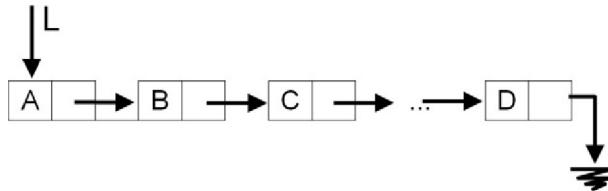


Capítulo 4

Listas Encadeadas



Seus objetivos neste capítulo

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

4.1 Alocação Sequencial versus Alocação Encadeada

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

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

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

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

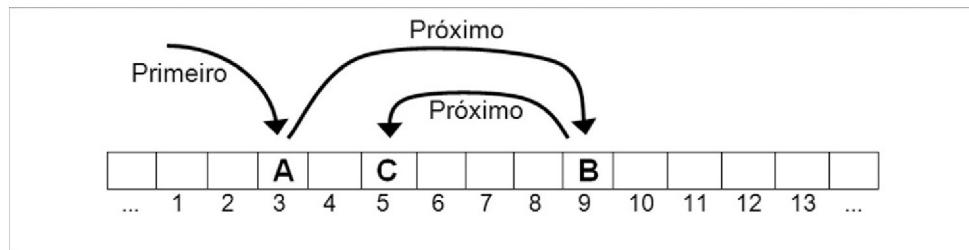


Figura 4.1 Indicação explícita da sequência dos elementos.

Definição: Alocação Encadeada de Memória para um conjunto de elementos

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

Figura 4.2 Definição de Alocação Encadeada de Memória para um conjunto de elementos.

4.2 Listas Encadeadas: conceito e representação

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

- Cada Nô N(i) é um conjunto composto por dois campos: o primeiro campo contém a informação a ser armazenada naquele Nô, e o segundo campo contém a indicação do Próximo elemento da Lista. Na [Figura 4.3](#), a informação armazenada no Nô é representada pelos caracteres A, B, C e D. A indicação do Próximo elemento é representada pelas setas horizontais.
- Os Nós da Lista não estão, necessariamente, em posições adjacentes de memória.
- O acesso aos elementos da Lista ocorre através de um ponteiro para o Primeiro elemento da Lista. Na [Figura 4.3](#), o Primeiro elemento da Lista é indicado pelo ponteiro L.
- O acesso aos demais elementos da Lista ocorre sempre a partir do Primeiro elemento e, a seguir, pela indicação de qual é o Próximo na sequência. No exemplo da [Figura 4.3](#), não é possível ter acesso direto ao elemento da Lista que contém o valor C. Para acessar o elemento que armazena o valor C, é preciso acessar o Primeiro elemento da Lista através do ponteiro L, depois seguir a indicação do Próximo elemento para chegar ao Nô(2) e então ao Nô(3).
- O último Nô da Lista — Nô(N) — aponta para um endereço de memória inválido chamado NULL; isso indica que Nô(N) guarda o último elemento do conjunto.
- Em uma Lista Encadeada vazia — sem elementos, o ponteiro para o Primeiro da Lista aponta para o endereço NULL.

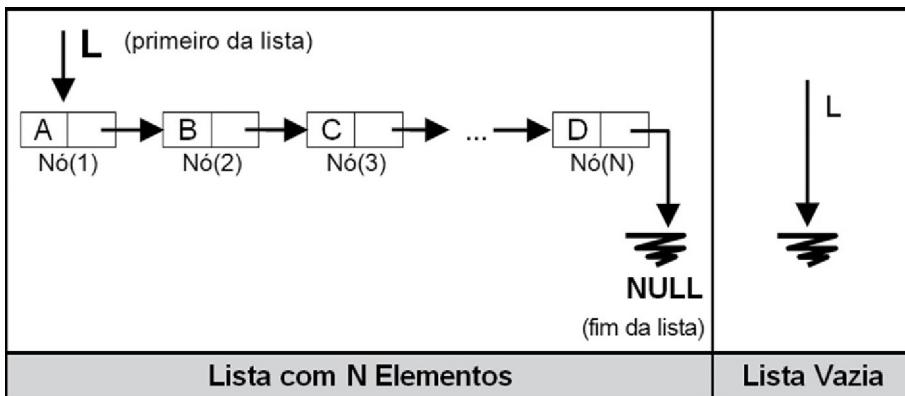


Figura 4.3 Representação usual de uma Lista Encadeada.

4.2.1 Notação para manipulação de Listas Encadeadas

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

Node é o tipo definido para o Nó da Lista Encadeada. O tipo Node é um Registro com dois campos: o campo Info — utilizado para armazenar informação, e o campo Next —

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

Figura 4.4 Notação conceitual para manipulação de Listas Encadeadas.

utilizado para indicar o próximo elemento da Lista. O campo Next e as variáveis L, P, P1 e P2 são do tipo NodePtr. O tipo NodePtr é um ponteiro para uma estrutura do tipo Node. Ou seja, variáveis do tipo NodePtr podem apontar para os Nós da Lista Encadeada.

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

4.2.2 Entendendo o funcionamento das operações

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

Acessando e atualizando o campo Info de um Nô

Considerando como ponto de partida a situação da [Figura 4.5a](#), aplicamos a operação **X = P→Info**. Aplicar essa operação significa que a variável X passará a ter o valor do campo Info do Nô apontado pelo Ponteiro P. Ou seja, a variável X passará a ter o valor B, conforme mostra a [Figura 4.5b](#).

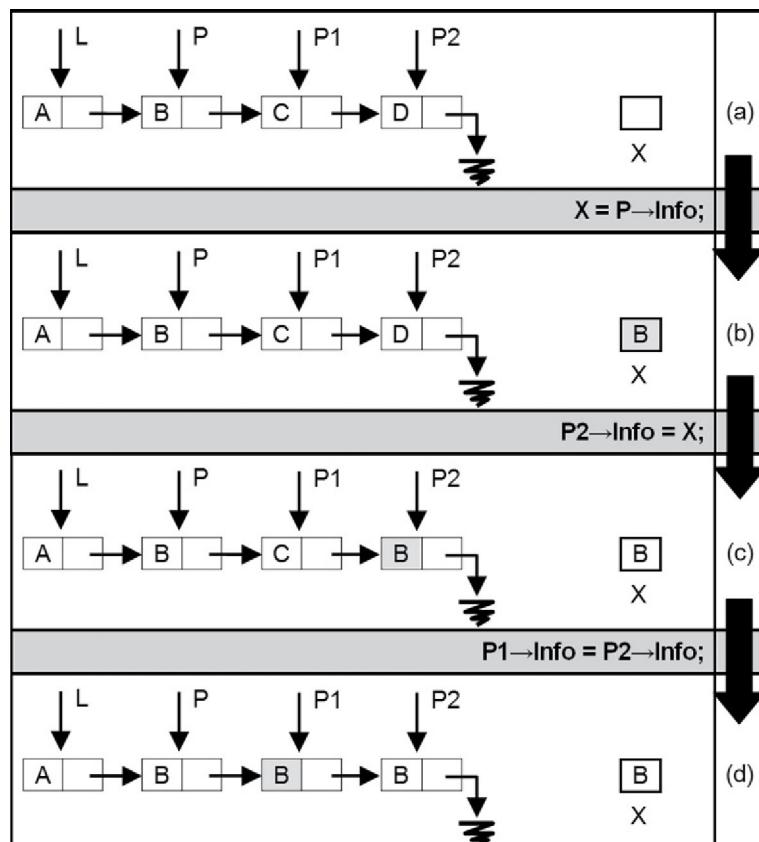


Figura 4.5 Operações para acessar e atualizar a informação armazenada.

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

Com a aplicação da operação **P1→Info = P2→Info**, o campo Info do nó apontado por P1 recebe o valor do campo Info do Nó apontado por P2. Veja o resultado na [Figura 4.5d](#). Um modo alternativo de fazer a leitura do comando **P1→Info = P2→Info** seria “Info de P1 recebe Info de P2”.



Acessando e atualizando o campo Next de um Nó

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



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

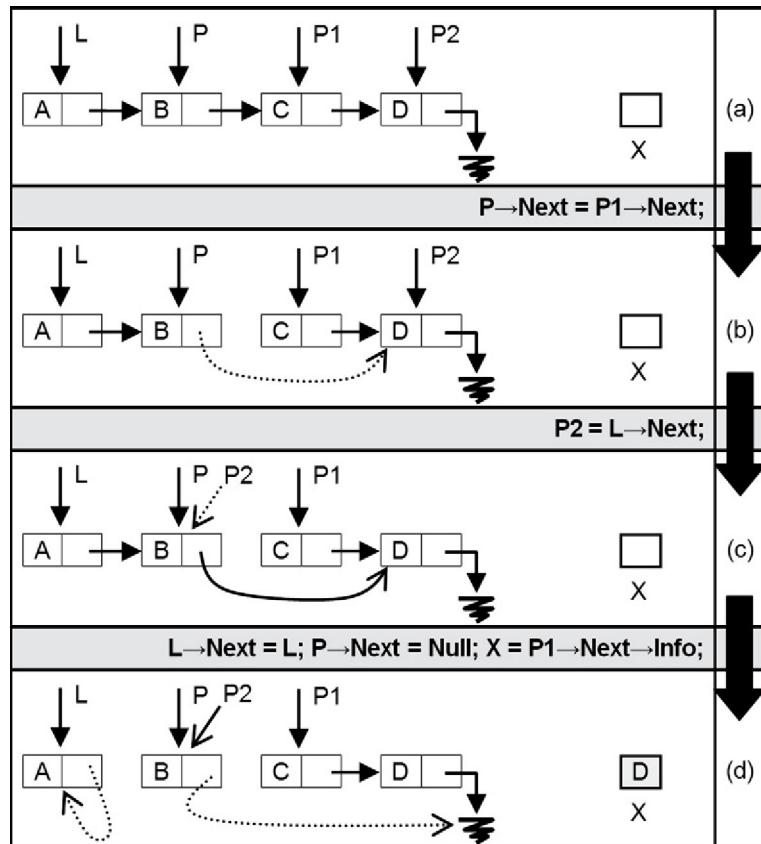


Figura 4.6 Operações para acessar e atualizar a indicação do próximo elemento da Lista.

Para passar da situação da [Figura 4.6c](#) para a situação da [Figura 4.6d](#), aplicamos três operações. A primeira operação, **L→Next = L**, faz o campo Next do Nó apontado por L apontar para o próprio L. A segunda operação, **P→Next = Null**, faz o campo Next do Nó apontado por P passar a apontar para o valor Null. Na terceira operação, **X = P1→Next→Info**, a variável X recebe a informação armazenada no campo Info do Nó apontado por **P1→Next**. Utilizando parênteses para tirar a ambiguidade, o comando poderia ser expresso como **X = (P1→Next)→Info**. Ou seja, X passa a ter o valor D. Um modo alternativo de fazer a leitura do comando **X = P1→Next→Info** seria: "X recebe Info do Next de P1".

Movendo ponteiros

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

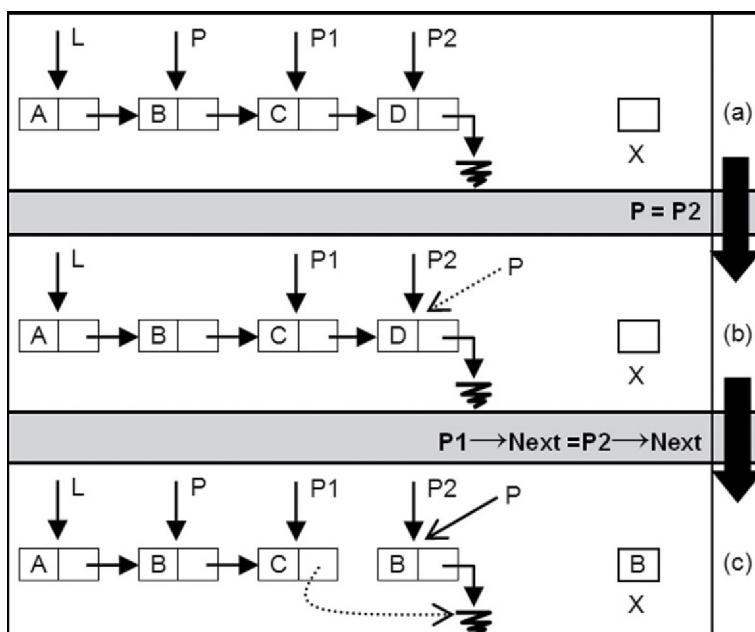


Figura 4.7 Operações para mover ponteiros.

Alocando e desalocando Nós

As operações para alocar e desalocar Nós são ilustradas na [Figura 4.8](#). Inicialmente, à situação inicial da [Figura 4.8a](#) aplicamos a operação **DeleteNode(P1)**. Essa operação desaloca o Nó apontado por P1, ou seja, libera a memória, para que possa ser utilizada para outras finalidades. Na prática, funciona como se o Nó apontado por P1 simplesmente deixasse de existir. Veja, na situação da [Figura 4.8b](#), que o Nó que era apontado por P1 na

Figura 4.8a simplesmente sumiu! Mas note na [Figura 4.8b](#) que P1 ainda continua a existir e parece apontar “para o nada”. A operação `DeleteNode` desalocou o Nó apontado por P1, mas não desalocou P1. Nessa situação, P1 está apontando para uma posição de memória que a princípio não está mais armazenando uma informação consistente e precisa ser atualizado para não causar erros na sequência do programa. Aplicamos então a operação **P1 = Null** para atualizar o valor de P1, que passa a apontar para Null ([Figura 4.8c](#)).

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

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

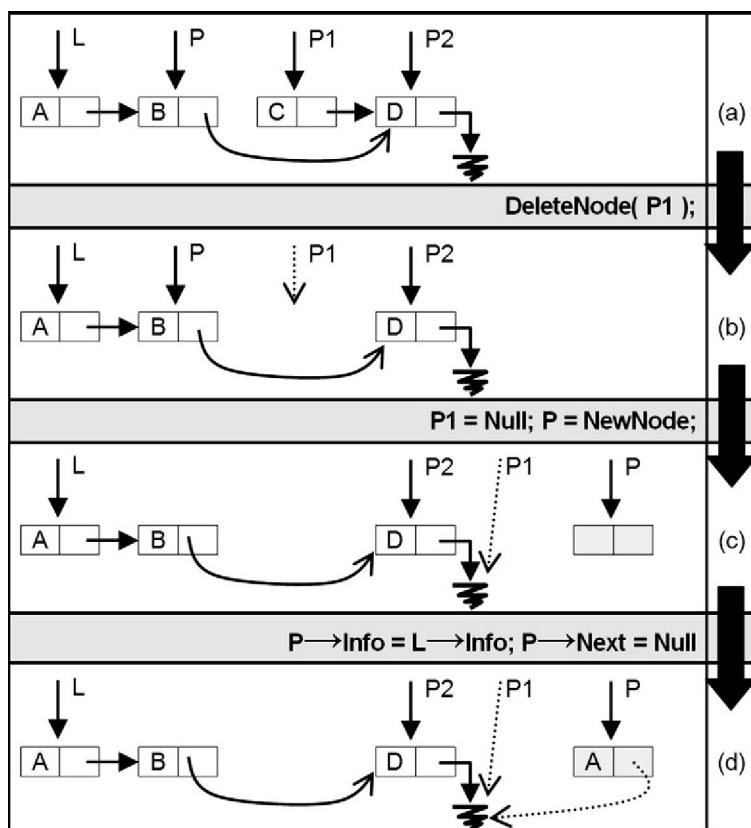


Figura 4.8 Operações para alocar e desalocar Nós.



4.3 Implementando uma Pilha como uma Lista Encadeada

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

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

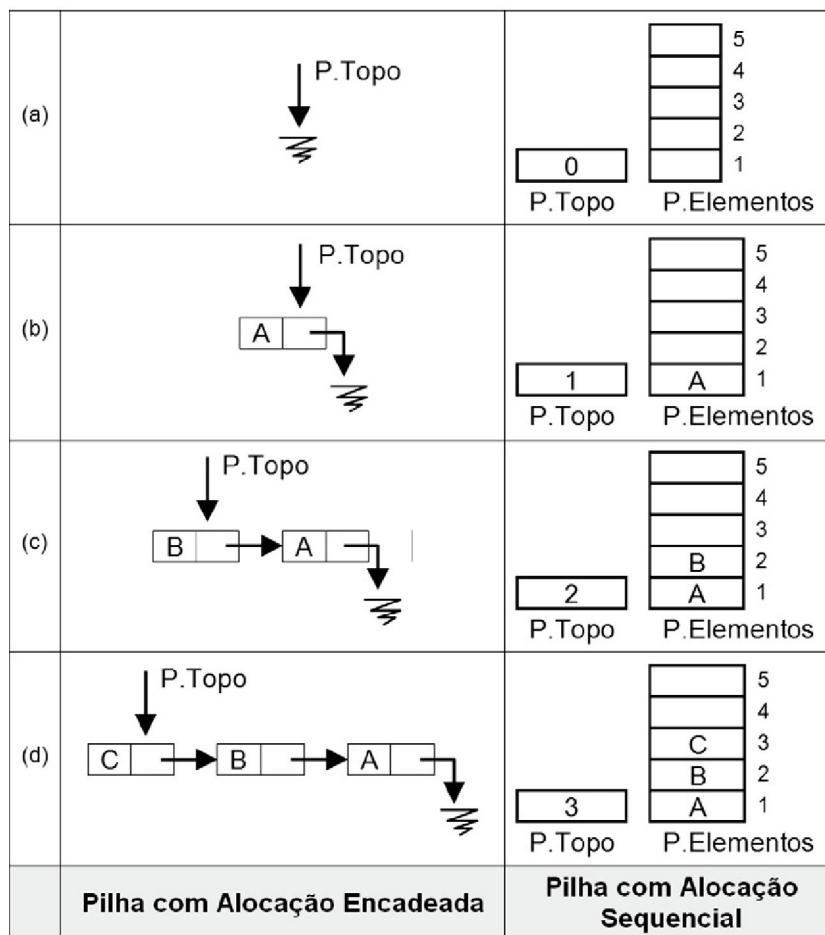


Figura 4.9 Pilha como Lista Encadeada.



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

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

Exercício 4.1 Operação Empilha

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

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

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

```
Empilha (parâmetro por referência P do tipo Pilha, parâmetro X do tipo Char, parâmetro por referência DeuCerto do tipo Boolean) {  
    /* Empilha o elemento X na Pilha P, ambos passados como parâmetro. O parâmetro DeuCerto deve indicar se a operação foi bem-sucedida ou não */  
    Variável PAux do tipo NodePtr;  
    /* PAux é uma variável auxiliar do tipo NodePtr. O tipo Pilha, nesta implementação encadeada, contém o campo P.Topo, que também é do tipo NodePtr, ou seja, um ponteiro para Nó, conforme definido na Figura 4.6 */  
    Se (Cheia(P)== Verdadeiro)           // se a Pilha P estiver cheia...  
        Então  DeuCerto = Falso;          // ... não podemos empilhar  
        Senão { PAux = NewNode;          // aloca um novo Nó  
                 PAux→Info = X;            // armazena o valor de X no novo Nó  
                 PAux→Next = P.Topo;       // o próximo deste novo nó será o elemento do topo  
                 P.Topo = PAux;           // o topo da pilha passa a ser o novo Nó  
                 DeuCerto = Verdadeiro;}; // a operação deu certo  
    } // fim do procedimento Empilha
```

Figura 4.10 Algoritmo conceitual — Empilha.

Vamos executar passo a passo esse algoritmo da operação Empilha? A [Figura 4.11](#) mostra a execução passo a passo da operação Empilha partindo da situação em que a Pilha P está vazia ([Figura 4.11a](#)).



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



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

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

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

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

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

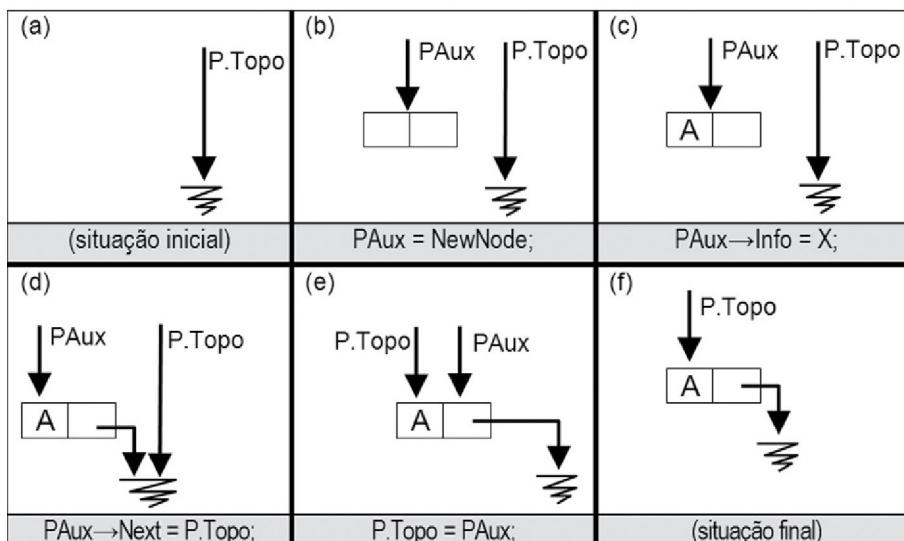


Figura 4.11 Execução da Operação Empilha partindo da situação inicial de Pilha Vazia.

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



A [Figura 4.11](#) mostrou a execução da operação Empilha tendo como situação inicial a Pilha P vazia. Com a execução da operação Empilha, a Pilha P passou a ter um elemento. Desenhe passo a passo a execução da operação Empilha partindo agora da situação inicial em que a Pilha P contém um elemento e levando a Pilha P a ter dois elementos. O elemento a ser inserido deve passar a ser o topo da Pilha. Considere que a Pilha nunca estará cheia.

Dica importante

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

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

Exercício 4.3 Operação Desempilha

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

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

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

Exercício 4.4 Operação Cria

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

Cria (parâmetro por referência P do tipo Pilha);

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

Exercício 4.5 Operação Vazia

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

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

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

Exercício 4.6 Operação Cheia

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

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

/* Nessa implementação conceitual, a operação cheia retorna sempre o valor Falso. Ou seja, a Pilha nunca estará cheia */

A [Figura 4.12](#) apresenta soluções para os Exercícios 4.3 a 4.6. Na operação Cria, basta apontar P para Null e estaremos criando uma Pilha vazia, conforme mostra o diagrama da [Figura 4.11a](#). A operação Vazia consiste em verificar se a Pilha está vazia ou não. Em uma Pilha vazia, nessa implementação encadeada, P.Topo aponta para Null ([Figura 4.11a](#)). Quando uma Pilha não é vazia, o valor de P.Topo é diferente de Null ([Figura 4.11f](#) ou coluna à esquerda das [Figuras 4.9b, 4.9c e 4.9d](#)). Conforme já mencionamos, nessa implementação conceitual de uma Pilha Encadeada a operação Cheia retorna sempre falso. Na operação Desempilha, o primeiro passo é verificar se a Pilha está vazia. Não é possível desempilhar um elemento de uma Pilha vazia. Caso a Pilha não estiver vazia, o parâmetro X retorna o valor do campo Info do nó apontado por P.Topo, ou seja, X retorna o valor

```
Cria (parâmetro por referência P do tipo Pilha) {
    /* Cria a Pilha P, inicializando a Pilha como vazia - sem nenhum elemento. */
    P.Topo = Null; // P.Topo passa a apontar para Null. Isso indica que a Pilha está vazia.
} // fim do Cria

Boolean Vazia (parâmetro por referência P do tipo Pilha) {
    /* Retorna Verdadeiro se a Pilha P estiver sem nenhum elemento; Falso caso contrário */
    Se (P.Topo == Null)
        Então Retorne Verdadeiro; // a Pilha P está vazia
    Senão Retorne Falso; // a Pilha P não está vazia
} // fim do Vazia

Boolean Cheia (parâmetro por referência P do tipo Pilha) {
    /* Nessa implementação conceitual, a operação cheia retorna sempre Falso */
    Retorne Falso; // nessa implementação conceitual, a Pilha nunca estará cheia.
} // fim do Cheia

Desempilha(parâmetro por referência P do tipo Pilha, parâmetro por referência X do tipo Char, parâmetro por referência DeuCerto do tipo Boolean) {
    /* Se a Pilha P estiver vazia, o parâmetro DeuCerto deve retornar Falso. Caso a Pilha P não esteja vazia, a operação Desempilha deve retornar o valor do elemento do Topo da Pilha no parâmetro X. O Nó em que se encontra o elemento do Topo deve ser desalocado, e o Topo da Pilha deve então ser atualizado para o próximo elemento */
    Variável PAux do tipo NodePtr; // ponteiro para Nó, auxiliar
    Se (Vazia( P ) == Verdadeiro)
        Então DeuCerto = Falso;
    Senão {
        DeuCerto = Verdadeiro;
        X = P.Topo→Info;
        PAux = P.Topo ; // aponta PAux para P.Topo, para guardar esse endereço
                        // antes de mudar P.Topo de lugar
        P.Topo = P.Topo→Next; // o topo da Pilha avança para o próximo elemento.
        DeleteNode( PAux ); // desaloca o Nó que armazenava o elemento do topo
    }
} // fim do Desempilha
```

Figura 4.12 Operações Cria, Vazia, Cheia e Desempilha.

do elemento que está no topo da Pilha ($X=P.\text{Topo} \rightarrow \text{Info}$) — [Figura 4.13b](#). Em seguida colocamos a variável auxiliar PAux apontando para onde aponta $P.\text{Topo}$ ($\text{PAux} = P.\text{Topo}$) — [Figura 4.13c](#). Avançamos o Topo da Pilha para o próximo elemento ($P.\text{Topo} = P.\text{Topo} \rightarrow \text{Next}$) — [Figura 4.13d](#) — e em seguida desalocamos o Nôo que continha o elemento que estava no Topo da Pilha ($\text{DeleteNode}(\text{PAux})$) — [Figura 4.13e](#). Para isso foi preciso guardar em PAux o endereço do Nôo que queríamos desalocar; note na [Figura 4.13d](#) que $P.\text{Topo}$ já não está mais apontando para esse Nôo, agora apontado apenas por PAux.

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

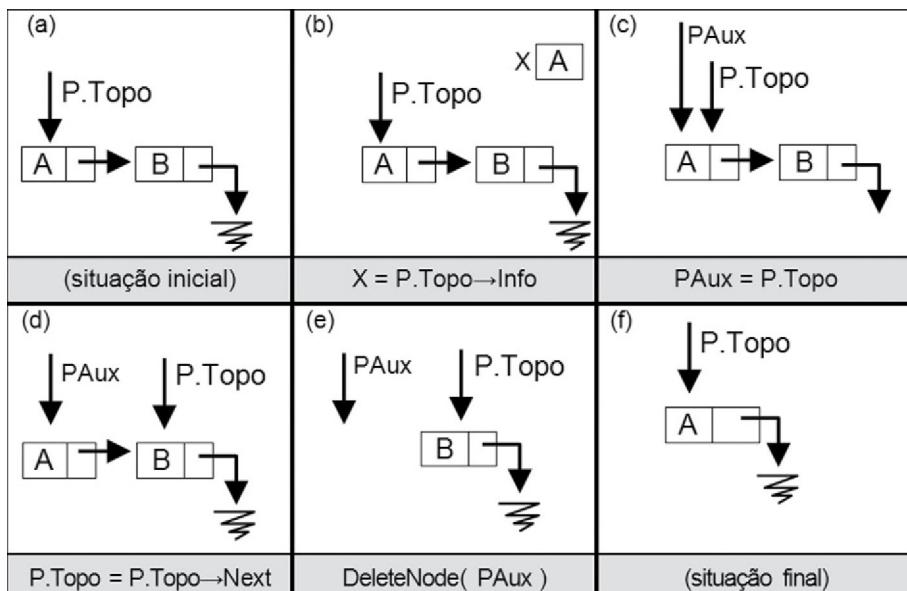


Figura 4.13 Execução da Operação Desempilha partindo da situação inicial de Pilha com dois elementos.

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

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

4.4 Implementando uma Fila como uma Lista Encadeada

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

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

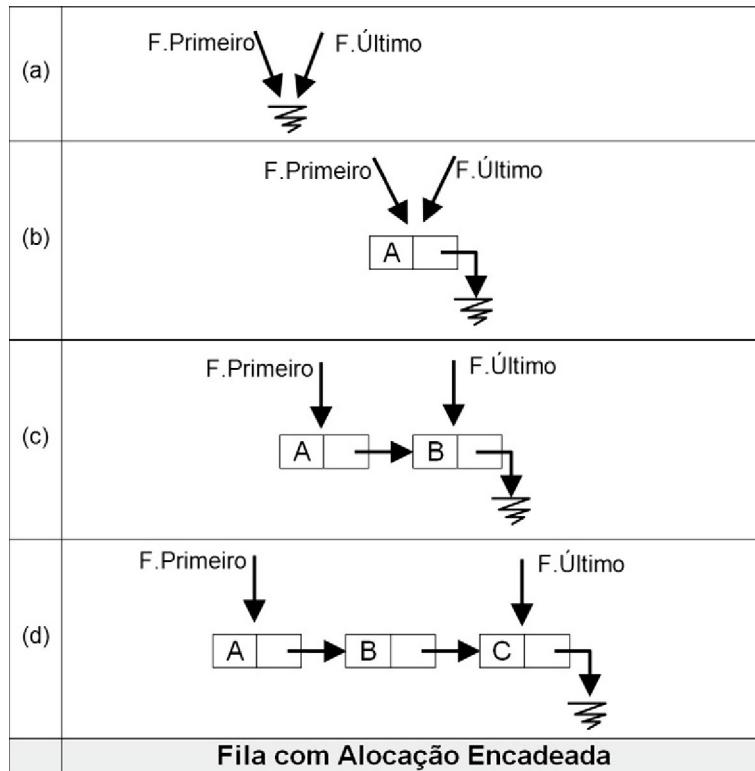


Figura 4.14 Fila como Lista Encadeada.

Exercício 4.8 Operação Cria a Fila

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

Cria (parâmetro por referência F do tipo Fila);

/* Cria a Fila F, inicializando como vazia - sem nenhum elemento - [Figura 4.14a](#) */

Exercício 4.9 Operação para testar se a Fila está vazia

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

Boolean Vazia (parâmetro por referência F do tipo Fila);

/* Retorna Verdadeiro se a Fila F estiver vazia - sem nenhum elemento; Falso caso contrário */

Exercício 4.10 Operação para testar se a Fila está cheia

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

Boolean Cheia (parâmetro por referência F do tipo Fila);

/* Nessa implementação conceitual, a operação cheia retorna sempre o valor Falso */

Exercício 4.11 Operação Insere na Fila

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

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

Exercício 4.12 Operação Retira da Fila

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

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

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

Exercício 4.13 Fila como Lista Encadeada Circular

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

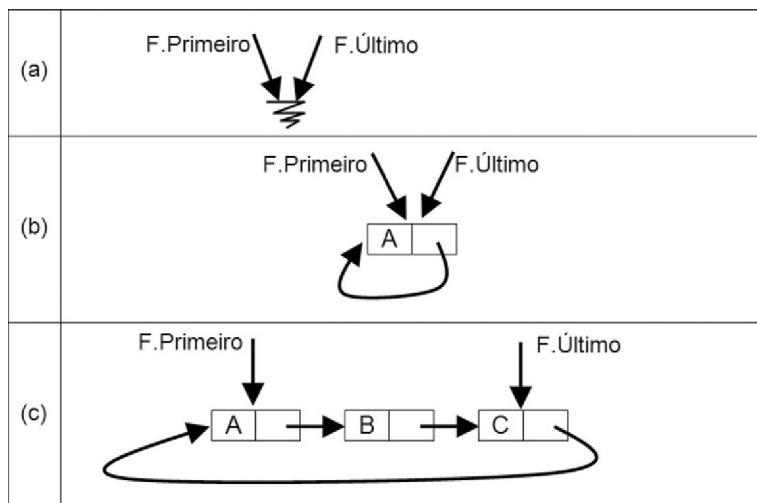


Figura 4.15 Fila como Lista Encadeada Circular.

Exercício 4.14 Fila Circular com Ponteiro só para o Último

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

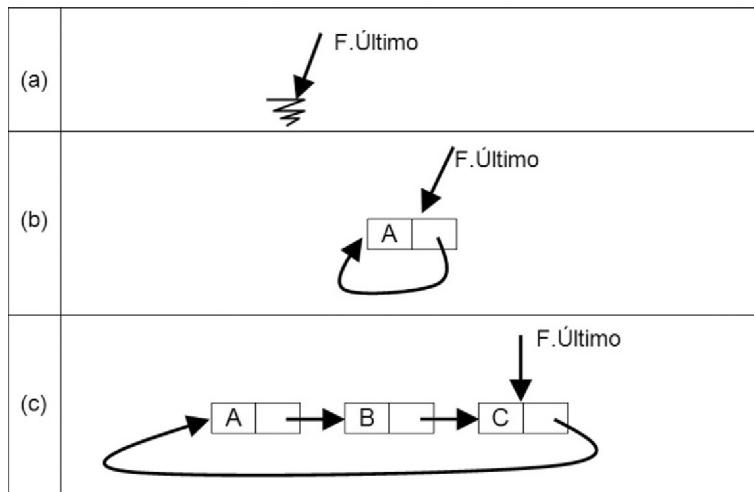


Figura 4.16 Fila como Lista Encadeada Circular com o ponteiro apenas para o Último elemento da Fila.

Exercício 4.15 Pilha como Lista Encadeada Circular

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

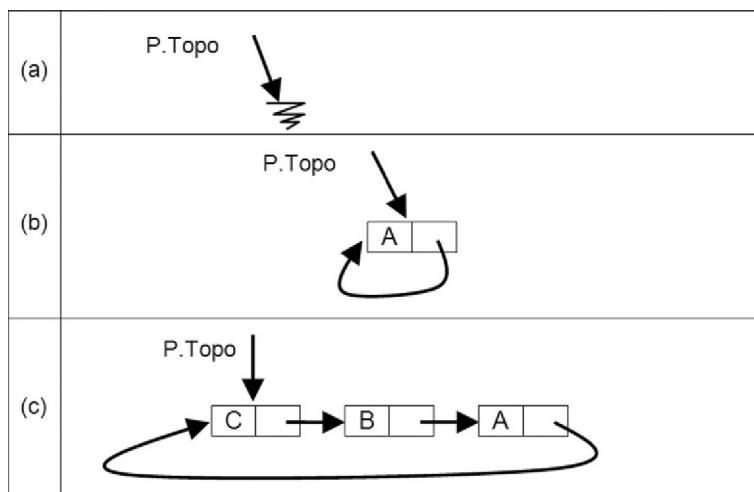


Figura 4.17 Pilha como Lista Encadeada Circular.

Consulte nos Materiais Complementares

Vídeos sobre Listas Encadeadas

Animações sobre Listas Encadeadas



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

Exercícios de fixação

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

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

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

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

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

Soluções para alguns dos exercícios

Exercícios 4.8, 4.9, 4.10, 4.11 e 4.12

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

Defina o tipo NodePtr = ponteiro para Node;

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

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

```

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

```

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

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

Variável PAux do tipo NodePtr;

```

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

```



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

Variável PAux do tipo NodePtr;

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

Referências e leitura adicional

Drozdek, A. *Estrutura de dados e algoritmos em C++*. São Paulo: Thomson, 2002.

Langsam, Y.; Augenstein, M. J.; Tenenbaum, A. M. *Data Structures Using C and C++*. 2nd ed. Upper Saddle River. New Jersey: Prentice Hall, 1996.