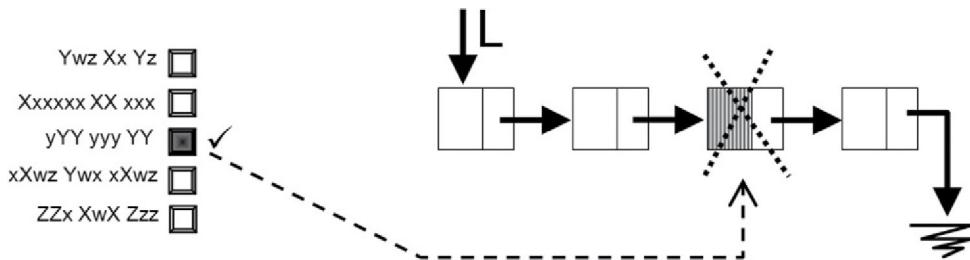


# Capítulo 6

## 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循环的, 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.

	Atualizando uma Lista de Compras
Ywz Xx Yz <input type="checkbox"/>	
Xxxxxx XX xxx <input type="checkbox"/>	
<b>Esguicho <input checked="" type="checkbox"/> ✓</b>	<p>Se queremos retirar o item <i>Esguicho</i> da Lista:</p> <ul style="list-style-type: none"> <li>• Procuramos na Lista o item cujo valor é <i>Esguicho</i>.</li> <li>• Retiramos da Lista, especificamente, o item cujo valor é <i>Esguicho</i> — esteja esse item no começo, no meio ou no final da Lista.</li> </ul>
xXwz Ywx xwz <input type="checkbox"/>	
ZZx XwX Zzz <input type="checkbox"/>	

**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.**

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](#).

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,

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.

Listas 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.

que continha quatro elementos (A, B, C, D), passará a ter apenas três elementos (A, C, D), conforme ilustra a [Figura 6.5b](#).

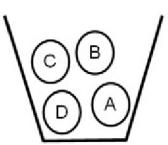
### *Operações para percorrer uma Lista*

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.

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'.

Listá 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.

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/? e 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.

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.**

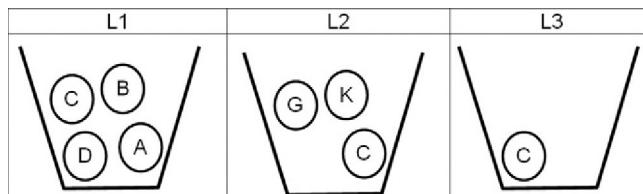
### *Exercício 6.2 Interseção de duas Listas L1 e L2*

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.

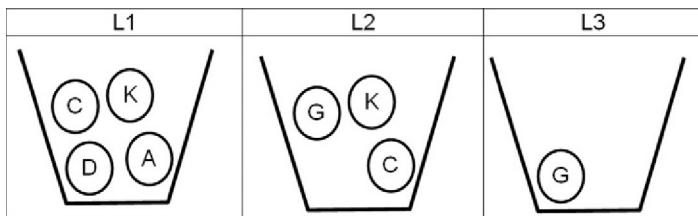
Inteiro PosiçãoNoConjunto (parâmetro por referência L do tipo Lista, parâmetro X do tipo char);
 /\* Retorna valor numérico que indica a posição de X na Lista L. Por exemplo, retorna 1 se X for o primeiro elemento da
 Lista L, 2 se for o segundo, e assim por diante. Se X não estiver em L, retorna o valor 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](#).

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\*/



**Figura 6.10** L3 contém os que estão em L2 e 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.

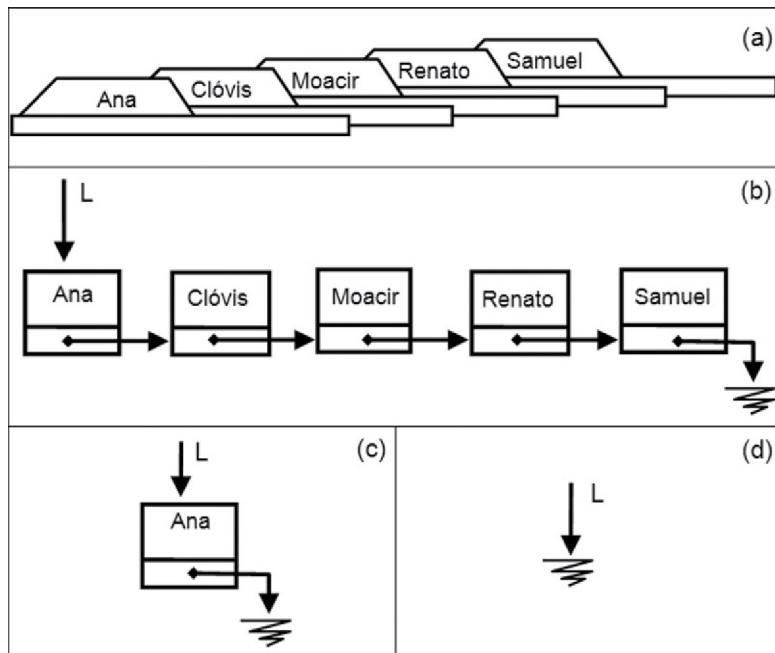
A [Figura 6.11a](#) ilustra a situação de um Cadastro com cinco pastinhas, ordenadas alfabeticamente. Na [Figura 6.11b](#), esse Cadastro com cinco elementos é representado como uma Lista Encadeada Ordenada. O ponteiro L aponta para o primeiro Nô da Lista, que armazena o valor Ana. Os demais elementos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

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](#).

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



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

#### 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.

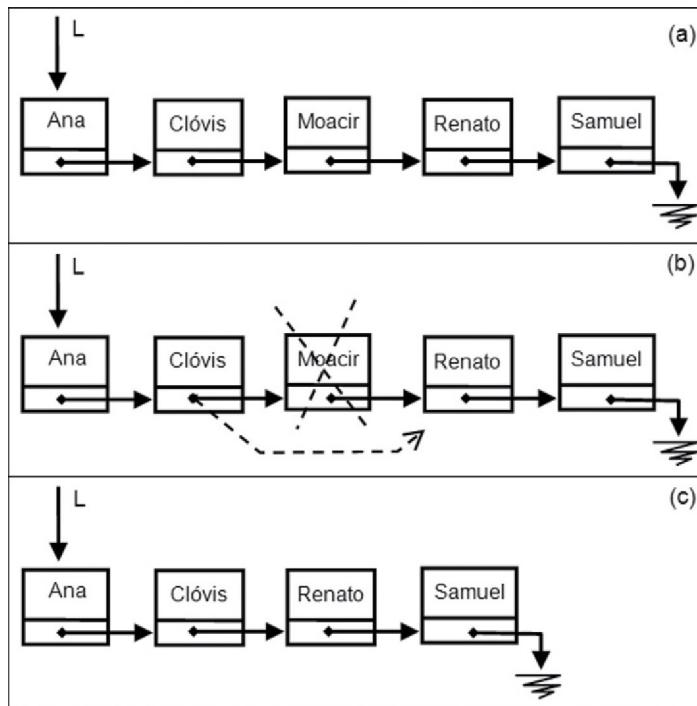
**Figura 6.12** Retirando elemento de uma Lista Cadastral implementada como uma Lista Encadeada Ordenada.



Nem sempre o Nó contendo o valor do elemento que queremos remover estará “no meio” da Lista, como no exemplo da [Figura 6.13](#). É possível que o elemento a ser removido esteja no início ou no final da Lista; aliás, é possível que o elemento que queremos remover nem esteja na Lista. E a operação que retira um elemento da Lista precisa tratar todos esses casos.

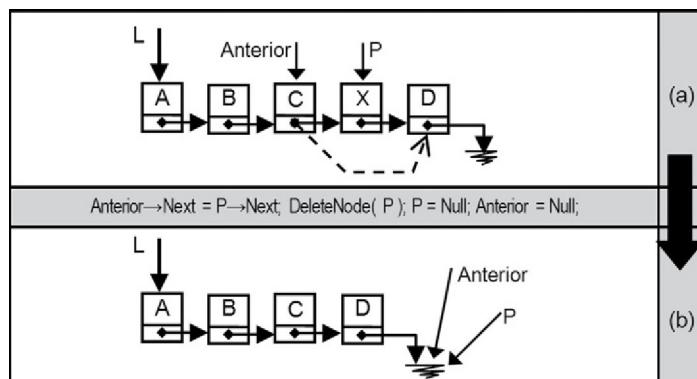
Os diagramas das [Figuras 6.14 a 6.19](#) ilustram diferentes situações para a operação Retira. Em todas essas situações, L é o ponteiro para o início da Lista, e X é o elemento que queremos remover.

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.



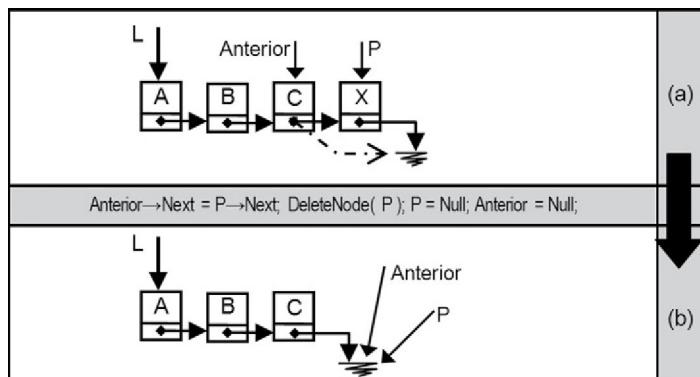
**Figura 6.13** Removendo o Nó que contém o valor *Moacir*.

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  $\text{Anterior} \rightarrow \text{Next} = P \rightarrow \text{Next}$ . Em seguida, aplicamos o comando  $\text{DeleteNode}(P)$ , que desaloca o nó apontado por P. Com o  $\text{DeleteNode}$ , 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](#).



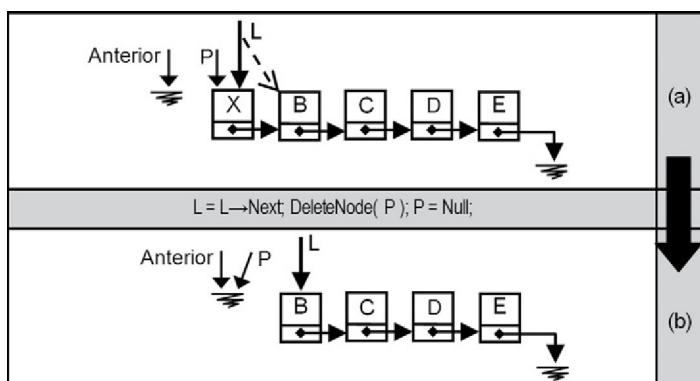
**Figura 6.14** Retirando elemento de uma Lista Ordenada. Caso 1: X no meio da Lista.

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  $P \rightarrow \text{Next}$  está apontando para um Nó, e no Caso 1' está apontando para Null.



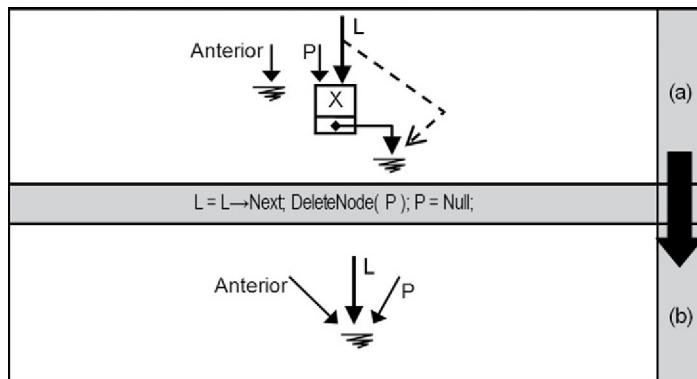
**Figura 6.15** Retirando elemento de uma Lista Ordenada. Caso 1': X no final da Lista.

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  $L = L \rightarrow \text{Next}$  e então liberar o Nó apontado por P com o comando  $\text{DeleteNode}(P)$ . Apenas para deixar P apontando para uma posição bem definida, apontamos P para Null e chegamos à situação da [Figura 6.16b](#).



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

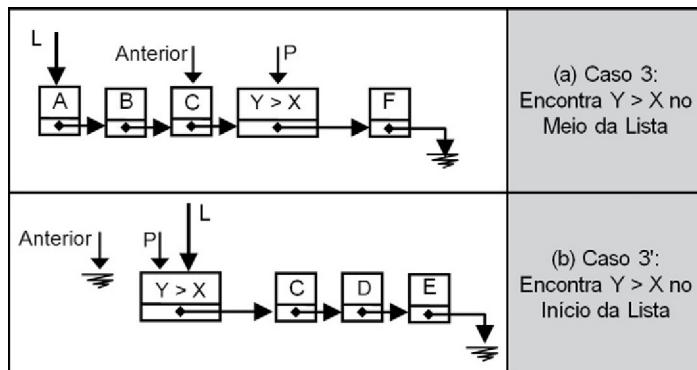
Na [Figura 6.17a](#), tratamos o Caso 2', situação em que X está no primeiro Nó da Lista, assim como no Caso 2. Mas, no Caso 2', esse primeiro Nó da Lista é também o único Nó da Lista. Os comandos aplicados no Caso 2' são os mesmos aplicados no Caso 2. Porém, no Caso 2', ao avançar L com o comando  $L = L \rightarrow \text{Next}$ , o ponteiro L passará a apontar para Null (seta pontilhada e [Figura 6.17b](#)).



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

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 Nô 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.

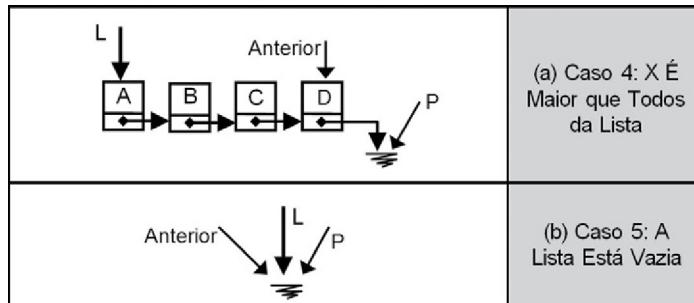
No Caso 3, encontramos um valor  $Y > X$  no meio da Lista — [Figura 6.18a](#). No Caso 3', encontramos  $Y > X$  logo no início da Lista — [Figura 6.18b](#).



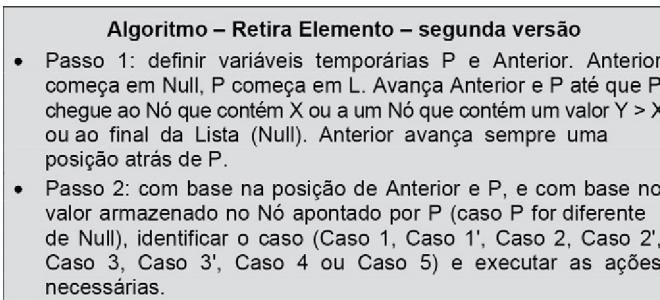
**Figura 6.18** Retira de Lista Ordenada. X não está na Lista. Caso 3: encontra  $Y > X$  no meio da Lista. Caso 3': encontra  $Y > X$  no início da Lista.

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](#).



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



**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 Nó que contém X.

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](#)).

```

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.

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](#).

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 ( $(P \neq \text{Null}) \text{ E } (\text{Info}(P) < X)$ ). 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.

```

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 Booleano) {
    /* 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.

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  $((P \neq \text{Null}) \text{ E } (P \rightarrow \text{Info} == X))$ . Não basta perguntar se  $P \rightarrow \text{Info} == X$  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,  $P \rightarrow \text{Info}$  não existe. O resultado da execução de  $P \rightarrow \text{Info}$  quando P está apontando para Null não é previsível e poderá resultar em um erro. Evitamos esse erro ao perguntarmos se  $((P \neq \text{Null}) \text{ e } (P \rightarrow \text{Info} == X))$ . 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 \*/

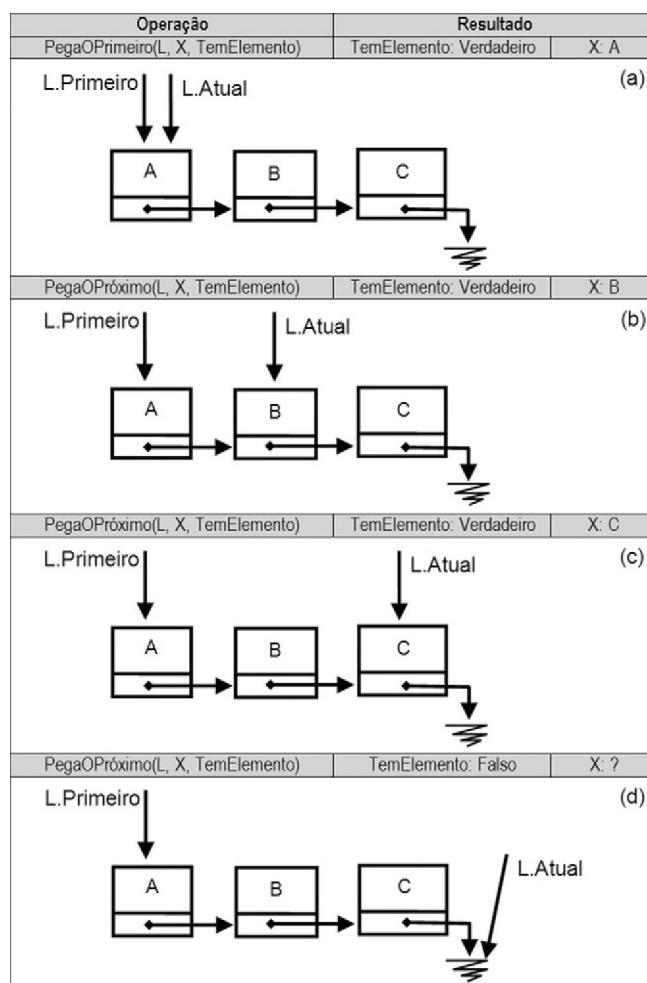
### *Implementando operações para percorrer a Lista Cadastral*

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 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.

*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 PegaOPróximo, 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 \*/

```
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 6.13 Revisar os Exercícios 6.5 a 6.10 adaptando as soluções para uma Lista com dois ponteiros*

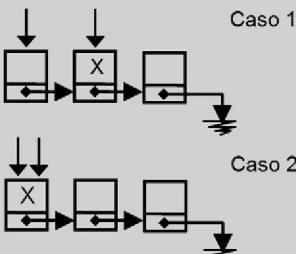
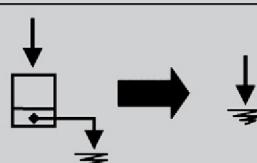
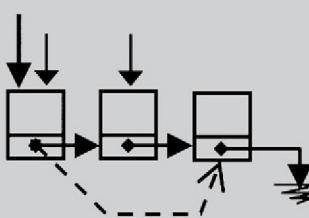
Na implementação dos Exercícios 6.5 a 6.10, utilizamos uma Lista L com um único ponteiro permanente — o ponteiro L. Contudo, para possibilitar a implementação das operações para percorrer a Lista, utilizamos a Lista L com dois ponteiros: L.Primeiro e L.Atual. Ajuste as soluções propostas para os Exercícios 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](#).

*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

Passos para construir uma boa solução	
<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.	
<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.	
<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.	Para tratar o Caso 2: <ul style="list-style-type: none"> <li>• DeleteNode(P);</li> <li>• L=NULL;</li> </ul>
<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.	Algoritmo geral Se Caso 1 Então [tratar Caso 1] Senão Se Caso 2 Então [tratar Caso 2] Senão Se Caso 3 Então [tratar Caso 3] Senão...
<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, parta 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.	

**Figura 6.25** Passos para construir uma boa solução.

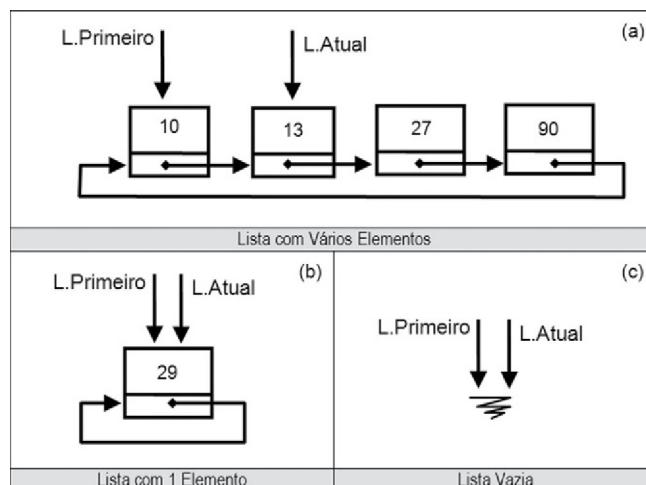
Lista Circular o campo Next do último Nó da Lista deve apontar para o primeiro Nó da Lista, em vez de apontar para Null ([Figuras 6.26a e 6.26b](#)). Quando a Lista estiver vazia, os ponteiros L.Primeiro e L.Atual devem apontar para Null ([Figura 6.26c](#)). Implemente as operações Cria, Vazia, Cheia, EstáNaLista, Insere, Retira, PegaOPrimeiro e PegaOPróximo.

### Exercício 6.15 Operação Destroi

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 \*/



**Figura 6.26** Lista Cadastral implementada como Lista Encadeada Ordenada Circular.

### Exercício 6.16 Lista Cadastral com elementos repetidos implementada como uma Lista Encadeada Ordenada Circular

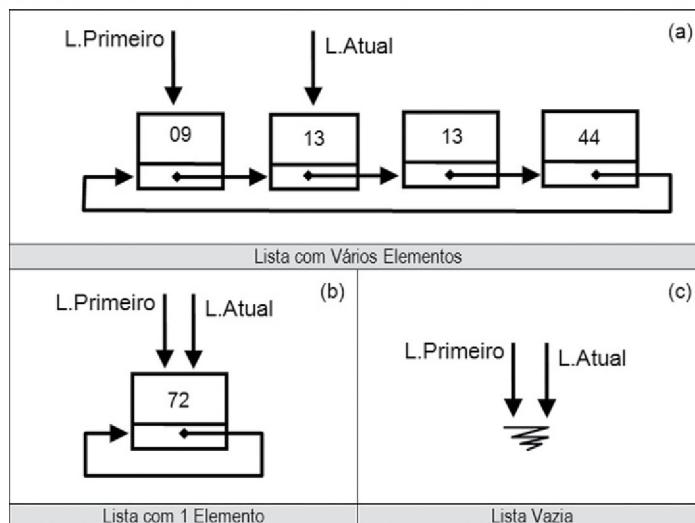
Implemente uma Lista Cadastral com elementos repetidos, através de uma Lista Encadeada Ordenada Circular. Conforme mostra o diagrama da [Figura 6.27a](#), a lista deve permitir a entrada de elementos repetidos. Considerando que a lista é ordenada, os elementos repetidos devem estar juntos. Implemente 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.

### 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 \*/

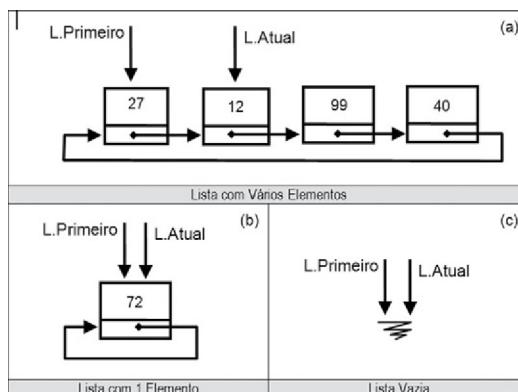
**Figura 6.27** Lista Cadastral com elementos repetidos.

*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.

*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

**Figura 6.28** Lista Cadastral implementada como uma Lista Encadeada Circular Não Ordenada.

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

Ywz Xx Yz <input type="checkbox"/> XXXXX XX xxx <input type="checkbox"/> yYY yy YY <input checked="" type="checkbox"/> ✓ xXwz Ywx xXwz <input type="checkbox"/> ZZx XwX Zzz <input type="checkbox"/>	
<b>Lista de Compras: Lista Cadastral Sem Elementos Repetidos</b>	<b>Carrinho de Compras: Lista Cadastral Com Elementos Repetidos</b>

**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

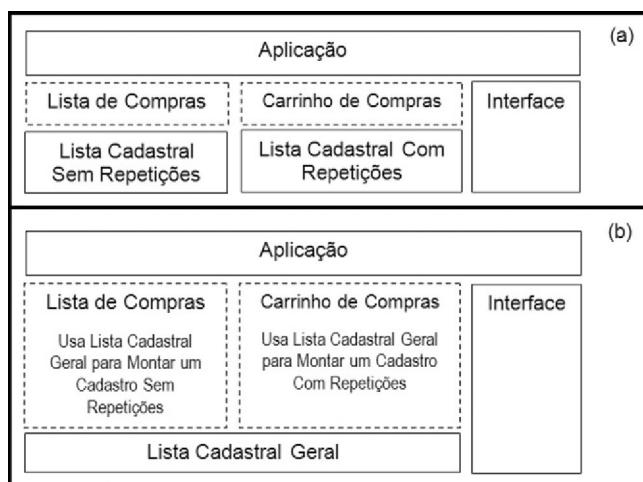
quanto como uma Lista com repetições, como na [Figura 6.30b](#). Qual dessas duas alternativas — A ou B — você considera mais adequada ao seu projeto? Você pode ainda pensar em uma outra alternativa.

*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.



**Figura 6.30** Projeto Spider Shopping — soluções alternativas quanto à arquitetura do software.



### *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 adequada 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 adequada 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 adequada 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

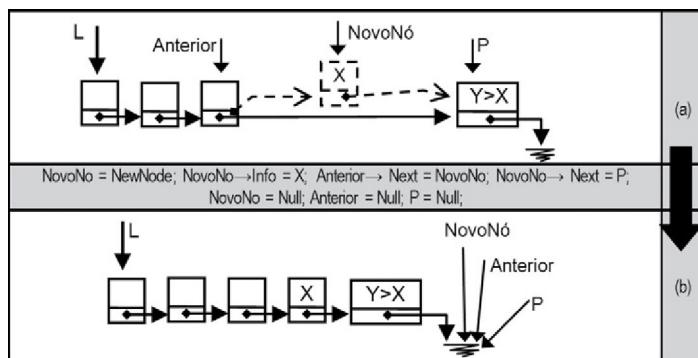
### Exercício 6.6 Operação Insere Elemento em uma Lista Cadastral implementada como uma Lista Encadeada Ordenada

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.

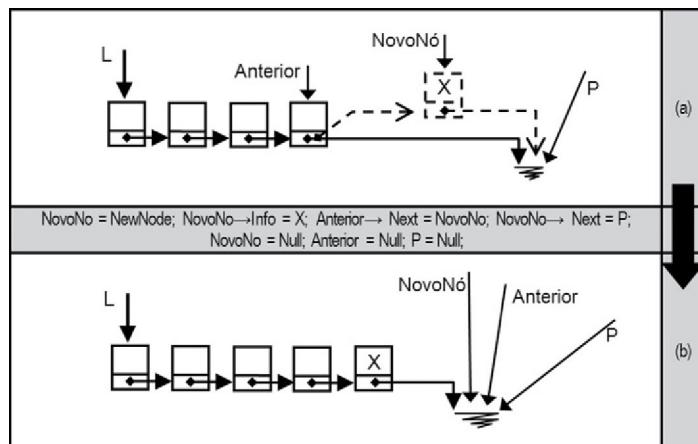
Algoritmo geral em dois passos — Insere Elemento

- 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 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 3 ou Caso 4) e executar as ações necessárias.

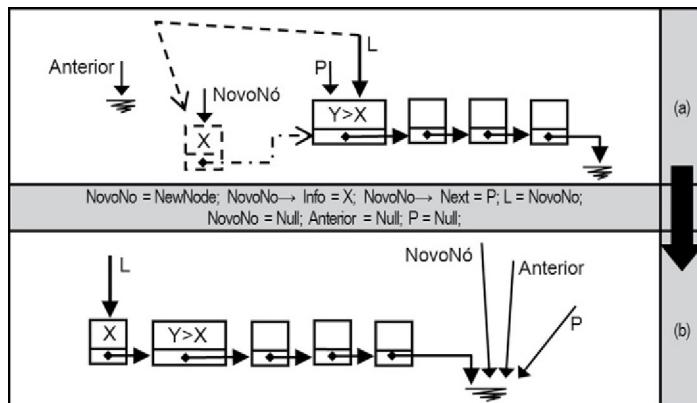
Caso 1: X não está na Lista e deve ser inserido no meio da Lista



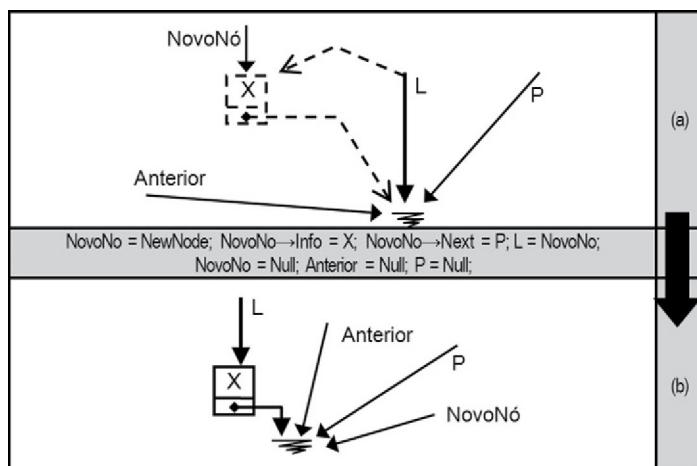
Caso 1': X não está na Lista e deve ser inserido no final da Lista



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 Node Ptr; // 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 inicio...
```

```
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 PegaOPróximo, 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 PegaOPróximo

```

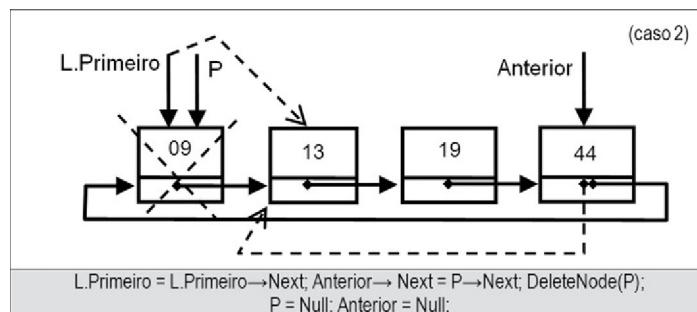
### Exercício 6.13 Revisar os Exercícios 6.7 a 6.12 adaptando as soluções para uma Lista com dois ponteiros

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 Destroi de uma Lista Cadastral

Comentário:

- A forma mais apropriada para proporcionar portabilidade e reusabilidade é implementar o Destroi através das operações primitivas.

Destroi (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 Destroi

### 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;
Repete
    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  $P \rightarrow \text{Info} \neq X$ ”.

*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 intersecção 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;

```
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).