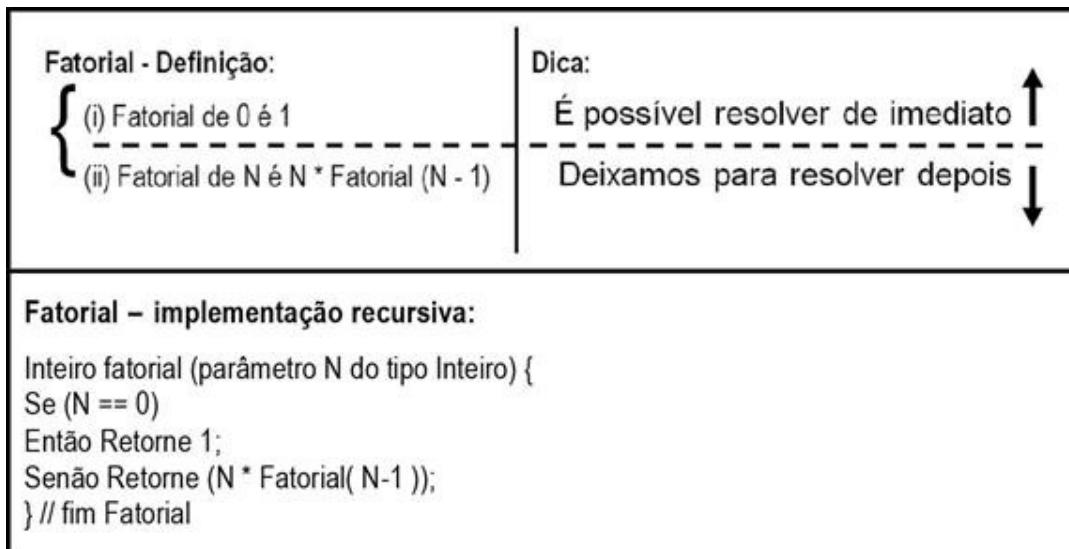
**FIGURA 8.6** R2 não é uma Árvore Binária de Busca.

## 8.3 Algoritmos recursivos para Árvores Binárias de Busca

Devido à natureza hierárquica e recursiva das Árvores, os algoritmos para manipulação de Árvores podem ser implementados de modo mais fácil e natural se utilizarmos a recursividade.

### Revisão sobre algoritmos recursivos

Um subprograma é recursivo quando faz chamadas a ele mesmo. Um exemplo clássico de algoritmos recursivos é o cálculo do fatorial. A Figura 8.7 apresenta a definição de fatorial. Note, na cláusula ii, que se trata de uma definição recursiva.



**FIGURA 8.7** Fatorial: recursividade na definição e na implementação.

Para resolver um problema recursivamente, precisamos identificar os casos em que conseguimos resolver o problema de imediato e os casos em que não conseguimos fazê-lo. Nos casos em que conseguimos resolver de imediato, simplesmente apresentamos a solução. Nos casos em que não conseguimos resolver de imediato, precisamos decompor o problema em problemas menores, nos aproximando da solução. No fundo, deixamos para resolver o problema em outro momento, fazendo uma chamada recursiva. Na [Figura 8.7](#), a linha tracejada delimita as situações em que conseguimos resolver de imediato e as situações em que precisamos deixar para resolver em outro momento, decompondo o problema em problemas menores através de uma ou mais chamadas recursivas.

A [Figura 8.7](#) apresenta também uma implementação recursiva para o cálculo do fatorial de  $N$ . A situação em que conseguimos resolver o problema de imediato ocorre quando  $N$  é igual a zero. Nesse caso, resolvemos o problema simplesmente retornando o valor 1. Nos demais casos, não conseguimos resolver de imediato, decomponemos o problema em um problema menor e retornamos o resultado da expressão  $N * \text{Fatorial}(N - 1)$ . Ou seja, o resultado de fatorial de  $N$  será obtido pela multiplicação do valor  $N$  pelo resultado do fatorial de  $N - 1$ . Calculamos então o fatorial de  $N - 1$  através de uma chamada recursiva, ou seja, uma chamada ao próprio procedimento fatorial.

Vamos exemplificar a execução do algoritmo recursivo da [Figura 8.7](#) calculando o fatorial de  $N$  quando  $N$  é igual a 3. Na primeira chamada ao procedimento fatorial,  $N$  é igual a 3 e o algoritmo retornará o resultado  $3 * 2 * 1$ .

Fatorial (2). Para calcular o fatorial de 2, fazemos a segunda chamada ao procedimento Fatorial, agora com N igual a 2. Nessa segunda chamada, o resultado será  $2 * \text{Fatorial}(1)$ . Calculamos o fatorial de 1 na terceira chamada, e o resultado será  $1 * \text{Fatorial}(0)$ . Na quarta chamada, N é zero, e só então conseguimos resolver o problema em definitivo, sem necessidade de novas chamadas recursivas. Quando N é igual a zero, retornamos o valor 1 como resultado. Esse resultado do fatorial (0) é então repassado à terceira chamada, que o multiplica por 1 e resulta no valor 1. Esse resultado do fatorial (1) é então repassado à segunda chamada, que o multiplica por 2 e resulta no valor 2. Esse resultado do fatorial (2) é então repassado à primeira chamada, que o multiplica por 3 e resulta no valor 6, que é o resultado do fatorial de 3. A [Figura 8.8](#) ilustra a sequência de obtenção de resultados nas quatro chamadas do procedimento Fatorial, para o cálculo do fatorial de 3.

Chamada	N	Resultado
Primeira	3	$\text{Fatorial}(3) = 3 * \text{Fatorial}(2)$
Segunda	2	$\text{Fatorial}(2) = 2 * \text{Fatorial}(1)$ 2 resultado do Fatorial(2) = 2
Terceira	1	$\text{Fatorial}(1) = 1 * \text{Fatorial}(0)$ 1 resultado do Fatorial(1) = 1
Quarta	0	$\text{Fatorial}(0) = 1$ resultado do Fatorial(0) = 1 1

**FIGURA 8.8** Chamadas recursivas para cálculo do fatorial de 3.

## O valor X está na Árvore?

Queremos saber se um valor X está em uma Árvore Binária de Busca de Raiz R. Sendo uma Árvore Binária de Busca, os valores armazenados em cada Nó da Subárvore Esquerda de R precisam ser menores do que o valor armazenado no Nó apontado por R; e os valores armazenados em cada Nó da Subárvore Direita de R precisam ser maiores do que o valor armazenado no Nó apontado por R.

Considere, por exemplo, que o valor de X é 39. Podemos ter quatro situações para X e para o Nó apontado por R, conforme mostra a [Figura 8.9](#). No caso 1, estamos querendo saber se o valor X está em uma Árvore vazia. Essa é uma situação que conseguimos resolver de imediato: X não está na Árvore; apresentamos esse resultado e encerramos o algoritmo.

Caso	X	R	Conclusão
Caso 1: Árvore Vazia	39		X não está na árvore. Encerramos o algoritmo.
Caso 2: $R \rightarrow \text{Info} = X$	39		X está na árvore. Encerramos o algoritmo.
			<b>É possível resolver de imediato e encerrar o algoritmo</b>
			<b>Deixamos para resolver depois, com recursividade</b>
Caso 3: $X < R \rightarrow \text{Info}$	39		Se X estiver na Árvore, estará na Subárvore Esquerda de R. O algoritmo não acaba ainda. Fazemos uma chamada recursiva.
Caso 4: $X > R \rightarrow \text{Info}$	39		Se X estiver na Árvore, estará na Subárvore Direita de R. O algoritmo não acaba ainda. Fazemos uma chamada recursiva.

**FIGURA 8.9** Casos do algoritmo que verifica se um valor X está em uma Árvore Binária de Busca R.

No caso 2, estamos procurando X na Árvore R e sabemos que  $R \rightarrow \text{Info} = X$ . Ou seja, no caso 2, o valor X está armazenado no Nó apontado por R. Também conseguimos resolver essa situação de imediato: temos certeza de que X está na Árvore; apresentamos esse resultado e encerramos o algoritmo.

No caso 3, o Nó apontado por R armazena uma Informação que é maior do que X. O que fazemos nesse caso? Nesse momento da execução ainda não temos como saber com certeza se X está na Árvore. Mas uma coisa nós sabemos: como

se trata de uma Árvore Binária de Busca, e como X é menor do que a Informação armazenada no Nó apontado por R, se X estiver na Árvore estará na Subárvore Esquerda de R. Então deixamos para dar uma resposta definitiva posteriormente e fazemos uma chamada recursiva. Analogamente, no caso 4, se X estiver na Árvore estará na Subárvore Direita de R, e também resolvemos com uma chamada recursiva.

## Exercício 8.2 Algoritmo recursivo: X está na Árvore?

Desenvolva um algoritmo recursivo para verificar se um valor X faz parte de



uma Árvore R. X e R são passados como parâmetros.

```
Boolean EstáNaÁrvore (parâmetro por referência R do tipo ABB, parâmetro X do tipo Inteiro);
/* Verifica se o valor X está na Árvore de Raiz R, retornando Verdadeiro para o caso de X estar na Árvore e Falso para o caso de não estar */
```

Tratando separadamente cada um dos quatro casos especificados na [Figura 8.9](#), chegamos ao algoritmo da [Figura 8.10](#). Nos casos 1 e 2, conseguimos dar uma resposta definitiva, e encerramos o algoritmo retornando os valores Falso e Verdadeiro, respectivamente.

```
Boolean EstáNaÁrvore (parâmetro por referência R do tipo ABB, parâmetro X do tipo Inteiro) {

    /* Verifica se o valor X está ou não está na Árvore de Raiz R, retornando Verdadeiro para o caso de X estar na Árvore e Falso para o caso de não estar */

    Se (R == Null)
        Então Retorne Falso; // Caso 1: Árvore vazia; X não está na Árvore;
    Senão Se (X == R→Info)
        Então Retorne Verdadeiro; // Caso 2: X está na árvore; acabou o algoritmo
    Senão Se (R→Info > X)
        Então Retorne ( Está_Na_Árvore (R→Esq, X) );
        // Caso 3: se estiver na Árvore, estará na Subárvore Esquerda

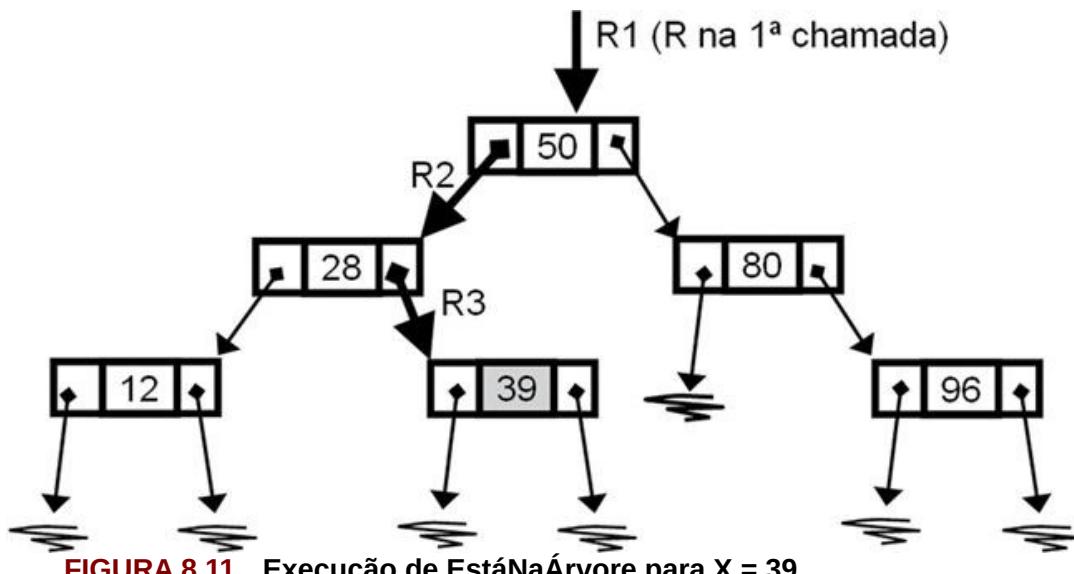
        Senão Retorne ( Está_Na_Árvore (R→Dir, X) );
        // Caso 4: se estiver na Árvore, estará na Subárvore Direita
} // fim EstáNaÁrvore
```

**FIGURA 8.10** Algoritmo conceitual — EstáNaÁrvore.

No caso 3, sabemos que  $R \rightarrow \text{Info} > X$ . Ou seja, o valor armazenado no Nó apontado por  $R$  é maior do que o valor que procuramos. Se  $X$  estiver na Árvore, estará na Subárvore Esquerda de  $R$ . Assim, chamamos recursivamente o procedimento `EstáNaÁrvore` e passamos como primeiro parâmetro  $R \rightarrow \text{Esq}$ . Ou seja, verificamos se o valor  $X$  está na Árvore apontada por  $R \rightarrow \text{Esq}$ . Analogamente, no caso 4, verificamos se o valor  $X$  está na Árvore apontada por  $R \rightarrow \text{Dir}$ .

## Execução do algoritmo `EstáNaÁrvore` para $X = 39$

Para exemplificar a execução do algoritmo recursivo da Figura 8.10, considere que estamos procurando  $X$  com valor igual a 39 na Árvore  $R$  esquematizada na Figura 8.11. A execução implicará três chamadas ao procedimento `EstáNaÁrvore`. Nas três chamadas, o valor de  $X$  será o mesmo, mas o valor de  $R$  será alterado. Os valores de  $R$  para a primeira, segunda e terceira chamadas estão identificados na Figura 8.11 por  $R_1$ ,  $R_2$  e  $R_3$ , respectivamente.



**FIGURA 8.11** Execução de `EstáNaÁrvore` para  $X = 39$ .

Na primeira chamada ao procedimento `EstáNaÁrvore`, a Árvore  $R$  (identificada por  $R_1$ , na primeira chamada) não é vazia, e o valor armazenado no Nó apontado por  $R_1$  é 50, maior que 39, que é o valor de  $X$ . Identificamos, nessa primeira chamada, o caso 3: situação em que  $R \rightarrow \text{Info} > X$ . De acordo com o algoritmo, fazemos então uma chamada recursiva através do comando `Retorne(EstáNaÁrvore( $R \rightarrow \text{Esq}$ ,  $X$ ))`. Com esse comando, o resultado da

primeira chamada ao procedimento `EstáNaÁrvore` retornará exatamente o resultado da segunda chamada, que buscará o valor de X na Árvore apontada por  $R1 \rightarrow \text{Esq}$ .

Entramos, então, na segunda chamada. R, na segunda chamada identificado por R2, agora aponta para o Nó que armazena o valor 28. Ou seja, nessa segunda chamada, identificamos o caso 4, pois  $R2 \rightarrow \text{Info} < X$ . De acordo com o algoritmo, fazemos uma chamada recursiva através do comando `Retorne(EstáNaÁrvore(R → Dir, X))`. Com esse comando, o resultado da segunda chamada ao procedimento `EstáNaÁrvore` retornará precisamente o resultado da terceira chamada, que buscará o valor de X na Árvore apontada por  $R2 \rightarrow \text{Dir}$ .

Entramos então na terceira chamada. R (identificado por R3) agora aponta para o Nó que armazena o valor 39. Identificamos o caso 2 ( $X = R \rightarrow \text{Info}$ ). De acordo com o algoritmo, retornamos o resultado Verdadeiro nessa terceira chamada.

Ao encerrar a terceira chamada, voltamos para a segunda chamada, no ponto onde foi feita a chamada recursiva. O resultado do comando `Retorne(EstáNaÁrvore(R2 → Dir, X))` será `Retorne(Verdadeiro)`, haja visto que o resultado da terceira chamada foi Verdadeiro. Analogamente, voltamos para a primeira chamada, no ponto onde foi feita a chamada recursiva. O resultado do comando `Retorne(EstáNaÁrvore(R1 → Esq, X))` será `Retorne(Verdadeiro)`, haja visto que o resultado da segunda chamada foi Verdadeiro. O resultado Verdadeiro na primeira chamada encerra a execução. A [Figura 8.12](#) ilustra a sequência em que os resultados são obtidos e transferidos entre as três chamadas do procedimento `EstáNaÁrvore`, para a Árvore da [Figura 8.11](#) e X com valor 39.

Chamada	Caso	Resultado
Primeira	3	<code>Retorne(EstáNaÁrvore(R1 → Esq, X))</code>
Segunda	4	<code>Verdadeiro</code>  <code>Retorne(EstáNaÁrvore(R2 → Dir, X))</code>
Terceira	2	<code>Verdadeiro</code>  <code>Retorne Verdadeiro</code>



**FIGURA 8.12** Resultados de `EstáNaÁrvore` para  $X = 39$ .

## Execução do algoritmo EstáNaÁrvore para X=70

Vamos fazer uma segunda execução do algoritmo recursivo da Figura 8.10, agora para verificar se X com valor 70 está na Árvore da Figura 8.13. Essa execução implicará três chamadas ao procedimento EstáNaÁrvore. Nas três chamadas, o valor de X será o mesmo; o valor de R será alterado e identificado na Figura 8.13 por R1, R2 e R3, respectivamente.

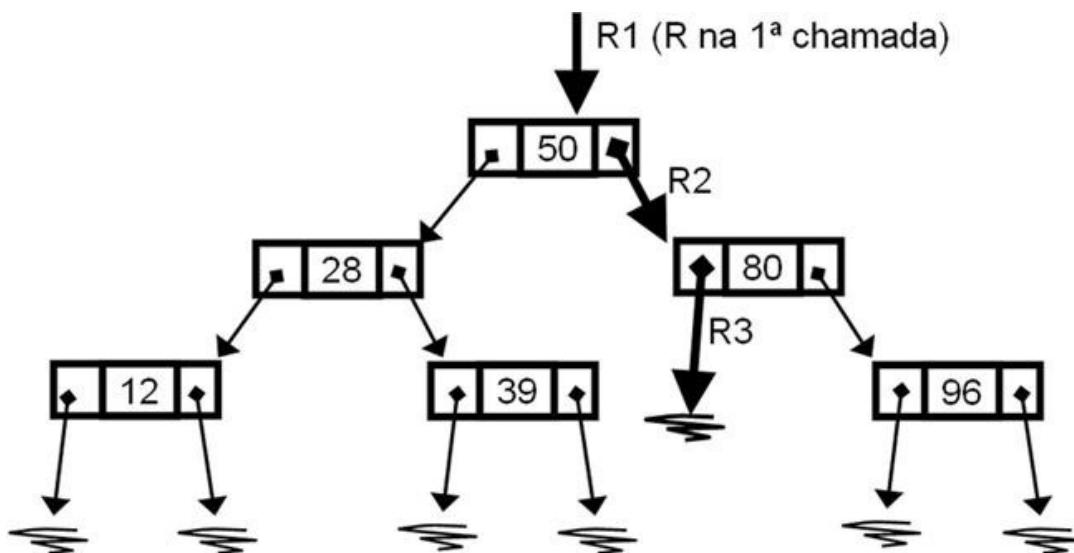


FIGURA 8.13 Execução de EstáNaÁrvore para  $X = 70$ .

Na primeira chamada ao procedimento EstáNaÁrvore, a Árvore R (R1) não é vazia, e o valor armazenado no Nό apontado por R1 é 50, menor que 70, que é o valor de X. Identificamos, nessa primeira chamada, o caso 4: situação em que  $R \rightarrow \text{Info} < X$ . De acordo com o algoritmo, fazemos uma chamada recursiva através do comando `Retorne(EstáNaÁrvore(R → Dir, X))`. Com esse comando, o resultado da primeira chamada ao procedimento EstáNaÁrvore retornará exatamente o resultado da segunda chamada, que buscará o valor de X na Árvore apontada por  $R1 \rightarrow \text{Dir}$ .

Entramos, então, na segunda chamada. R (R2) agora aponta para o Nό que armazena o valor 80. Ou seja, nessa segunda chamada, identificamos o caso 3, pois  $R2 \rightarrow \text{Info} > X$ . De acordo com o algoritmo, fazemos uma chamada recursiva através do comando `Retorne(EstáNaÁrvore (R → Esq, X))`. Com esse comando, o resultado da segunda chamada ao procedimento EstáNaÁrvore retornará precisamente o resultado da terceira chamada, que buscará o valor de

X na Árvore apontada por R2 → Esq.

Entramos então na terceira chamada. R (R3) agora aponta para Null. Identificamos o caso 1 (Árvore vazia). De acordo com o algoritmo, retornamos o resultado Falso nessa terceira chamada para indicar que X não está na Árvore.

Ao encerrar a terceira chamada, voltamos para a segunda chamada, no ponto onde foi feita a chamada recursiva. O resultado do comando Retorne (EstáNaÁrvore(R2 → Esq, X)) será Retorne(Falso), haja visto que o resultado da terceira chamada foi Falso. Analogamente, voltamos para a primeira chamada, no ponto onde foi feita a chamada recursiva. O resultado do comando Retorne (EstáNaÁrvore(R1 → Dir, X)) será Retorne(Falso), haja visto que o resultado da segunda chamada foi Falso. O resultado Falso na primeira chamada encerra a execução. A [Figura 8.14](#) ilustra a sequência em que os resultados são obtidos e transferidos entre as três chamadas do procedimento EstáNaÁrvore, para a Árvore da [Figura 8.13](#) e X com valor 70. Ao elaborar algoritmos recursivos, procure seguir as sugestões da [Figura 8.15](#).

Chamada	Caso	Resultado
Primeira	3	Retorne (EstáNaÁrvore(R1→Dir, X))
Segunda	4	Falso Retorne (EstáNaÁrvore(R2→Esq, X))
Terceira	2	Falso Retorne Falso

**FIGURA 8.14** Resultado de EstáNaÁrvore para X = 70.

### Ao elaborar algoritmos recursivos:

- Liste todos os casos, identificando-os como caso 1, caso 2, e assim por diante.
- Identifique os casos em que é possível dar uma resposta de imediato, e proponha a resposta.
- Identifique os casos em que não é possível resolver de imediato, e procure resolver com uma ou mais chamadas recursivas.

**FIGURA 8.15** Sugestões para elaborar algoritmos recursivos.

## Exercício 8.3 Algoritmo recursivo: imprimir uma Árvore

Desenvolva um algoritmo recursivo para imprimir uma Árvore Binária de Busca de Raiz R, com elementos do tipo Inteiro. Faça uma versão do algoritmo que imprima as informações em ordem crescente e outra versão que imprima em ordem decrescente.

```
ImprimeTodos (parâmetro por referência R do tipo ABB);  
/* Imprime todos os elementos da Árvore de Raiz R */
```

A [Figura 8.16](#) apresenta duas versões do algoritmo que imprime todos os elementos de uma Árvore Binária de Busca. Para imprimir recursivamente todos os elementos de uma Árvore Binária de Busca, temos dois casos a tratar: caso 1 — a Árvore é vazia; caso 2 — a Árvore não é vazia. No caso 1, não é preciso fazer nada, pois não existem elementos a imprimir. No caso 2, imprimimos o elemento que está na raiz e então, recursivamente, imprimimos todos os elementos das Subárvore Esquerda e Direita ([Figura 8.16a](#)). Quando utilizamos essa sequência — primeiro a Raiz, depois todos da Subárvore Esquerda, depois todos da Subárvore Direita, estamos percorrendo a Árvore em “pré-ordem”. O prefixo “pré” indica que a Raiz está sendo tratada previamente. Se executarmos essa versão do algoritmo com a Árvore R da [Figura 8.13](#), os elementos serão impressos na seguinte ordem: 50, 28, 12, 39, 80, 96.

### (a) Percorso em "pré-ordem": Raiz, Esquerda, Direita

```
ImprimeTodos (parâmetro por referência R do tipo ABB) {  
{  
/* Imprime todos os elementos da Árvore R na seguinte sequência: primeiro imprime a  
Raiz, depois as Subárvore Esquerda e Direita. Percorso em "pré-Ordem": Raiz,  
Esquerda, Direita */  
  
Se (R != Null)  
Então {    Escreva(R→Info);    // imprime a informação da raiz  
            ImprimeTodos(R→Esq);    // imprime todos da Subárvore Esquerda  
            ImprimeTodos(R→Dir); }    // imprime todos da Subárvore Direita  
} // fim ImprimeTodos – Pré-Ordem
```

### ImprimeTodos (parâmetro por referência R do tipo ABB)

```
/* Imprime todos os elementos da Árvore R na seguinte sequência: primeiro imprime  
todos da Subárvore Esquerda, depois imprime a Raiz, depois imprime todos da  
Subárvore Direita. Os elementos serão impressos em ordem crescente. Percorso em "In  
Ordem": Raiz impressa entre os elementos das Subárvore Esquerda e Direita */
```

```
Se R != Null  
Então {    ImprimeTodos(R→Esq);    // imprime todos da Subárvore Esquerda  
            Escreva(R→Info);    // imprime a informação da raiz  
            ImprimeTodos(R→Dir); }    // imprime todos da Subárvore Direita  
} // fim ImprimeTodos - In Ordem
```

### (b) Percorso em "In Ordem": Esquerda, Raiz, Direita

**FIGURA 8.16** Algoritmo para imprimir os elementos da Árvore.

Na versão da [Figura 8.16b](#), após verificar que estamos tratando o caso 2 (Árvore não vazia), imprimimos recursivamente todos os elementos da Subárvore Esquerda, depois imprimimos o elemento da Raiz, e só então imprimimos todos os elementos da Subárvore Direita. Quando utilizamos essa sequência — primeiro todos da Subárvore Esquerda, depois o elemento da Raiz, depois todos os elementos da Subárvore Direita — estamos percorrendo a Árvore em “in ordem”. O prefixo “in” indica que a Raiz está sendo tratada entre as Subárvore. Se executarmos essa versão do algoritmo com a Árvore R da [Figura 8.13](#), os elementos serão impressos em ordem crescente: 12, 28, 39, 50, 80, 96.

Se desejarmos imprimir os elementos em ordem decrescente, basta imprimir primeiramente os elementos da Subárvore Direita, depois o elemento da Raiz e então os elementos da Subárvore Esquerda. Também é possível percorrer uma

Árvore em “pós-ordem”: elementos da Subárvore Esquerda, depois os elementos da Subárvore Direita e então o elemento da Raiz.

## Exercício 8.4 Soma dos elementos de uma Árvore

Implemente uma função recursiva que retorne a soma do valor de cada um dos Nós de uma Árvore Binária R. Considere elementos do tipo Inteiro.

```
Inteiro Soma (parâmetro por referência R do tipo ÁrvoreBinária);  
/* Retorna a soma do valor dos elementos de R. Elementos do tipo Inteiro */
```

## Exercício 8.5 Número de Nós com um único Filho

Implemente uma função recursiva que retorne o número de Nós de uma Árvore R que contém um único Filho. Em um Nó com um único Filho, um dos filhos deve ser nulo e o outro filho deve ser não nulo.

```
Inteiro NósComUmÚnicoFilho (parâmetro por referência R do tipo ÁrvoreBinária);  
/* Retorna o número de Nós de R com um único Filho */
```

## Exercício 8.6 Árvores são iguais?

Desenvolva um algoritmo recursivo para verificar se duas Árvores Binárias são iguais. Considere que duas Árvores vazias são iguais. Duas Árvores não vazias são iguais se apresentarem exatamente a mesma disposição dos elementos.

```
Boolean Iguais (parâmetros por referência R1, R2 do tipo ÁrvoreBinária);  
/* Verifica se R1 e R2 são iguais */
```

## Exercício 8.7 É Árvore Binária de Busca?

Implemente uma função recursiva que verifica se uma Árvore Binária R, passada como parâmetro, é uma Árvore Binária de Busca, conforme a definição da Figura 8.4. Sugestões: leia cuidadosamente a definição antes de iniciar o desenvolvimento. Use subprogramas auxiliares, se necessário.

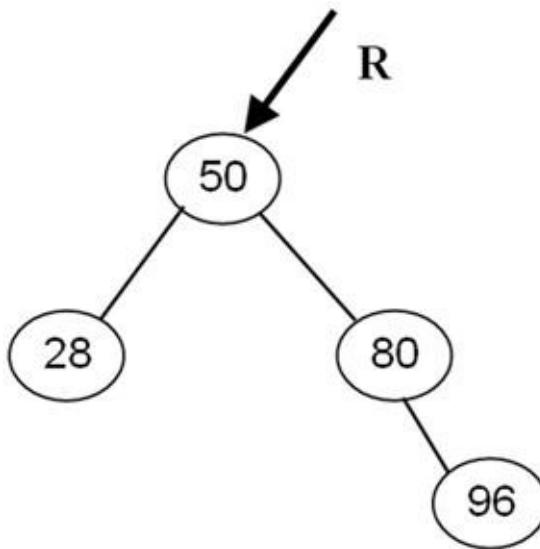
```

Boolean É_ABB (parâmetro por referência R do tipo ÁrvoreBinária);
/* verifica se a Árvore Binária R é uma Árvore Binária de Busca, segundo a definição da Figura 8.4. Elementos
do tipo Inteiro */

```

## 8.4 Árvores Binárias de Busca: inserir e eliminar elementos

Considere a Árvore Binária de Busca da [Figura 8.17](#). Se quisermos inserir o valor 37, em que lugar da Árvore esse novo elemento poderá ser colocado? Lembre-se de que temos um critério a respeitar: valores menores que a Informação armazenada na Raiz devem ficar na Subárvore Esquerda; valores maiores que a Informação armazenada na Raiz devem ficar na Subárvore Direita; esse critério deve ser respeitado para cada Nô da Árvore.



**FIGURA 8.17 ABB: onde inserir o valor 37?**

Uma possível estratégia é inserir todos os novos valores como Nós Terminais (sem Filhos). Inserindo novos valores em Nós Terminais, evitamos alterações na estrutura da Árvore, pois não precisaremos mudar de lugar nenhum dos valores já existentes.

Para não quebrar o critério de ordenação que define uma Árvore Binária de busca, existe um único lugar na Árvore onde podemos inserir um valor como Nô Terminal. Na ABB da [Figura 8.17](#), podemos inserir o valor 37 como um Nô Terminal à direita do Nô que armazena o valor 28. Se colocado em um Nô

Terminal em qualquer outro local, o valor 37 quebraria o critério que define uma ABB. Analogamente, se quisermos inserir o valor 55 como um Nó Terminal, ele teria que ser posicionado à esquerda do Nó que armazena o valor 80; se quisermos inserir o valor 12, ele teria que ser colocado à esquerda de 28, e assim por diante.

### Inserindo novos valores em uma ABB

- Inserir novos elementos como Nós Terminais (sem Filhos).
- Procurar o lugar certo, considerando o critério que define uma ABB e então inserir.

**FIGURA 8.18** ABB: estratégia geral para inserir novos valores.

## Algoritmo Insere

Se quisermos inserir um novo valor X em uma ABB de Raiz R, quatro possíveis situações podem acontecer — [Figura 8.19](#). No caso 1, estamos querendo inserir X em uma Árvore vazia. Se a Árvore estiver vazia, podemos inserir X na própria Raiz da Árvore. A ABB passará a ter um único Nó, que armazenará o valor X. Esse novo Nó passará a ser a Raiz da Árvore.

Caso	X	R	Conclusão
Caso 1: Árvore Vazia	37		Encontramos o local onde X deve ser inserido. Inserimos e encerramos o algoritmo.
Caso 2: $R \rightarrow \text{Info} = X$	37		X já está na árvore. Não inserimos (para não permitir elementos repetidos) e encerramos o algoritmo.
Caso 3: $X < R \rightarrow \text{Info}$	37		X deve ser inserido na Subárvore Esquerda de R. O algoritmo não acaba ainda. Fazemos uma chamada recursiva.
Caso 4: $X > R \rightarrow \text{Info}$	37		X deve ser inserido na Subárvore Direita de R. O algoritmo não acaba ainda. Fazemos uma chamada recursiva.

**FIGURA 8.19** ABB: casos do algoritmo Insere.

No caso 2, queremos inserir X na ABB R e logo na Raiz achamos um valor igual a X. Como não podemos ter dois valores iguais na ABB, simplesmente não inserimos X e encerramos o algoritmo.

No caso 3, sabemos que  $R \rightarrow \text{Info} > X$ , ou seja, X é menor que a Informação que está armazenada na Raiz. Não temos certeza quanto ao local onde poderemos inserir X. Aliás, nem temos certeza ainda se poderemos inserir X. Mas sabemos que, se X não estiver na Árvore, ele deverá ser inserido na Subárvore Esquerda de R. Analogamente, no caso 4, em que  $R \rightarrow \text{Info} < X$ , se X não estiver na Árvore, deverá ser inserido na Subárvore Direita de R.

### Exercício 8.8 Algoritmo recursivo: Insere em uma ABB

Desenvolva um algoritmo recursivo para inserir um novo valor X em uma Árvore Binária de Busca de Raiz R. X e R são passados como parâmetros. X

deve ser inserido como um Nô Terminal

```

Insere (parâmetro por referência R do tipo ABB, parâmetro X do tipo Inteiro, parâmetro por referência Ok do tipo Boolean);
/* Insere o valor X na ABB de Raiz R, como um Nó terminal, sem Filhos. Ok retorna Verdadeiro para o caso de X ter sido
inserido e Falso caso contrário. */

```

Tratando separadamente cada um dos quatro casos especificados na [Figura 8.19](#), chegamos ao algoritmo da [Figura 8.20](#). Nos casos 1 e 2, conseguimos dar uma resposta definitiva, e encerramos o algoritmo retornando os valores Falso e Verdadeiro, respectivamente. Nos casos 3 e 4, não é possível encerrar o algoritmo, e fazemos uma chamada recursiva.

```

Insere (parâmetro por referência R do tipo ABB, parâmetro X do tipo Inteiro, parâmetro
por referência Ok do tipo Boolean) {
/* Insere o valor X na ABB de Raiz R, como um Nó terminal, sem Filhos. Ok retorna
Verdadeiro para o caso de X ter sido inserido e Falso caso contrário. */

Variável P do tipo NodePtr;

Se (R == Null)
Então {   P = NewNode;    // Caso 1: Achou o lugar; insere e encerra o algoritmo
          P→Info = X;
          P→Dir = Null;
          P→Esq = Null;
          R = P;
          P = Null;
          Ok = Verdadeiro; }
Senão {   Se (X == R→Info)
            Então Ok = Falso; // Caso 2: X já está na árvore; não insere; acaba o algoritmo
            Senão {   Se (R→Info > X)
                        Então /* Caso 3: tenta inserir X na Subárvore Esquerda de R */
                        Insere (R→Esq, X , Ok)
                        Senão /* Caso 4: tenta inserir X na Subárvore Direita de R */
                        Insere(R→Dir, X, Ok);
                    } // fim senão
            } // fim senão
} // fim Insere ABB

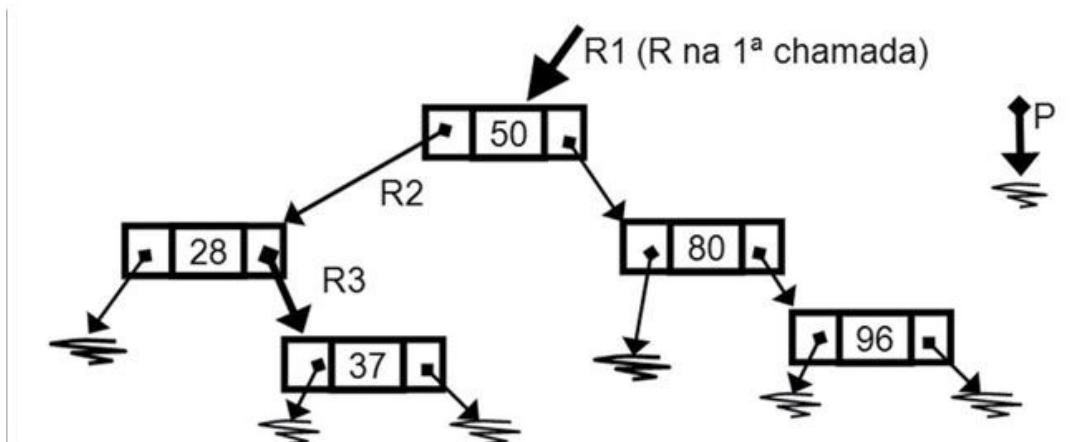
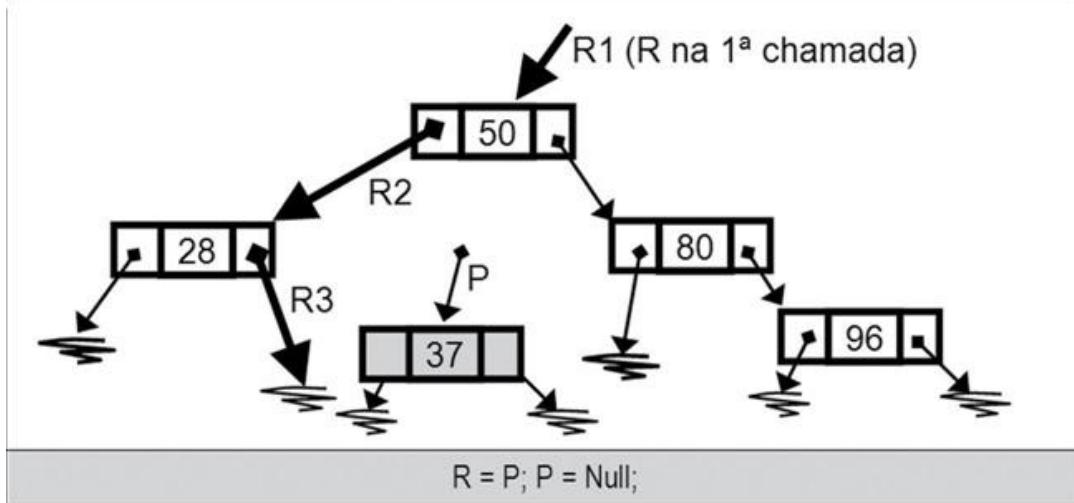
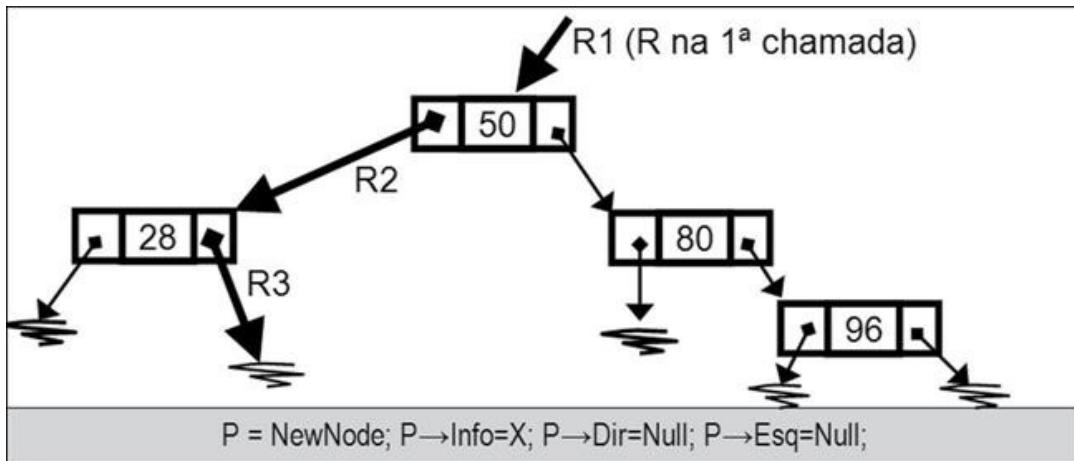
```

**FIGURA 8.20** Algoritmo conceitual — Insere em ABB.

## Execução do algoritmo Insere

Considere a inserção de  $X = 37$  na Árvore da [Figura 8.21a](#). R1, R2 e R3 fazem referência ao parâmetro R na primeira, segunda e terceira chamadas ao

procedimento Insere, respectivamente. Na primeira chamada identificamos o caso 3 (Figura 8.19), pois  $R1 \rightarrow \text{Info}$  é 50, maior que X, que tem valor 37. De acordo com o algoritmo, fazemos uma chamada recursiva para inserir X na Subárvore Esquerda de R1.



### **FIGURA 8.21 Execução de Insere para X = 37.**

Na segunda chamada, R2 aponta para o Nó cujo valor é 28. Nessa segunda chamada, identificamos o caso 4, pois  $R2 \rightarrow \text{Info} < X$ . De acordo com o algoritmo, fazemos uma chamada recursiva para inserir X na Subárvore Direita de R2.

Na terceira chamada, R3 está apontando para Null e, assim, identificamos o caso 1. Nesse momento inserimos X. De acordo com o algoritmo para tratar o caso1, aplicamos os comandos  $P = \text{NewNode}$ ;  $P \rightarrow \text{Info} = X$ ;  $P \rightarrow \text{Dir} = \text{Null}$ ;  $P \rightarrow \text{Esq} = \text{Null}$  e chegamos à situação da [Figura 8.21b](#).

A partir da situação da [Figura 8.21b](#), aplicamos o comando  $R = P$ . Como estamos na terceira chamada, R aqui faz referência a R3. Logo, R3 passa a apontar para onde aponta P. Note, no algoritmo da [Figura 8.20](#), que o parâmetro R é passado por referência. Logo, se atualizamos R3, será atualizado também o campo  $R2 \rightarrow \text{Dir}$ , pois desencadeamos a terceira chamada com o comando  $\text{Insere}(R2 \rightarrow \text{Dir}, X, \text{Ok})$ . No fundo, o nosso R3 é o  $R2 \rightarrow \text{Dir}$ .

O parâmetro Ok receberá o valor Verdadeiro na terceira chamada. Como também é um parâmetro passado por referência, Ok ficará verdadeiro também na segunda e na primeira chamadas. P é uma variável temporária, que simplesmente apontamos para Null, e encerramos o algoritmo.

## **Algoritmo Remove**

Na tentativa de remover um valor X de uma ABB de Raiz R, podemos nos deparar com quatro casos, detalhados na [Figura 8.22](#). Note que são as mesmas situações identificadas para o algoritmo Insere ([Figura 8.19](#)) e para o algoritmo que verifica se um valor X está na Árvore ([Figura 8.9](#)). As situações são as mesmas; as ações que devem ser desencadeadas em cada situação é que mudam.

Caso	X	R	Conclusão
Caso 1: Árvore Vazia	37		Se a Árvore está vazia, X não está na Árvore. Não removemos nenhum Nó e encerramos o algoritmo.
Caso 2: $R \rightarrow \text{Info} = X$	37		Encontramos X. Removemos X e fazemos os ajustes necessários na Árvore.
Caso 3: $X < R \rightarrow \text{Info}$	37		A procura por X deve continuar na Subárvore Esquerda de R. O algoritmo não acaba e fazemos uma chamada recursiva.
Caso 4: $X > R \rightarrow \text{Info}$	37		A procura por X deve continuar na Subárvore Direita de R. O algoritmo não acaba e fazemos uma chamada recursiva.

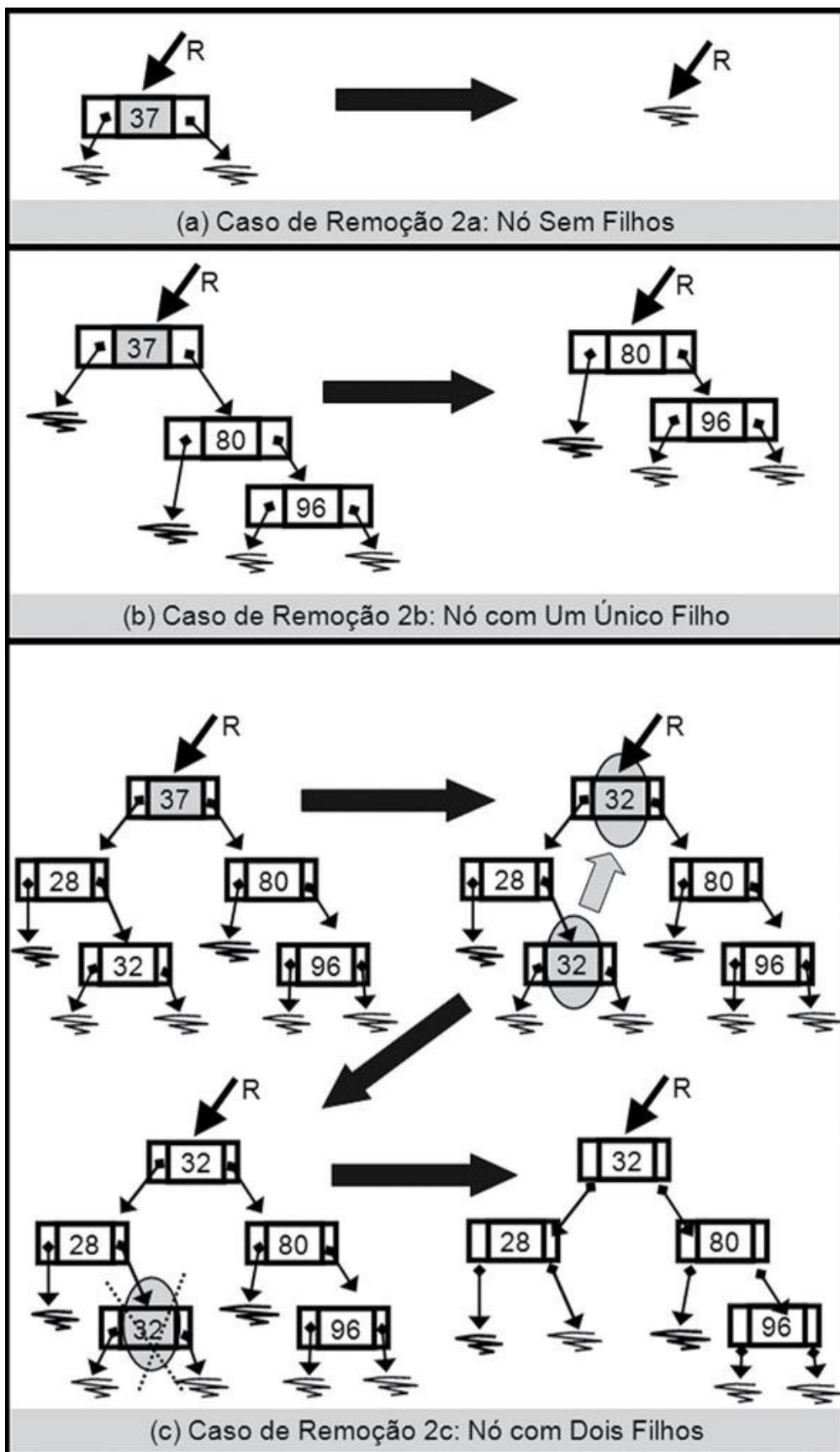
**FIGURA 8.22** ABB: casos do algoritmo Remove.

No caso 1, estamos querendo remover X de uma Árvore vazia. Se a árvore está vazia, X não está na Árvore, não removemos nenhum Nó e encerramos o algoritmo.

No caso 3 encontramos em  $R \rightarrow \text{Info}$  um valor maior que X, e, no caso 4, um valor menor que X. Nesses dois casos, ainda não temos certeza se encontraremos X na Árvore. Mas, pelos critérios que definem uma ABB, sabemos que no caso 3 a busca por X deve continuar na Subárvore Esquerda, e, no caso 4, na Subárvore Direita. Portanto, nos casos 3 e 4, continuamos o algoritmo fazendo uma chamada recursiva.

No caso 2, encontramos o valor que queremos remover. Precisamos agora remover efetivamente o Nó que contém o valor X e fazer os ajustes necessários na Árvore. Para exemplificar as possíveis situações de remoção e ajustes na Árvore, considere as três situações da Figura 8.23. Na Figura 8.23a, o Nó que contém o valor a ser removido, 37, não possui Filhos. Na Figura 8.23b, o Nó que contém o valor a ser removido possui um único Filho. E, na Figura 8.23c, o Nó que contém o valor a ser removido possui dois Filhos. Os ajustes na Árvore serão diferentes nesses três casos.





**FIGURA 8.23** Removendo Nós com zero, um ou dois Filhos.

Quando o Nó a ser removido não possui Filhos, eliminamos o Nó, e o ponteiro R passa a apontar para Null, como ilustrado na [Figura 8.23a](#). Quando o Nó a ser removido possui um único Filho, eliminamos o Nó, e o ponteiro R passa a apontar para o Filho não nulo. Veja, na [Figura 8.23b](#), que R passa a apontar para o Nó que contém o elemento de valor 80.

E se o Nó a ser removido da ABB tivesse dois Filhos, como na [Figura 8.23c](#)? Como podemos ajustar a Árvore nesse caso? Lembre-se de que é preciso respeitar o critério de ordenação que define uma Árvore Binária de Busca.

Com o objetivo de minimizar a movimentação dos demais Nós da Árvore, uma possível estratégia para ajustar a Árvore seria substituir o valor que queremos remover (37) por um outro valor da Árvore. Na Árvore inicial da [Figura 8.23c](#), temos dois valores que podemos colocar no lugar do valor 37 sem que o critério de ordenação de uma ABB seja quebrado. Você saberia dizer quais são esses dois valores?

Podemos substituir o valor 37 pelo valor 32, que é o maior valor da Subárvore Esquerda do Nó que armazena o valor 37. Sendo o maior valor da Subárvore Esquerda, se colocarmos 32 no lugar de 37 não haverá nenhum valor na Subárvore Esquerda maior que 32. E também não haverá nenhum valor na Subárvore Direita que seja menor que 32. Ou seja, o critério que define uma ABB não será quebrado. Também poderíamos substituir 37 pelo valor 80, que é o menor valor da Subárvore Direita do Nó que armazena o valor 37. Nesse caso, o critério de ordenação de uma ABB também não seria quebrado.

A [Figura 8.23c](#) mostra uma sequência de operações para ajuste da Árvore na remoção de um Nó com dois Filhos. Primeiramente substituímos o valor 37 pelo valor 32, que é o maior valor da Subárvore Esquerda de R. Em seguida, para a Árvore não ficar com dois valores 32, removemos o valor 32 que está na Subárvore Esquerda de R.

Note que o maior valor da Subárvore Esquerda de R nunca terá dois Filhos. Na verdade, o maior valor nunca terá o Filho Direito porque, se tivesse um Filho Direito, esse Filho Direito seria maior que ele.

A [Figura 8.24](#) resume as ações para ajuste de uma ABB para remoção de um Nó com zero, um ou dois Filhos.

(a) Caso de remoção 2a: Nós sem Filhos	Remover o Nós; apontar R para Null.
(b) Caso de remoção 2b: Nós com um único Filho	Remover o Nós e apontar R para o Filho não nulo.
(c) Caso de remoção 2c: Nós com dois Filhos	Encontrar o maior valor da Subárvore Esquerda de R; substituir R→Info por esse maior valor; remover o maior valor da Subárvore Esquerda de R.

**FIGURA 8.24** Ajuste da Árvore para remoção de Nós com zero, um ou dois Filhos.

## Exercício 8.9 Algoritmo Remove em uma ABB

Desenvolva um algoritmo recursivo para remover um valor X de uma Árvore Binária de Busca de Raiz R. X e R são passados como parâmetros. Como sugestão, trate os casos previstos nas [Figuras 8.22 a 8.24](#).

```
Remove (parâmetro por referência R do tipo ABB, parâmetro X do tipo Inteiro, parâmetro por referência Ok do tipo Boolean);
/* Remove o valor X da ABB de Raiz R. Ok retorna Verdadeiro para o caso de X ter sido encontrado e removido, e
Falso caso contrário. */
```

## Exercício 8.10 ABB: Cria, Vazia, Destroi

Para criar o Tipo Abstrato de Dados ABB, reúna as operações EstáNaÁrvore, Insere e Remove, implementadas nos [Exercícios 8.2, 8.8 e 8.9](#), e implemente as operações Cria, Vazia e Destroi. A operação Cria inicializa a Árvore como vazia; a operação Vazia verifica se uma ABB está ou não vazia; a operação Destroi remove todos os Nós da Árvore e a deixa vazia.

## 8.5 Por que uma Árvore Binária de Busca é boa?

Considere, por exemplo, que queremos desenvolver um sistema de votação por telefone, como em programas de televisão em que o telespectador vota e decide quem ganha o prêmio. Queremos que o sistema de votação tenha as seguintes características:

- Cada número de telefone pode votar uma única vez.

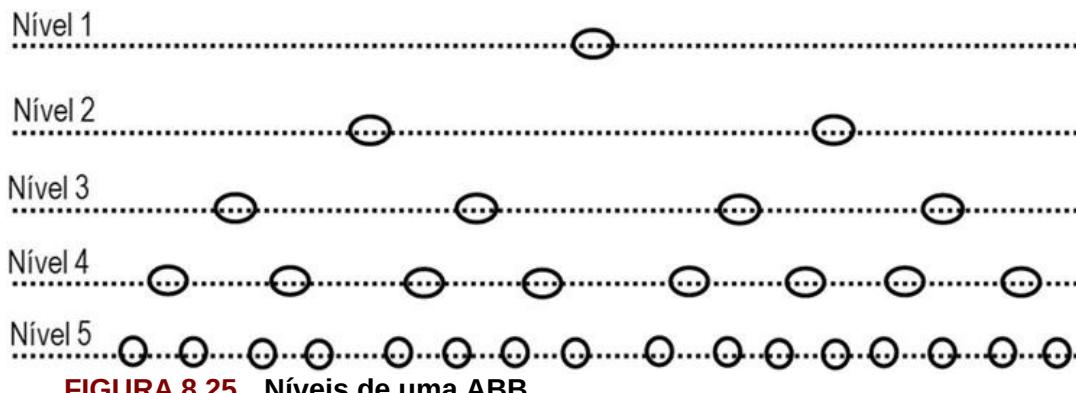
- Um sistema de informação deve armazenar todos os números que já ligaram.
- A cada nova ligação recebida, o sistema verifica se aquele número já votou; o voto é computado apenas se o número ainda não votou.
- O resultado parcial da votação deve estar sempre disponível, sendo atualizado automaticamente a cada voto.

Imagine que estejamos utilizando uma Árvore Binária de Busca para armazenar os números de telefone que já votaram. Cada número de telefone que já votou é armazenado em um Nô de ABB. Suponha que em determinado momento a ABB tenha um milhão de Nôs uniformemente distribuídos. Ou seja, um milhão de números de telefone armazenados. Surge uma nova ligação e é preciso saber se aquele número já votou, ou seja, se aquele número está na Árvore. Quantos nôs da ABB teriam que ser visitados, no máximo, para saber se o telefone que está ligando no momento está na Árvore?

### **Exercício 8.11 Quantos Nôs é preciso visitar?**

Analise o algoritmo do [Exercício 8.2](#) e calcule quantos Nôs é preciso visitar, no máximo, para verificar se determinado valor está armazenado em uma Árvore Binária de Busca com um milhão de Nôs uniformemente distribuídos.

Para responder à questão do [Exercício 8.11](#), primeiramente observe o diagrama da [Figura 8.25](#) e tente responder: quantos níveis teria uma Árvore com um milhão de Nôs uniformemente distribuídos?



No nível 1, é possível armazenar um único elemento, e teríamos que visitar, no máximo, um Nô (esse Nô do nível 1) para saber se um elemento X está na Árvore. Do nível 1 ao nível 2 é possível armazenar três elementos, e pelo algoritmo do [Exercício 8.2](#) precisaríamos visitar, no máximo, dois Nôs para

saber se um valor X está na Árvore. Isso porque, ao chegar na Raiz do nível 1, visitamos o Nó e, se o valor X que procuramos for menor que a informação armazenada na Raiz, continuaremos procurando no nível 2 apenas na Subárvore Esquerda. Se X for maior que a informação armazenada na Raiz, continuaremos procurando no nível 2 apenas na Subárvore Direita. Ou seja, no nível 2 procuramos em apenas uma das Subárvores; nunca em ambas.

Do nível 1 ao nível 3 cabem sete elementos. Quantos Nós temos que visitar, no máximo, para saber se X está na ABB? Três visitas, no máximo: uma visita para cada nível. Do nível 1 ao nível 4, quatro visitas, no máximo, e cabem 15 elementos. Do nível 1 ao nível 5, são cinco visitas, no máximo, e cabem 31 elementos.

Em uma ABB de N níveis, com Nós uniformemente distribuídos, precisamos visitar, no máximo, N Nós para saber se um valor X está ou não na Árvore. Quantos elementos cabem em uma ABB de N níveis?

Em uma ABB de N níveis cabem  $2^N - 1$  Nós. Observe, na [Figura 2.26](#), que uma ABB com 20 níveis pode abrigar, aproximadamente, um milhão de elementos. Em 30 níveis, a ABB pode abrigar aproximadamente um bilhão; em 40 níveis, aproximadamente um trilhão, e assim por diante. Uma análise semelhante pode ser consultada em [Drozdek \(2002, p. 227\)](#). Veja também Celes (2004, p. 194).

Se uma ABB uniformemente distribuída tiver 20 níveis, ela terá cerca de um milhão de elementos, e poderemos saber se um elemento X está nessa Árvore visitando, no máximo, 20 Nós.

Encontrar um valor X entre um milhão de elementos visitando, no máximo, 20 Nós é um bom desempenho, não é? Esse desempenho de uma Árvore Binária de Busca é especialmente significativo quando a quantidade de elementos armazenados é grande. Para encontrar um valor X em uma Lista Encadeada com um milhão de elementos poderia ser necessário visitar, no pior caso, um milhão de Nós.

Esse excelente desempenho só é possível se a Árvore Binária de Busca tiver seus Nós uniformemente distribuídos, como na [Figura 8.25](#). Os algoritmos de inserção e eliminação de Nós que elaboramos não garantem que a Árvore permaneça uniformemente distribuída. No próximo capítulo, ajustaremos os algoritmos para que o equilíbrio da Árvore e seu desempenho sejam garantidos.

Níveis na Árvore	Quantos Nós cabem na Árvore
1	1
2	3
3	7
4	15
5	31
<b>N</b>	$2^N - 1$
10	1.023
13	8.191
16	65.535
18	262.143
20	1 milhão (aproximadamente)
30	1 bilhão (aproximadamente)
40	1 trilhão (aproximadamente)

**FIGURA 8.26** Níveis e quantidade de elementos em uma ABB uniformemente distribuída.

### ABB proporciona agilidade em consultas

Uma ABB permite consultas rápidas, mesmo quando a quantidade de elementos é grande.

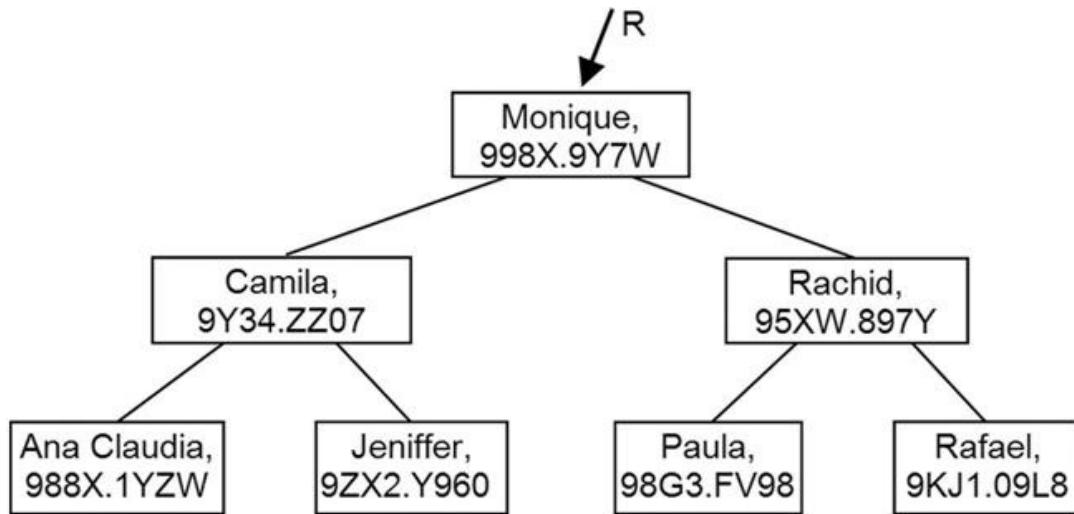
## 8.6 Aplicações de Árvores

Nos exemplos e algoritmos que estudamos até aqui, cada Nô da Árvore Binária de Busca abrigava apenas a informação utilizada como chave de pesquisa. Mas podemos inserir outras informações no Nô, além da chave de pesquisa.

Por exemplo, suponha que temos uma ABB cuja chave de pesquisa seja o nome da pessoa. Isso significa que cada Nô daquela ABB armazena o nome de uma pessoa e também que a ABB está ordenada em função dos nomes. Suponha ainda que os Nôs armazenem o número de telefone daquelas pessoas, como no diagrama da [Figura 2.27](#).

Um exemplo de consulta a essa estrutura seria: “Qual é o telefone da Ana Cláudia?”. Buscamos, então, o Nô da ABB que contém o valor ‘Ana Cláudia’

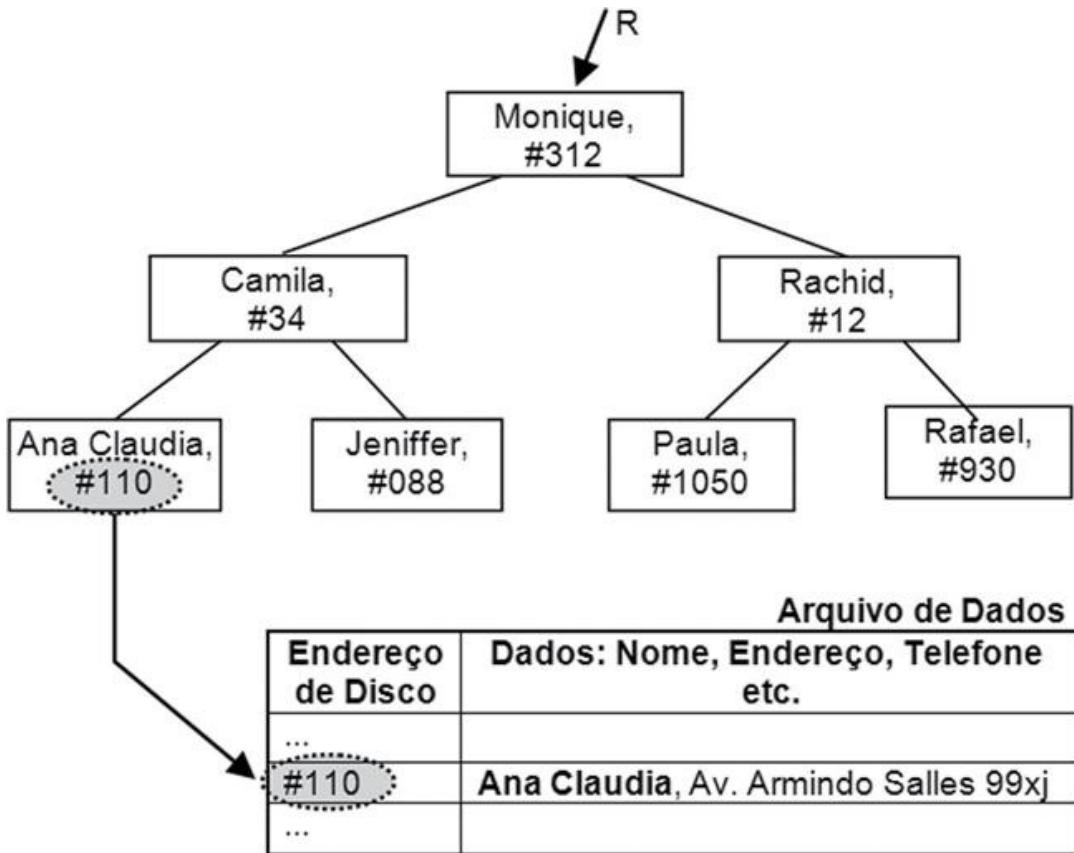
(algoritmo da [Figura 8.10](#)) e retornamos o telefone armazenado no Nó encontrado.



**FIGURA 8.27** ABB com chave de busca e outras informações armazenadas no Nô.

## Índices para arquivos

Em vez de armazenar dentro do próprio Nô um conjunto de informações associadas a uma chave de pesquisa, podemos armazenar no Nô a chave de pesquisa e o endereço de disco onde estão armazenadas as informações relativas àquela chave. Por exemplo, no diagrama da [Figura 8.28](#) buscamos o Nô que contém o nome *Ana Cláudia*. No mesmo Nô, encontramos o endereço de disco onde estão armazenados os dados da Ana Cláudia. Então vamos ao endereço de disco indicado e encontramos todas as informações disponíveis sobre a *Ana Cláudia*.



**FIGURA 8.28** ABB como índice para um arquivo.

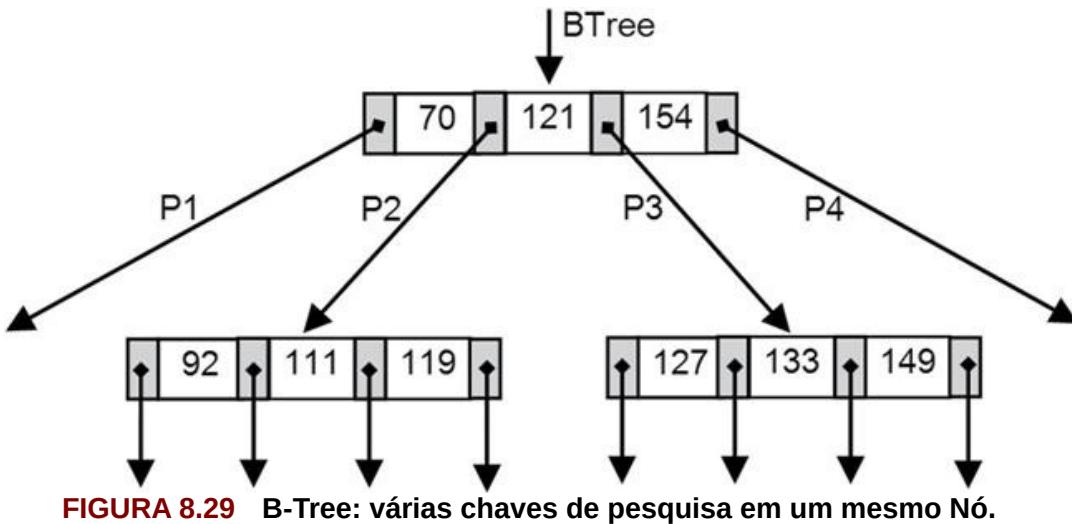
Esse é o conceito de Índice para Arquivos. A maioria dos sistemas de gerenciamento de banco de dados utiliza alguma variação de Árvore para implementar seus índices. Se, ao programar ou configurar um sistema de banco de dados, você indicar que determinado campo do arquivo será indexado, o sistema montará um índice semelhante ao da [Figura 8.28](#), tendo como chave de pesquisa as informações do campo em questão, por exemplo, o campo Nome da Pessoa.

## B-Trees

Uma generalização interessante das Árvores Binárias de Busca, direcionada ao armazenamento em disco, é a B-Tree e suas variações. Nas Árvores Binárias de Busca, temos uma única chave de pesquisa armazenada em cada Nô. Em uma B-Tree podemos ter várias chaves de pesquisa em um mesmo Nô.

Considere a B-Tree da [Figura 8.29](#) e tente imaginar como seria o algoritmo para localizar uma chave X em uma Árvore como essa. Suponha, por exemplo, que queremos encontrar a chave de valor 92. O Nô que está na Raiz não contém

a chave 92. Então, em que direção continuamos a busca?

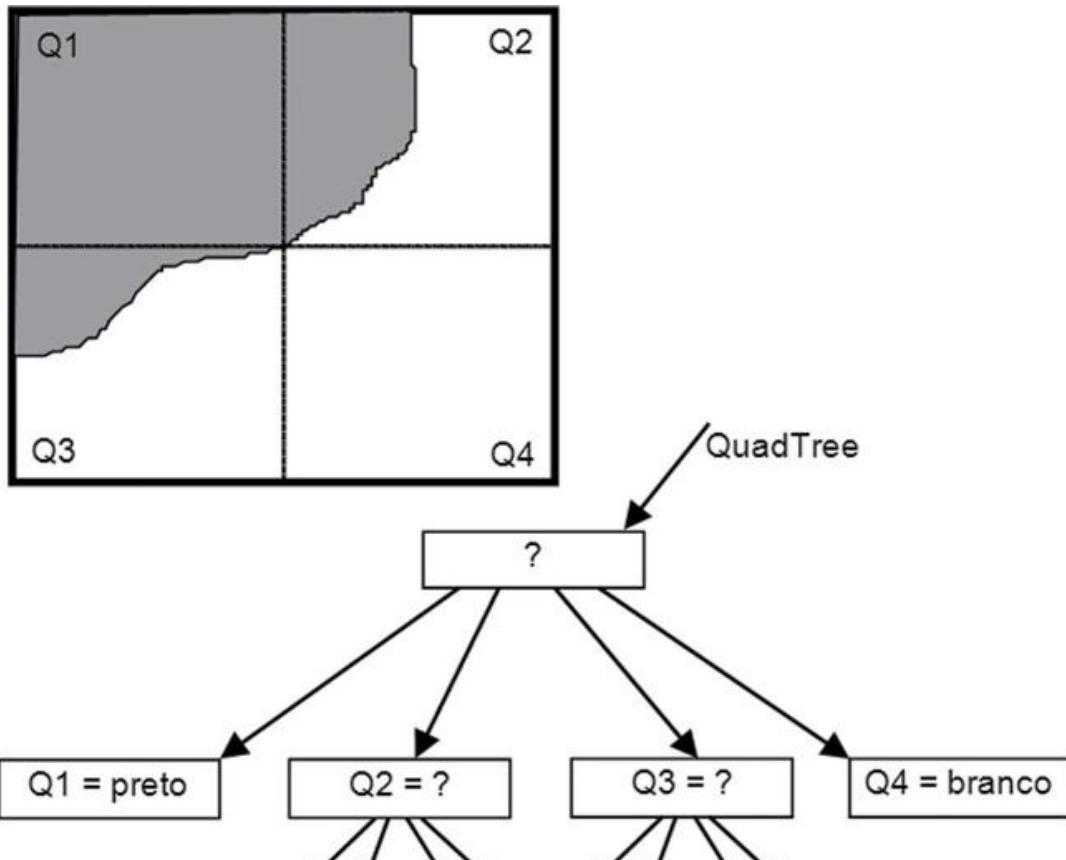


**FIGURA 8.29** B-Tree: várias chaves de pesquisa em um mesmo N o.

Continuamos a busca pelo valor 92 na Sub rvore apontada pelo ponteiro P2, pois 92 est a entre os valores 70 e 121. Se estiv essemos buscando um valor menor que 70, continuar  mos a busca na Sub rvore apontada pelo ponteiro P1; se o valor procurado fosse maior que 154, continuar  mos a busca na Sub rvore apontada por P4; se o valor procurado estivesse entre 121 e 154, continuar  mos a busca na Sub rvore apontada por P3.

## QuadTree

Quadtree    uma estrutura utilizada para armazenar informa es espaciais. A Figura 8.30 ilustra um tipo de QuadTree — *Region QuadTree* — utilizado para armazenar regi es.



**FIGURA 8.30** Region QuadTree.

No exemplo da [Figura 8.30](#), subdividimos uma região em quatro quadrantes: Q1, Q2, Q3 e Q4. O quadrante Q1 é homogêneo na cor preta. Armazenamos essa informação em um Nô, e não precisamos armazenar mais nada com relação ao quadrante Q1. Analogamente, o quadrante Q4 é homogêneo na cor branca. Armazenamos essa informação em um Nô, e não precisamos armazenar mais nada com relação a Q4. Mas os quadrantes Q2 e Q3 são heterogêneos. Então, subdividimos recursivamente Q2 e Q3, gerando para cada um deles mais quatro Nôs, e assim sucessivamente.

Além da *Region QuadTree*, temos a *Point QuadTree* ou *Point-Region QuadTree*, para armazenamento e busca de informações unidimensionais (ou seja, um *Ponto*, com coordenadas X, Y) em uma região.

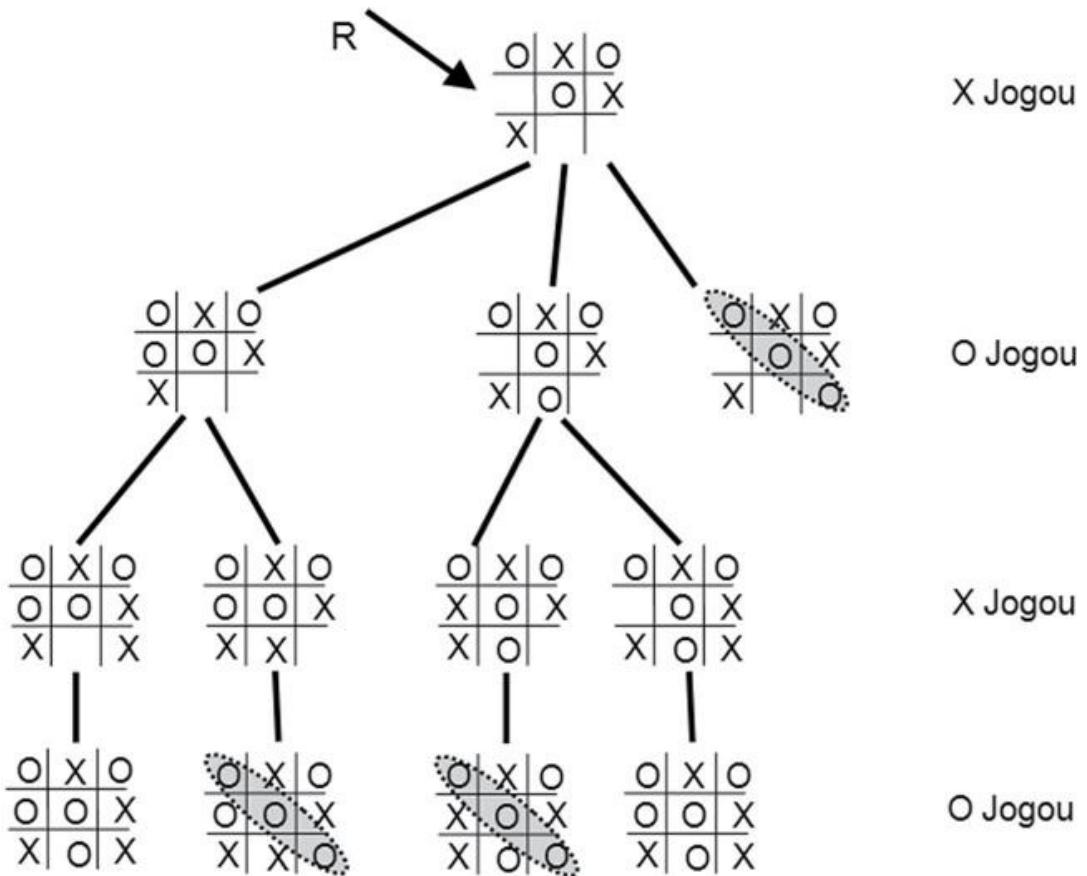
Existem diversas variações desse tipo de estrutura para armazenamento de informações espaciais em Árvores. Possíveis benefícios do uso de QuadTrees são a compressão de informações e, principalmente, a agilidade no processamento de operações sobre informações espaciais.

## 8.7 Avanço de projeto: o Desafio 4

Suponha que queremos desenvolver um jogo em que o computador joga contra um jogador humano. Quando for sua vez de jogar, o computador precisa *escolher* a melhor jogada. Árvores de Decisão para jogos — adaptações de estruturas do tipo Árvore — podem ser utilizadas para essa finalidade.

Árvores de Decisão para jogos servem primeiramente para simular as jogadas futuras e depois servem para ajudar o computador a escolher a melhor jogada. Simplificadamente, o computador procura nas Subárvores de determinada situação de jogo as jogadas em que o computador vence, as jogadas que levam a situações em que ainda é possível vencer, jogadas que levam a situações em que não há risco de derrota, e assim por diante.

[Langsam, Augenstein e Tenembaum \(1996, p. 321-327\)](#) propuseram um algoritmo que utiliza uma Árvore de Decisão como a da [Figura 8.31](#) e avaliam a chance de vitória em uma situação do *jogo da velha*. O algoritmo considera, entre outros fatores, a quantidade de diagonais, horizontais ou verticais ainda em aberto para uma jogada vencedora.



**FIGURA 8.31** Exemplo de Árvore de Decisão para jogos.

No exemplo da [Figura 8.31](#), na Raiz da Árvore temos uma situação de jogo em que 'X' (o jogador humano) acabou de jogar. A partir dessa situação inicial temos três possíveis jogadas para 'O' (o computador). Uma dessas três opções é uma jogada vencedora. Assim, com base no primeiro nível da Árvore abaixo da Raiz, o computador já consegue tomar uma decisão.

Mas podem existir situações em que uma jogada vencedora não é encontrada no primeiro nível da simulação. Então, a simulação pode avançar mais níveis e aplicar um algoritmo que avalia, para cada Subárvore, a chance de vitória; com base nessa avaliação, o computador escolhe a melhor jogada. Suponha que para cada situação de jogo temos um valor numérico — uma “nota” de 0 a 10 — indicando o quanto aquela situação é boa. Jogadas vencedoras têm nota 10. A nota da Subárvore será o valor máximo entre as notas de todas as situações de jogo presentes naquela Subárvore.

Essa mesma estratégia — simulação das jogadas futuras para escolha da melhor opção — pode ser aplicada a outros jogos mais complexos como, por exemplo, o xadrez.

## **Árvores de Decisão dando “inteligência” a um jogo**

A simulação de jogadas futuras é um exemplo de Árvore de Decisão utilizada para dar “inteligência” a um jogo. Árvores de Decisão também podem ser utilizadas para direcionar a lógica do jogo para a situação à esquerda da Raiz caso o usuário humano responda “sim” e para a situação à direita da Raiz caso o usuário humano responda “não”. Árvores de Decisão podem indicar ao jogo o que fazer caso o usuário humano escolha levar a princesa pela estrada estreita ou se escolher levá-la pela estrada larga, e assim por diante.

### **Exercício 8.12 Discutir aplicações de Árvores em jogos**

Identifique alguns jogos que podem ser implementados com o uso de uma estrutura do tipo Árvore. Identifique também outras aplicações, fora do mundo dos games. Sugestão de uso acadêmico: faça uma discussão em grupo. Ao final, cada grupo apresenta a todos os estudantes um novo projeto de Jogo, que ilustre bem a estrutura Árvore.

### **Exercício 8.13 Avançar o projeto do Desafio 4: defina regras, escolha um nome e inicie o desenvolvimento do seu jogo**

No Desafio 4 você deverá desenvolver uma adaptação do *jogo da velha* ou outra aplicação que utilize uma estrutura do tipo Árvore. Se for desenvolver um *jogo da velha*, dê personalidade própria ao seu jogo e escolha um nome que enfatize suas características marcantes. Se for criar um jogo totalmente novo, para cumprir os propósitos acadêmicos pretendidos no Desafio 4, mantenha a característica fundamental: um jogo que utilize uma ou mais Árvores. Sugestão para uso acadêmico: desenvolva o projeto em grupo. Tome as principais decisões em conjunto e divida o trabalho entre os componentes do grupo, cada qual ficando responsável por parte das atividades.

Inicie agora o desenvolvimento do seu jogo referente ao Desafio 4!

### **Agilidade e suporte a decisões**

Árvores são estruturas hierárquicas que proporcionam agilidade na busca e processamento de informações, mesmo quando a quantidade de elementos armazenados é grande. Árvores de Decisão podem ser utilizadas para dar inteligência a um jogo, auxiliando na escolha da melhor opção de jogada ou em outras decisões. É possível adaptar estruturas bem conhecidas, como as

Árvores Binárias de Busca, e propor Árvores diferenciadas que atendam a necessidades específicas de sua aplicação.

## Consulte nos Materiais Complementares

Vídeos sobre Árvores



Animações sobre Árvores



<http://www.elsevier.com.br/edcomjogos>

## Exercícios de fixação

**Exercício 8.14** Considere a Árvore da Figura 8.31. Indique:

- o número máximo de Subárvores;
- a Altura da Árvore;
- os Nós Terminais.

**Exercício 8.15** Qual seria a sequência de processamento dos elementos ao percorrermos a Árvore Binária de Busca da Figura 8.17 em:

- Pré-ordem (sequência: Raiz, Subárvore Esquerda, Subárvore Direita)?
- In ordem (sequência: Subárvore Esquerda, Raiz, Subárvore Direita)?
- Pós-ordem (sequência: Subárvore Esquerda, Subárvore Direita, Raiz)?

**Exercício 8.16** Compare o uso de uma Árvore Binária de Busca e uma Lista, para armazenamento de um conjunto de elementos. Em quais situações o uso da ABB seria vantajoso? Quais seriam as desvantagens?

**Exercício 8.17** Desenvolva um algoritmo para calcular a Altura (número de níveis) de uma Árvore Binária R.

**Exercício 8.18** Execute algum simulador de operações em uma Árvore Binária de Busca, como, por exemplo, o Tree Explorer (link 1). Execute operações para inserir e eliminar elementos.

## Soluções para alguns dos exercícios

## Exercício 8.4 Algoritmo recursivo: soma dos elementos de uma Árvore

```
Inteiro Soma (parâmetro por referência R do tipo ÁrvoreBinária) {  
/* Retorna a soma do valor de cada um dos elementos de R. Elementos do tipo Inteiro */  
  
Se (R == Null)           // Caso 1: árvore vazia  
Então Retorne 0 ;        // a soma dos elementos é zero  
Senão Retorne R→Info + Soma (R→Esq) + Soma (R→Dir);  
    // Caso 2: R é não nulo. Adiciona R→Info à soma dos resultados das chamadas recursivas para as subárvore  
} // fim Soma
```

## Exercício 8.5 Algoritmo recursivo: número de Nós com um único Filho

```
Inteiro NósCom1ÚnicoFilho (parâmetro por referência R do tipo ÁrvoreBinária) {  
/* Retorna o número de Nós de R com um único Filho */  
Se (R == Null)           // Caso 1: árvore vazia  
Então Retorne 0;          // número de nós com 1 único filho é zero  
Senão Se ((R→Dir==Null) E (R→Esq!=Null)) OU ((R→Dir!=Null) E (R→Esq==Null))  
    Então Retorne (1 + NósCom1ÚnicoFilho (R→Esq) + NósCom1ÚnicoFilho (R→Dir));  
        /* Caso 2: 1 dos filhos de R é não nulo e o outro é nulo. Pelo menos 1 dos Nós da Árvore possui 1 único Filho.  
        Adiciona 1 à soma do resultados das chamadas recursivas para as subárvore esquerda e direita */  
    Senão Retorne (0+NósCom1ÚnicoFilho (R→Esq)+NósCom1ÚnicoFilho (R→Dir));  
        /* Caso 3: o nó da raiz não tem 1 único Filho. Logo, adiciona 0 (zero) à soma dos resultados das chamadas  
        recursivas para as subárvore esquerda e direita */  
} // fim NósCom1ÚnicoFilho
```

## Exercício 8.6 Algoritmo recursivo: Árvores são iguais?

```
Boolean Iguais (parâmetros por referência R1, R2 do tipo ÁrvoreBinária) {  
/* Verifica se R1 e R2 são iguais. Duas Árvores vazias são iguais. Duas Árvores não vazias são  
iguais se armazenam valores iguais em suas raízes, se suas Subárvore Esquerdas são iguais e  
suas Subárvore Direitas também são iguais */  
Se ((R1 == Null) E (R2 == Null)) // Caso 1: Ambas vazias  
Então Retorne Verdadeiro;  
Senão // Caso 2: uma vazia e a outra não vazia  
    Se ((R1==Null) E (R2!=Null)) OU ((R1!=Null) E (R2==Null))  
        Então Retorne Falso;  
    Senão // nenhuma das árvores é vazia  
        Se ((R1→Info == R2→Info) E (Iguais(R1→Esq, R2→Esq)) E (Iguais(R1→Dir, R2→Dir)))  
            Então Retorne Verdadeiro; // são iguais!  
        Senão Retorne Falso; // não são iguais.  
} // fim Iguais
```

## Exercício 8.7 Algoritmo recursivo: é Árvore Binária de Busca?

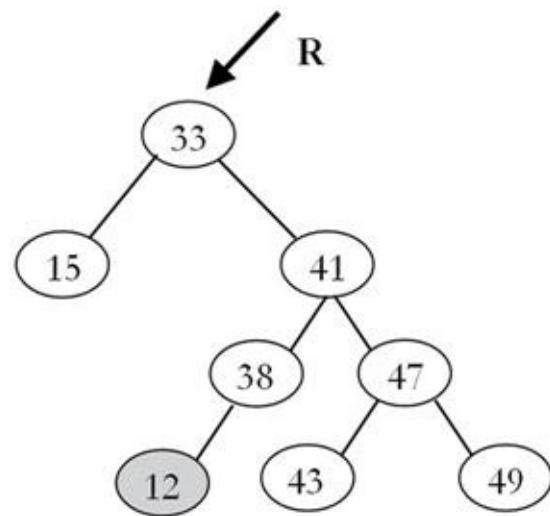
```
Boolean É_ABB (parâmetro por referência R do tipo ÁrvoreBinária) {
    // verifica se R é ou não é uma Árvore Binária de Busca, segundo a definição da Figura 8.4.
    Elementos do tipo Inteiro.
    Se (R == Null) // Caso 1: árvore vazia
        Então Retorne Verdadeiro;
    Senão /* Caso 2: árvore não nula. Tenta falsificar verificando se "TemAlguemMaior", ou seja, se
        há algum Nô de valor
            menor que R→Info na subárvore esquerda de R, ou se "TemAlguemMenor", ou seja, se
            há algum Nô de valor
                maior que R→Info na subárvore direita */
            Se (TemAlguémMaior(R→Esq, R→Info) OU TemAlguémMenor(R→Dir, R→Info))
                Então Retorne Falso;
            Senão /* pelo Nô atual está tudo ok, mas é preciso verificar se as subárvores esquerda
                e direita também são
                    ABBs. Este é o critério 3 da definição da Figura 8.4 */
                Se (É_ABB(R→Esq) E (É_ABB(R→Dir)) // subárvore esq é ABB E subárvore dir
                    Então Retorne Verdadeiro;
                Senão Retorne Falso;
            } fim É_ABB

    Boolean TemAlguemMaior (parâmetro por referência R do tipo ÁrvoreBinária, parâmetro X do tipo
    Inteiro) {
        Se (R == Null)
            Então Retorne Falso; // não tem nenhum valor maior que X na árvore R
        Senão Se (R→Info > X)
            Então Retorne Verdadeiro;
            Senão Retorne (TemAlguemMaior( R→Esq, X) OU TemAlguemMaior( R→Dir, X));
        } fim TemAlguémMaior

    Boolean TemAlguemMenor (parâmetro por referência R do tipo ÁrvoreBinária, parâmetro X do tipo
    Inteiro) {
        Se (R == Null)
            Então Retorne Falso; // não tem nenhum valor menor que X na árvore R
        Senão Se (R→Info < X)
            Então Retorne Verdadeiro;
            Senão Retorne (TemAlguemMenor( R→Esq, ) OU TemAlguemMenor( R→Dir, X));
        } // fim TemAlguémMenor
```

**Comentário:** Não basta verificar se a informação do filho direito de R é maior que a informação de R e verificar se a informação do filho esquerdo de R é menor que a informação de R. É preciso verificar a informação de *cada Nô* das subárvores esquerda e direita. Veja na Figura uma situação em que é preciso

verificar *cada* Nó da subárvore. O Nó que contém o valor 12 faz com que a Árvore não seja uma Árvore Binária de Busca \*/



### Exercício 8.9 Algoritmo Remove em uma ABB

```

Remove (parâmetro por referência R tipo ABB, parâmetro X do tipo Inteiro, parâmetro por
referência Ok tipo Boolean) {
    // Remove X da ABB R. Ok retorna Verdadeiro para o caso de X ter sido encontrado e removido, e
    // Falso caso contrário
    Variável Aux do tipo NodePtr;
    Se (R == Null)
        Então Ok = Falso; // Caso 1: Árvore vazia: não remove e encerra o algoritmo.
    Senão Se (R→Info > X)
        Então Remove (R→Esq, X , Ok); // Caso 3: remove X da Subárvore Esq de R
        Senão Se (R→Info < X)
            Então Remove(R→Dir, X, Erro); // Caso 4: remove X da Subárv Dir de R
            Senão
        /* Caso 2: Encontrou X - Remove e Ajusta a Árvore. Existem três casos: Nó com 0, 1 ou 2 Filhos –
        Figuras 8.23 e 8.24 */
        {
        Aux = R;
        Ok = Verdadeiro;

        Se (R→Esq = Null E R→Dir = Null) // Caso 2a: Zero Filhos
        Então { DeleteNode(Aux); R = Null; } // fim Caso 2a
        Senão Se (R→Dir != Null E Esq( R ) != Null) // Caso 2c: 2 Filhos
            Então {
                /* Acha o Nó que contém o Maior Elemento da Subárvore Esquerda
                de R. O maior é o elemento mais à direita da Subárvore. Ele nunca terá
                o Filho Direito. */
                Aux = R→Esq;
                Enquanto (Aux→Dir != Null) Faça Aux = Aux→Dir;

                /* Substitui o valor de R→Info - que é o elemento que estarmos querendo
                eliminar - pelo valor do Maior da Subárvore Esquerda de R. A Árvore
                ficará com 2 elementos com o mesmo valor. */
                R→Info = Aux→Info; // Aux aponta para o Nó que contém o Maior

                /* Remove o valor repetido da Subárvore Esquerda de R, através de
                uma chamada recursiva. Atenção aos parâmetros. Não estamos mais
                removendo X, mas R→Info, que está repetido. Não estamos mais
                removendo de R e sim de R→Esq */
                Remove(R→Esq, R→Info, Ok);
            } // fim Caso 2c

            Senão {
                // Caso 2b: 1 Único Filho
                Se (R→Esq == Null)
                    Então R = R→Dir; // "puxa" o Filho Direito; Filho esquerdo é nulo
                Senão R = R→Esq; // "puxa" o Filho Esquerdo; Filho direito é nulo
                DeleteNode (Aux);
            } // fim Caso 2b
        } // fim remove
}

```

## Exercício 8.10 ABB Cria, Vazia, Destrói

Sugestão: na operação Destrói, destrua (recursivamente) a Subárvore Esquerda,

destrua (recursivamente) a Subárvore Direita e então destrua o Nó da Raiz. Ao final, aponte a Raiz para Null.

## Exercício 8.14

Altura: 4; Número Máximo de Subárvores: 3; Nós Terminais (ou Nós Folha): todos os Nós do nível 4, e a Jogada Vencedora do nível 2.

## Exercício 8.15

- Pré-ordem: 50, 28, 80, 96;
- In ordem: 28, 50, 80, 96;
- Pós-ordem: 28, 96, 80, 50.

## Exercício 8.17 Algoritmo recursivo: altura de uma Árvore

```
Inteiro Altura (parâmetro por referência R do tipo ÁrvoreBinária) {  
/* Retorna a altura (número de níveis) de uma Árvore R. Elementos do tipo Inteiro. */  
Variáveis AlturaEsq, AlturaDir do tipo Inteiro;  
Se (R == Null)           // Caso 1: árvore vazia  
Então Retorne 0;          // a altura da Árvore é zero  
Senão { /* Caso 2: R é não nulo. Logo, a altura da árvore é pelo menos 1. Então, a altura da árvore  
será 1 + a altura da  
    AlturaEsq = Altura (R→Esq);           // altura da subárvore esquerda de R  
    AlturaDir = Altura(R→Dir);           // altura da subárvore direita de R  
    Se AlturaEsq > AlturaDir  
        Então Retorne 1 + AlturaEsq;  
    Senão Retorne 1 + AlturaDir ; }  
} // fim Altura subárvore esquerda ou 1 + a altura da subárvore direita - dependendo de qual for maior */
```

## Links

1. Rocha, V.; Vervloet, M.; Franco, A.; Andrade, C. A. *Tree Explorer*. EDGames, 2013. Disponível em: <http://edgames.dc.ufscar.br> (acesso em: setembro de 2013).

## Referências e leitura adicional

1. Celes W, Cerqueira R, Rangel JL. *Introdução à estrutura de dados*. Rio de Janeiro: Elsevier; 2004.

2. Drozdek A. *Estruturas de dados e algoritmos em C++*. São Paulo: Thomson; 2002.
3. Langsam Y, Augenstein MJ, Tenenbaum AM. *Data Structures Using C and C++*. Upper Saddle River NJ USA: Prentice Hall; 1996.
4. Pereira SL. *Estruturas de dados fundamentais: conceitos e aplicações*. São Paulo: Érica; 1996.

---

## CAPÍTULO 9

# Árvores Balanceadas

---

### Seus objetivos neste capítulo

- Entender o conceito de Balanceamento e sua importância para a eficiência das Árvores Binárias de Busca.
- Desenvolver habilidade para elaborar algoritmos sobre Árvores Binárias de Busca Balanceadas e para adaptar a lógica do Balanceamento a novas situações, se necessário.

### 9.1 Conceito de Balanceamento

O excelente desempenho de uma Árvore Binária de Busca só é atingido se os Nós estiverem uniformemente distribuídos ao longo de toda a Árvore. A Árvore não pode crescer para apenas um dos lados, esquerdo ou direito. É preciso crescer equilibradamente, para ambos os lados: para cada Nós, as alturas das Subárvores Esquerda e Direita precisam ser iguais ou, pelo menos, parecidas.

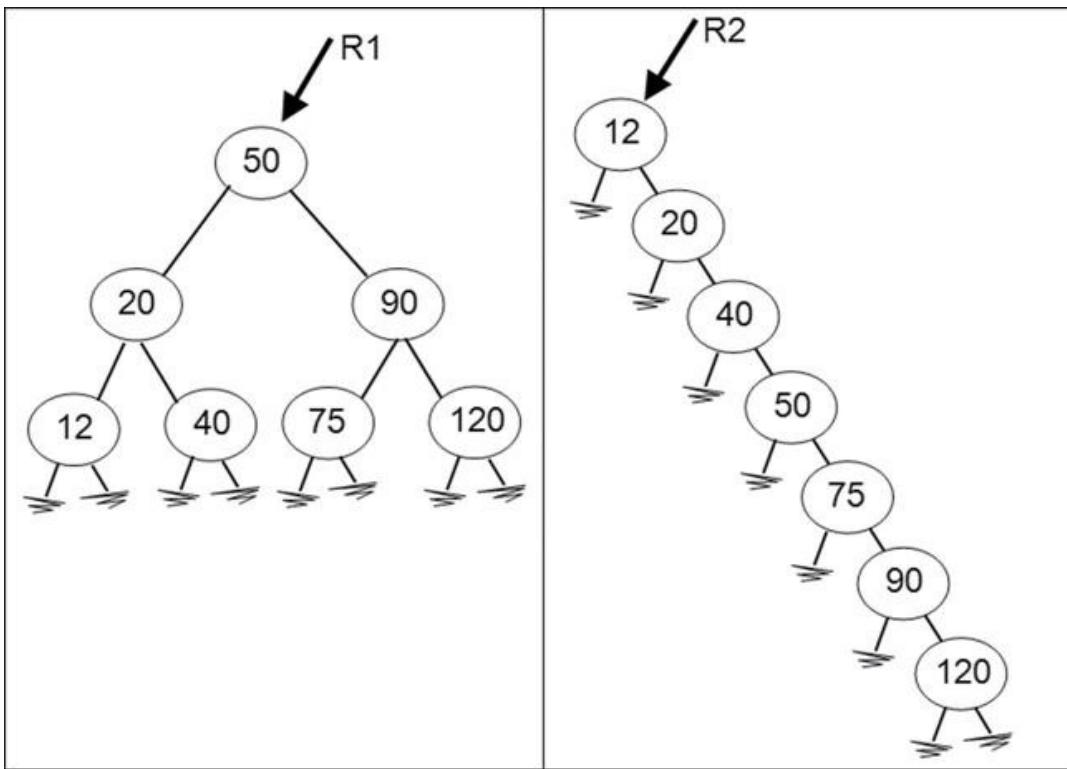
Os algoritmos para inserção e eliminação de Nós em uma ABB que elaboramos no [Capítulo 8](#) não garantem que haja equilíbrio entre as alturas das Subárvores. Assim, o equilíbrio e o desempenho podem se degradar a cada operação.

### Exercício 9.1 Inserir sequência de valores

Aplique o algoritmo Insere, desenvolvido no [Capítulo 8](#), e insira em uma ABB de Raiz R1, inicialmente vazia, os seguintes valores, na sequência A: 50, 20, 40, 12, 90, 75, 120. Em seguida, em uma segunda Árvore de Raiz R2, também inicialmente vazia, insira os mesmos valores, mas na sequência B: 12, 20, 40, 50, 75, 90, 120.

A [Figura 9.1](#) mostra como ficariam as Árvores R1 e R2, dada a utilização do algoritmo Insere que desenvolvemos no [Capítulo 8](#), para a inserção das sequências de valores A e B do [Exercício 9.1](#). Foram inseridos os mesmos valores em R1 e em R2, mas em R1 os valores foram inseridos na sequência A e

em R2 foram inseridos na sequência B.



**FIGURA 9.1** Valores inseridos em sequências diferentes.

As Árvores R1 e R2 da [Figura 9.1](#) ilustram uma deficiência dos algoritmos de inserção e eliminação em uma ABB que elaboramos no [Capítulo 8](#): dependendo da ordem de inserção dos elementos, as Árvores podem ficar equilibradas, como R1 na [Figura 9.1a](#), ou desequilibradas, como R2 na [Figura 9.1b](#). Por Árvore “equilibrada” entendemos uma Árvore em que os tamanhos de suas Subárvores Esquerda e Direita são iguais ou compatíveis.

Refletindo sobre a eficiência das Árvores Binárias de Busca no [Capítulo 8](#), chegamos à conclusão de que, se uma ABB *uniformemente distribuída* tiver 20 níveis, ela terá cerca de um milhão de elementos, e poderemos saber se um elemento X está ou não nessa Árvore visitando, no máximo, 20 Nós. A conclusão traz como ressalva que esse desempenho é obtido em “uma ABB uniformemente distribuída”, como R1, na [Figura 9.1a](#).

A Árvore R2, na [Figura 9.1b](#), não é uniformemente distribuída. R2 é uma árvore desequilibrada ou ainda desbalanceada, pois a Subárvore Direita é muito maior que a Subárvore Esquerda. Logo, devido a esse desequilíbrio, em uma Árvore como R2 não teríamos a mesma eficiência que obtemos em Árvores

Balanceadas, como R1. Se uma Árvore desbalanceada como R2 tiver um milhão de elementos, no pior caso teremos que visitar um milhão de Nós para ter certeza de que um valor X está na Árvore.

Visitar, no máximo, 20 Nós ou, no máximo, um milhão de Nós: que diferença de desempenho!

## Árvores Binárias de Busca Balanceadas

Uma Árvore Binária de Busca está Balanceada se, para cada NÓ, as alturas de suas Subárvores diferem de, no máximo, 1. Se chamarmos a altura da Subárvore Direita de Árvore R de Hd, e a altura da Subárvore Esquerda de R de He, podemos dizer que R está balanceada se  $He = Hd$  ou se  $Hd = He + 1$ , ou ainda se  $Hd = He - 1$ . Esses valores precisam ser válidos para cada NÓ da Árvore R. Em qualquer outra circunstância, a Árvore não estará Balanceada.

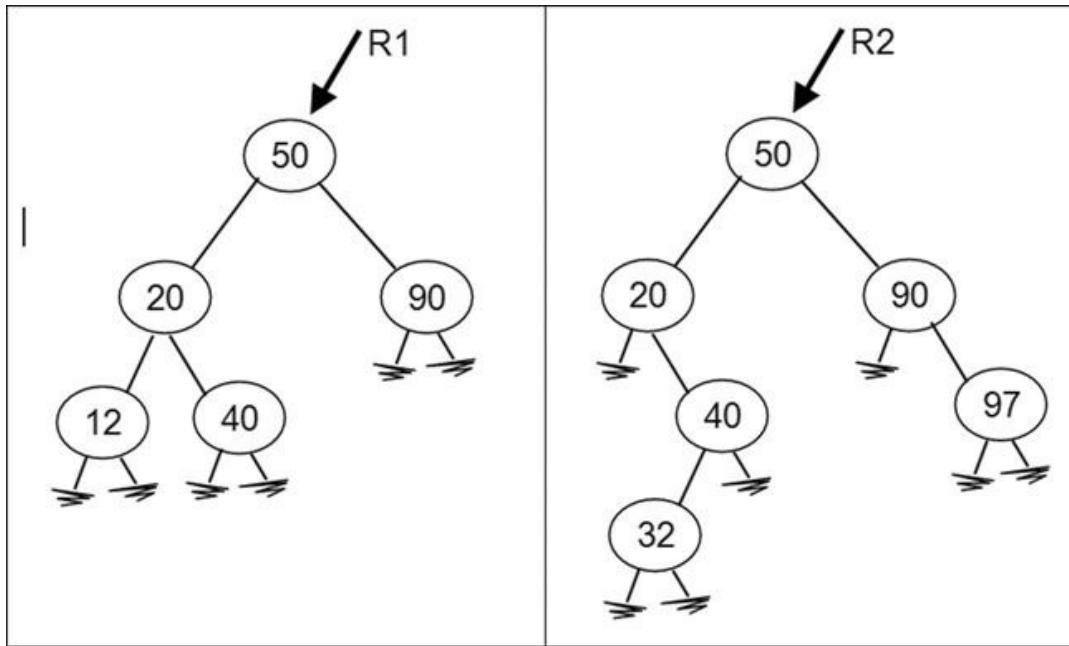
### Exercício 9.2 As Árvores estão Balanceadas?

Analise as Árvores da Figura 9.3 e verifique se estão Balanceadas, de acordo com a definição da Figura 9.2.

#### Definição: Árvore Binária de Busca Balanceada (ABBB)

Uma Árvore Binária de Busca é dita Balanceada se e somente se, para cada NÓ da Árvore, as alturas de suas Subárvores diferem de, no máximo, 1.

**FIGURA 9.2** Definição de Árvore Binária de Busca Balanceada.



**FIGURA 9.3** Estão Balanceadas?

A Árvore R1 está Balanceada, pois para cada Nô de Árvore as alturas de suas Subárvore diferem de, no máximo, 1. Mas a Árvore R2 não está Balanceada. Você saberia explicar por quê? Você saberia indicar em qual Nô de R2 o desbalanceamento fica evidente?

Ao analisarmos, em R2, o Nô que contém o valor 50, vemos que a altura de sua Subárvore Direita Hd é 2, e a altura da Subárvore Esquerda He é 3; assim, o critério de balanceamento não é quebrado. Hd e Hd diferem, no máximo, em 1 em todos os Nôs de R2, exceto o Nô que contém o valor 20. Para esse Nô, Hd = 2 e He = 0. A diferença entre Hd e He para esse Nô difere em 2 e, assim, o critério de balanceamento é quebrado. Se o critério é quebrado em um dos Nôs da Árvore, então a Árvore como um todo não está balanceada.

## Como manter uma Árvore Balanceada?

Se utilizarmos os algoritmos para inserção e eliminação que elaboramos no [Capítulo 8](#), a Árvore Binária de Busca perderá gradativamente seu equilíbrio e seu desempenho. Em algum momento poderemos promover uma reorganização geral de todos os valores que fazem parte da Árvore, para torná-la novamente Balanceada. O bom desempenho seria mantido apenas por algum tempo: com a execução de operações para inserir e eliminar valores, o desempenho iria diminuir gradativamente, até que uma nova reorganização geral dos elementos seja executada.

Uma estratégia melhor seria não permitir o desbalanceamento da Árvore, em nenhum momento. Dessa forma, o bom desempenho seria garantido durante todo o tempo, e não apenas logo após uma reorganização geral. Para manter a Árvore constantemente Balanceada, precisamos alterar os algoritmos de inserção e eliminação. Os novos algoritmos deverão: monitorar o Balanceamento da Árvore e, quando necessário, desencadear ações de Rebalanceamento.

#### Estratégia para manter uma ABB Balanceada

Alterar os algoritmos de inserção e eliminação de modo a:

- a) Monitorar o Balanceamento da Árvore.
- b) Desencadear ações de rebalanceamento, sempre que necessário.

**FIGURA 9.4** Estratégia para manter uma ABB Balanceada.

Essa estratégia de manter a Árvore sempre balanceada foi proposta por [Adelson-Velskii e Landis \(1962\)](#), dando origem ao nome “Árvore AVL” (AVL são as iniciais de Adelson-Velskii e Landis). Posteriormente, as Árvores AVL foram tratadas em livros como os de [Langsam, Augenstein e Tenembbaum \(1996\)](#), e [Drozdek \(2002\)](#), e em materiais didáticos como os de Devadas et al. (2009), Leser (2011), Buricea (links 1, 2 e 3) e Camargo.

Para monitorar o Balanceamento, vamos manter, para cada Nô, o Fator de Balanceamento (nome adotado por [Drozdek, 2002](#)), ou Bal, definido como o valor da Altura da Subárvore Direita (Hd) menos o valor da Altura da Subárvore Esquerda (He). Pela definição de Árvores Balanceadas da [Figura 9.2](#), o Fator de Balanceamento poderá assumir os valores: -1, 0 e 1. Com outros valores, a Árvore estará desbalanceada.

#### Fator de Balanceamento: $\text{Bal} = \text{Hd} - \text{He}$

Altura da Subárvore Direita menos a altura da Subárvore Esquerda.

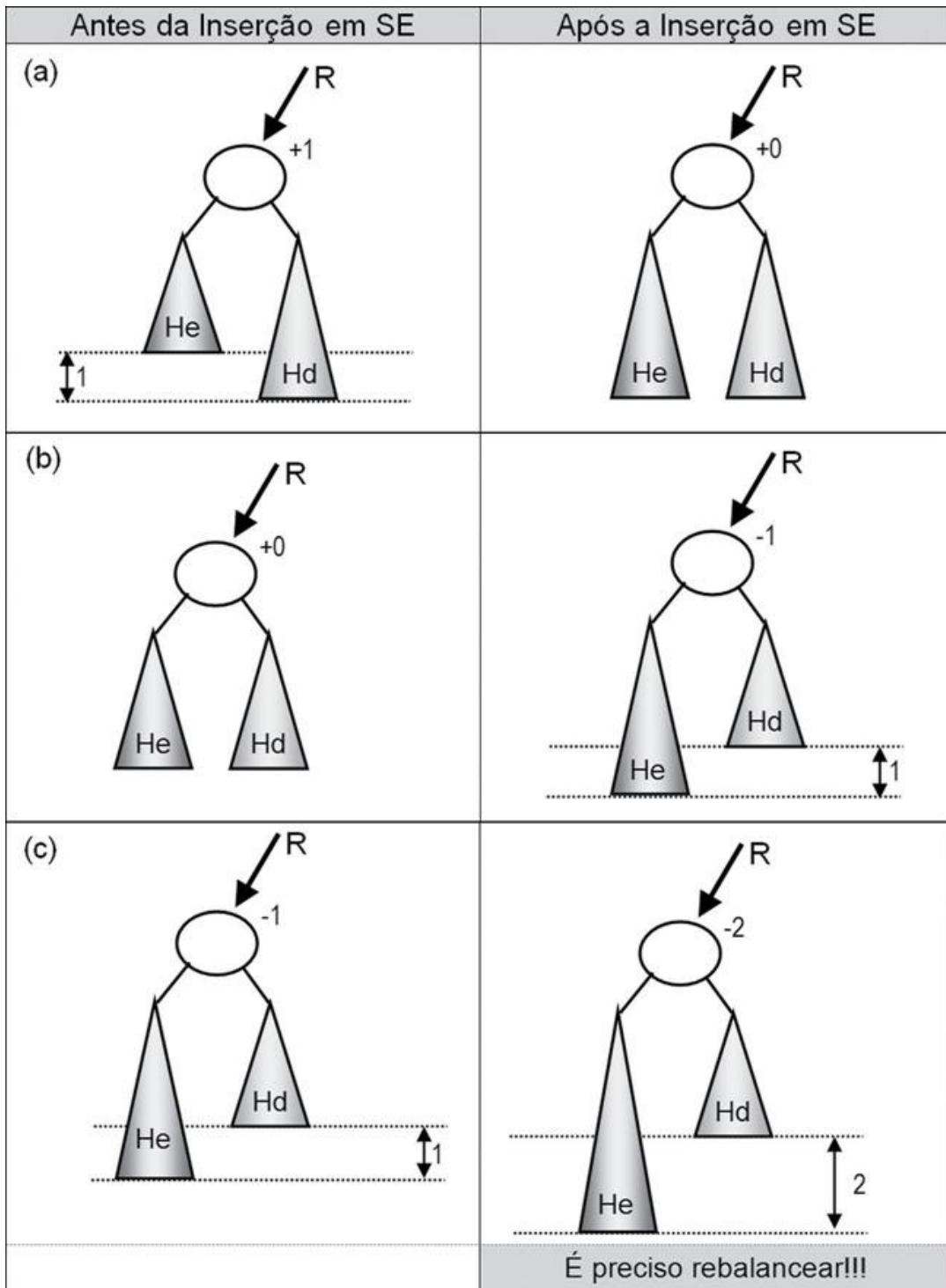
**FIGURA 9.5** Definição: Fator de Balanceamento.

### Exercício 9.3 Calcular o Fator de Balanceamento

Calcule o Fator de Balanceamento para cada Nô das Árvores R1 e R2 da [Figura 9.3](#).

## 9.2 Inserir elementos em uma ABB Balanceada

Seja uma Árvore R com Subárvore Esquerda SE e Subárvore Direita SD, com alturas  $He$  e  $Hd$ , respectivamente. Se um novo elemento é inserido em SE, causando aumento na altura  $He$ , três casos podem ocorrer, conforme ilustra a Figura 9.6.



**FIGURA 9.6** Monitorando o Balanceamento na inserção, com aumento da Altura He.

Se, antes da inserção, He era menor que Hd em 1, o Fator de Balanceamento de R era +1. Após a inserção, com o aumento da altura He, o Fator de Balanceamento passará a ser igual a 0, pois He passará a ser igual a Hd. O critério de

balanceamento não será quebrado nesse caso ([Figura 9.6a](#)).

Se, antes da inserção, as alturas das Subárvores Esquerda e Direita eram iguais, o Fator de Balanceamento era 0. Após a inserção, com aumento da altura  $H_e$ , o Fator de Balanceamento de R passará a ser -1, pois  $H_d$  será menor que  $H_e$  em 1, mas o critério de balanceamento não será quebrado ([Figura 9.6b](#)).

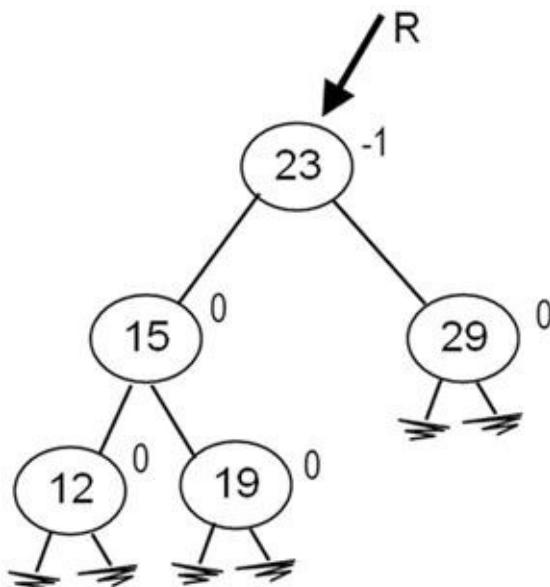
Se, antes da inserção,  $H_e$  era maior que  $H_d$  em 1, o balanceamento era -1. Após a inserção, com aumento da altura  $H_e$ ,  $H_e$  será maior que  $H_d$  em 2. O critério de balanceamento, nesse caso, é violado, ou seja, a inserção de um elemento está causando desbalanceamento, e a Árvore precisará ser ajustada ([Figura 9.6c](#)).

Observe ainda, nas situações da [Figura 9.6](#), a altura total da Árvore. Na [Figura 9.6a](#), antes da inserção de um novo valor em  $H_e$ , a altura da Árvore R era  $H_d + 1$  (altura da Subárvore Direita mais um nível, em função do Nô apontado por R). Após a inserção do novo valor, a altura total da Árvore continua sendo  $H_d + 1$ . Na [Figura 9.6b](#), na situação inicial a altura é  $H_d + 1$  e na situação final a altura é  $H_d + 2$ . Na [Figura 9.6c](#), a altura inicial era  $H_d + 2$  e a altura final é  $H_d + 3$ . Nos casos 9.6b e 9.6c, a inserção de um novo valor em SE fez crescer  $H_e$  e também a Árvore como um todo. Na situação da [Figura 9.6a](#), a inserção de um novo valor em SE fez crescer  $H_e$ , mas a altura da Árvore como um todo permaneceu a mesma.

## Exercício 9.4 Causaria desbalanceamento?

Considerando como situação inicial a Árvore da [Figura 9.7](#), verifique:

- A inserção do valor 25 causaria desbalanceamento?
- A inserção do valor 40 causaria desbalanceamento? Considere a Árvore exatamente como mostra a [Figura 9.7](#), sem a chave 25.
- A inserção do valor 9 causaria desbalanceamento? Considere a Árvore exatamente como mostra a [Figura 9.7](#) (sem 25 e 40).
- A inserção do valor 13 causaria desbalanceamento? Considere a Árvore como mostra a [Figura 9.7](#) (sem 9, 25 e 40).
- A inserção do valor 17 causaria desbalanceamento? Considere a Árvore como mostra a [Figura 9.7](#) (sem 9, 13, 25 e 40).
- A inserção do valor 21 causaria desbalanceamento? Considere a Árvore como mostra a [Figura 9.7](#) (sem 9, 13, 17, 25 e 40).



**FIGURA 9.7** Monitorando o Balanceamento na Inserção, com aumento da Altura He.

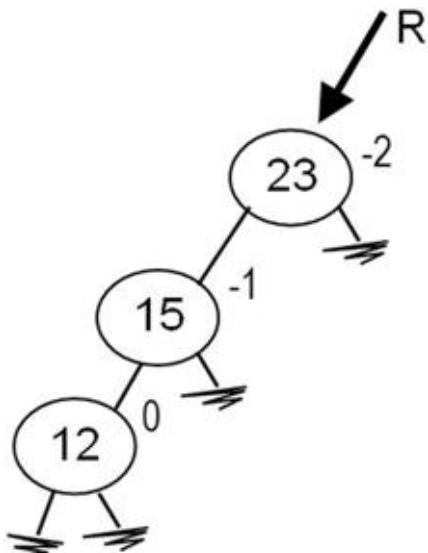
A inserção dos valores 25 ou 40 não causaria desbalanceamento, pois eles seriam inseridos logo abaixo do Nó que contém o valor 29. Na verdade, a Árvore ficaria até mais equilibrada com a inserção de 25 ou 40. Já a inserção dos valores 9, 13, 17 ou 21 causaria desbalanceamento, pois eles seriam inseridos logo abaixo dos Nós de valor 12 ou 19, aumentando o tamanho da Subárvore Esquerda de R. Como a Subárvore Esquerda de R já era maior do que a Subárvore Direita, com a inserção de 9, 13, 17 ou 21 o critério de平衡amento seria violado.

A inserção dos valores 9, 13, 17 ou 21 exemplifica a situação genérica da Figura 9.6c, em que a Subárvore Esquerda já é maior, e cresce ainda mais, causando desbalanceamento. Conforme nossa estratégia para manter a Árvore balanceada (Figura 9.4), ao detectar que a inserção de um novo valor causou desbalanceamento, precisamos ajustar a Árvore para que volte a ser balanceada. Chamamos a esse processo de rebalanceamento.

## Exercício 9.5 Como rebalancear?

Na Figura 9.8, o valor 12 acabou de ser inserido e causou desbalanceamento. Como essa Árvore pode ser rebalanceada? Não se esqueça de que é preciso respeitar o critério de balanceamento (a diferença das alturas pode ser de, no máximo, 1) e também o critério que define uma Árvore Binária de Busca (valores da Subárvore Esquerda devem ser menores, valores da Subárvore

Direita devem ser maiores, para cada Nó da Árvore).

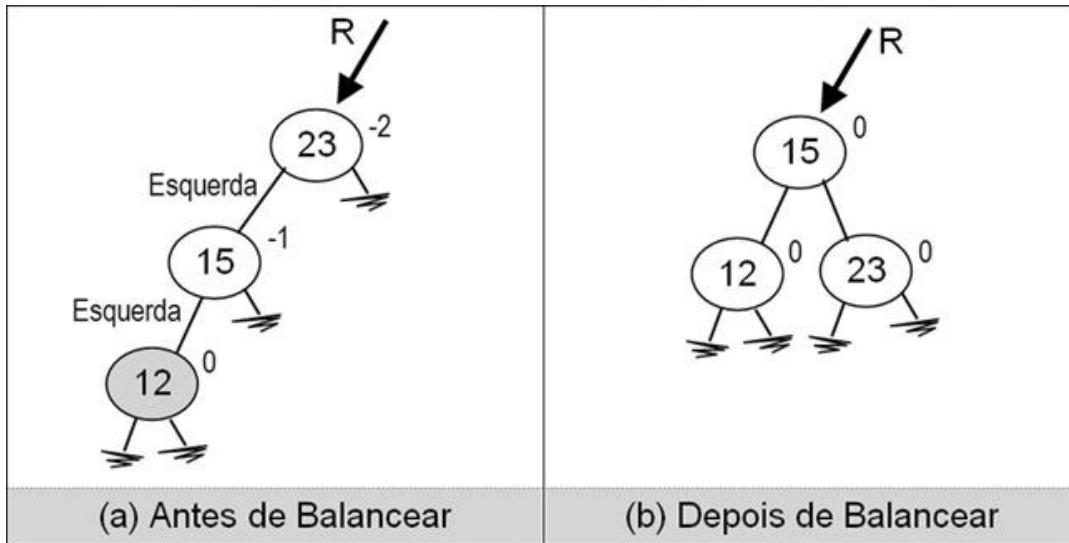


**FIGURA 9.8** Inserção do valor 12 causou desbalanceamento.

## Casos de Rebalanceamento: caso 1 — Rotação Simples EE

O Exercício 9.5 é um exemplo do caso de rebalanceamento Rotação Simples EE ou, ainda, Rotação Simples do tipo Esquerda-Esquerda. Essa forma de denominar os casos de rebalanceamento — adotada neste livro — é compatível com a adotada no material didático de Camargo.

A Figura 9.9 mostra a situação inicial (Figura 9.9a): o Nó que contém o valor 12 acabou de ser inserido e causou o desbalanceamento. O nome Rotação Simples Esquerda-Esquerda reflete a seguinte situação: a partir do Nó em que foi detectado o desbalanceamento (Nó apontado por R), em direção ao Nó que causou o desbalanceamento, temos que seguir para a (Subárvore) Esquerda e depois outra vez para a Subárvore Esquerda.



**FIGURA 9.9** Insere em ABBB: caso 1 — Rotação Simples EE.

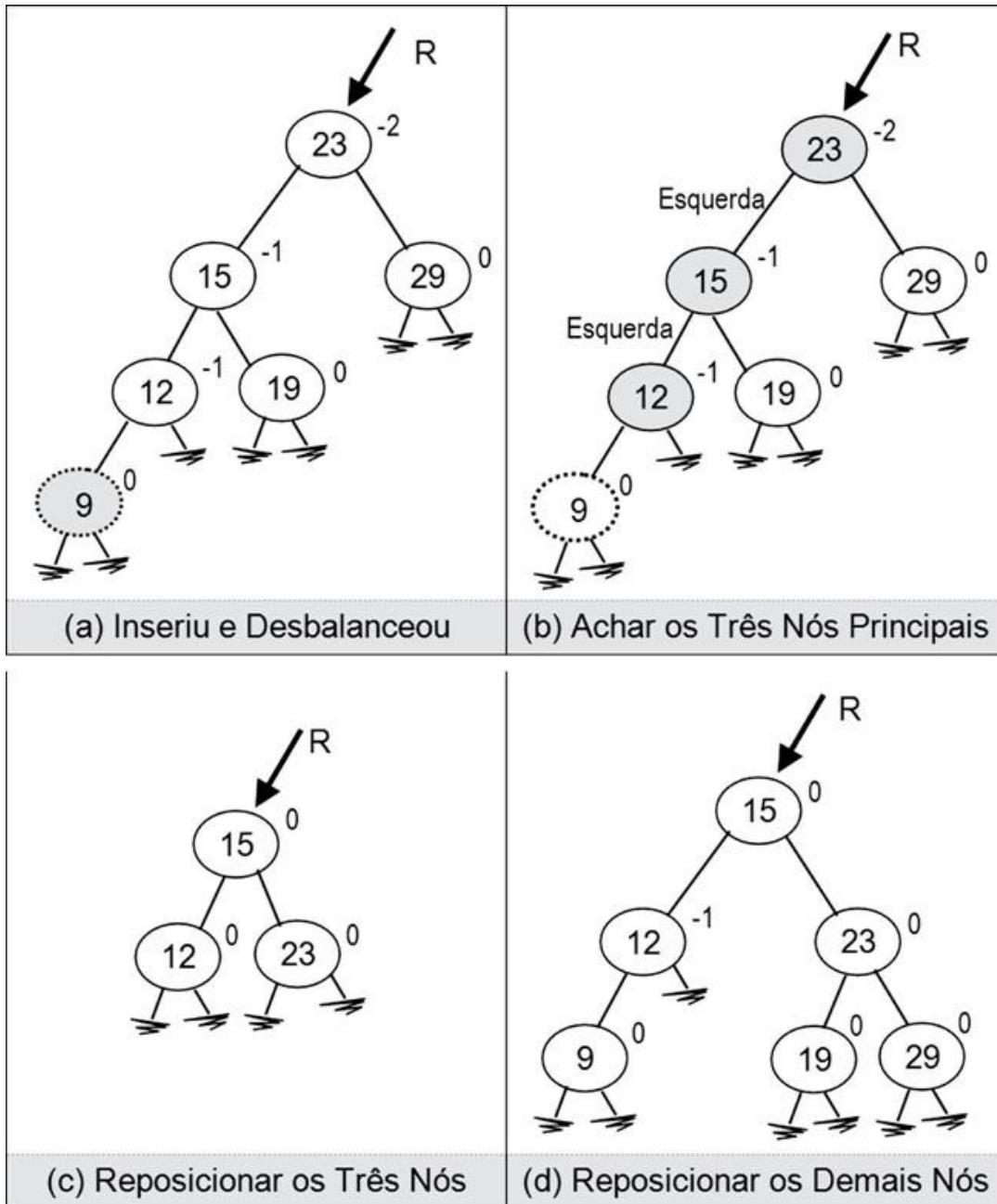
Exemplo com apenas três valores.

A Figura 9.9b mostra a situação final da Árvore, após ter sido rebalanceada. Você consegue achar outra configuração para uma Árvore, com esses mesmos três valores, que respeite o critério de uma ABB e também o critério de balanceamento? Tente achar outra configuração, que não a da Figura 9.9b.

Não há outra configuração que atenda a ambos os critérios. A situação da Figura 9.9b é a única solução. A lógica de平衡amento nesse exemplo com apenas três valores é bastante intuitiva. Lembre-se desse exemplo com apenas três valores se estiver em dúvida quanto ao caso 1 — Rotação Simples EE.

### Exemplo com mais de três valores

Temos um segundo exemplo da Rotação simples Esquerda-Esquerda na Figura 9.10. O valor 9 acabou de ser inserido, causando desbalanceamento. Na Figura 9.10a, a altura da Subárvore Esquerda de R é 3 (três níveis) e a altura da Subárvore Direita de R é 1 (um nível). A diferença entre a altura das Subárvores é 2, o que viola o critério de balanceamento.



**FIGURA 9.10** Insere em ABBB: caso 1 — Rotação Simples EE.

Para rebalancear a Árvore, o primeiro passo é identificar os três Nós principais. O primeiro desses Nós é aquele em que foi detectado o desbalanceamento, ou seja, o Nό apontado por R. Nesse Nό apontado por R, a diferença entre as alturas de suas subárvoreas é 2, o que viola o critério de balanceamento.

Para identificar os outros dois Nós principais para o rebalanceamento, caminhamos em direção ao Nό que causou o desbalanceamento (ou seja, o Nό

que acabou de ser inserido) e encontramos os Nós com valores 15 e 12 ([Figura 9.10b](#)). Como caminhamos para a esquerda e outra vez para a esquerda, fica caracterizado o caso 1 — Rotação Simples EE.

Note que os valores dos três Nós principais são exatamente os mesmos valores e nas mesmas posições do [Exercício 9.5](#) que apresentava uma Árvore com apenas esses três valores. A lógica do rebalanceamento desses três Nós principais será a mesma lógica aplicada no [Exercício 9.5](#), resultando na situação da [Figura 9.10c](#). Após identificar os três Nós principais, pense em uma Árvore com apenas esses três valores. A solução será bastante intuitiva.

Para posicionar os demais valores da Árvore — 9, 19 e 29 — é preciso seguir o critério que define uma Árvore Binária de Busca: valores menores vão para a Subárvore Esquerda, valores maiores vão para a Subárvore Direita. Considerando que os valores 12, 15 e 23 já estão posicionados segundo mostra a [Figura 9.10c](#), em qual lugar deve ser posicionada a chave 9, de modo a respeitar o critério que define uma Árvore Binária de Busca? A única posição possível para a chave 9 é à esquerda da chave 12, pois 9 é menor do que 12. Se o 9 ficasse à direita do 12, quebraria o critério. Se o 9 ficasse abaixo do 23 (seja à esquerda ou à direita), estaria à direita do 15, o que também quebraria o critério.

Seguindo o mesmo raciocínio que utilizamos para posicionar a chave 9, qual o único lugar no qual podemos posicionar os valores 19 e 29, sem quebrar a definição de Árvore Binária de Busca? A única solução possível é posicionar o 19 à esquerda do 23, e o 29 à direita do 23, como mostra a [Figura 9.10d](#). Não há outra solução que não quebre o critério que define uma ABB.

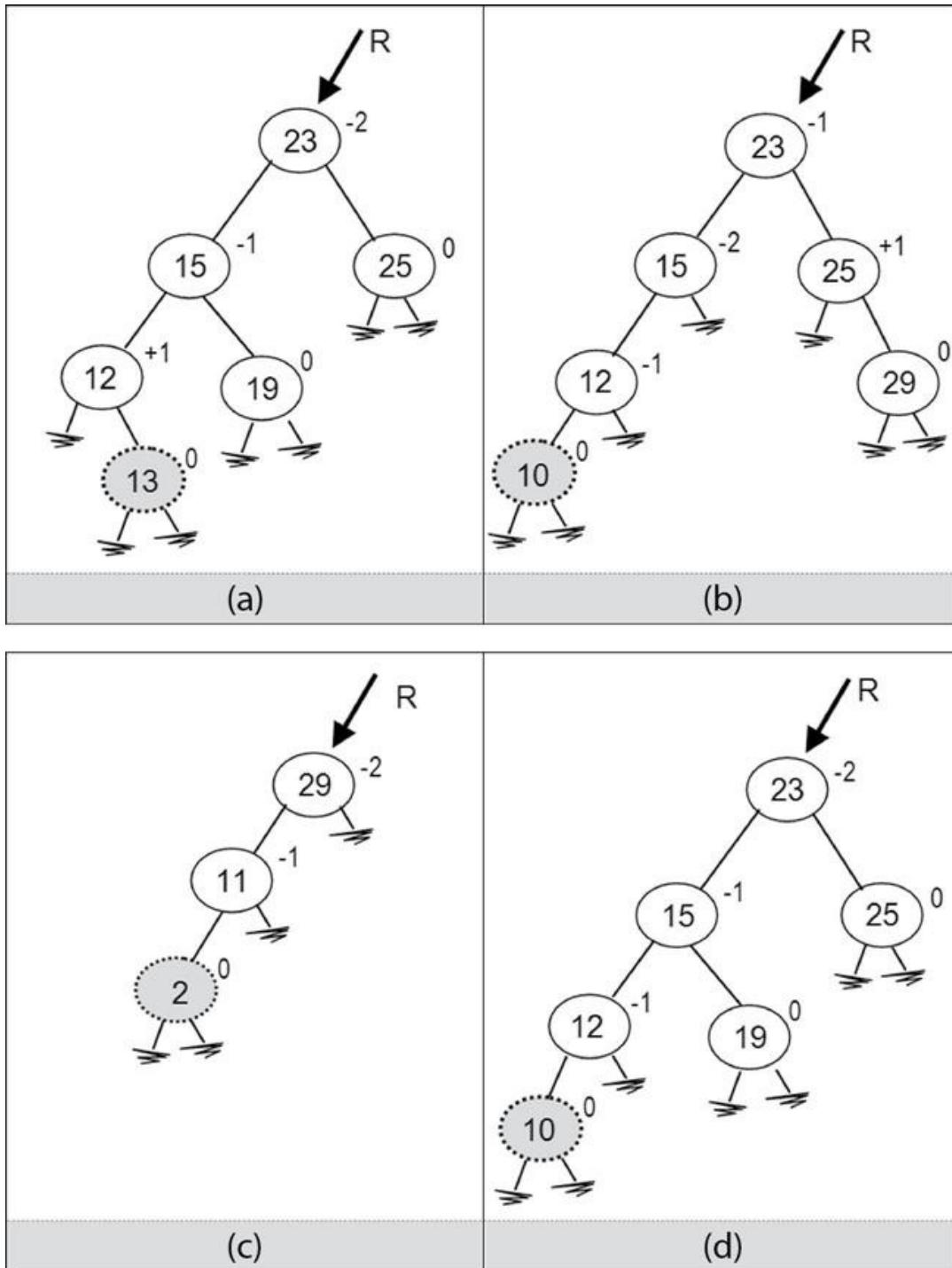
## Exercício 9.6 Rebalanceamento manual — EE

Nas situações da [Figura 9.12](#), um novo valor acabou de ser inserido e causou desbalanceamento. O valor inserido foi posicionado em um Nô com fundo cinza. Em cada exemplo, faça o rebalanceamento pelo caso 1: Rotação Simples EE. Desenhe a Árvore resultante respeitando o critério que define uma ABB e o critério de Balanceamento. Siga o roteiro da [Figura 9.11](#).

## **Para rebalancear uma Árvore manualmente**

- Passo 1: identificar os três Nós principais.** O primeiro desses três é o Nó em que foi detectado o desbalanceamento (a diferença de altura das subárvoreas é maior que 1). Selecione os outros dois Nós caminhando em direção ao Nó que causou o desbalanceamento (o Nó que foi inserido).
- Passo 2: reposicionar os três Nós principais.** Considere uma Árvore com apenas esses três nós e os reposicione na única configuração possível que respeite ambos os critérios: ABB e Balanceamento.
- Passo 3: reposicione os demais valores respeitando o critério que define uma ABB.** Posicione os demais valores abaixo dos três Nós principais. Só há um local possível para cada valor ou Subárvore a ser reposicionada.
- Passo 4: atualize o Fator de Balanceamento de cada Nó.** Conforme definido na Figura 9.5,  $Bal = Hd - He$ , ou seja, o Fator de Balanceamento de um Nó R é a altura da Subárvore Direita de R menos a altura da Subárvore Esquerda de R.

**FIGURA 9.11** Passos para rebalancear uma Árvore.

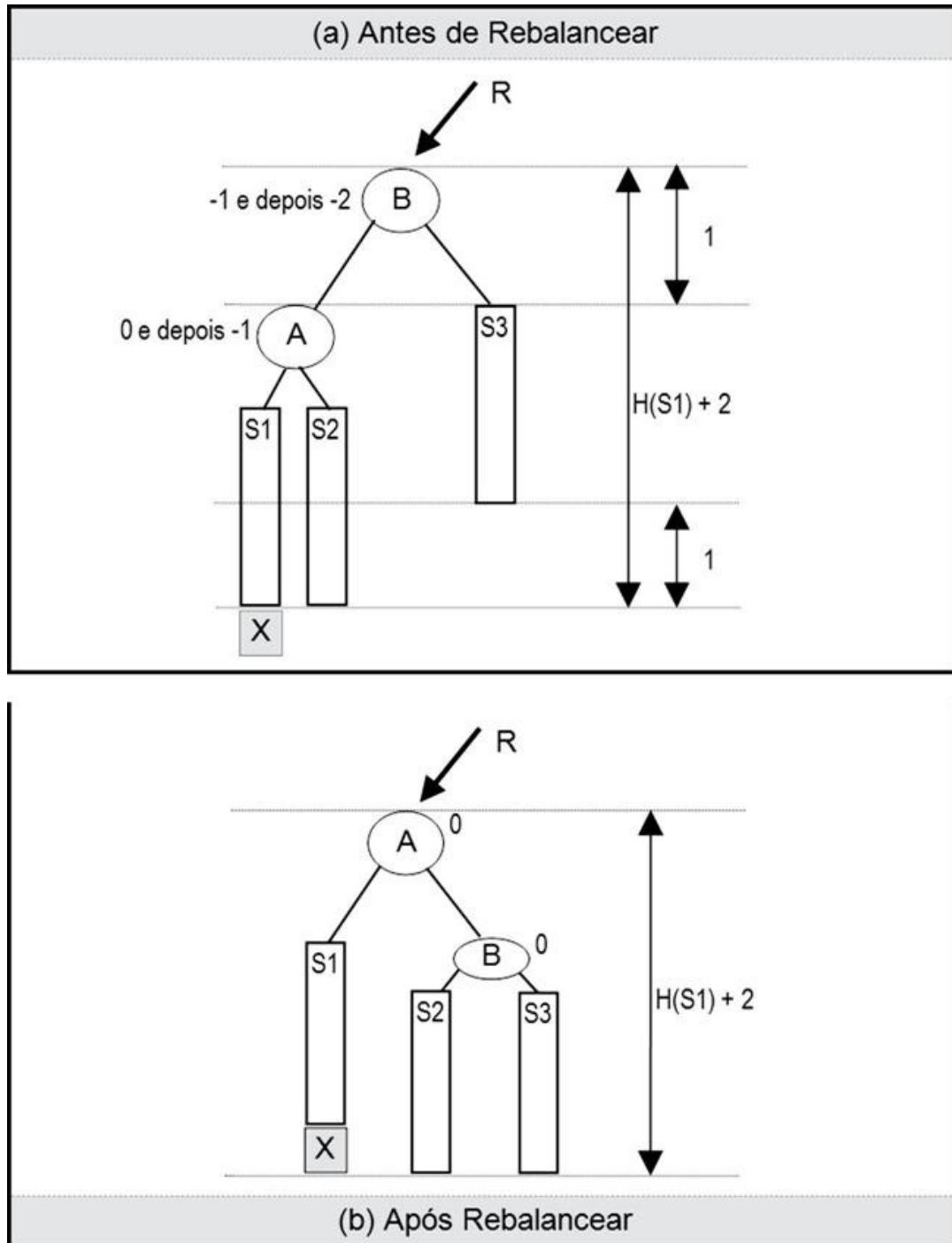


**FIGURA 9.12** Exemplos do caso 1 — Rotação Simples EE.

### Generalização do caso 1: Rotação Simples EE — Insere

A Figura 9.13 apresenta um diagrama genérico do rebalanceamento pelo caso 1: Rotação Simples EE do algoritmo que insere um novo valor na Árvore. Esse

estilo de diagrama é uma adaptação do estilo adotado por Knuth (1998, p. 461) e também por Camargo (p. 3-4).



**FIGURA 9.13** Generalização do caso 1 — Rotação Simples EE — estilo de diagrama adaptado de Knuth (1998) e Camargo.

Note na [Figura 9.13a](#) que um novo valor X acabou de ser inserido, causando desbalanceamento. O Fator de Balanceamento do Nó que contém o valor 'A' antes da inserção de X era zero, pois a altura das Subárvores Esquerda e Direita era igual. Após a inserção de X, o Fator de Balanceamento do Nó que contém 'A' passou a ser -1, pois  $H_e$  passou a ser maior que  $H_d$  em 1. Mesmo com a inserção de X, o Nó que contém 'A' continua balanceado.

O Fator de Balanceamento do Nó que contém o valor 'B' era -1 antes da inserção de X. Com a inserção de X, passou a ser -2. Ou seja, a Subárvore Esquerda do Nó que contém 'B' já era maior que a Subárvore Direita; com a inserção de X em S1, passou a ser ainda maior, violando o critério de balanceamento. Será preciso rebalancear a Árvore.

O rebalanceamento no diagrama da [Figura 9.13](#) generaliza o rebalanceamento dos exemplos numéricos das [Figuras 9.9](#) e [9.10](#), e também dos quatro casos do [Exercício 9.6](#). Procure identificar a equivalência entre os exemplos numéricos e o diagrama genérico da [Figura 9.13](#).

A [Figura 9.14](#) mostra um trecho de algoritmo que implementa o rebalanceamento pelo caso 1: Rotação Simples EE. É um trecho do algoritmo que insere um valor X em uma Árvore Binária de Busca Balanceada (ABBB).

```

Variável Filho do tipo NodePtr; // Filho é ponteiro auxiliar, que apontará R→Esq
/* movimentando as Subárvores e os ponteiros */
Filho = R→Esq; // Filho aponta para o Nó que contém o valor 'A' - Figura 9.13
R→Esq = Filho→Dir; // R→Esq passa a apontar para S2 - Figura 9.13
Filho→Dir = R; // Filho→Dir passa a apontar o Nó que contém 'B' - Figura 9.13
/* ajustando os balanceamentos */
R→Bal = 0; // atualizando o Fator de Balanceamento de R
Filho→Bal = 0; // atualizando o Fator de Balanceamento de Filho
/* mudando a Raiz da árvore*/
R = Filho; // o Nó que contém 'A' passará a ser a Raiz - Figura 9.13
/* atualizando a variável MudouAltura */
MudouAltura = Falso; // após inserir X e rebalancear, a altura da Árvore continua sendo
// a mesma:  $H(S1) + 2$ .

```

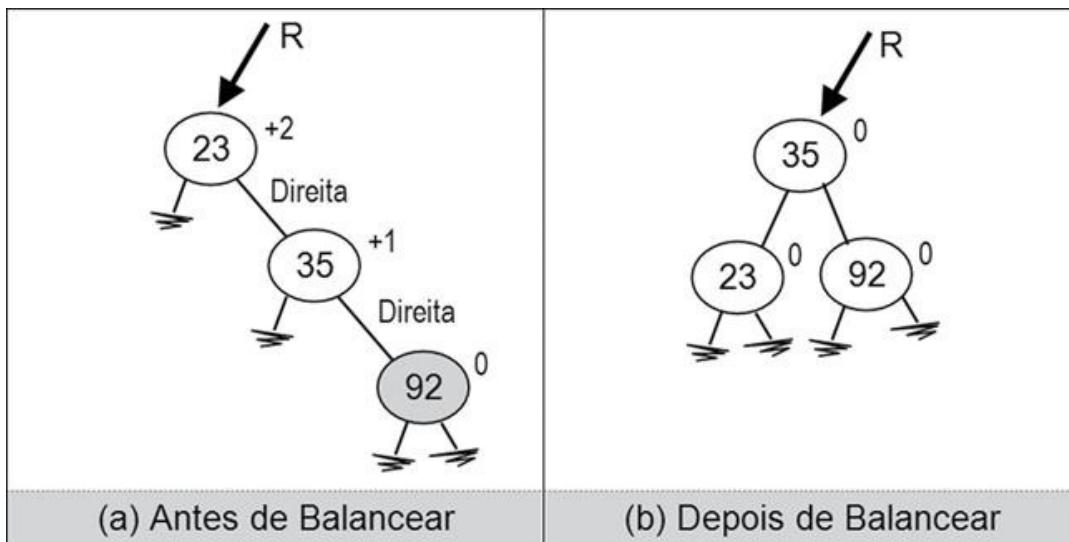
**FIGURA 9.14** Insere em ABBB — caso 1: Rotação Simples EE.

O algoritmo da [Figura 9.14](#) providencia a movimentação das Subárvores e dos ponteiros, partindo da situação da [Figura 9.13a](#) e levando à situação da [Figura 9.13b](#). Em seguida, o algoritmo ajusta os balanceamentos, muda a raiz da Árvore e atualiza uma variável chamada MudouAltura.

Note que a altura da Subárvore S1 é igual à altura da Subárvore S2 e também igual à altura da Subárvore S3. Ou seja,  $H(S1) = H(S2) = H(S3)$ . Antes da inserção de X, a altura total da Árvore era  $H(S1) + 2$ . Após a inserção de X e o rebalanceamento, a altura da Árvore continua sendo  $H(S1) + 2$  (veja na [Figura 9.13b](#)). Por isso, no algoritmo da [Figura 9.14](#) a variável MudouAltura recebeu o valor Falso.

## Casos de rebalanceamento: caso 2 — Rotação Simples DD

O caso 2 — Rotação Simples Direita-Direita (DD) é um caso de rebalanceamento absolutamente simétrico à Rotação Simples Esquerda-Esquerda (EE). A [Figura 9.15](#) apresenta um exemplo do caso de rebalanceamento Rotação Simples DD. A [Figura 9.15](#) mostra a situação inicial ([Figura 9.15a](#)): o Nó que contém o valor 92 acabou de ser inserido e causou o desbalanceamento. A [Figura 9.15b](#) mostra a situação final da Árvore após ter sido rebalanceada. Não há outra configuração possível que atenda os critérios que definem uma ABBB.

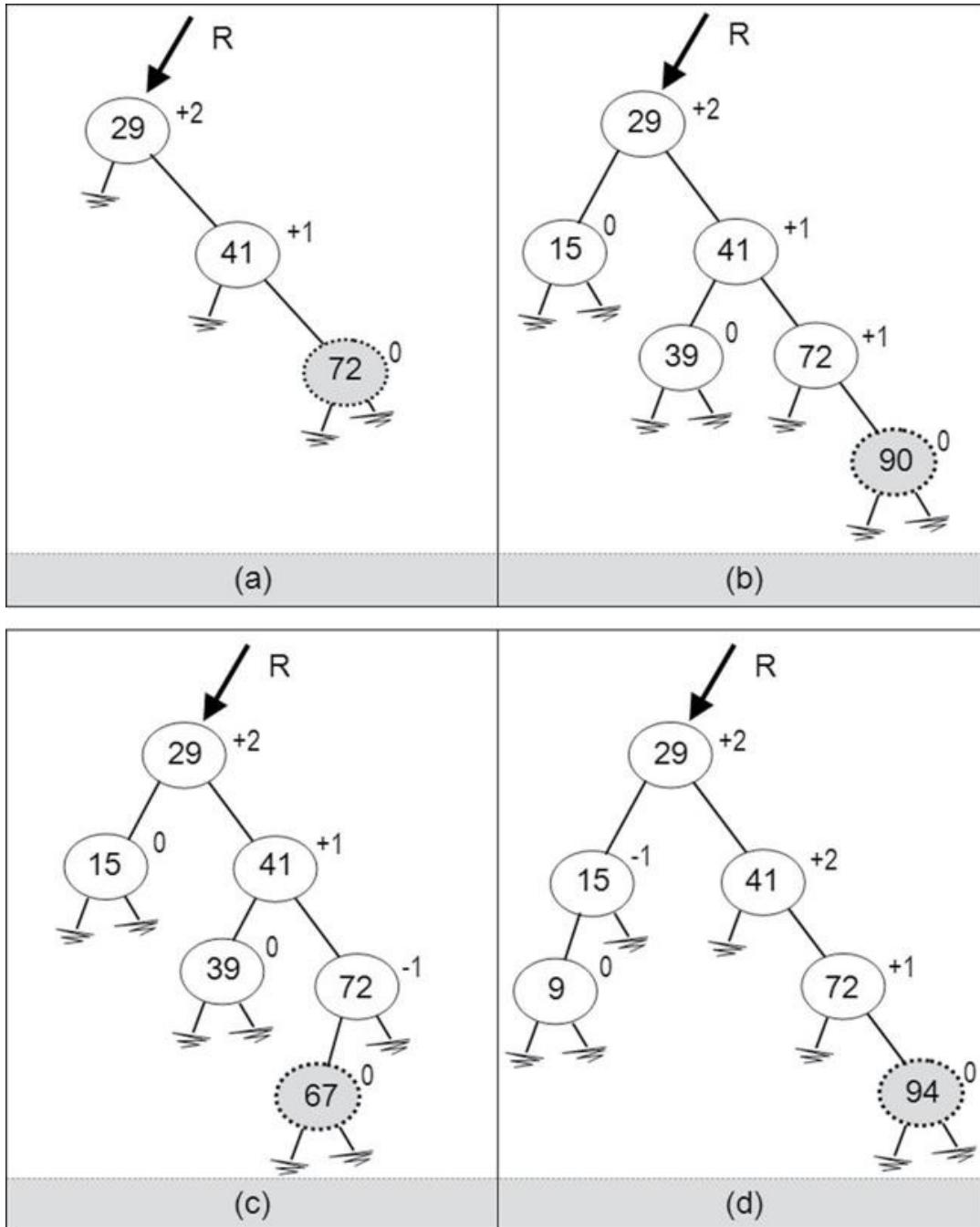


**FIGURA 9.15** Insere em ABBB: caso 2 — Rotação Simples DD.  
Exemplo com apenas três valores.

## Exercício 9.7 Rebalanceamento — caso 2: Rotação Simples DD

Nas situações da [Figura 9.16](#), um novo valor acabou de ser inserido e causou desbalanceamento. O valor inserido está posicionado em um Nó destacado com

fundo cinza. Em cada caso, faça o rebalanceamento pelo caso 2 — Rotação Simples DD. Desenhe a Árvore resultante respeitando o critério que define uma ABB e também o critério de Balanceamento. Siga o roteiro da [Figura 9.11](#).



**FIGURA 9.16** Exemplos do caso 2 — Rotação Simples DD.

## **Exercício 9.8 Diagrama — caso 2: Rotação Simples DD**

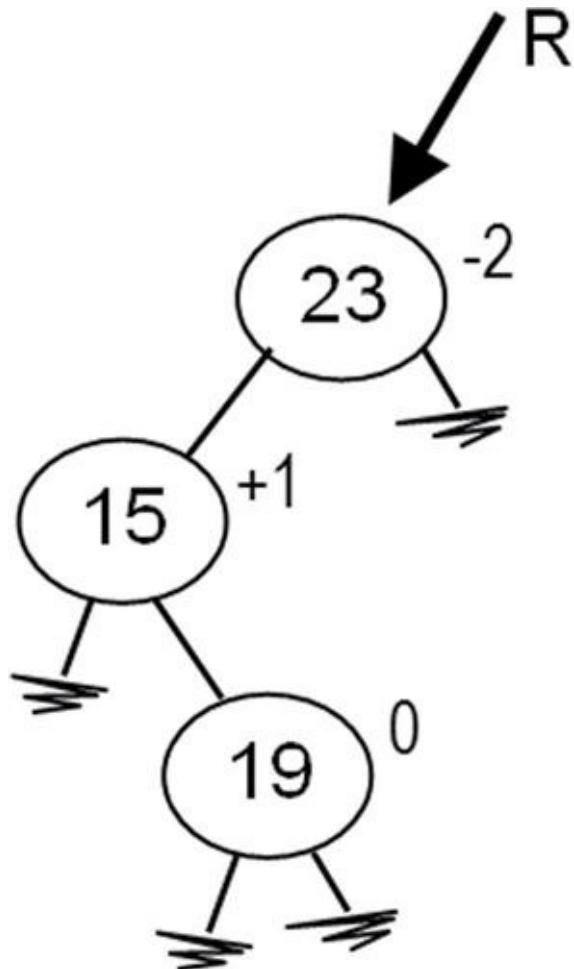
Generalizamos a Rotação Simples EE através do diagrama da [Figura 9.13](#). Desenvolva um diagrama genérico (simétrico ao do [Figura 9.13](#)) para o caso 2: Rotação Simples DD.

## **Exercício 9.9 Algoritmo — caso 2: Rotação Simples DD**

Generalizamos a Rotação Simples EE através do diagrama da [Figura 9.13](#) e então desenvolvemos o trecho de algoritmo da [Figura 9.14](#). Desenvolva um trecho de algoritmo que implemente o caso 2: Rotação Simples DD. O trecho de algoritmo resultante deve ser análogo ao trecho de algoritmo da [Figura 9.14](#).

## **Exercício 9.10 Como rebalancear?**

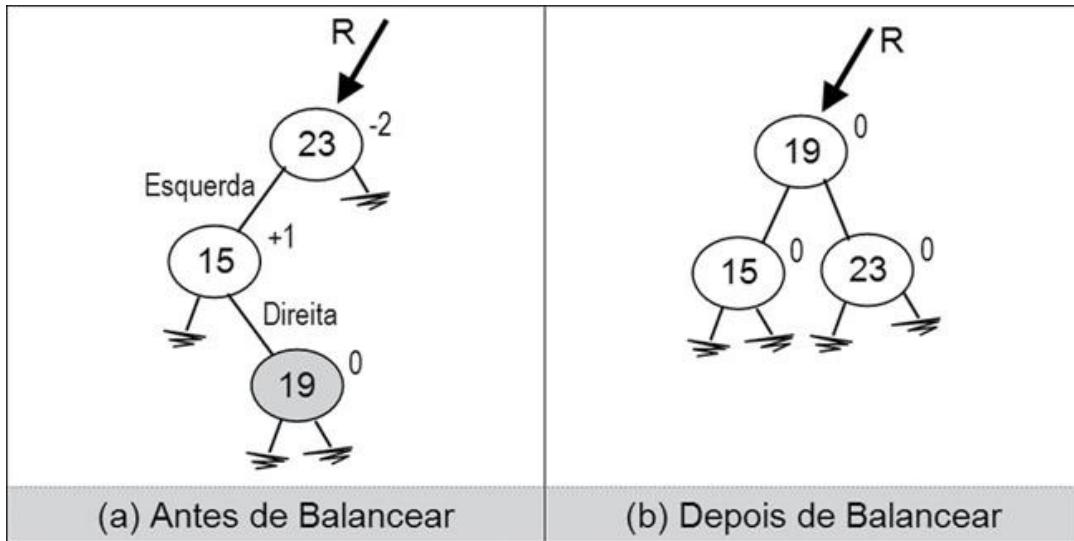
Na [Figura 9.17](#), o valor 19 acabou de ser inserido e causou desbalanceamento. Como essa Árvore pode ser rebalanceada? Desenhe a nova Árvore respeitando o critério de balanceamento ([Figura 9.4](#)) e também o critério que define uma Árvore Binária de Busca ([Figura 8.4](#)).



**FIGURA 9.17** Inserção do valor 19 causou desbalanceamento.

### Casos de rebalanceamento: caso 3 — Rotação Dupla ED

O Exercício 9.10 é um exemplo do caso de rebalanceamento Rotação Dupla ED ou Rotação Dupla do tipo Esquerda-Direita (nomenclatura de Camargo). A Figura 9.18 mostra a situação inicial (Figura 9.18a): o Nó que contém o valor 19 acabou de ser inserido e causou o desbalanceamento da Árvore. O nome Rotação Dupla Esquerda-Direita reflete a situação em que, a partir do Nó em que foi detectado o desbalanceamento (Nó apontado por R), em direção ao Nó que causou o desbalanceamento, temos que seguir para a Subárvore Esquerda e depois para a Subárvore Direita.



**FIGURA 9.18** Insere em ABBB: caso 3 — Rotação Dupla ED.

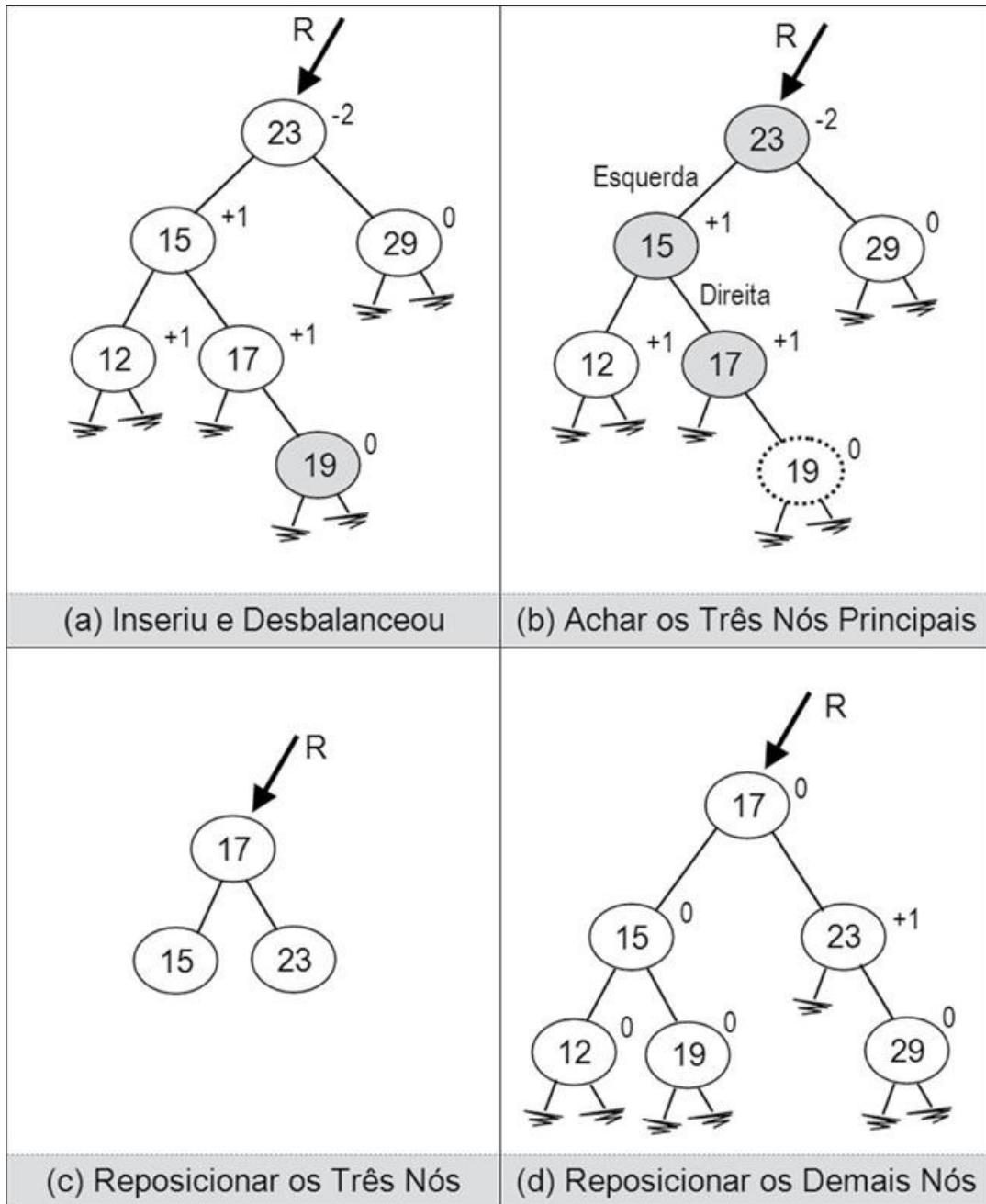
**Exemplo com apenas três valores.**

A Figura 9.18b mostra a situação final da Árvore após ter sido rebalanceada. Você consegue achar outra configuração para uma Árvore com esses mesmos três valores que respeite o critério de uma ABB e também o critério de balanceamento? Tente achar outra configuração que não a da Figura 9.18b.

Não há outra configuração que atenda a ambos os critérios. A situação da Figura 9.18b é a única solução. A lógica de balanceamento nesse exemplo com apenas três valores na Árvore é bastante intuitiva. Lembre-se desse exemplo com apenas três valores na Árvore se estiver em dúvida quanto ao caso 3 — Rotação Dupla ED.

### Exemplo com mais de três valores — caso ED

Temos um segundo exemplo da Rotação Dupla Esquerda-Direita na [Figura 9.19](#). Note, na [Figura 9.19a](#), que o valor 19 acabou de ser inserido, causando desbalanceamento, pois a altura da Subárvore Esquerda de R é 3 (três níveis) e a altura da Subárvore Direita de R é 1 (um nível). A diferença entre a altura das Subárvores é 2, o que viola o critério de balanceamento.



**FIGURA 9.19** Insere em ABBB: caso 3 — Rotação Dupla ED.

Para rebalancear a Árvore, o primeiro passo é identificar os três Nós principais. O primeiro desses Nós é aquele em que foi detectado o desbalanceamento, ou seja, o NÓ apontado por R. Para identificar os outros dois Nós principais para o rebalanceamento, caminhamos em direção ao NÓ que causou o desbalanceamento (ou seja, o NÓ que acabou de ser inserido) e encontramos os Nós com valores 15 e 17 (Figura 9.19b). Como caminhamos para a esquerda e depois para a direita, fica caracterizado o caso 3 — Rotação Dupla ED.

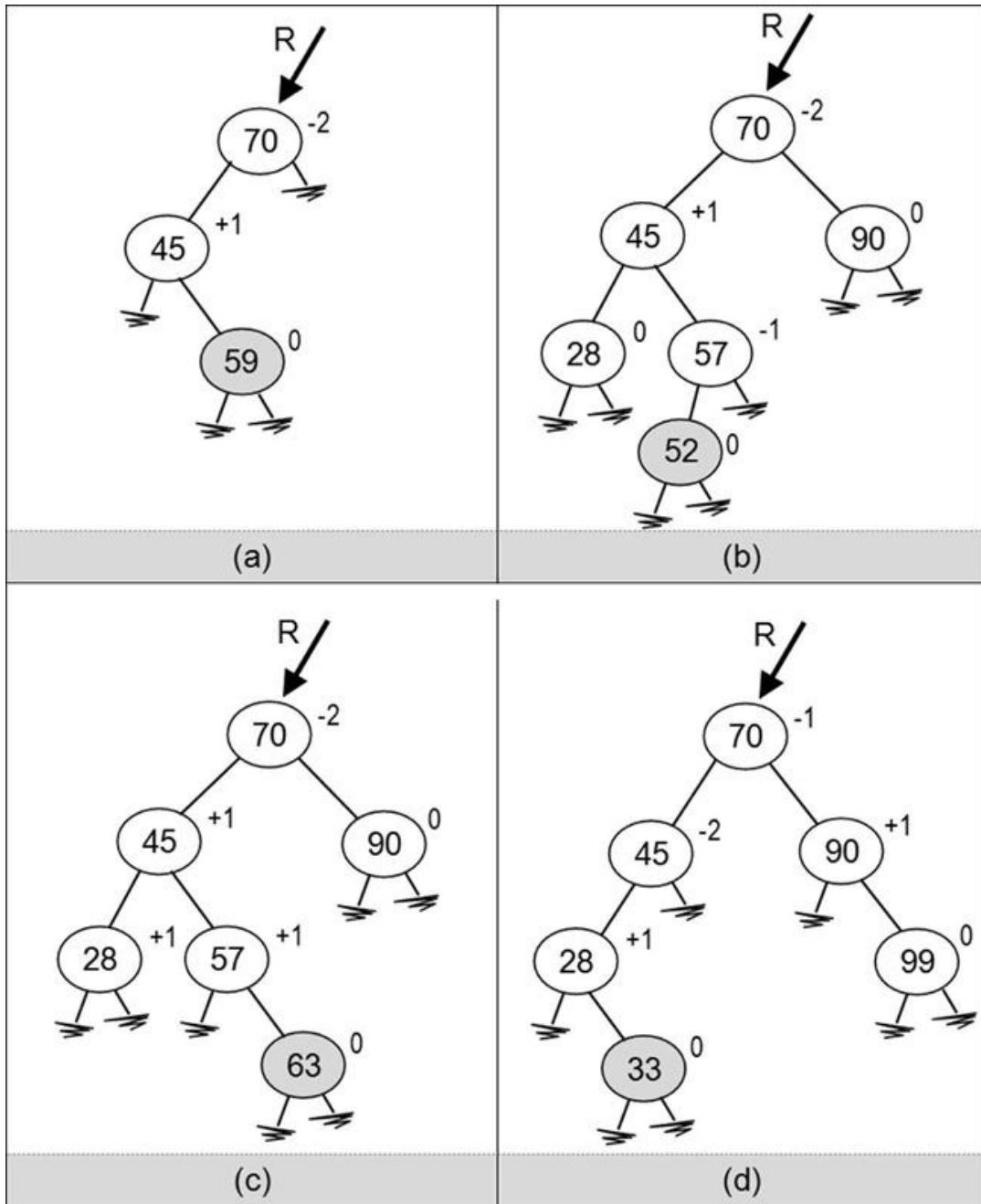
Dupla ED.

Note que os valores dos três Nós principais são exatamente os mesmos e nas mesmas posições do [Exercício 9.10](#), que apresentava uma Árvore com apenas três valores. A lógica do rebalanceamento desses três Nós principais será a mesma lógica aplicada no [Exercício 9.10](#), resultando na situação da [Figura 9.19c](#). Após identificar os três Nós principais, pense em uma Árvore com apenas esses três valores. A solução será bastante intuitiva.

Para posicionar os demais valores da Árvore (12, 19 e 29), é preciso seguir o critério que define uma Arvore Binária de Busca: valores menores vão para a Subárvore Esquerda, valores maiores vão para a Subárvore Direita. Considerando que os valores 15, 17 e 23 já estão posicionados ([Figura 9.19c](#)), em qual lugar deve ser posicionada a chave 12 de modo a respeitar o critério que define uma Árvore Binária de Busca? A única posição possível para a chave 12 (sem deslocamento dos valores já posicionados) é à esquerda da chave 15, pois 12 é menor do que 15. Seguindo o mesmo raciocínio, posicionamos o 19 à direita do 15, e o 29 à direita do 23, como mostra a [Figura 9.19d](#). Não há outra posição possível, sem deslocamento dos três valores principais já posicionados, que não quebre o critério que define uma ABB.

## **Exercício 9.11 Rebalanceamento manual — ED**

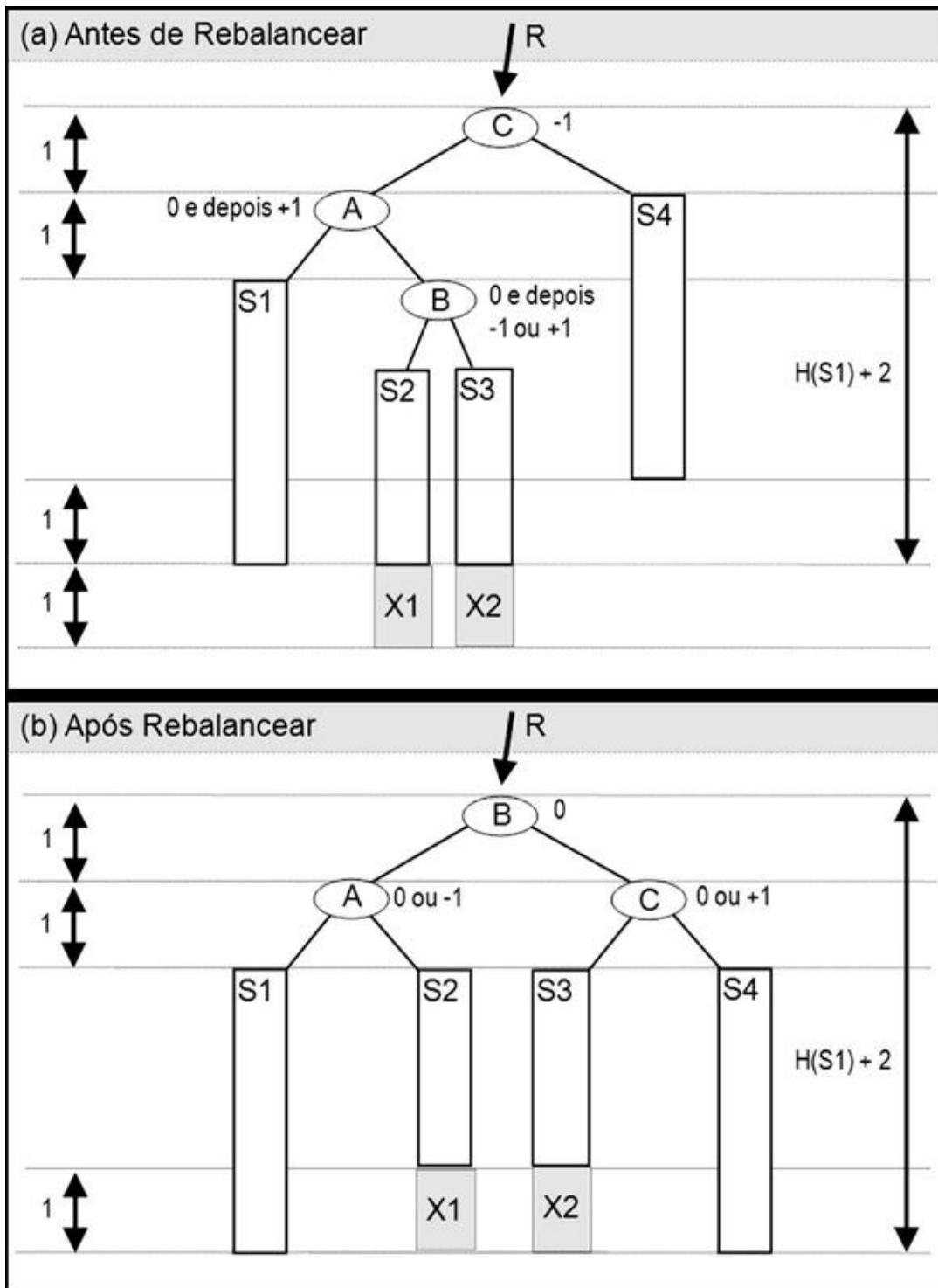
Na [Figura 9.20](#), um novo valor (destaque em cinza) foi inserido e causou desbalanceamento. Em cada caso, faça o rebalanceamento pelo caso 3: Rotação Dupla ED. Desenhe a Árvore resultante respeitando o critério que define uma ABB e também o critério de Balanceamento. Siga o roteiro da [Figura 9.11](#).



**FIGURA 9.20** Exemplos do caso 3 — Rotação Dupla ED.

### Generalização do caso 3: Rotação Dupla ED — Insere

A Figura 9.21 apresenta um diagrama genérico do rebalanceamento pelo caso 3: Rotação Dupla ED do algoritmo que insere um novo valor na Árvore. Um novo valor X1 ou um novo valor X2 (ou um ou outro; nunca ambos) acabou de ser inserido, causando desbalanceamento.



**FIGURA 9.21** Generalização do caso 3 — Rotação Dupla ED — estilo de diagrama adaptado de [Knuth \(1998\)](#) e Camargo.

O Fator de Balanceamento do Nó que contém o valor 'B' antes da inserção de X era zero, pois a altura das Subárvores Esquerda e Direita era igual. Após a

inserção de X1 o Fator de Balanceamento do Nó que contém 'B' passou a ser  $-1$ . Se, em vez de X1, houve a inserção de X2, o Fator de Balanceamento do Nó que contém 'B' passou a ser  $+1$ .

O Fator de Balanceamento do Nó que contém o valor 'A' antes da inserção de X1 ou X2 era zero e passou a ser  $+1$  após a inserção de X1 ou X2. Considerando apenas o Nó que contém o valor 'A' e o Nó que contém o valor 'B', a Árvore continua balanceada.

O Fator de Balanceamento do Nó que contém o valor 'C' antes da inserção de X1 ou X2 era  $-1$ . Após a inserção de X1 ou X2, o Fator de Balanceamento passou a ser  $-2$ , quebrando o critério de balanceamento. Ou seja, a Subárvore Esquerda do Nó que contém 'C' já era maior que a Subárvore Direita; com a inserção de X1 ou X2, a Subárvore Esquerda passou a ser ainda maior, violando o critério. Será preciso rebalancear a Árvore.

O rebalanceamento ilustrado no diagrama da [Figura 9.21](#) generaliza os exemplos numéricos das [Figuras 9.18](#) e [9.19](#), e também os quatro casos do [Exercício 9.11](#). Procure identificar a equivalência entre os exemplos numéricos e o diagrama genérico da [Figura 9.21](#).

A [Figura 9.22](#) mostra o trecho de algoritmo que implementa o rebalanceamento pelo caso 3: Rotação Dupla ED. É um trecho do algoritmo que insere um valor X em uma Árvore Binária de Busca Balanceada. O algoritmo providencia a movimentação das Subárvores e dos ponteiros, partindo da situação da [Figura 9.21a](#) e levando à situação da [Figura 9.21b](#). Em seguida, o algoritmo ajusta os balanceamentos, muda a raiz da Árvore e atualiza uma variável chamada MudouAltura.

```

variável Filho do tipo NodePtr; // Filho é ponteiro auxiliar, que apontará R→Esq
variável Neto do tipo NodePtr; // Neto é ponteiro auxiliar, que apontará Filho→Dir

/* posicionando os ponteiros Filho e Neto */
Filho = R→Esq; // Filho aponta para o Nó que contém o valor 'A' - Figura 2.21
Neto = Filho→Dir; // Filho aponta para o Nó que contém o valor 'B' - Figura 2.21

/* movimentando a Subárvore S2 */
Filho→Dir = Neto→Esq; // Filho→Dir passa a apontar para S2 - Figura 2.21
Neto→Esq = Filho; // Neto→Esq passa a apontar Nó que contém 'A', Figura 2.21

/* movimentando a Subárvore S3 */
R→Esq = Neto→Dir; // R→Esq passa a apontar para S3 - Figura 2.21
Neto→Dir = R; // Neto→Dir passa a apontar o Nó que contém 'C'

/* ajustando os balanceamentos */
Caso Neto→Bal for
    -1 : { R→Bal = 1; // inseriu X1. Exemplo numérico na Figura 9.20b.
            Filho→Bal = 0;
            Neto→Bal = 0; }
    +1 : { R→Bal = 0; // inseriu X2. Exemplo numérico na Figura 9.19 e...
            Filho→Bal = -1; // ... na Figura 9.20c
            Neto→Bal = 0; }
    0 : { R→Bal = 0; // inseriu o Nó apontado por Neto, que contém...
            Filho→Bal = 0; // ... o valor 'B', na Figura 9.21. Exemplo numérico...
            Neto→Bal = 0; } // ... na Figura 9.18

/* mudando a Raiz da árvore*/
R = Neto; // o Nó que contém 'B' passará a ser a Raiz - Figura 2.21

/* atualizando a variável MudouAltura */
MudouAltura = Falso; // após inserir X1 ou X2 e rebalancear...
// ... a altura da Árvore continua a mesma: H(S1) + 2.

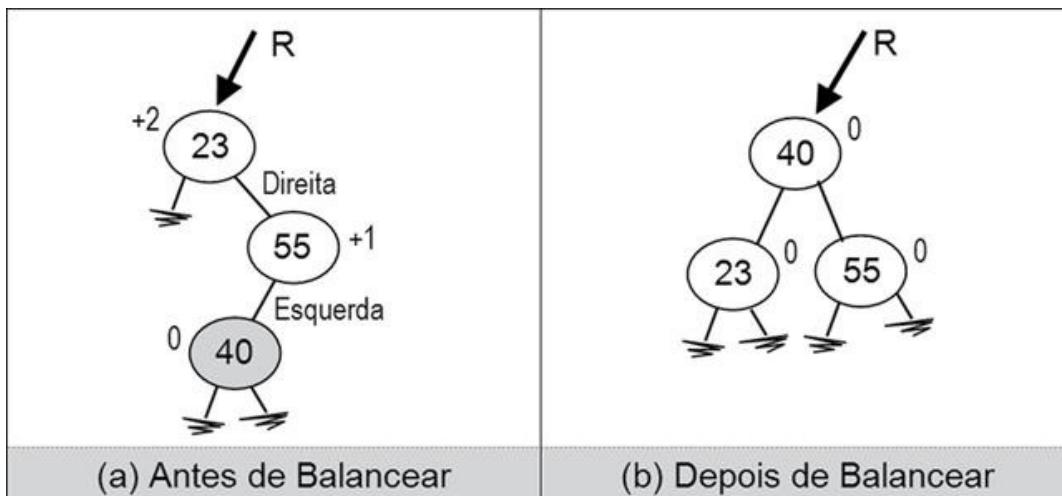
```

**FIGURA 9.22** Insere em ABBB — algoritmo do caso 3: Rotação Dupla ED.

Note que a altura da Subárvore S1 é igual à altura da Subárvore S4; a altura da Subárvore S2 é igual à altura da Subárvore S3; e S1 e S4 são maiores que S2 e S3 em 1. Ou seja,  $H(S1) = H(S4)$ ;  $H(S2) = H(S3)$ ;  $H(S1) = H(S2) + 1$ . Antes da inserção de X1 ou X2, a altura total da Árvore era  $H(S1) + 2$ . Após a inserção de X1 ou X2 e o rebalanceamento, a altura da Árvore continua sendo  $H(S1) + 2$  (veja na [Figura 9.21b](#)). Por isso, no algoritmo da [Figura 9.22](#), a variável MudouAltura recebeu o valor Falso.

## Casos de rebalanceamento: caso 4 — Rotação Dupla DE

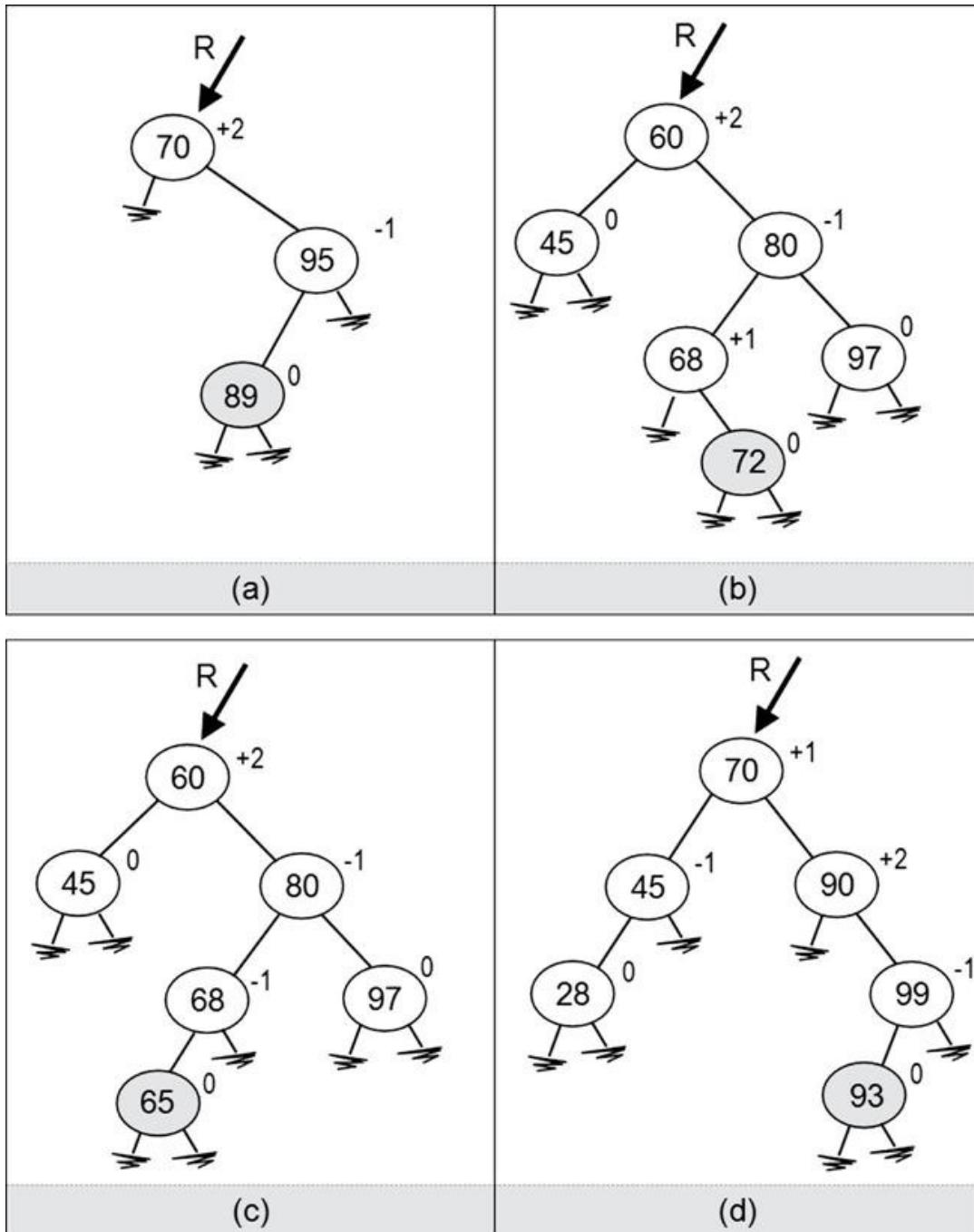
O caso 4 — Rotação Dupla Direita-Esquerda (DE) é um caso de rebalanceamento absolutamente simétrico à Rotação Dupla Esquerda-Direita (ED). A [Figura 9.23](#) apresenta um exemplo do caso de rebalanceamento Rotação Dupla DE. A [Figura 9.23a](#) mostra a situação inicial ([Figura 9.23a](#)): o Nó que contém o valor 40 acabou de ser inserido e causou o desbalanceamento. A [Figura 9.23b](#) mostra a situação final da Árvore após ter sido rebalanceada. Não há outra configuração possível que atenda os critérios que definem uma ABBB.



**FIGURA 9.23** Insere em ABBB: caso 4 — Rotação Dupla DE.  
Exemplo com apenas três valores.

## Exercício 9.12 Rebalanceamento — caso 4: Rotação Dupla DE

Nas situações da [Figura 9.24](#), um novo valor acabou de ser inserido e causou desbalanceamento. O valor inserido está posicionado em um Nó destacado com fundo cinza. Em cada caso, faça o rebalanceamento pelo caso 4 — Rotação Dupla DE. Desenhe a Árvore resultante respeitando o critério que define uma ABB e também o critério de Balanceamento. Siga o roteiro da [Figura 9.11](#).



**FIGURA 9.24** Exemplos do caso 4 — Rotação Dupla DE.

### Exercício 9.13 Diagrama — caso 4: Rotação Dupla DE — diagrama

Generalizamos a Rotação Dupla ED através do diagrama da Figura 9.21. Desenvolva um diagrama genérico (simétrico ao da Figura 9.21) para o caso 4:

Rotação Dupla DE.

### **Exercício 9.14 Algoritmo — caso 4: Rotação Dupla DE**

Generalizamos a Rotação Dupla ED através do diagrama da [Figura 9.21](#) e então desenvolvemos o trecho de algoritmo da [Figura 9.22](#). Desenvolva um trecho de algoritmo que implemente o caso 4: Rotação Dupla DE. O trecho de algoritmo resultante deve ser análogo ao trecho de algoritmo da [Figura 9.22](#).

### **Exercício 9.15 Algoritmo Insere — ABB Balanceada (ABBB)**

No [Capítulo 8](#), elaboramos um algoritmo que insere um novo valor em uma Árvore Binária de Busca — ABB ([Figura 8.20](#)). Adapte esse algoritmo fazendo-o monitorar o balanceamento da Árvore e desencadear ações de rebalanceamento, sempre que necessárias, de modo a manter a Árvore sempre balanceada. Para monitorar o balanceamento, tome como ponto de partida o diagrama da [Figura 9.6](#) e faça um diagrama análogo para a inserção de um valor X na Subárvore Direita de R. Como ações de rebalanceamento, considere os casos 1, 2, 3 e 4 estudados anteriormente.

A [Figura 9.25](#) apresenta o algoritmo para a operação que insere um valor X em uma Árvore Binária de Busca Balanceada (ABBB). O algoritmo da [Figura 9.25](#) foi elaborado tendo como base, em alguns aspectos, o algoritmo de Camargo (p. 6-8).

```

Insere (parâmetro por referência R tipo ABBB, parâmetro X tipo Inteiro, parâmetro por
referência Ok tipo Boolean, parâmetro por referência MudouAltura tipo Boolean) {

/* Insere o valor X na ABBB de Raiz R como um Nó terminal (sem Filhos). Monitora o
balanceamento da Árvore e desencadeia ações de rebalanceamento, caso necessário. Ok
retorna Verdadeiro se X foi inserido e Falso caso contrário. MudouAltura retorna
Verdadeiro se a inserção de X aumentou a altura da Árvore e Falso caso contrário */

Variáveis P, Filho, Neto do tipo NodePtr; // NodePtr = Ponteiro para Nó;

Se (R == Null)
Então { /* Caso 1 da Figura 8.19 - Achou o lugar; insere! */
    P = NewNode; P→Info = X; P→Bal = 0; P→Dir = Null; P→Esq = Null;
    R = P; P = Null; Ok = Verdadeiro; MudouAltura = Verdadeiro;
} // fim do Caso 1 da Figura 8.19
Senão Se (X == R→Info)
    Então /* Caso 2 da Figura 8.19: X já está na árvore; não insere! */
        { Ok = Falso; MudouAltura = Falso; } // fim do Caso 2 da Figura 8.19
    Senão { Se (R→Info > X)
        Então /* Caso 3 da Figura 8.19: tenta inserir X na Sub Esq de R */
            { Insere (R→Esq, X , Ok, MudouAltura);
            // monitora balanceamento voltando de inserir na Sub Esq de R
            Se MudouAltura // Se a Subárvore esquerda cresceu.
            Então Caso R→Bal for:
                +1: { R→Bal = 0; MudouAltura = Falso;} // Figura 9.6a
                0: R→Bal= -1; // Figura 9.6b.
                // MudouAltura continua Verdadeiro
                -1: /* Figura 9.6c. É preciso rebalancear! */
                    { Filho = R→Esq;
                    Se (Filho→Bal == +1)
                        Então RotDuplaEDInsere; // Figura 9.22
                    Senão RotSimplesEEInsere; } // Figura 9.14
            } // fim do Caso 3 da Figura 8.19
        Senão /* Caso 4 da Figura 8.19: tenta inserir X na Sub Dir de R */
            { Insere(R→Dir, X, Ok, MudouAltura);
            // monitora balanceamento voltando de inserir na Sub Dir de R
            Se MudouAltura // se a Subárvore Direita cresceu...
            então Caso R→Bal for:
                -1: { R→Bal = 0; MudouAltura = Falso; };
                0: R→Bal=1; // MudouAltura continua Verdadeiro
                +1: { Filho = R→Dir; // É preciso rebalancear!
                    Se (Filho→Bal = -1)
                        Então RotDuplaDeInsere; // Exercício 9.15
                    Senão RotSimplesDDInsere; } // Exercício 9.9
            }; // fim do Caso 4 da Figura 8.19;
    } /* fim do algoritmo Insere em ABBB */
}

```

**FIGURA 9.25 Algoritmo conceitual — Insere em ABBB — algoritmo**



adaptado de Camargo (p. 6-8).

Para ajustar o algoritmo que insere um valor X em uma Árvore Binária de Busca não balanceada ([Figura 8.20](#)), cada Nó da Árvore precisará armazenar o seu Fator de Balanceamento. Assim, além dos campos Info, Dir e Esq, cada Nó precisará ter também o campo Bal — Fator de Balanceamento.

O balanceamento de um Nó apontado por um ponteiro R precisa ser ajustado logo após a inserção de um novo valor em uma de suas Subárvores. No algoritmo da [Figura 9.25](#), esse monitoramento é implementado logo após as chamadas recursivas que inserem um valor X nas Subárvores Esquerda ou Direita de R.

A [Figura 9.6](#) ilustra o monitoramento e o ajuste do balanceamento quando inserimos um novo valor na Árvore, aumentando a altura da Subárvore Esquerda de R. Assim, logo após retornar da chamada recursiva que insere o novo valor X na Subárvore Esquerda de R — destacada em negrito na [Figura 9.25](#) — tratamos os casos *a*, *b* e *c* da [Figura 9.6](#), trecho de algoritmo também destacado em negrito na [Figura 9.25](#). Compare esse trecho do algoritmo com os diagramas da [Figura 9.6](#).

Nos casos *a* e *b*, o Fator de Balanceamento de R é ajustado, mas a Árvore continua balanceada. No caso *c*, a Árvore precisa ser rebalanceada. Considerando que estamos retornando de uma inserção na Subárvore Esquerda, podem ser desencadeados os casos de rebalanceamento EE (Esquerda-Esquerda) ou ED (Esquerda-Direita), dependendo do Fator de Balanceamento do Filho Esquerdo de R. Se o Fator de Balanceamento do Filho Esquerdo de R for +1 (veja exemplo na [Figura 9.18](#)), desencadeamos o caso ED; caso contrário, desencadeamos o Caso EE (veja exemplo na [Figura 9.15](#)).

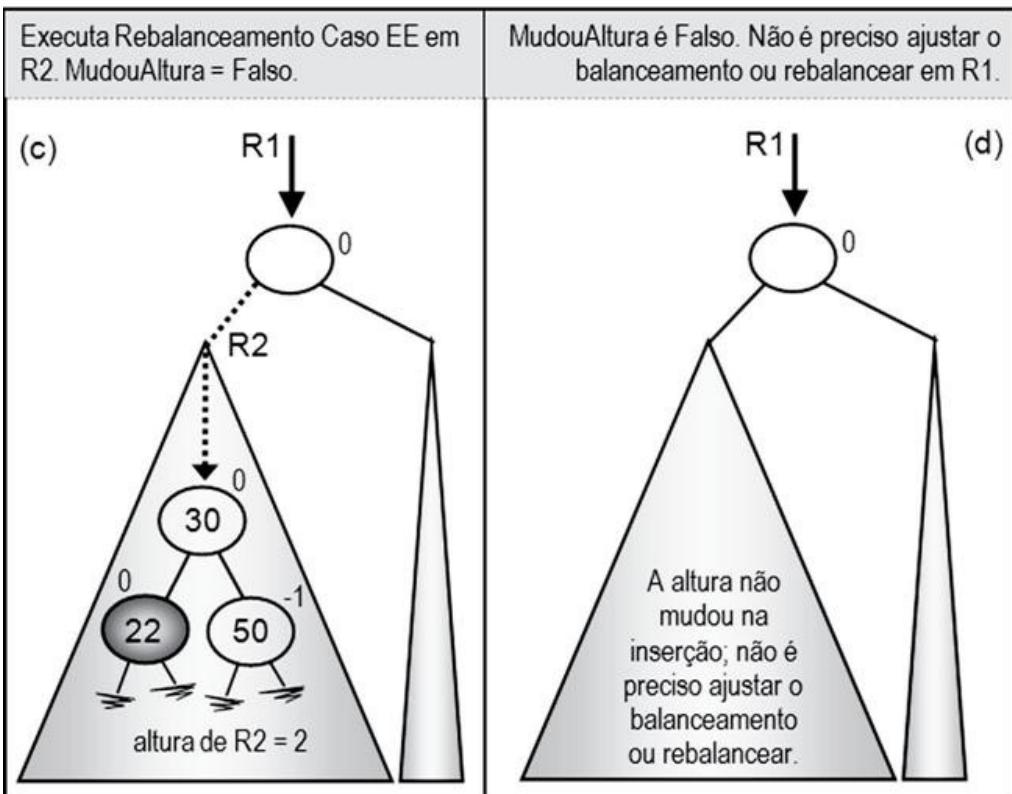
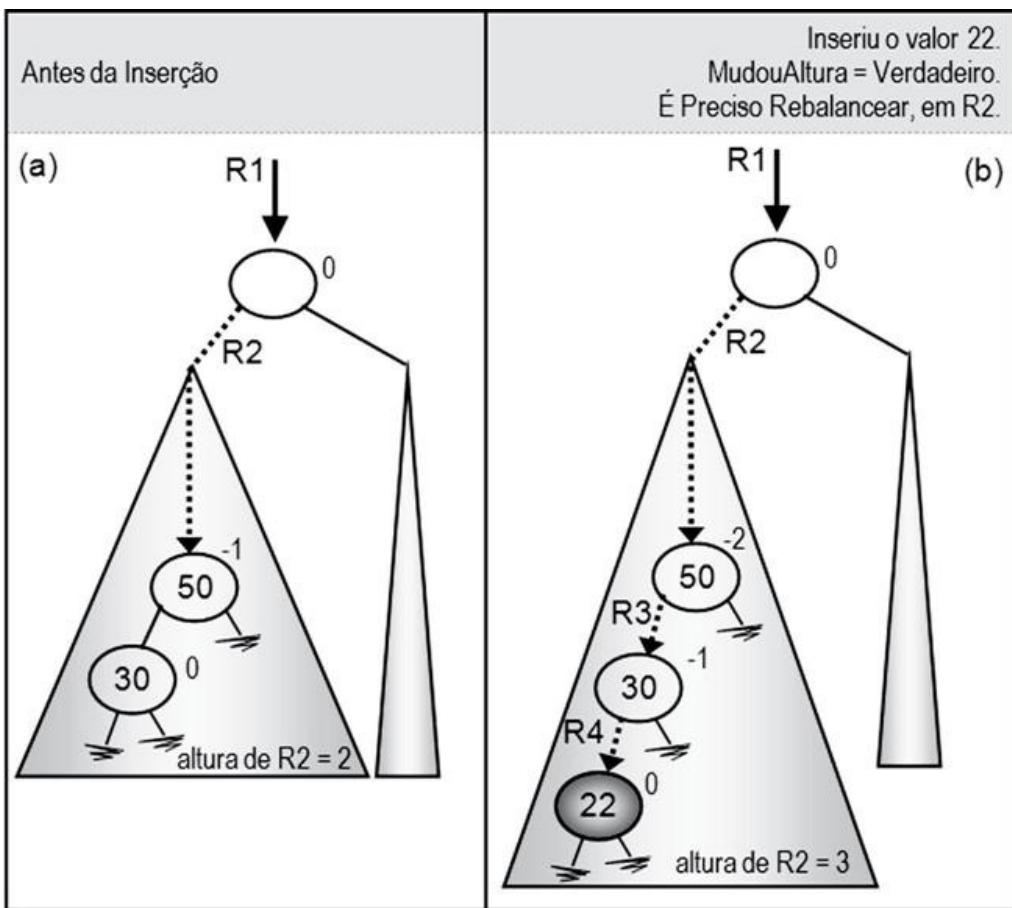
Analogamente, ao retornarmos de uma chamada recursiva para inserir um valor X na Subárvore Direita de R, podemos ajustar o Balanceamento de R ou desencadear o rebalanceamento pelos casos DE ou DD. Os algoritmos dos casos EE e ED foram apresentados nas [Figuras 9.14](#) e [9.22](#); os algoritmos dos casos DD e DE são objeto dos [Exercícios 9.9](#) e [9.15](#).

A variável MudouAltura recebe o valor Verdadeiro quando um novo Nó é inserido e o valor Falso quando detectamos que, dada a disposição dos demais Nós, a inserção do novo valor não alterou a altura da Árvore (veja a [Figura 9.6a](#)).

MudouAltura também recebe o valor Falso nos processos de rebalanceamento. Relembre nas [Figuras 9.13](#) e [9.14](#) que, antes da inserção do novo valor X, a

altura da Árvore era  $H(S1) + 2$  e após a inserção e o rebalanceamento pelo caso EE a altura da Árvore continuava sendo  $H(S1) + 2$ . Situação semelhante pode ser observada nas [Figuras 9.21 e 9.22](#), referentes ao caso ED.

Uma vez que a variável MudouAltura receber o valor Falso, não será mais necessário ajustar o balanceamento ou desencadear processos de rebalanceamento. Considere, por exemplo, a situação da [Figura 9.26](#). Antes da inserção do novo valor 22, a altura de R2 — ou seja, a altura da Árvore R na segunda chamada recursiva — era 2 ([Figura 9.26a](#)).



**FIGURA 9.26** Insere em ABBB: exemplo de execução.

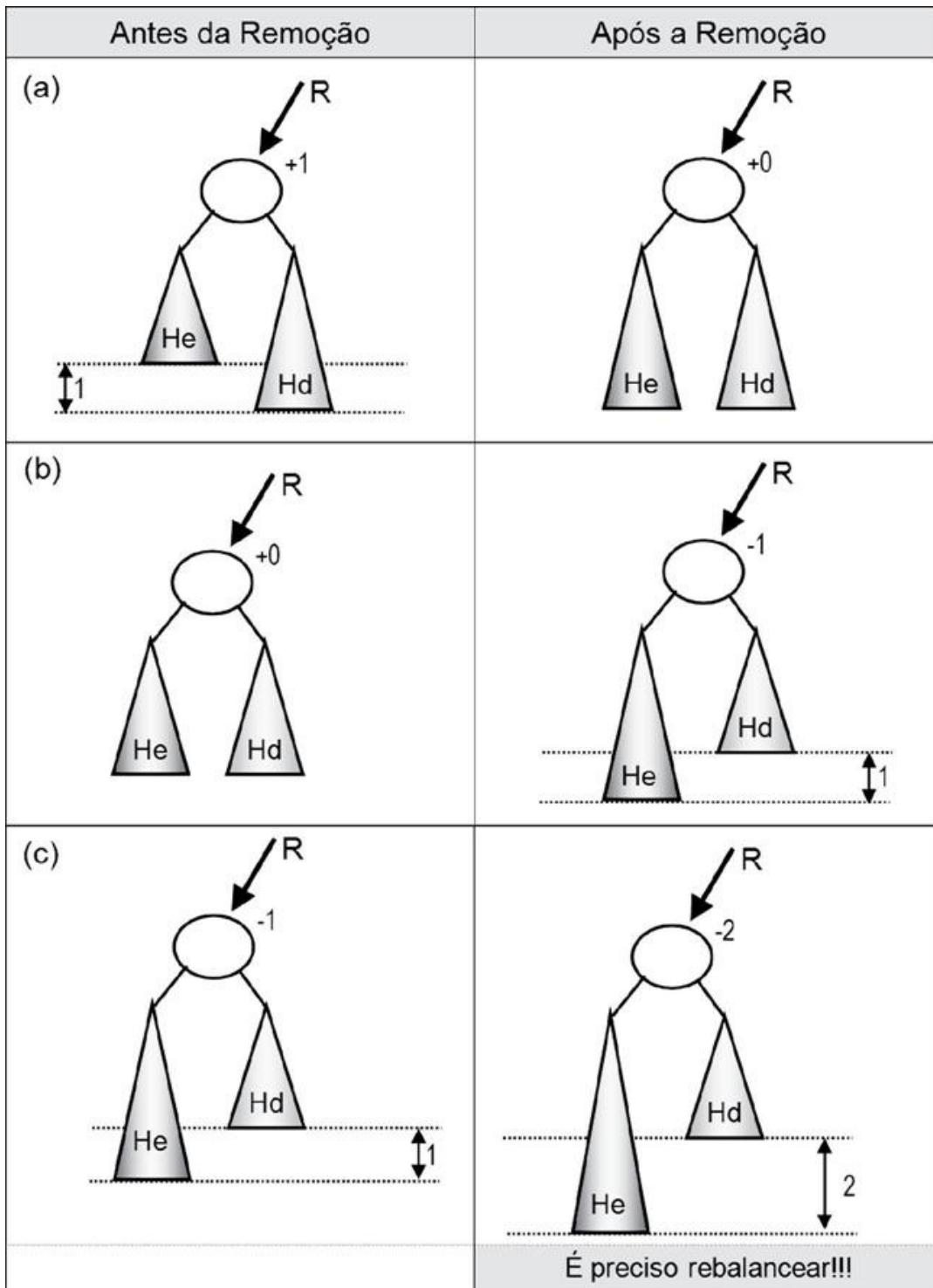
Em R4, ou seja, na quarta chamada recursiva, inserimos efetivamente o novo valor, 22, e com isso a variável MudouAltura recebe o valor Verdadeiro. Voltamos para R3 (terceira chamada recursiva) e ajustamos o Fator de Balanceamento de zero para -1. A variável MudouAltura continuou com o valor Verdadeiro, pois a altura de R3 passou de 1 para 2, mas não detectamos a necessidade de rebalancear em R3. Voltamos então para R2 (segunda chamada recursiva). Após a inserção, a altura de R2 passou a ser 3 e detectamos a necessidade de rebalancear ([Figura 2.26b](#)).

Em R2 executamos o rebalanceamento, caso EE (Rotação Simples Esquerda-Esquerda). Com a execução do rebalanceamento, a altura de R2 volta a ser 2. A altura de R2 antes da inserção era 2; após a inserção e o rebalanceamento, a altura de R2 voltou a ser 2; por isso, a variável MudouAltura recebe o valor Falso ([Figura 2.26c](#)).

Voltamos a R1 (primeira chamada recursiva) com a variável MudouAltura com o valor Falso. Isso significa que solicitamos a inserção de um novo valor na Subárvore Esquerda de R1 através de uma chamada recursiva. Após a execução dessa chamada recursiva, o novo valor foi inserido mas a altura da Subárvore Esquerda continua a mesma. Logo, o balanceamento de R1 não precisa ser ajustado. Como já mencionamos anteriormente, uma vez que a variável MudouAltura recebe o valor Falso, não é mais necessário ajustar o balanceamento ou desencadear processos de rebalanceamento.

## 9.3 Remover elementos de uma ABB Balanceada

Os ajustes necessários no algoritmo que remove um valor X de uma Árvore Binária de Busca Balanceada (ABBB) são análogos aos ajustes que realizamos no algoritmo Insere. Assim como fizemos na [Figura 9.6](#) para o algoritmo Insere, a [Figura 9.27](#) ilustra o ajuste do balanceamento de um Nô apontado por R após a remoção de um valor X, com diminuição da altura da Subárvore Direita.



**FIGURA 9.27** Monitorando o Balanceamento na remoção, com redução da Altura da Subárvore Direita.

Compare a [Figura 9.27](#) com a [Figura 9.6](#): os diagramas são idênticos, o que indica que a operação que remove um valor da Subárvore Direita tem

implicações muito parecidas com as implicações da operação que insere um novo valor na Subárvore Esquerda.

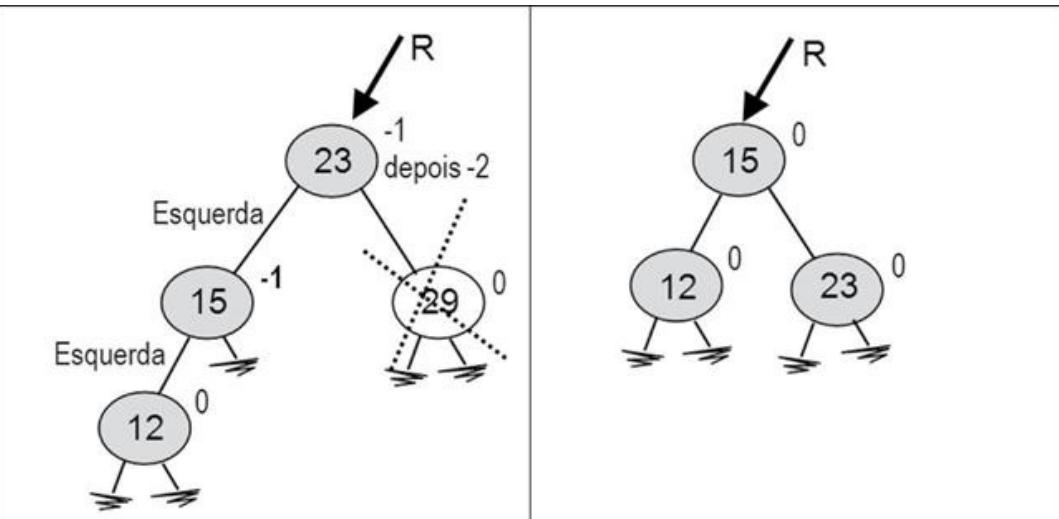
Na [Figura 9.27a](#), antes da remoção (lado esquerdo do diagrama), o Balanceamento de R tem valor +1. Isso significa que a Subárvore Direita de R era maior que a Subárvore Esquerda, em 1. A remoção de um valor da Subárvore Direita de R causou redução da altura  $H_d$ , que passou a ser igual a  $H_e$ . Após a remoção (lado direito do diagrama), o Balanceamento de R passou a ser zero.

Na [Figura 9.26b](#), antes da remoção, as alturas das Subárvores  $H_d$  e  $H_e$  eram iguais e o Balanceamento de R era zero. Após a remoção de um valor da Subárvore Direita de R,  $H_d$  passou a ser menor que  $H_e$  em 1 e o Balanceamento de R passou a ser -1.

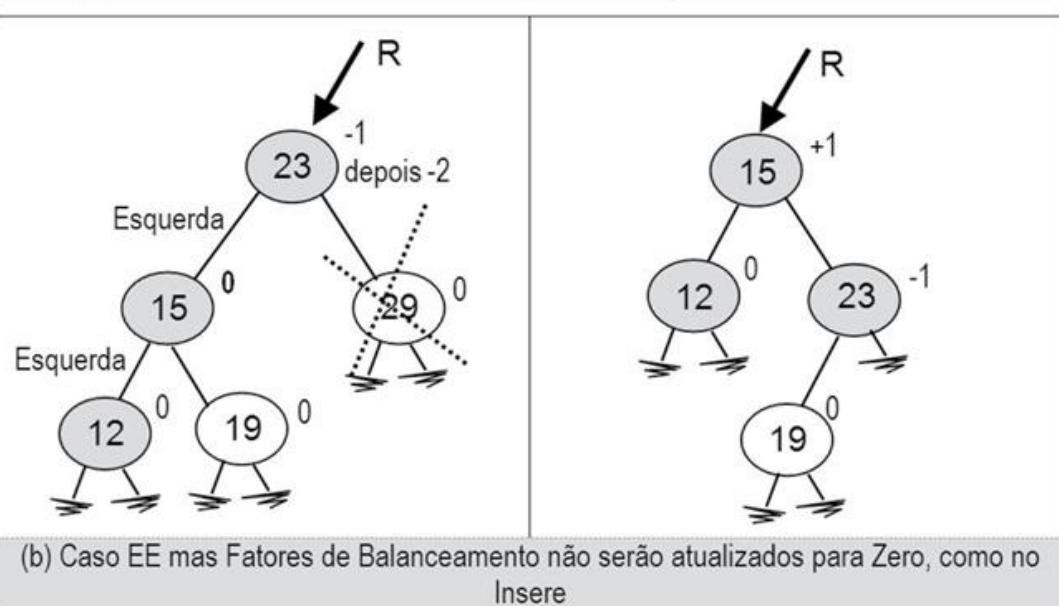
A situação inicial da [Figura 9.26c](#) mostra  $H_e$  maior que  $H_d$  (em 1) e, consequentemente, o Balanceamento de R tendo valor -1. Com a remoção de um valor da Subárvore Direita de R,  $H_d$  diminuiu e passou a ser menor que  $H_e$  em 2, o que quebra o critério de balanceamento. Assim, na situação da [Figura 9.26c](#), a operação de remoção implica a necessidade de rebalanceamento da Árvore.

## Casos de rebalanceamento

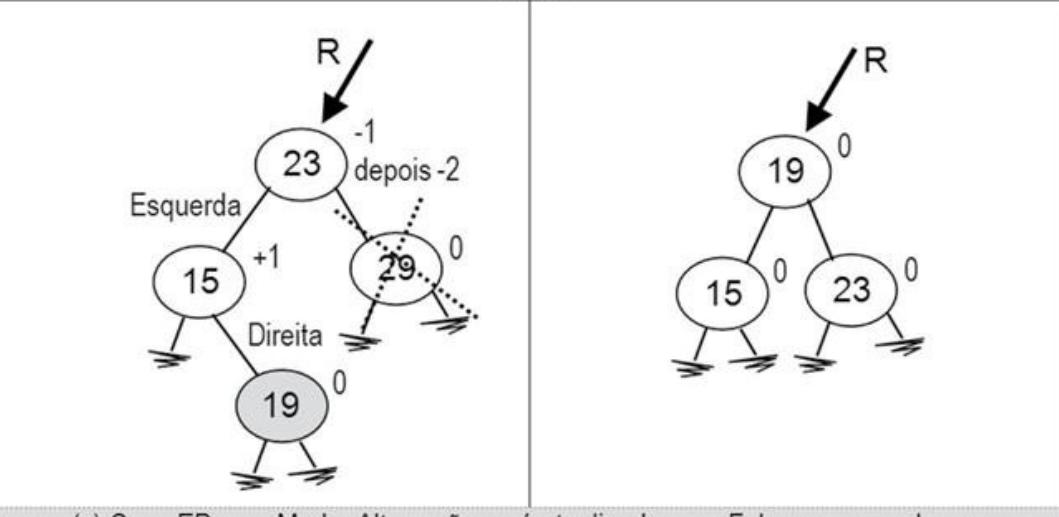
A analogia entre inserir um valor na Subárvore Esquerda e eliminar um valor da Subárvore Direita é válida também nos casos de rebalanceamento. Note, na [Figura 9.28a](#), que a remoção do valor 29 gera uma situação idêntica à da [Figura 9.9](#), que exemplifica o caso de rebalanceamento Rotação Simples do tipo EE do algoritmo Insere. Similarmente, na [Figura 9.28c](#), a remoção do valor 29 gera uma situação idêntica à da [Figura 9.18](#), que exemplifica o caso de rebalanceamento Rotação Dupla do tipo ED do algoritmo Insere.



(a) Caso EE mas MudouAltura não será atualizada para Falso, como no Insere



(b) Caso EE mas Fatores de Balanceamento não serão atualizados para Zero, como no Insere



(c) Caso ED mas MudouAltura não será atualizada para Falso, como no Insere

---

**FIGURA 9.28** Remove de ABBB: casos de rebalanceamento



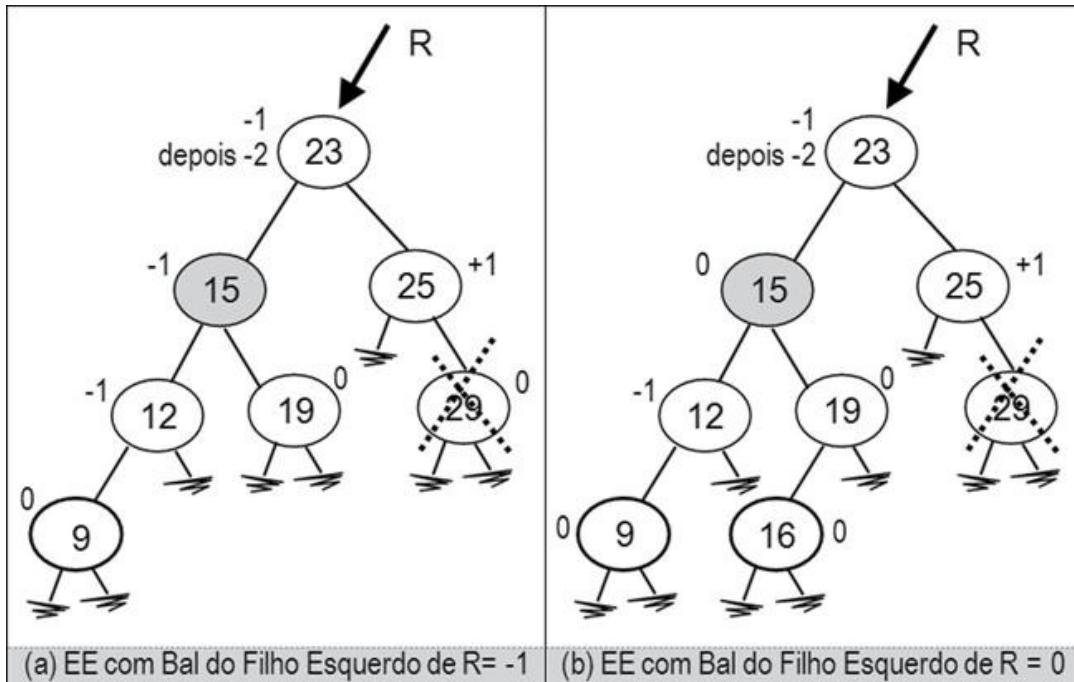
análogos aos do Insere, mas com ajustes.

Contudo, a Figura 9.28 mostra também que, embora os casos de rebalanceamento sejam, em essência, os mesmos que estudamos no algoritmo Insere EE, ED, DD e DE, alguns ajustes precisam ser realizados para sua aplicação na operação Remove. A movimentação das Subárvores e ponteiros é a mesma nos casos do Insere e do Remove, mas os ajustes nos valores do Fator de Balanceamento e no valor da variável MudouAltura precisam ser adaptados para a operação Remove. Por exemplo, na Figura 9.28a, ao contrário do que ocorre no caso EE do Insere, a variável MudouAltura precisa continuar com valor Verdadeiro, pois a altura mudou de 3 para 2 com a remoção e rebalanceamento. Essa diferença com relação à variável MudouAltura também ocorre no caso ED (Figura 9.28c).

Na Figura 9.28b, após o rebalanceamento, o Fator de Balanceamento do Nó que contém o valor 15 é +1, e o do Nó que contém o valor 23 é -1. No caso EE do Insere, o Fator de Balanceamento de todos os Nós movimentados passa a ser zero. Note que o Balanceamento do Filho Esquerdo de R na Figura 9.28a é -1 e na Figura 9.28b é zero. No Insere, no caso EE, o rebalanceamento do Filho Esquerdo de R sempre será -1.

### **Exercício 9.16 Remove ABBB: rebalanceamento — caso EE**

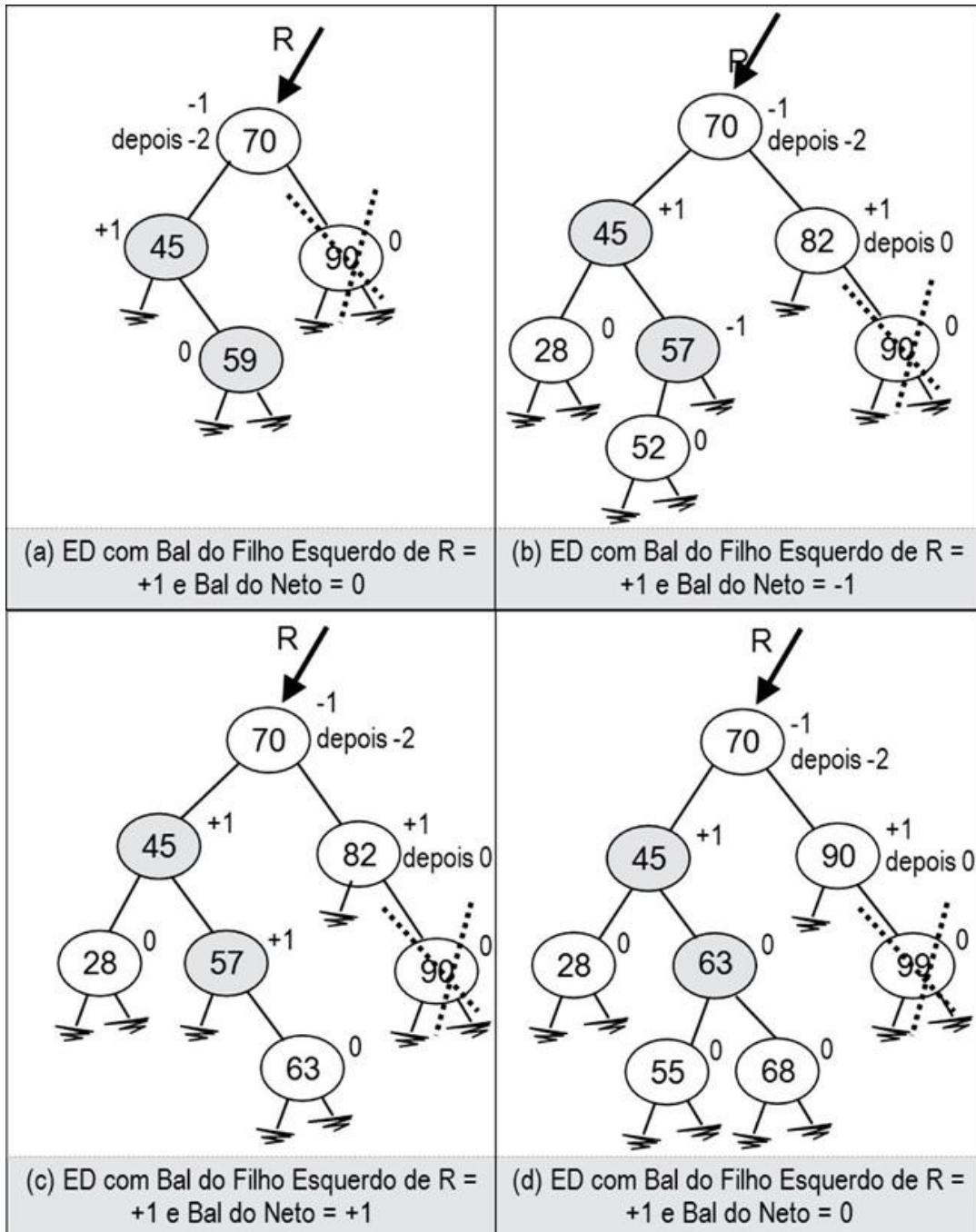
Aplique manualmente o caso de rebalanceamento EE do Remove nos exemplos numéricos da Figura 9.29. A movimentação das Subárvores e ponteiros deve ser idêntica ao que ocorre no caso EE do Insere, mas os valores do Fator de Balanceamento e da variável MudouAltura devem ser ajustados. Aplique o caso EE do Remove quando o Balanceamento do Filho Esquerdo de R for -1 ou zero.



**FIGURA 9.29** Remove de ABBB: exemplos do caso 5 — rotação simples EE do Remove.

### Exercício 9.17 Remove de ABBB: rebalanceamento manual ED

Aplique manualmente o caso de rebalanceamento ED do Remove nos exemplos numéricos da [Figura 9.30](#). A movimentação das Subárvores e ponteiros deve ser idêntica ao que ocorre no caso ED do Insere, mas os valores do Fator de Balanceamento e da variável MudouAltura devem ser ajustados. Aplique o caso ED do Remove somente quando o Balanceamento do Filho Esquerdo de R for +1.



**FIGURA 9.30 Remove de ABBB: exemplos do caso 7 — Rotação Dupla ED do Remove.**

## Exercício 9.18 Generalização do caso 5: Rotação Simples EE do Remove — diagrama e algoritmo

Analogamente ao que fizemos para o algoritmo Insere nas Figuras 9.13 e 9.14, faça um diagrama generalizando a Rotação Simples EE do Remove e desenvolva

um algoritmo que implemente esse caso de rebalanceamento.

### **Exercício 9.19 Generalização do caso 6: Rotação Simples DD do Remove — diagrama e algoritmo**

O caso DD do Remove é simétrico ao caso EE do Remove. Faça um diagrama generalizando a Rotação Simples DD do Remove e desenvolva um algoritmo que implemente esse caso de rebalanceamento.

### **Exercício 9.20 Generalização do caso 7: Rotação Dupla ED do Remove — diagrama e algoritmo**

Analogamente ao que fizemos para o algoritmo Insere nas [Figuras 9.21](#) e [9.22](#), faça um diagrama generalizando a Rotação Dupla ED do Remove e desenvolva um algoritmo que implemente esse caso de rebalanceamento.

### **Exercício 9.21 Generalização do caso 8: Rotação Dupla DE do Remove — algoritmo**

O caso DE do Remove é simétrico ao caso ED do Remove. Desenvolva um algoritmo generalizando a Rotação Dupla DE do Remove.

### **Exercício 9.22 Algoritmo Remove para uma Árvore Binária de Busca Balanceada (ABBB)**

No [Capítulo 8](#) elaboramos um algoritmo para remover um valor X de uma Árvore Binária de Busca — ABB ([Exercício 8.11](#)). Adapte esse algoritmo fazendo-o monitorar o balanceamento da Árvore e desencadear ações de rebalanceamento, sempre que necessário, de modo a manter a Árvore sempre balanceada. Para realizar o monitoramento do balanceamento, tome como ponto de partida o diagrama da [Figura 9.27](#). Como ações de rebalanceamento, considere os casos EE, DD, ED e DE do Remove, objeto dos [Exercícios 9.18](#) a [9.21](#).

```
Remove (parâmetro por referência R tipo ABBB, parâmetro X tipo Inteiro, parâmetro por referência Ok tipo Boolean, parâmetro por referência MudouAltura tipo Boolean) {
```

```
/* Remove o valor X da ABBB de Raiz R. Monitora o balanceamento e dispara processos de rebalanceamento, sempre que necessário, para manter a Árvore sempre balanceada. Ok retorna Verdadeiro se X for encontrado e removido, e Falso caso contrário. MudouAltura retorna Verdadeiro se na remoção de X a altura da Árvore diminuiu e Falso caso contrário. */
```

## **Exercício 9.23 Implemente uma Árvore Binária de Busca Balanceada (ABBB) em uma linguagem de programação**

Implemente uma Árvore Binária de Busca Balanceada (ABBB) como um Tipo Abstrato de Dados, com operações Cria, Vazia, Insere, Remove e Destroi. Implemente em uma linguagem de programação, como C++.

### **Desempenho depende de balanceamento!**

Para que uma Árvore Binária de Busca mantenha seu excelente desempenho durante todo o tempo, ela precisa estar balanceada durante todo o tempo. Para isso, os algoritmos para inserir e eliminar elementos precisam monitorar o balanceamento da Árvore e desencadear operações de rebalanceamento sempre que necessário.

### **Consulte nos Materiais Complementares**

Vídeos sobre Árvores Balanceadas



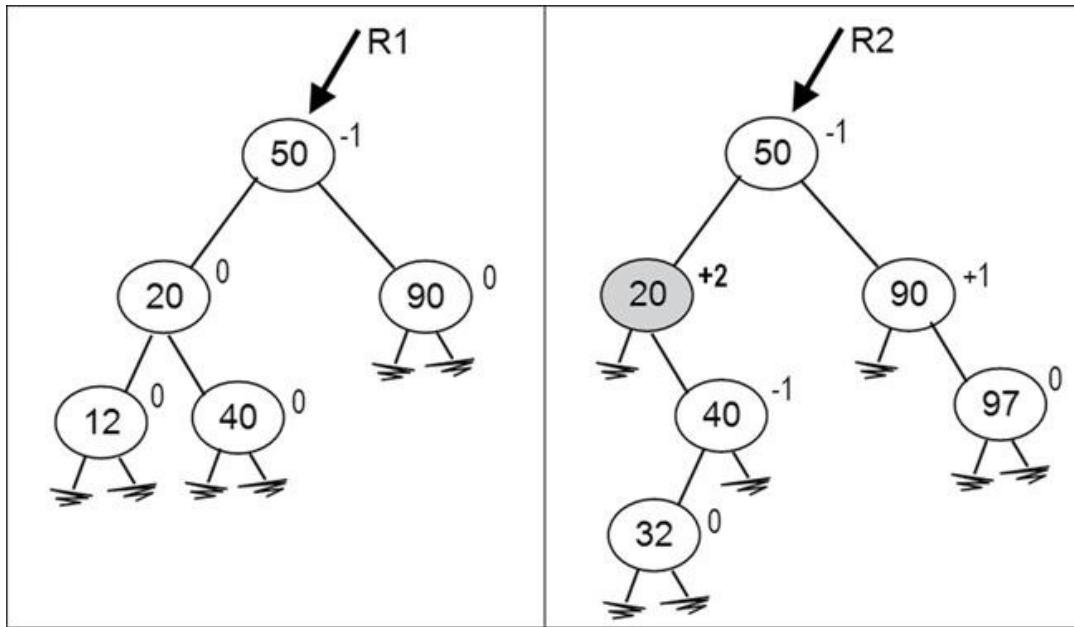
<http://www.elsevier.com.br/edcomjogos>

## **Exercícios de fixação**

**Exercício 9.24** Execute algum simulador de operações em uma Árvore Binária de Busca Balanceada, como, por exemplo, o Tree Explorer (link 4). Execute operações para inserir e eliminar elementos.

## **Soluções para alguns dos exercícios**

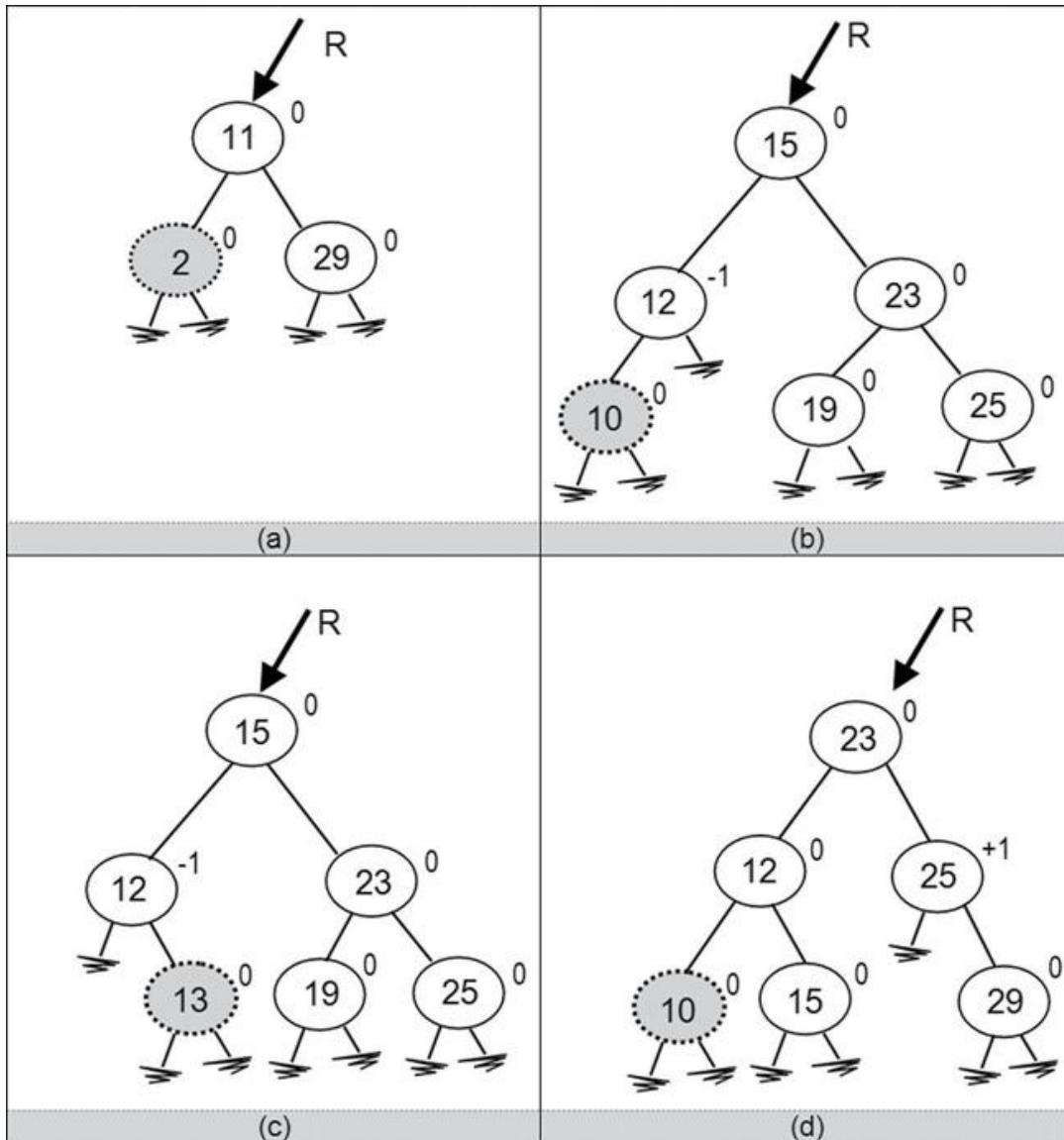
### **Exercício 9.3 Fator de Balanceamento**



### Exercício 9.4 Causaria desbalanceamento?

A inserção de 25 e de 40 não causaria desbalanceamento. A inserção dos demais valores causaria.

### Exercício 9.6 Rebalanceamento manual



No caso *d*, note que o Nó em que o desbalanceamento foi detectado é o Nó que contém o valor 15, e não o Nó apontado por R.

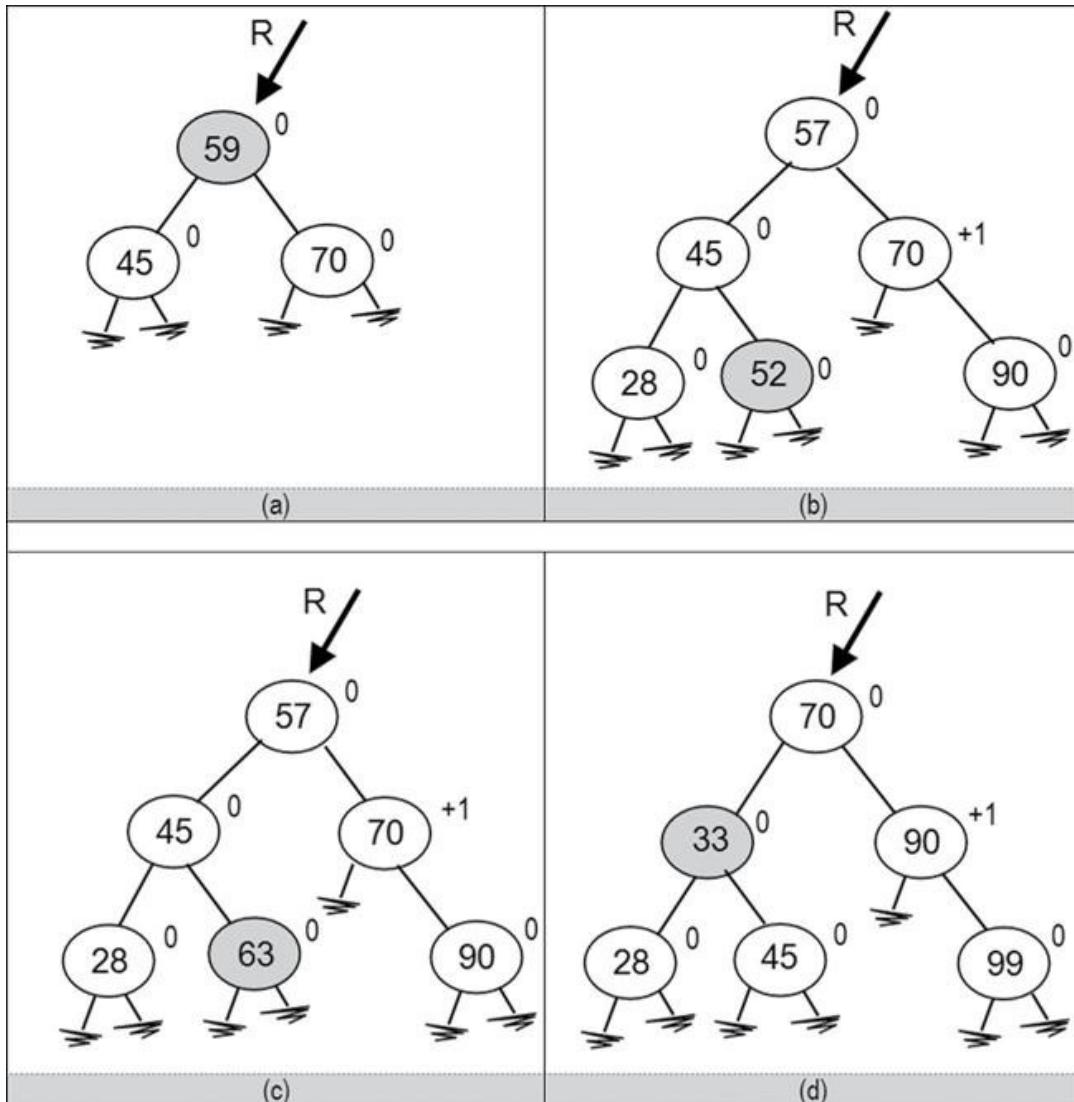
### **Exercício 9.9 Generalização do caso 2 — Rotação Simples DD do Insere — algoritmo**

```

/* caso absolutamente simétrico ao caso EE; o que era Esq passa a ser Dir e vice-versa */
variável Filho do tipo NodePtr;           // Filho é ponteiro auxiliar, que apontará R→Dir
/* movimentando as Subárvores e os ponteiros */
Filho = R→Dir;
R→Dir = Filho→Esq;
Filho→Esq = R;
/* ajustando os balanceamentos */
R→BAL = 0;                      // atualizando o Fator de Balanceamento de R
Filho→Bal = 0;                   // atualizando o Fator de Balanceamento de Filho
/* mudando a Raiz da árvore*/
R = Filho;
/* atualizando a variável MudouAltura */
MudouAltura = Falso;            // após inserir X e rebalancear, a altura da Árvore...
                                // ... continua a mesma: H(S1) + 2.

```

## Exercício 9.11 Rebalanceamento manual — ED



### Exercício 9.14 Generalização do caso 4: Rotação Dupla DE — Insere — algoritmo

```

/* caso simétrico ao caso ED; o que era Esq passa a ser Dir e vice-versa */
variável Filho do tipo NodePtr;
variável Neto do tipo NodePtr;

/* posicionando os ponteiros Filho e Neto */
Filho = R→Dir;
Neto = Filho→Esq;

/* movimentando a Subárvore S2 */
Filho→Esq = Neto→Dir;
Neto→Dir = Filho;

/* movimentando a Subárvore S3 */
R→Dir = Neto→Esq;
Neto→Esq = R;

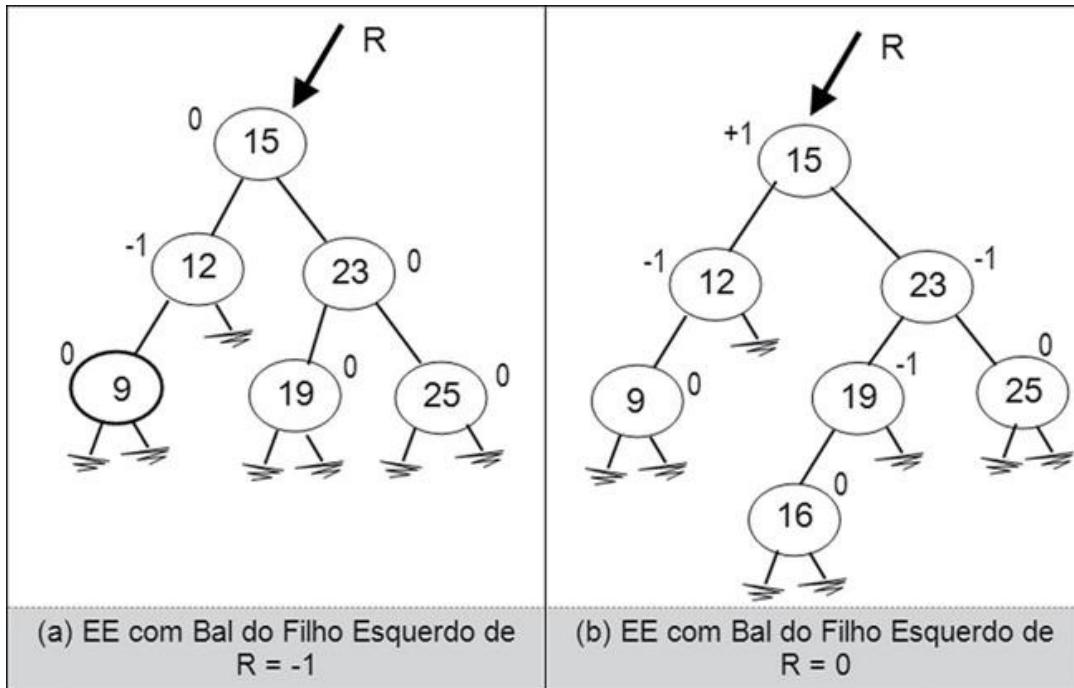
/* ajustando os balanceamentos */
Caso Neto→Bal for
    -1 : { R→Bal = 0;           // inseriu X2.
            Filho→Bal = 1;
            Neto→Bal = 0; };
    +1 : { R→Bal = -1;          // inseriu X1.
            Filho→Bal = 0;
            Neto→Bal = 0; };
    0 : { R→Bal = 0;           // inseriu o Nó apontado por Neto; caso com apenas três valores
            Filho→Bal = 0;
            Neto→Bal = 0; };

/* mudando a Raiz da árvore*/
R = Neto;           // o Nó que contém 'B' passará a ser a Raiz - Figura 2.21

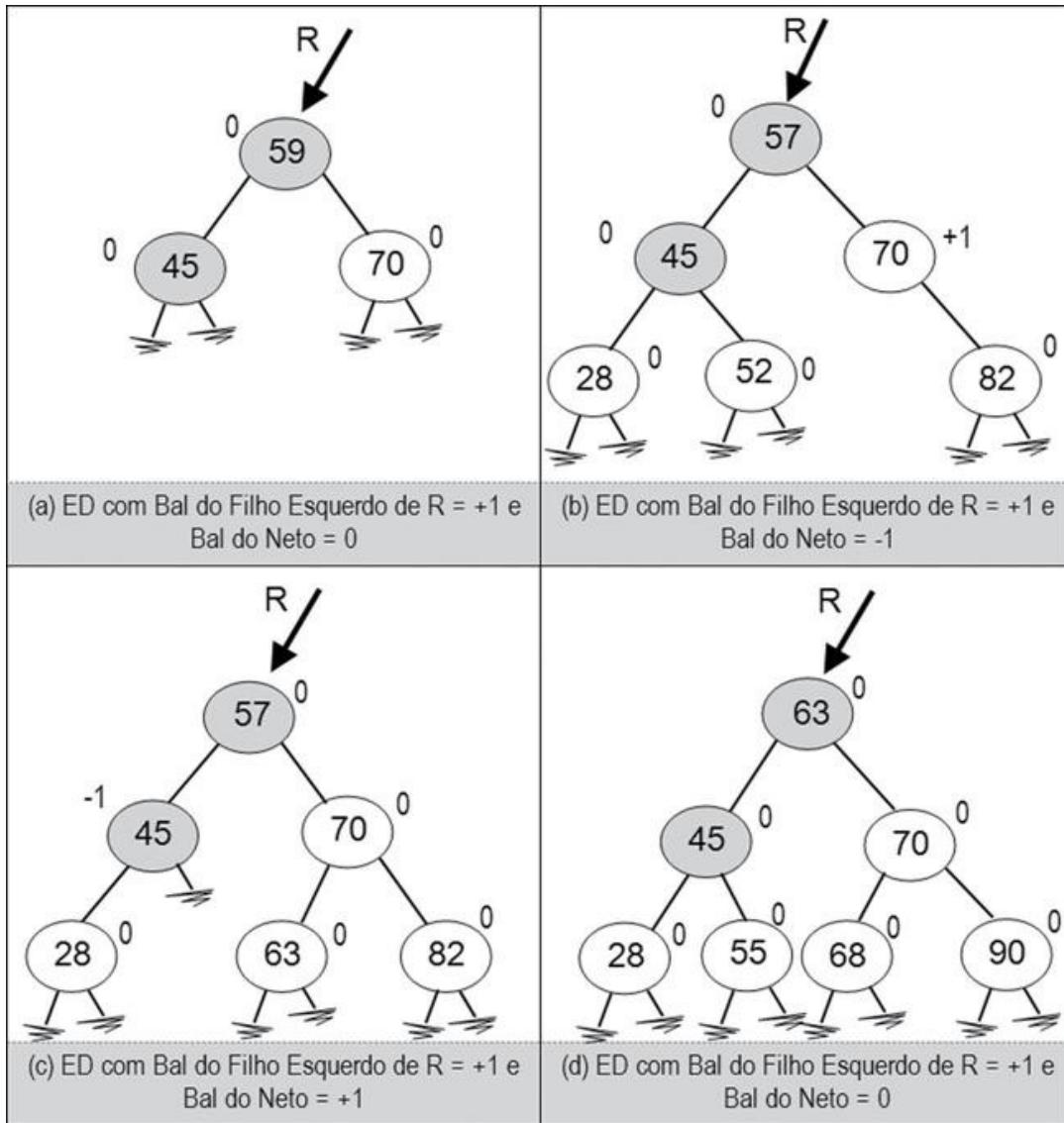
/* atualizando a variável MudouAltura */
MudouAltura = Falso;      // após inserir X1 ou X2 e rebalancear a altura da Árvore continua a
mesma: H(S1) + 2.

```

## Exercício 9.16 Remove de uma ABBB: rebalanceamento manual — caso EE



### Exercício 9.17 Remove de ABBB: rebalanceamento manual — caso ED



## Exercício 9.18 Generalização do caso 5: Rotação Simples EE do Remove — diagrama e algoritmo

Dividimos o caso EE do Remove em dois subcasos, para mostrar algumas diferenças: caso EE(a): Quando  $\text{Filho} \rightarrow \text{Bal} = -1$ ; caso EE(b): Quando  $\text{Filho} \rightarrow \text{Bal} = \text{zero}$ . Para exemplificar as diferenças, note que, quando  $\text{Bal do Filho}$  é  $-1$ , a altura da Árvore muda; quando o  $\text{Bal do Filho}$  é zero, a altura da Árvore não muda. Observe também os Balanceamentos.

Caso EE (a) do Remove: quando  $\text{Filho} \rightarrow \text{Bal} = -1$

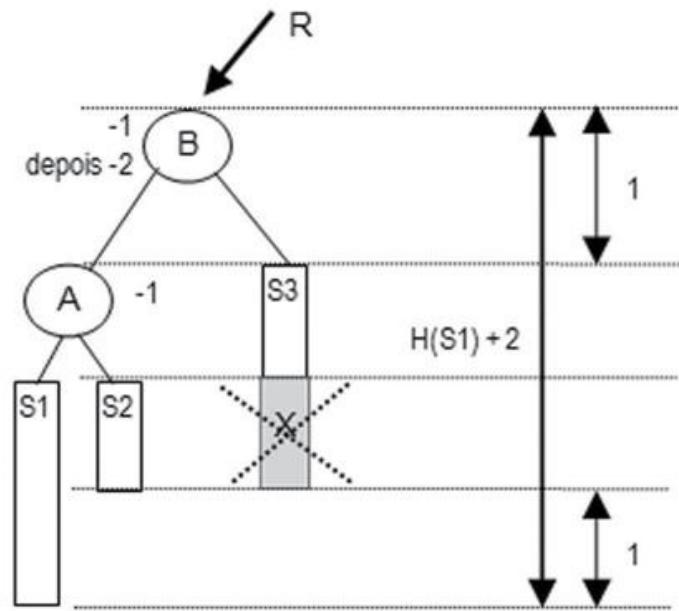
```

variável Filho do tipo NodePtr;           // Filho é ponteiro auxiliar, que apontará R→Esq
/* movimentando as Subárvores e os ponteiros */
Filho = R→Esq;
Se (Filho→Bal == -1)
Então {   R→Esq = Filho→Dir;
            Filho→Dir = R;
            /* ajustando os balanceamentos */
            R→BAL = 0;    // atualizando o Fator de Balanceamento de R
            Filho→Bal = 0; // atualizando o Fator de Balanceamento de Filho
            /* mudando a Raiz da árvore*/
            R = Filho;
            /* a variável MudouAltura continua Verdadeiro */
            MudouAltura = Verdadeiro;  }

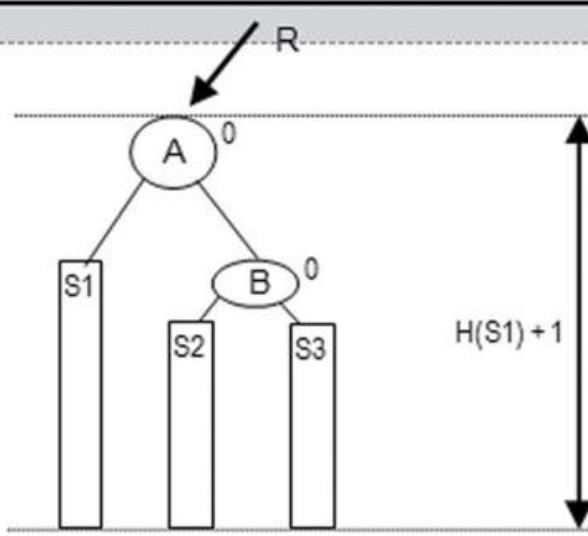
```

Caso EE (a) do Remove: quando Filho → Bal = -1

(a) Elimina X e é preciso Rebalancear;  $H(S2) = H(S3)$ ;  $H(S1) = H(S2) + 1$



(b) Após Rebalancear



\* Estilo de diagramas adaptado de [Knuth \(1998\)](#) e Camargo.  
Caso EE (b) do Remove: quando Filho → Bal = 0

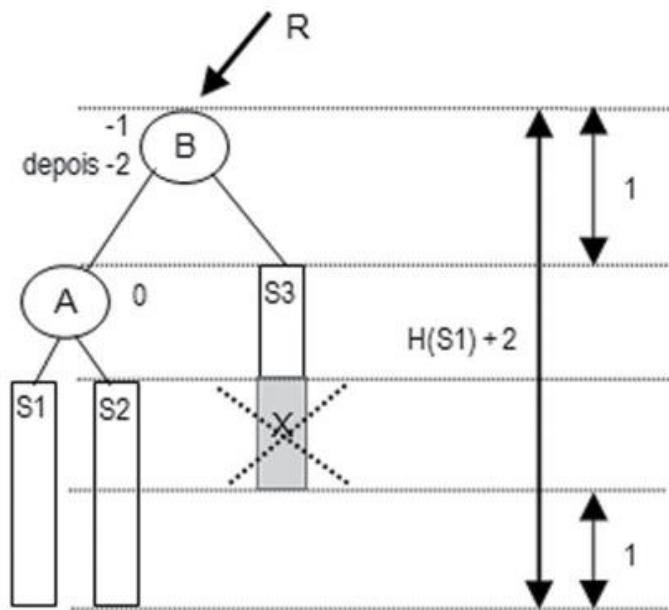
```

variável Filho do tipo NodePtr;           // Filho é ponteiro auxiliar, que apontará R→Esq
/* movimentando as Subárvores e os ponteiros */
Filho = R→Esq;
Se (Filho→Bal == 0)
Então {   R→Esq = Filho→Dir;
            Filho→Dir = R;
            /* ajustando os balanceamentos */
            R→BAL = -1;    // atualizando o Fator de Balanceamento de R
            Filho→Bal = +1; // atualizando o Fator de Balanceamento de Filho
            /* mudando a Raiz da árvore*/
            R = Filho;
            /* atualizando a variável MudouAltura */
            MudouAltura = Falso;    }

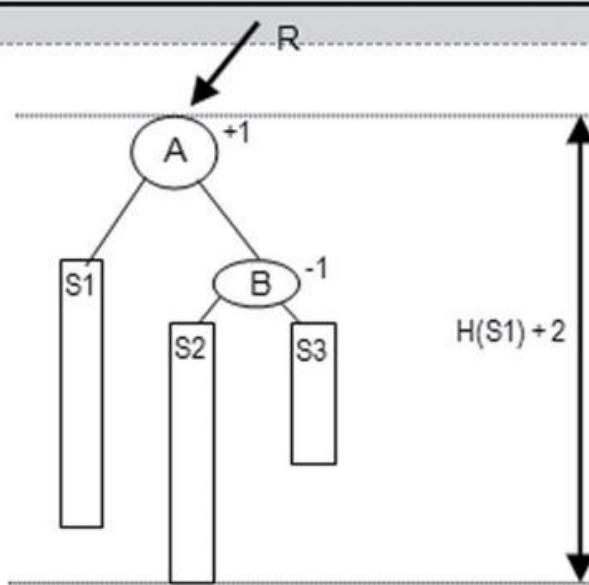
```

Caso EE (b) do Remove: quando Filho → Bal = 0

(a) Elimina X e é preciso Rebalancear;  $H(S1) = H(S2); H(S1) = H(S3) + 1$



(b) Após Rebalancear



\* Estilo de diagramas adaptado de [Knuth \(1998\)](#) e Camargo.

## Exercício 9.20 Generalização do caso 7: Rotação Dupla ED do Remove — algoritmo

```

variável Filho do tipo NodePtr;      // Filho é ponteiro auxiliar, que apontará R→Esq
variável Neto do tipo NodePtr;      // Neto é ponteiro auxiliar, que apontará Filho→Dir

Filho = R→Esq;          /* posicionando os ponteiros Filho e Neto */
Neto = Filho→Dir;

Filho→Dir = Neto→Esq;      /* movimentando a Subárvore S2 */
Neto→Esq = Filho;

R→Esq = Neto→Dir;      /* movimentando a Subárvore S3 */
Neto→Dir = R;

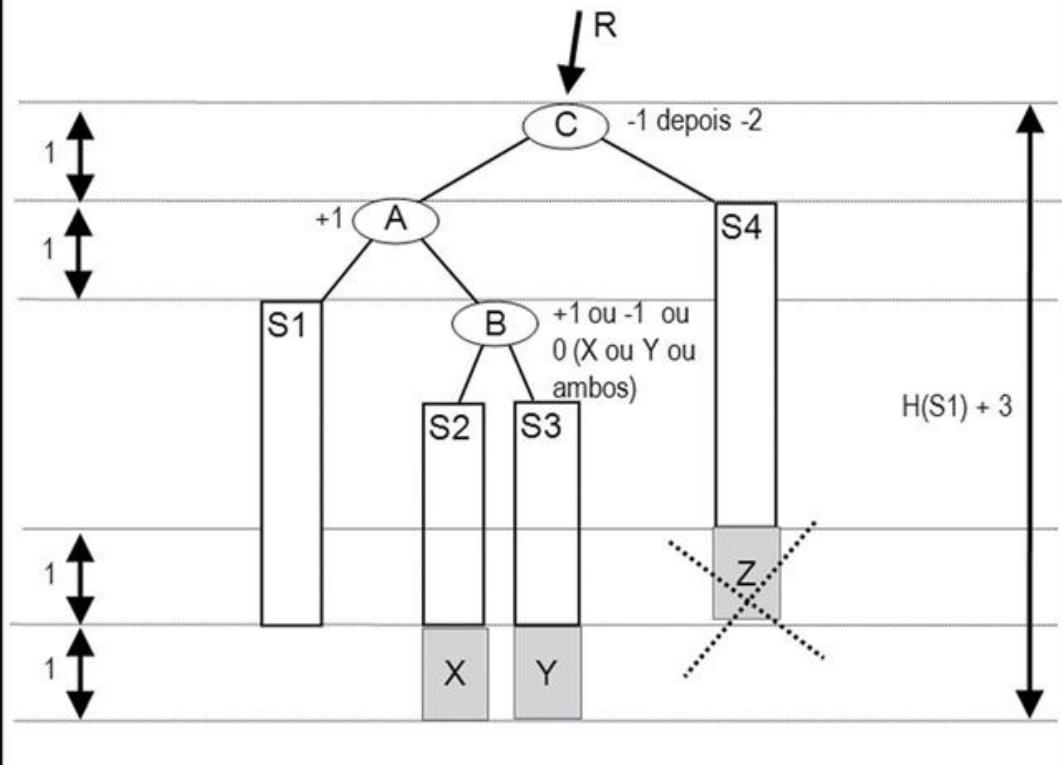
Caso Neto→Bal for      /* ajustando os balanceamentos */
-1 : { R→Bal = 1;
        Filho→Bal = 0;
        Neto→Bal = 0; };
+1 : { R→Bal = 0;
        Filho→Bal = -1;
        Neto→Bal = 0; };
0 : { R→Bal = 0;
        Bal (Filho) = 0;
        Neto→Bal = 0; };

R = Neto;    /* mudando a Raiz da árvore*/
MudouAltura = Verdadeiro; /* variável MudouAltura - continua Verdadeiro */

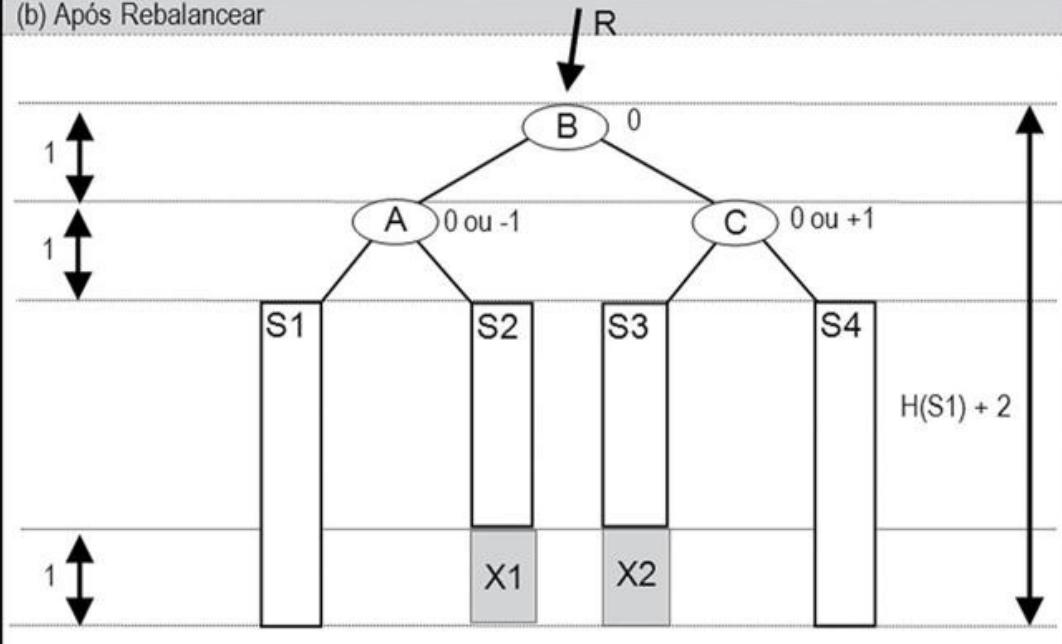
```

## Exercício 9.20 Generalização do caso 7: Rotação Dupla ED do Remove — diagrama

(a) Elimina Z e é preciso Rebalancear;  $H(S1) = H(S4)$ ;  $H(S2) = H(S3)$ ;  $H(S1) = H(S2) + 1$



(b) Após Rebalancear



\* Estilo de diagramas adaptado de [Knuth \(1998\)](#) e Camargo.

## Exercício 9.22 Algoritmo Remove para uma Árvore Binária

## de Busca Balanceada (ABBB)

```
/* Esse algoritmo foi elaborado tendo como base, em alguns aspectos, o algoritmo de Camargo, p. 10-13 */
Remove (parâmetro por referência R do tipo ABBB, parâmetro X do tipo Inteiro, parâmetro por
referência Ok do tipo Boolean, parâmetro por referência MudouAltura tipo Boolean) {
    /* Remove o valor X da ABB de Raiz R. Monitora o balanceamento e dispara processos de
    rebalanceamento, sempre que necessário, para manter a Árvore sempre balanceada. Ok retorna
    Verdadeiro se X for encontrado e removido, e Falso caso contrário. MudouAltura retorna
    Verdadeiro se na remoção de X a altura da Árvore diminuiu e Falso caso contrário */

    variável Aux do tipo NodePtr;           // NodePtr = Ponteiro para Nó;

    Se (R == Null)
        Então // Caso 1: Árvore Vazia: não remove.
        { Ok = Falso; MudouAltura = Falso; }

    Senão Se (R→Info > X)
        Então // Caso 3: remove X da Subárvore Esq de R
        { Remove (R→Esq, X , Ok, MudouAltura);

            /* Monitora balanceamento após eliminar da Subárvore Esquerda */
            Se MudouAltura
                Então Caso R→Bal for:
                    -1: R→Bal = 0; // MudouAltura continua Verdadeiro
                    0: { R→Bal = 1; MudouAltura = Falso;};
                    1: { Filho = R→Dir;    // vai precisar balancear
                        Se (Filho→Bal == -1)
                            Então RotDuplaDE-Remove
                        Senão RotSimplesDD-Remove; // casos Filho→Bal = 0 e +1
                    }; // fim do Caso
            } // fim do Caso 3
        Senão Se (R→Info < X)
            Então // Caso 4: remove X da Subárvore Dir de R
            { Remove(R→Dir, X, Ok, MudouAltura);

                /* Monitora balanceamento após eliminar da Subárvore Direita */
                Se MudouAltura
                    Então Caso R→Bal for:
                        1: R→Bal = 0; // MudouAltura continua Verdadeiro
                        0: { R→Bal = -1; MudouAltura = Falso;};
                        -1: { Filho= R→Esq; // vai precisar balancear
                            Se (Filho→Bal == 1)
                                Então RotDuplaED-Remove
                            Senão RotSimplesEE-Remove; // Filho→Bal=0 e -1
                        }; // fim do Caso
            } // fim do Caso 4
        Senão
    /* Caso 2: Encontrou X - Remove e Ajusta a Árvore. 3 casos: 0, 1 ou 2 Filhos */
    { Aux = R;
```

```

Se ((R→Esq == Null) E (R→Dir == Null ))
Então { DeleteNode(Aux); R=NULL; Ok=Verdadeiro; MudouAltura=Verdadeiro; } // Zero Filhos
Senão Se ((R→Dir != Null) E (R→Esq != Null))
    Então { /* 2 Filhos. Acha o Nó que contém o Maior Elemento da Subárvore Esquerda de R. O maior é o elemento
        mais à direita da Subárvore. Ele nunca terá o Filho Direito */
        Aux = R→Esq;
        Enquanto (Aux→Dir != Null) Faça
            Aux = Aux→Dir;
    }

    /* Substitui o valor de R→Info - que é o elemento que estamos querendo eliminar - pelo valor do
       Maior da Subárvore Esquerda de R. A Árvore ficará com 2 elementos com o mesmo valor. */
    R→Info = Aux→Info; // Aux aponta para o Nó que contém o Maior

    /* Remove o valor repetido da Subárvore Esquerda de R através de uma chamada recursiva.
       Atenção aos parâmetros. Não estamos mais removendo X, mas R→Info, que está repetido.
       Não estamos mais removendo de R, mas de R→Esq. */
    Remove(R→Esq, R→Info, Ok, MudouAltura);

    /* Monitora balanceamento após eliminar da Subárvore Esquerda */
    Se MudouAltura
        Então Caso R→Bal for:
            -1: R→Bal = 0; // MudouAltura continua Verdadeiro
            0: { R→Bal = 1; MudouAltura = Falso;};
            1: { Filho = R→Dir; // vai precisar balancear
                  Se (Filho→Bal == -1)
                      Então RotDuplaDE-Remove
                  Senão RotSimplesDD-Remove; // casos Filho→Bal = 0 e +1
            }; // fim do Caso
        // Ok e MudouAltura só são ajustadas nos casos com 0 e 1 Filhos
    } // fim - 2 Filhos

    Senão
        { // 1 Único Filho
            Se (R→Esq == Null)
                Então R = R→Dir; // "puxa" o Filho Direito; Filho esquerdo é nulo
            Senão R = R→Esq; // "puxa" o Filho Esquerdo; Filho direito é nulo
            DeleteNode (Aux);
            Ok=Verdadeiro;
            MudouAltura=Verdadeiro;
        } // fim 1 único filho
    } // fim Remove
}

```

## Links

1. Buricea, M. *Height Balanced Trees*. Lecture Notes. Universitatea Din Craiova. Disponível em: <http://software.ucv.ro/~mburicea/lab6ASD.pdf> (acesso em: novembro de 2013).
2. Devadas S.; Daskalakis, K.; Dzunic, Z.; Onak, K.; Schwendner, A. Singh, R. *Introduction to Algorithms*. Llecture Notes. Massachusetts Institute of Technology, 2009. Disponível em: [http://courses.csail.mit.edu/6.006/fall09/lecture\\_notes/lecture04.pdf](http://courses.csail.mit.edu/6.006/fall09/lecture_notes/lecture04.pdf) (acesso

- em: novembro de 2013).
3. Leser, U. *Algorithms and Data Structures*. AVL: Balanced Search Trees. Lecture Notes. Humboldt Universitat, Zu Berlin, 2011. Disponível em [http://www.informatik.hu-berlin.de/forschung/gebiete/wbi/teaching/archive/ss11/vl\\_algorithmen/15\\_avl\\_](http://www.informatik.hu-berlin.de/forschung/gebiete/wbi/teaching/archive/ss11/vl_algorithmen/15_avl_) (acesso em: novembro de 2013).
  4. Rocha. V.; Vervloet, M.; Franco, A.; Andrade, C. A. *Tree Explorer*. EDGames, 2013. Disponível em: <http://edgames.dc.ufscar.br> (acesso em: setembro de 2013).

## Referências e leitura adicional

1. Adelson-Velskii, G. M.; Landis, E. M. *An Algorithm for the Organization of Information*. Soviet Mathematics 3 (1962), 1259-1263 - conforme citado por Drozdek (2002). p. 232 e 268.
2. Camargo, H. A. *Apostila da disciplina Estruturas de Dados*. Departamento de Computação e Estatística. Universidade Federal de São Carlos, s/d.
3. Drozdek A. *Estruturas de dados e algoritmos em C++*. São Paulo: Thomson; 2002.
4. Knuth D. The Art of Computer Programming. In: *V3 Sorting and Searching*. 2. ed. Reading, MA: Addison- Wesley; 1998:458–481.
5. Langsam, Y; Augenstein, M. J.; Tenenbaum, A. M. *Data Structures Using C and C++*. 2nd ed. Upper Saddle River. New Jersey: Prentice Hall, 1996. p. 413-423.

---

# Seu próximo desafio

---

No livro *Estruturas de dados com jogos* você conheceu quatro Estruturas de Dados fundamentais: Pilhas, Filas, Listas Cadastrais e Árvores, além de diversas variações. São Estruturas de Dados para representação e armazenamento de conjuntos de informações em um programa.

Você aprendeu a implementar essas e outras estruturas com diversas técnicas de programação. Por exemplo, você utilizou o conceito de Alocação Encadeada e também o conceito de Alocação Dinâmica de Memória. As Listas Encadeadas são uma ferramenta realmente poderosa de programação. Continue utilizando. Continue praticando! Continue desenhando as estruturas ao elaborar e testar os algoritmos.

Você recebeu diversas dicas sobre como dar portabilidade e reusabilidade aos softwares que desenvolve, em especial os referentes à adoção do conceito de Tipos Abstratos de Dados. Em seus novos projetos, não se esqueça: manipule Estruturas de Dados exclusivamente pelos seus operadores. Aumente o volume da televisão pelos botões da televisão!

Você foi desafiado a desenvolver quatro jogos — aplicações de quatro Estruturas de Dados fundamentais que estudamos: Pilhas, Filas, Listas Cadastrais e Árvores. Os jogos serviram para ilustrar os conceitos e tornar seu crescimento mais divertido! Se você aceitou o desafio, certamente aprendeu e se divertiu. Já publicou seus jogos? Os amigos gostaram?

Que tal encarar desafios maiores agora? Se você percebeu que gosta mesmo de desenvolver jogos, que tal produzir games profissionalmente, para diversas plataformas — PC, web, smartphones, consoles e redes sociais? Que tal passar de fase: estudar o mercado, entender o mercado e fazer de seus jogos um sucesso de público e de renda?

Se você percebeu que os games são para você só uma diversão mesmo, a essência da sugestão ainda é válida: que tal aproveitar cada oportunidade de aprendizado para desenvolver um projeto audacioso e divertido? Aprender, em si, já é legal. Mas é muito mais interessante quando você aprende, aplica o que aprendeu, produz resultados impactantes, sente prazer e orgulho pelo que

produziu.

Seja qual for sua praia, seja audacioso ao escolher seus próximos desafios.  
Mergulhe fundo na vida! Viver pode ser muito divertido!



<http://www.elsevier.com.br/edcomjogos>