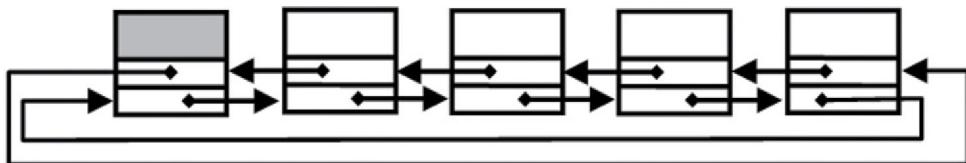


# Capítulo 7

## Generalização de Listas Encadeadas



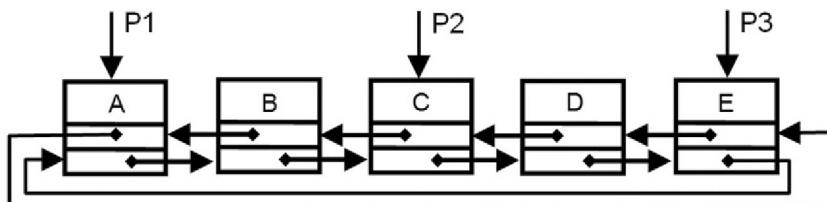
### Seus objetivos neste capítulo

- Estudar técnicas complementares para a implementação de Listas Encadeadas: Encadeamento Duplo, Nô 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

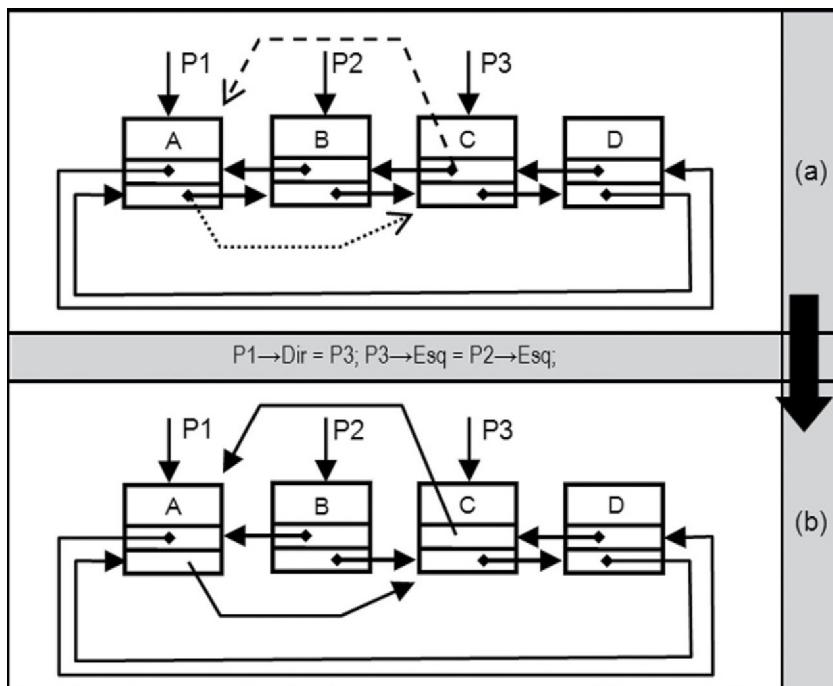
Em cada Nô das estruturas encadeadas que implementamos até o momento, temos um campo para armazenar a informação — que chamamos de Info — e um campo para indicar o próximo elemento — que chamamos de Next. Nessas implementações, podemos percorrer a lista em um único sentido: partindo do primeiro elemento 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 Nô 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 Nô. Na [Figura 7.1](#), o campo Dir do nô apontado por P2 está apontando para o Nô que contém o valor 'D'; o campo Esq do Nô apontado por P2 está apontando para o Nô 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.

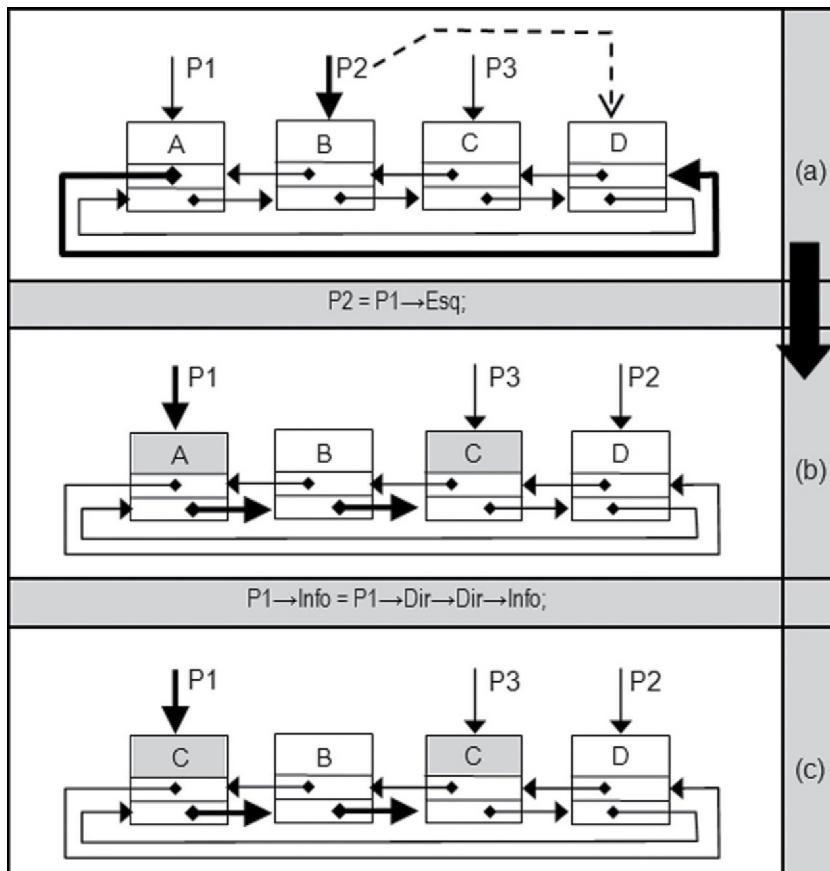


Como estamos manipulando uma Lista Circular, note na [Figura 7.3a](#) que o campo Esq do Nó apontado por P1 aponta para o Nó que contém o valor 'D'. Se, na situação da [Figura 7.3a](#), aplicarmos o comando  $P2 = P1 \rightarrow Esq$ , P2 passará a apontar para o Nó que contém o valor 'D' ([Figura 7.3b](#)).

O comando aninhado  $P1 \rightarrow Info = P1 \rightarrow Dir \rightarrow Dir \rightarrow Info$  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  $P1 \rightarrow Dir \rightarrow Dir$ .  $P1 \rightarrow Dir$  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



**Figura 7.3** Exemplificando a manipulação de uma Lista Duplamente Encadeada.

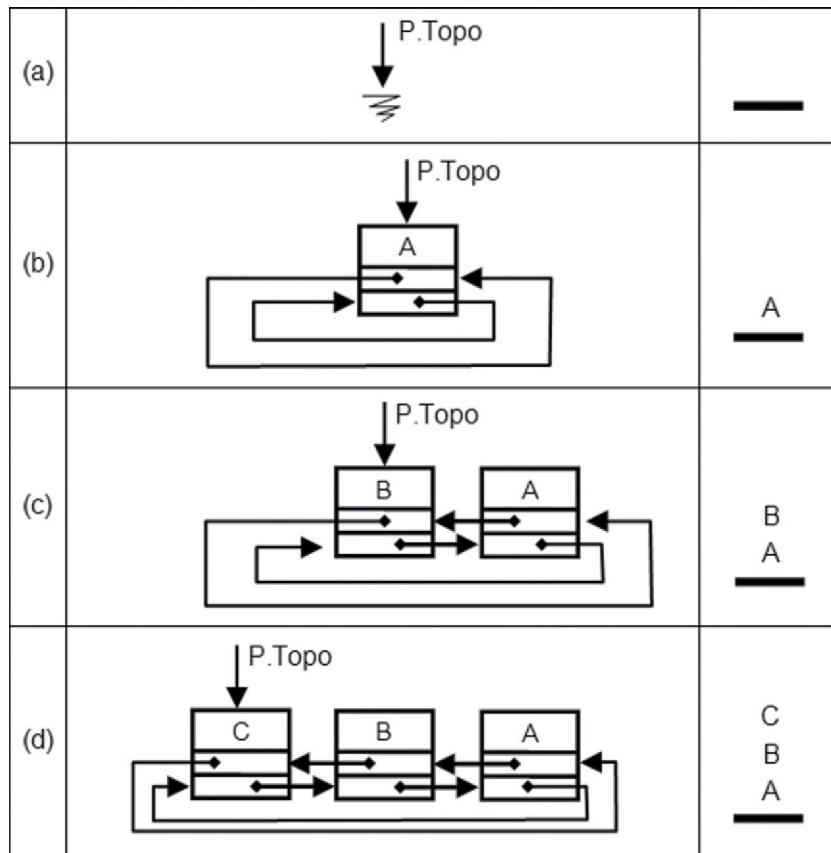


mais um elemento — o elemento de valor ‘B’, chegaremos à situação da [Figura 7.4c](#), na qual a Pilha possui dois elementos, sendo que o elemento ‘B’ está no Topo. Se, a partir da situação da [Figura 7.4c](#), empilharmos mais um elemento, agora de valor ‘C’, chegaremos a uma Pilha com três elementos, com o elemento ‘C’ no Topo, conforme mostra a [Figura 7.4d](#).

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.



**Figura 7.4 Pilha implementada como uma Lista Circular Duplamente Encadeada.**



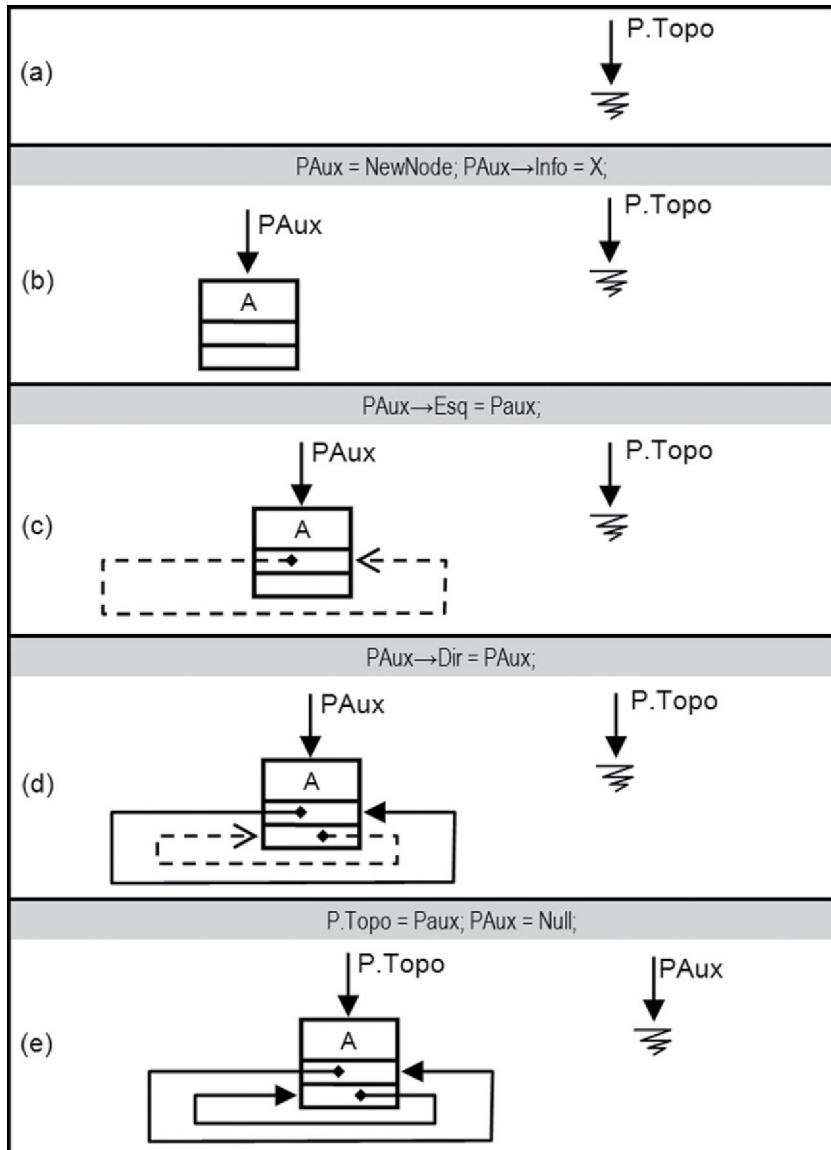
Empilha (parâmetro por referência P do tipo Pilha, parâmetro X do tipo Char, parâmetro por referência DeuCerto do tipo Boolean);

/\* 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.

Com uma variável auxiliar chamada PAux, alocamos um Nó através do comando conceitual PAux = NewNode. Em seguida armazenamos no Nó apontado por PAux a informação que queremos empilar. Fazemos isso através do comando PAux→Info = X. Com esses dois comandos, a situação da [Figura 7.5a](#) passa para a situação ilustrada na [Figura 7.5b](#).

A partir da situação da [Figura 7.5b](#), executamos o comando PAux→Esq = Paux. Através desse comando, o campo Esq do Nó apontado por PAux passa a apontar para o próprio PAux.



**Figura 7.5** Empilha elemento em uma Pilha vazia.



A seta pontilhada na [Figura 7.5c](#) destaca o efeito da execução de  $\text{PAux} \rightarrow \text{Esq} = \text{Paux}$ . Em seguida executamos o comando  $\text{PAux} \rightarrow \text{Dir} = \text{PAux}$ . Através desse comando, o campo Dir do Nô apontado por PAux passa a apontar para o próprio PAux, conforme ilustra a seta pontilhada da [Figura 7.5d](#).

Através do comando  $\text{P.Topo} = \text{PAux}$ , o ponteiro  $\text{P.Topo}$  passa a apontar para onde aponta  $\text{PAux}$ , ou seja, para o Nô que acabou de ser alocado. Movemos  $\text{PAux}$  para  $\text{Null}$ , haja visto

que PAux é uma variável temporária. Ao final da execução dessa sequência 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 operação Empilha para a situação inicial de uma Pilha com dois elementos. A situação da [Figura 7.6a](#) mostra a Pilha com dois elementos, sendo que o elemento B está no Topo. A [Figura 7.6a](#) também mostra o resultado da execução dos comandos PAux = NewNode e PAux→Info = X. Um novo Nó está alocado, e o campo Info desse novo Nó armazena a informação que queremos empilhar.

A partir da situação da [Figura 7.6a](#), executamos o comando PAux→Dir = 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 PAux→Esq = P.Topo→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](#)).

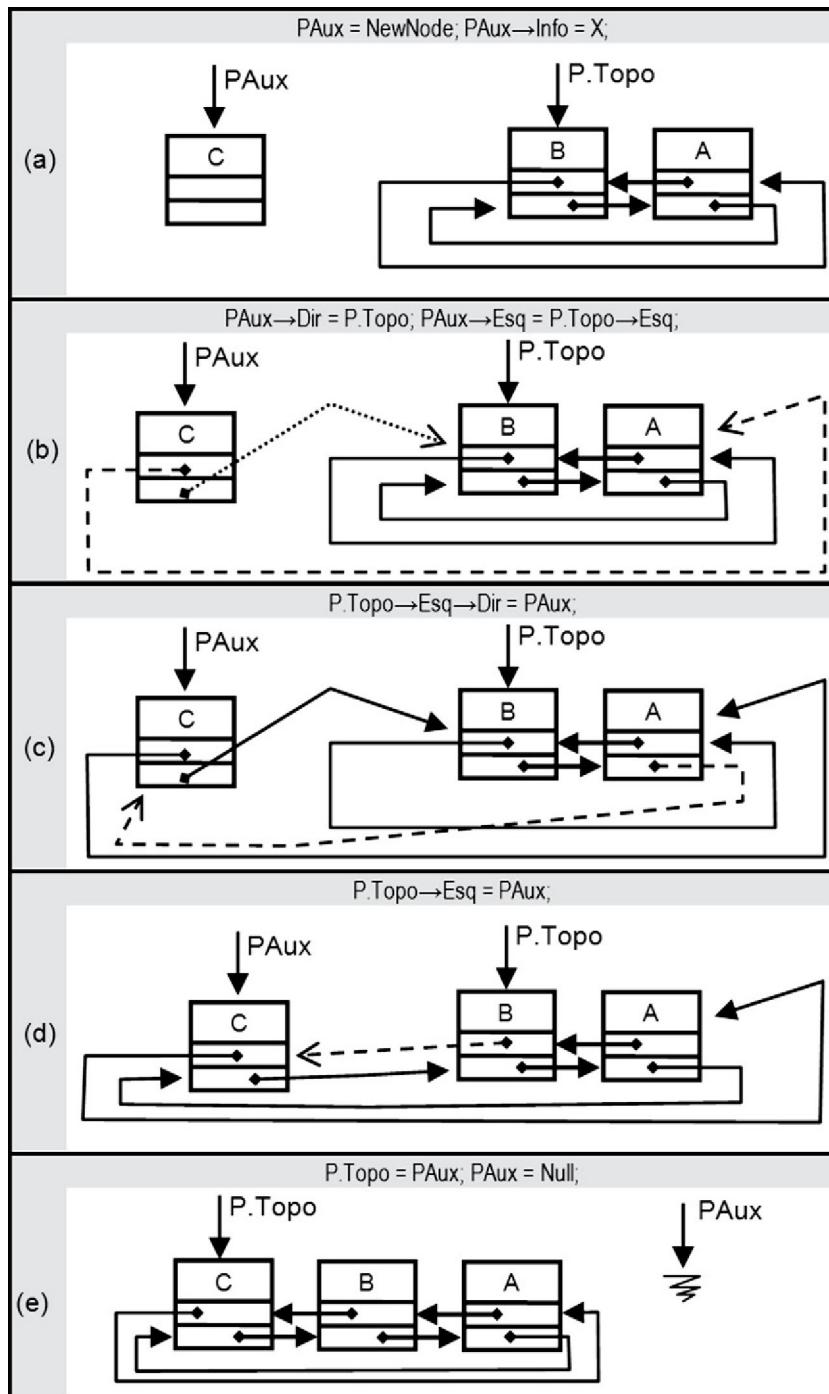
Em seguida, executamos o comando P.Topo→Esq→Dir = PAux, que produz o efeito indicado pela seta tracejada da [Figura 7.6c](#). Note que P.Topo→Esq aponta para o Nó que contém o valor 'A'. Portanto, com o comando aninhado P.Topo→Esq→Dir = PAux estamos apontando o campo Dir do Nó que contém o valor 'A' para PAux.

Caso você tenha alguma dificuldade para visualizar a execução do comando aninhado P.Topo→Esq→Dir = 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 NovoPonteiroAuxiliar = P.Topo→Esq. Em seguida você poderá executar NovoPonteiroAuxiliar→Dir = PAux. O resultado da execução desses dois comandos será o mesmo da execução do comando aninhado P.Topo→Esq→Dir = PAux, destacado pela seta tracejada da [Figura 7.6c](#).

Em seguida, a partir da situação da [Figura 7.6c](#), executamos o comando P.Topo→Esq = 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 P.Topo→Esq = PAux.

Através do comando P.Topo = 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](#).



**Figura 7.6** Empilha Elemento em uma Pilha não vazia.



```

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.

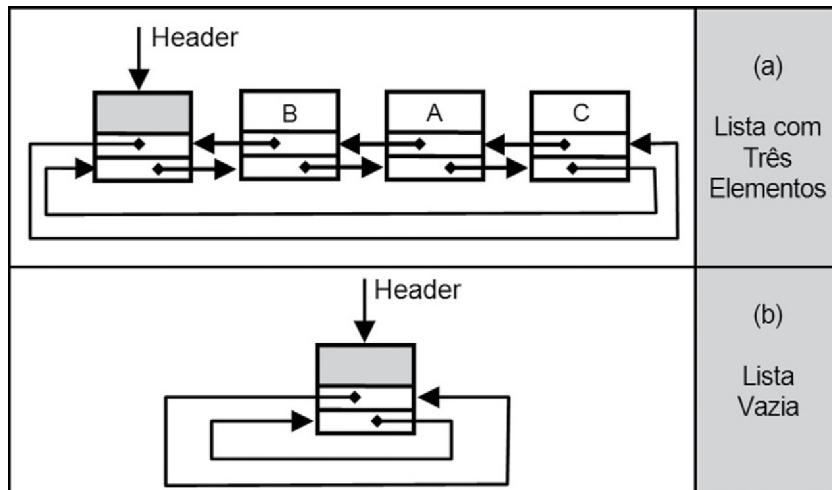
*Exercício 7.5 TAD Lista Cadastral implementada como Lista Circular Duplamente Encadeada Não Ordenada*

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 Nôs. Três desses Nôs estão armazenando os elementos B, A e C. O quarto Nô é o Nô Header, que não armazena nenhum elemento, e apenas marca o início da Lista.

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 Nô Header.



**Figura 7.8** Lista circular com Nô 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 Nô 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 Nó 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.

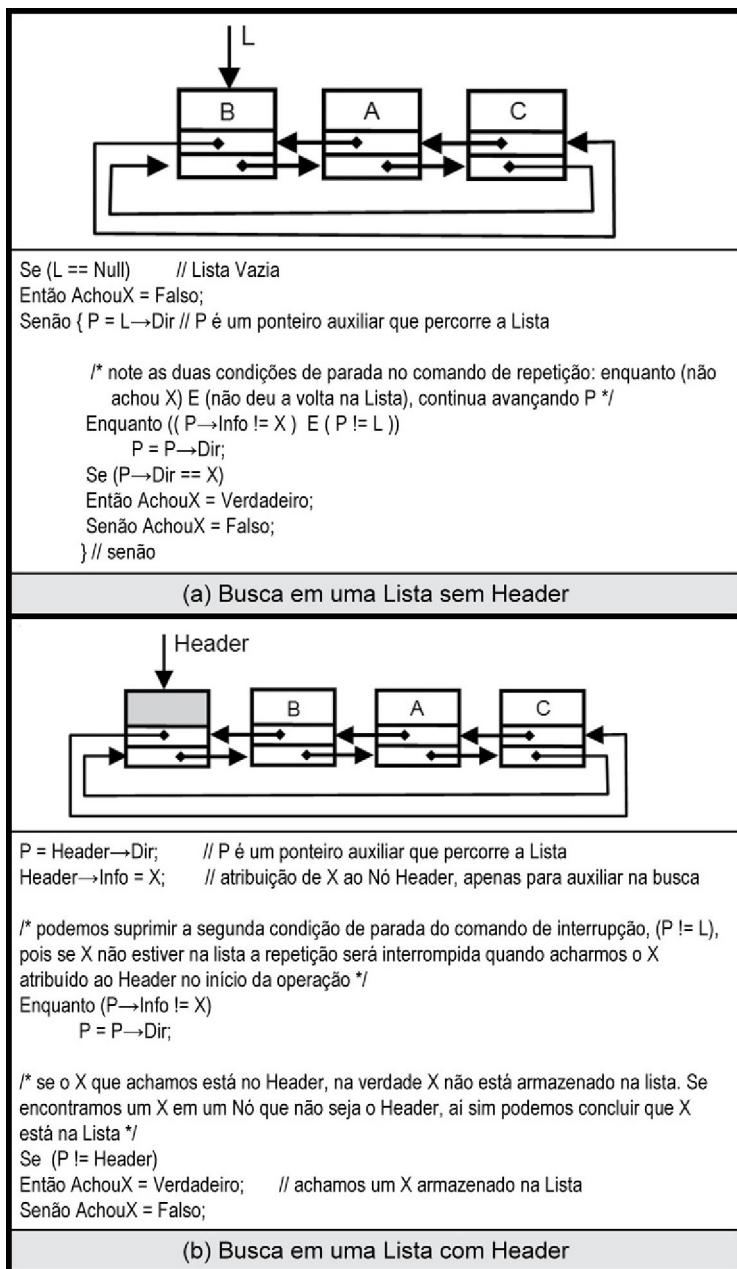
Quando temos uma implementação com Header, antes de iniciar a busca podemos atribuir o valor de X ao campo Info do Nó Header. Então podemos suprimir a segunda condição de parada, ( $P! = L$ ), pois se X não for encontrado em um dos Nós da Lista que armazenam elementos, será encontrado no Nó Header, interrompendo 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  $P != \text{Header}$ ) 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.



**Figura 7.9 Comparando a busca com o Nό Header à Busca sem o Nό Header.**

### Exercício 7.7 TAD Fila com Nό Header

Implemente uma Fila através de uma Lista Circular Duplamente Encadeada com Nό 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 Nó 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 Nó 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 Nó apontado por P.

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*

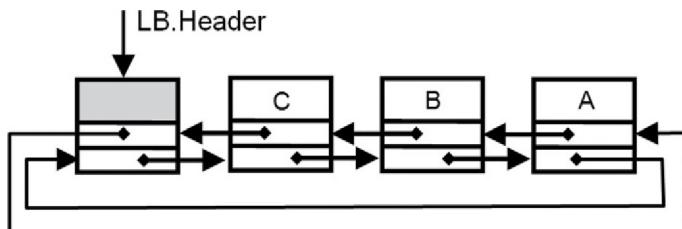
Implemente uma Pilha utilizando as Operações de Baixo Nível para Manipulação de Listas Encadeadas, especificadas na [Figura 7.11](#). Considere uma Lista Básica implementada como uma Lista Circular, Duplamente Encadeada e com Nó Header. Primeiramente escolha uma posição da Lista Genérica na qual ficará 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 Nó Header, para implementar uma Pilha podemos convencionar, por exemplo, que o topo da Pilha ficará sempre à direita do Nó Header, apontado pelo

<b>Operações</b>	<b>Funcionamento</b>
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.

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 Nó Header e ficará posicionado entre o Nó de valor ‘C’ e o Nó 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 Nó à direita

do qual queremos inserir o novo Nó, com o valor X. Como queremos inserir o novo elemento da Pilha à direita do Nó 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.

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.

```

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.

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.

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.

```

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

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.

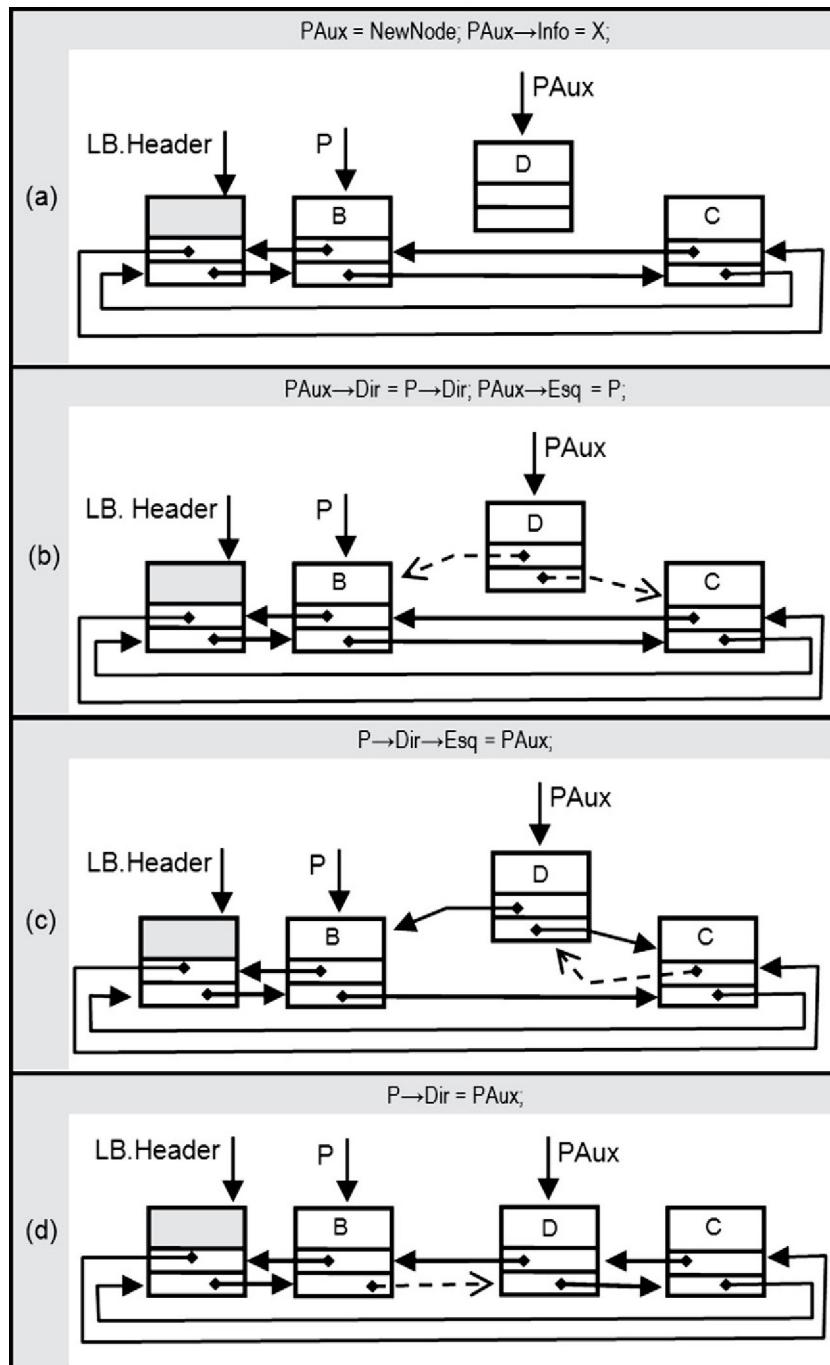


Figura 7.15 Execução — Operação InsereADireitaDeP.

### *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](#).

Em seguida, conforme ilustra a seta pontilhada da [Figura 7.15c](#), o campo Esq do Nó apontado por P→Dir passa a apontar para PAux. Finalmente, na [Figura 7.15d](#), o campo Dir do Nó 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 Nó imediatamente à direita do Nó apontado por P, passado como
       parâmetro */

    Variável auxiliar NovoNó do tipo NodePtr; // NodePtr = ponteiro para Nó

    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.**

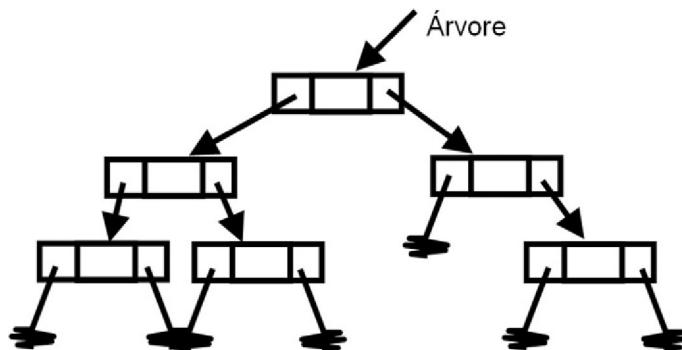
### *Exercício 7.12 Remove\_P, EstáNaLista e Info\_de\_P*

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 Nó 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

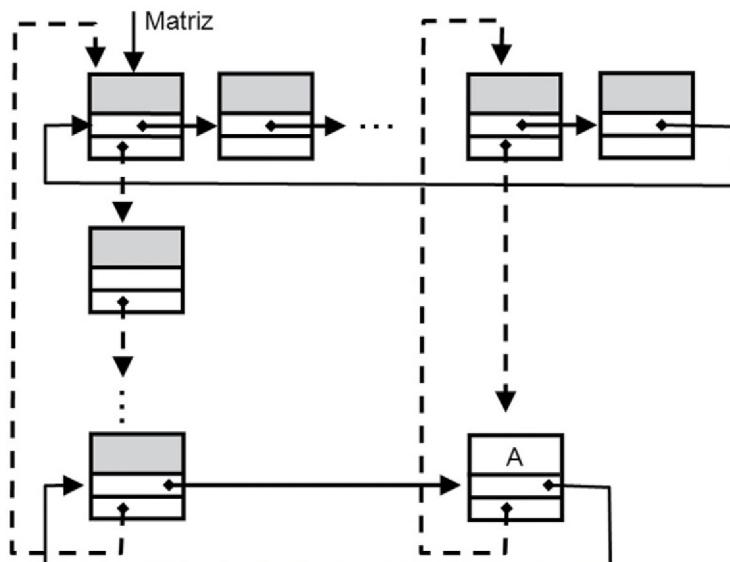
Até o momento estudamos Estruturas Encadeadas em que cada elemento possui um único elemento que é o seu próximo na sequência. No caso das Listas Duplamente Encadeadas, cada elemento possui um próximo e um anterior, os quais designamos como elemento da direita e elemento da esquerda.

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.

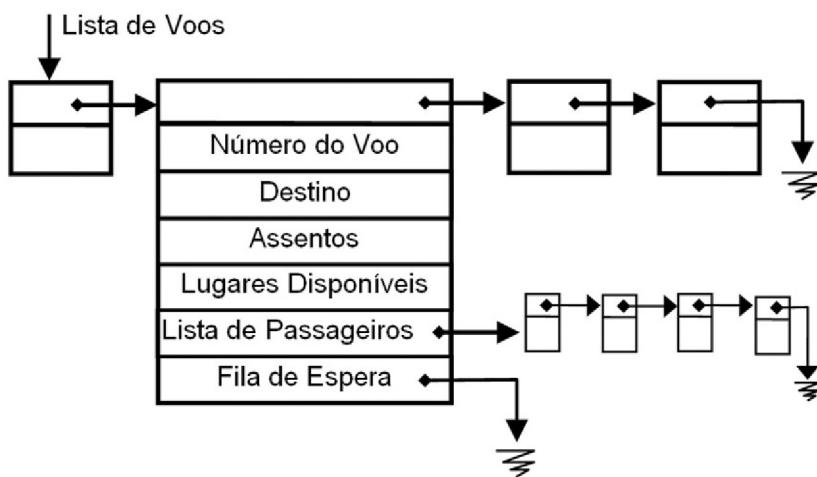
Na implementação encadeada da [Figura 7.18](#), cada elemento não nulo da Matriz está armazenado em um Nó. Cada Nó da estrutura é encadeado em duas Listas: uma Lista encadeando todos os elementos não nulos da mesma linha da matriz e uma Lista conectando todos os elementos não nulos na mesma coluna da Matriz.

As Listas de linhas e de colunas são implementadas como Listas Encadeadas Circulares, com Nó Header. Assim, temos um Nó Header para cada Linha e um Nó 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 Nó 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*

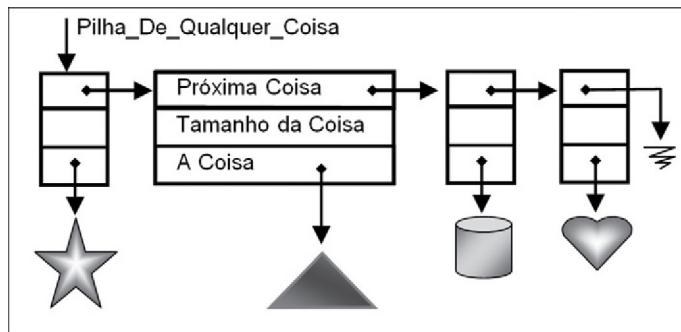
O uso de Templates permite elaborar um código genérico quanto ao tipo de elemento, mas no momento de criar a estrutura o tipo precisará ser definido. Uma vez definido o tipo do elemento, a estrutura só poderá receber elementos daquele tipo. Por exemplo, se criarmos uma Pilha de Inteiros, não poderemos 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](#).

### *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



**Figura 7.21** Inserindo elementos de tipos diferentes em uma única Pilha.

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



## Exercícios de fixação

**Exercício 7.15** Que vantagens você pode apontar quanto ao uso de um Nó 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
```



ELSEVIER

**Exercício 7.4 TAD Fila implementada como Lista Circular Duplamente Encadeada**

Sugestão: faça uma fila com dois ponteiros: F.Primeiro e F.Ultimo, 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 Nô Header*

Solução parcial — Operações Cria e EstáNaLista:

```
Cria (parâmetro por referência L do tipo Lista) {
/* Cria a Lista sem nenhum elemento mas com o Nô Header. O ponteiro H aponta para o Nô 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 Nô
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 Nô 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 Nô
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 Nô 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 Nô
Se (EstáNaLista(LB, X, P)==Verdadeiro) // P estará apontando para o Nô onde está X
Então Remove_P (LB, P, DeuCerto); // remove o Nô 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

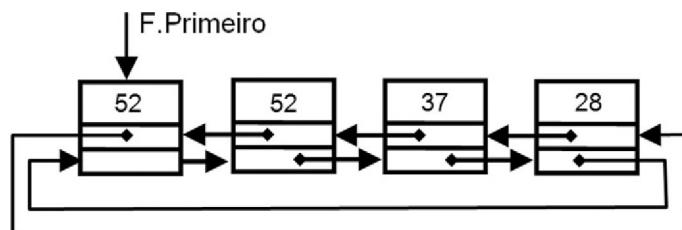
```

### *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 Nô 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);
}
```