A Figura 10.2 mostra os efeitos das operações ENQUEUE e DEQUEUE. Cada operação demora o tempo $O(1)$.

### Exercícios

***10.1-1*** Usando a Figura 10.1 como modelo, ilustre o resultado de cada operação na sequência PUSH($S$, 4), PUSH($S$, 1), PUSH($S$, 3), POP($S$), PUSH($S$, 8) e POP($S$) sobre uma pilha $S$ inicialmente vazia armazenada no arranjo $S[1 .. 6]$.

***10.1-2*** Explique como implementar duas pilhas em um único arranjo $A[1 .. n]$ de tal modo que nenhuma delas sofra um estouro a menos que o número total de elementos em ambas as pilhas juntas seja $n$. As operações PUSH e POP devem ser executadas no tempo $O(1)$.

***10.1-3*** Usando a Figura 10.2 como modelo, ilustre o resultado de cada operação na sequência ENQUEUE($Q$, 4), ENQUEUE($Q$, 1), ENQUEUE($Q$, 3), DEQUEUE($Q$), ENQUEUE($Q$, 8) e DEQUEUE($Q$) em uma fila $Q$ inicialmente vazia armazenada no arranjo $Q[1 .. 6]$.

***10.1-4*** Reescreva ENQUEUE e DEQUEUE para detectar o estouro negativo e o estouro de uma fila.

***10.1-5*** Enquanto uma pilha permite inserção e eliminação de elementos em apenas uma extremidade e uma fila permite inserção em uma extremidade e eliminação na outra extremidade, uma ***deque*** (double-ended queue, ou fila de extremidade dupla) permite inserção e eliminação em ambas as extremidades. Escreva quatro procedimentos de tempo $O(1)$ para inserir elementos e eliminar elementos de ambas as extremidades de uma deque construída a partir de um arranjo.

***10.1-6*** Mostre como implementar uma fila usando duas pilhas. Analise o tempo de execução das operações em filas.

***10.1-7*** Mostre como implementar uma pilha usando duas filas. Analise o tempo de execução das operações em pilhas.

## 10.2 LISTAS LIGADAS

Uma ***lista ligada*** é uma estrutura de dados na qual os objetos estão organizados em ordem linear. Entretanto, diferentemente de um arranjo, no qual a ordem linear é determinada pelos índices do arranjo, a ordem em uma lista ligada é determinada por um ponteiro em cada objeto. Listas ligadas nos dão uma representação simples e flexível para conjuntos dinâmicos, suportando (embora não necessariamente com eficiência) todas as operações que aparecem na lista à página 166.

Como mostra a Figura 10.3, cada elemento de uma ***lista duplamente ligada*** $L$ é um objeto com um atributo *chave* e dois outros atributos *ponteiros*: *próximo* e *anterior*. O objeto também pode conter outros dados satélites. Dado um elemento $x$ na lista, $x.próximo$ aponta para seu sucessor na lista ligada e $x.anterior$ aponta para seu predecessor. Se $x.anterior = \text{NIL}$, o elemento $x$ não tem nenhum predecessor e, portanto, é o primeiro elemento, ou ***início***, da lista. Se $x.próximo = \text{NIL}$, o elemento $x$ não tem nenhum sucessor e, assim, é o último elemento, ou ***fim***, da lista. Um atributo $L.início$ aponta para o primeiro elemento da lista. Se $L.início = \text{NIL}$, a lista está vazia.

Uma lista pode ter uma entre várias formas. Ela pode ser simplesmente ligada ou duplamente ligada, pode ser ordenada ou não e pode ser circular ou não. Se uma lista é ***simplesmente ligada***, omitimos o ponteiro *anterior* em cada elemento. Se uma lista é ***ordenada***, a ordem linear da lista corresponde à ordem linear de chaves armazenadas em elementos da lista; então, o elemento mínimo é o início da lista, e o elemento máximo é o fim. Se a lista é ***não ordenada***, os elementos podem aparecer em qualquer ordem. Em uma ***lista circular***, o ponteiro *anterior* do início da lista aponta para o fim, e o ponteiro *próximo* do fim da lista aponta para o início. Podemos imaginar uma lista circular

como um anel de elementos. No restante desta seção, supomos que as listas com as quais estamos trabalhando são listas não ordenadas e duplamente ligadas.

## Como fazer uma busca em uma lista ligada

O procedimento LIST-SEARCH($L$, $k$) encontra o primeiro elemento com chave $k$ na lista $L$ por meio de uma busca linear simples, retornando um ponteiro para esse elemento. Se nenhum objeto com chave $k$ aparecer na lista, o procedimento retorna NIL. No caso da lista ligada da Figura 10.3(a), a chamada LIST-SEARCH($L$, 4) retorna um ponteiro para o terceiro elemento, e a chamada LIST-SEARCH($L$, 7) retorna NIL.

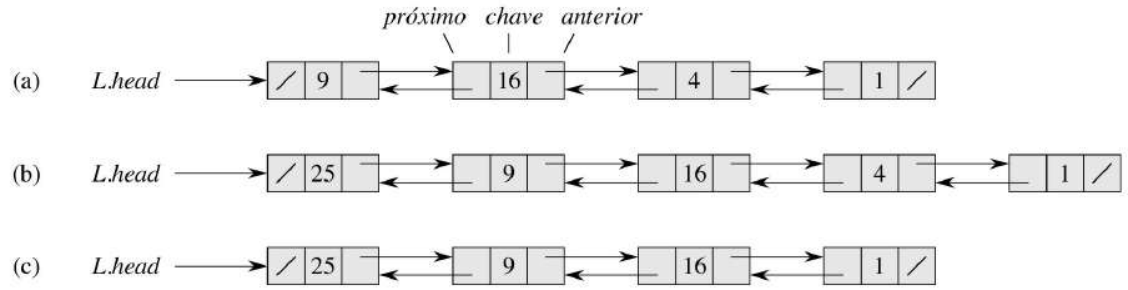


**Figura 10.3** (**a**) Uma lista duplamente ligada $L$ representando o conjunto dinâmico {1, 4, 9, 16}. Cada elemento na lista é um objeto com atributos para a chave e ponteiros (mostrados por setas) para o próximo objeto e para o objeto anterior. O atributo *próximo* do fim e o atributo *anterior* do início são NIL, indicados por uma barra diagonal. O atributo *L.início* aponta para o início. (**b**) Seguindo a execução de LIST-INSERT($L$, $x$), onde $x.chave = 25$, a lista ligada tem um novo objeto com chave 25 como o novo início. Esse novo objeto aponta para o antigo início com chave 9. (**c**) O resultado da chamada subsequente LIST-DELETE($L$, $x$), onde $x$ aponta para o objeto com chave 4.

```
LIST-SEARCH(L, k)
1  x = L.início
2  while x ≠ NIL e x.chave ≠ k
3      x = x.próximo
4  return x
```

Para fazer uma busca em uma lista de $n$ objetos, o procedimento LIST-SEARCH demora o tempo $Q(n)$ no pior caso, já que talvez tenha de pesquisar a lista inteira.

## Inserção em uma lista ligada

Dado um elemento $x$ cujo atributo *chave* já foi definido, o procedimento LIST-INSERT "emenda" $x$ à frente da lista ligada, como mostra a Figura 10.3(b).

```
LIST-INSERT(L, x)
1  x.próximo = L.início
2  if L.início ≠ NIL
3      L.início.anterior = x
4  L.início = x
5  x.anterior = NIL
```

(Lembre-se de que nossa notação de atributo pode ser usada em cascata, de modo que *L.início.anterior* denota o atributo *anterior* do objeto que *L.início* aponta.) O tempo de execução para LIST-INSERT para uma lista de $n$ elementos é $O(1)$.

## Eliminação em uma lista ligada

O procedimento LIST-DELETE remove um elemento $x$ de uma lista ligada $L$. Ele deve receber um ponteiro para $x$, e depois "desligar " $x$ da lista atualizando os ponteiros. Se desejarmos eliminar um elemento com determinada chave, deveremos primeiro chamar LIST-SEARCH, para reaver um ponteiro para o elemento.

```
LIST-DELETE(L, x)
1  if x.anterior ≠ NIL
2     x.anterior.próximo = x.próximo
3  else L.início = x.próximo
4  if x.próximo ≠ NIL
5     x.próximo.anterior = x.anterior
```

A Figura 10.3(c) mostra como um elemento é eliminado de uma lista ligada. LIST-DELETE é executado no tempo $O(1)$ mas, se desejarmos eliminar um elemento com uma dada chave, será necessário o tempo $Q(n)$ no pior caso porque primeiro devemos chamar LIST-SEARCH.

## Sentinelas

O código para LIST-DELETE seria mais simples se pudéssemos ignorar as condições de contorno no início e no fim da lista.

```
LIST-DELETE′(L, x)
1  x.anterior.próximo = x.próximo
2  x.próximo.anterior = x.anterior
```

Uma ***sentinela*** é um objeto fictício que nos permite simplificar condições de contorno. Por exemplo, suponha que suprimos com uma lista $L$ um objeto *L.nil* que representa NIL, mas tem todos os atributos dos outros objetos da lista. Onde quer que tenhamos uma referência a NIL no código da lista, nós a substituímos por uma referência à sentinela *L.nil.* Como mostra a Figura 10.4, essa mudança transforma uma lista duplamente ligada normal em uma ***lista circular duplamente ligada com uma sentinela***, na qual a sentinela *L.nil* se encontra entre o início e o fim; o atributo *L.nulo.próximo* aponta para o início da lista e *L.nil.anterior* aponta para o fim. De modo semelhante, tanto o atributo *próximo* do fim quanto o atributo *anterior* do início apontam para *L.nil.* Visto que *L.nil.próximo* aponta para o início, podemos eliminar totalmente o atributo *L.início*, substituindo as referências a ele por referências a *L.nil.próximo.* A Figura 10.4(a) mostra que uma lista vazia consiste apenas na sentinela e que *L.nil.próximo* e *L.nil.anterior* apontam para *L.nil*.

O código para LIST-SEARCH permanece o mesmo de antes, porém com as referências a NIL e *Linício* modificadas como já especificado:

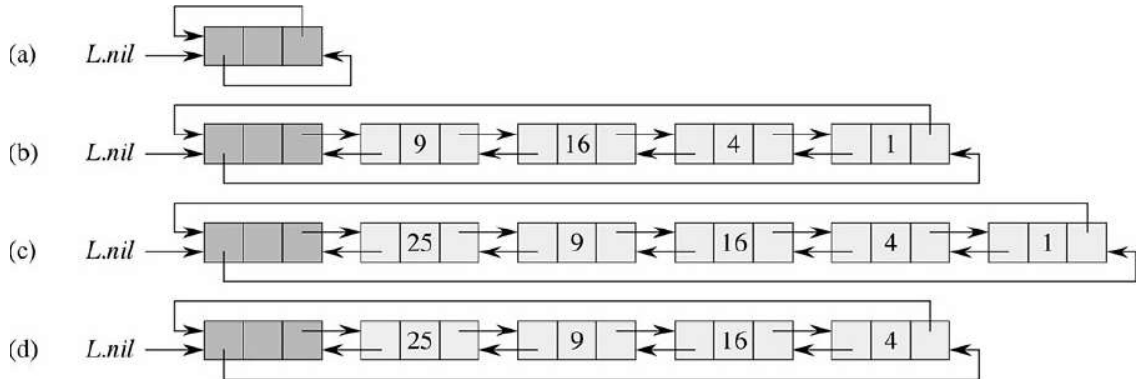


**Figura 10.4** Uma lista circular duplamente ligada com uma sentinela. A sentinela *L.nil* aparece entre o início e o fim. O atributo *L.início* não é mais necessário, visto que podemos acessar o início da lista por *L.nil.próximo*. (**a**) Uma lista vazia. (**b**) A lista ligada da Figura 10.3(a), com chave 9 no início e chave 1 no fim. (**c**) A lista após a execução de LIST-INSERT'$(L, x)$, onde $x.chave = 25$. O novo objeto se torna o início da lista. (**d**) A lista após a eliminação do objeto com chave 1. O novo fim é o objeto com chave 4.

```
LIST-SEARCH'(L, k)
1  x = L.nil.próximo
2  while x ≠ L.nil e x.chave ≠ k
3     x = x.próximo
4  return x
```

Usamos o procedimento de duas linhas LIST-DELETE' de antes para eliminar um elemento da lista. O seguinte procedimento insere um elemento na lista:

```
LIST-INSERT'(L, x)
1  x.próximo = L.nil.próximo
2  L.nil.próximo.anterior = x
3  L.nil.próximo = x
4  x.anterior = L.nil
```

A Figura 10.4 mostra os efeitos de LIST-INSERT' e LIST-DELETE' sobre uma amostra de lista.

Sentinelas raramente reduzem os limites assintóticos de tempo de operações de estrutura de dados, mas podem reduzir fatores constantes. O ganho da utilização de sentinelas dentro de laços em geral é uma questão de clareza de código em vez de velocidade; por exemplo, o código da lista ligada fica mais simples quando usamos sentinelas, mas poupamos apenas o tempo $O(1)$ nos procedimentos LIST-INSERT' e LIST-DELETE'. Contudo, em outras situações, a utilização de sentinelas ajuda a restringir o código em um laço, reduzindo assim o coeficiente de, digamos, $n$ ou $n_2$ no tempo de execução.

Devemos usar sentinelas com sensatez. Quando houver muitas listas pequenas, o armazenamento extra usado por suas sentinelas poderá representar desperdício significativo de memória. Neste livro, só utilizaremos sentinelas quando elas realmente simplificarem o código.

## Exercícios

***10.2-1*** Você pode implementar a operação de conjuntos dinâmicos INSERT em uma lista simplesmente ligada em tempo $O(1)$? E a operação DELETE ?

***10.2-2*** Implemente uma pilha usando uma lista simplesmente ligada $L$. As operações PUSH e POP ainda devem demorar o tempo $O(1)$.

***10.2-3*** Implemente uma fila por meio de uma lista simplesmente ligada $L$. As operações ENQUEUE e DEQUEUE ainda devem demorar o tempo $O(1)$.

***10.2-4*** Como está escrita, cada iteração do laço no procedimento LIST-SEARCH' exige dois testes: um para $x \neq L.nil$ e um para $x.chave \neq k$. Mostre como eliminar o teste para $x \neq L.nil$ em cada iteração.

***10.2-5*** Implemente as operações de dicionário INSERT, DELETE e SEARCH usando listas circulares simplesmente ligadas. Quais são os tempos de execução dos seus procedimentos?

***10.2-6*** A operação em conjuntos dinâmicos UNION utiliza dois conjuntos disjuntos $S_1$ e $S_2$ como entrada e retorna um conjunto $S = S_1 \cup S_2$ que consiste em todos os elementos de $S_1$ e $S_2$. Os conjuntos $S_1$ e $S_2$ são normalmente destruídos pela operação. Mostre como suportar UNION no tempo $O(1)$ usando uma estrutura de dados de lista adequada.

***10.2-7*** Dê um procedimento não recursivo de tempo $Q(n)$ que inverta uma lista simplesmente ligada de $n$ elementos. O procedimento só pode usar armazenamento constante além do necessário para a própria lista.

***10.2-8*** ★ Explique como implementar listas duplamente ligadas usando somente um valor de ponteiro $x.np$ por item, em vez dos dois valores usuais (*próximo* e *anterior*). Suponha que todos os valores de ponteiros podem ser interpretados como inteiros de $k$ bits e defina $x.np$ como $x.np = x.próximo$ XOR $x.anterior$ [$x$] o "ou exclusivo" de $k$ bits de $x.próximo$ e $x.anterior$. (O valor NIL é representado por 0.) Não esqueça de descrever as informações necessárias para acessar o início da lista. Mostre como implementar as operações SEARCH, INSERT e DELETE em tal lista. Mostre também como inverter essa lista em tempo $O(1)$.

## 10.3 IMPLEMENTAÇÃO DE PONTEIROS E OBJETOS

Como implementamos ponteiros e objetos em linguagens que não os oferecem? Nesta seção, veremos dois modos de implementar estruturas de dados ligadas sem um tipo de dados ponteiro explícito. Sintetizaremos objetos e ponteiros de arranjos e índices de arranjos.

### Uma representação de objetos em vários arranjos

Podemos representar uma coleção de objetos que têm os mesmos atributos usando um arranjo para cada atributo. Como exemplo, a Figura 10.5 mostra como podemos implementar a lista ligada da Figura 10.3(a) com três arranjos. A *chave* do arranjo contém os valores das chaves presentes atualmente no conjunto dinâmico, e os ponteiros são armazenados nos arranjos *próximo* e *anterior*. Para um dado índice de arranjo $x$, *chave*[$x$], *próximo*[$x$] e *anterior*[$x$] representam um objeto na lista ligada. Por essa interpretação, um ponteiro $x$ é simplesmente um índice comum para os arranjos *chave*, *próximo* e *anterior*.

Na Figura 10.3(a), o objeto com chave 4 vem após o objeto com chave 16 na lista ligada. Na Figura 10.5, chave 4 aparece em *chave*[2] e chave 16 aparece em *chave*[5]; assim, temos *próximo*[5] = 2 e *anterior*[2] = 5. Embora a constante NIL apareça no atributo *próximo* do fim e no atributo *anterior* do início, em geral usamos um inteiro (como 0 ou – 1) que não poderia, de modo algum, representar um índice real para os arranjos. Uma variável $L$ contém o índice do início da lista.

### Uma representação de objetos com um único arranjo

As palavras na memória de um computador, normalmente são endereçadas por inteiros de 0 a $M - 1$, onde $M$ é um inteiro adequadamente grande. Em muitas linguagens de programação, um objeto ocupa um conjunto contíguo de posições na memória do computador. Um ponteiro é simplesmente o endereço da primeira posição de memória do objeto, e podemos endereçar outras posições de memória dentro do objeto acrescentando um deslocamento ao ponteiro.

Podemos utilizar a mesma estratégia para implementar objetos em ambientes de programação que não fornecem ponteiros explícitos. Por exemplo, a Figura 10.6 mostra como usar um único arranjo $A$ para armazenar a lista ligada das Figuras 10.3(a) e 10.5. Um objeto ocupa um subarranjo contíguo $A[j \,..\, k]$. Cada atributo do objeto corresponde a um deslocamento na faixa 0 a $k - j$, e um ponteiro para o objeto é o índice $j$. Na Figura 10.6, os deslocamentos correspondentes a *chave*, *próximo* e *anterior* são 0, 1 e 2, respectivamente. Para ler o valor de *anterior*[$i$], dado um ponteiro $i$, adicionamos o valor $i$ do ponteiro ao deslocamento 2, lendo assim $A[i + 2]$.

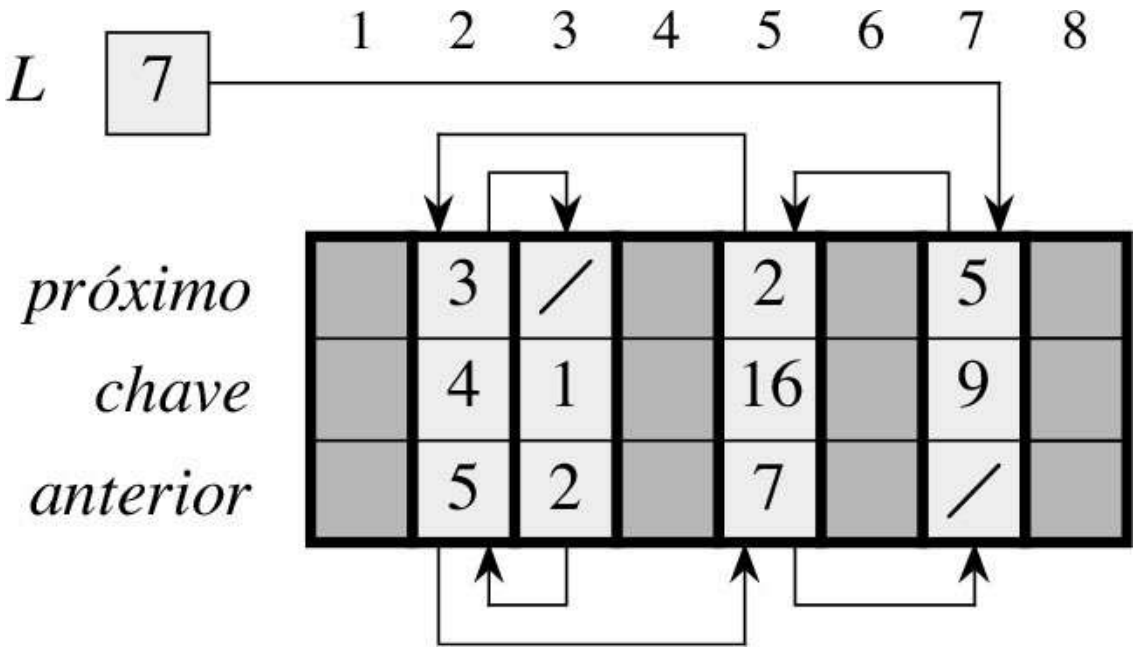


**Figura 10.5** A lista ligada da Figura 10.3(a) representada pelos arranjos *chave*, *próximo* e *anterior*. Cada fatia vertical dos arranjos representa um único objeto. Os ponteiros armazenados correspondem aos índices do arranjo mostrados na parte superior; as setas mostram como interpretá-los. As posições de objetos sombreadas em tom mais claro contêm elementos de listas. A variável $L$ mantém o índice do início.

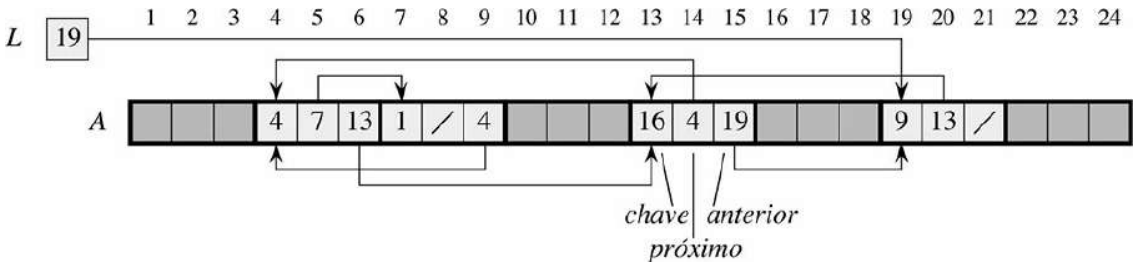


**Figura 10.6** A lista ligada das Figuras 10.3(a) e 10.5, representada em um único arranjo $A$. Cada elemento da lista é um objeto que ocupa um subarranjo contíguo de comprimento 3 dentro do arranjo. Os três atributos *chave*, *próximo* e *anterior* correspondem aos deslocamentos 0, 1 e 2, respectivamente, dentro de cada objeto. Um ponteiro para um objeto é um índice do primeiro elemento do objeto. Objetos que contêm elementos da lista estão sombreados em tom mais claro, e as setas mostram a ordenação da lista.

A representação de um único arranjo é flexível no sentido de que permite que objetos de diferentes comprimentos sejam armazenados no mesmo arranjo. O problema de administrar tal coleção heterogênea de objetos é mais difícil que o problema de administrar uma coleção homogênea, onde todos os objetos têm os mesmos atributos. Visto que a maioria das estruturas de dados que consideraremos são compostas por elementos homogêneos, será suficiente para nossa finalidade empregar a representação de objetos em vários arranjos.

## Alocação e liberação de objetos

Para inserir uma chave em um conjunto dinâmico representado por uma lista duplamente ligada, devemos alocar um ponteiro a um objeto que não está sendo utilizado na representação da lista ligada no momento considerado. Por isso, é útil gerenciar o armazenamento de objetos não utilizados na representação da lista ligada nesse momento, de tal modo que um objeto possa ser alocado. Em alguns sistemas, um ***coletor de lixo*** é responsável por determinar quais objetos não são utilizados. Porém, muitas aplicações são tão simples que podem assumir a responsabilidade pela devolução de um objeto não utilizado a um gerenciador de armazenamento. Agora, exploraremos o problema de alocar e liberar (ou desalocar) objetos homogêneos utilizando o exemplo de uma lista duplamente ligada representada por vários arranjos.

Suponha que os arranjos na representação de vários arranjos tenham comprimento $m$ e que em algum momento o conjunto dinâmico contenha $n \leq m$ elementos. Então, $n$ objetos representam elementos que se encontram atualmente no conjunto dinâmico, e os $m - n$ objetos restantes são ***livres***; os objetos livres estão disponíveis para representar elementos inseridos no conjunto dinâmico no futuro.

Mantemos os objetos livres em uma lista simplesmente ligada, que denominamos ***lista livre***. A lista livre usa apenas o arranjo *próximo*, que armazena os ponteiros *próximo* na lista. O início da lista livre está contido na variável global *livre*. Quando o conjunto dinâmico representado pela lista ligada $L$ é não vazio, a lista livre pode estar entrelaçada com a lista $L$, como mostra a Figura 10.7. Observe que cada objeto na representação está na lista $L$ ou na lista livre, mas não em ambas.

A lista livre funciona como uma pilha: o próximo objeto alocado é o último objeto liberado.

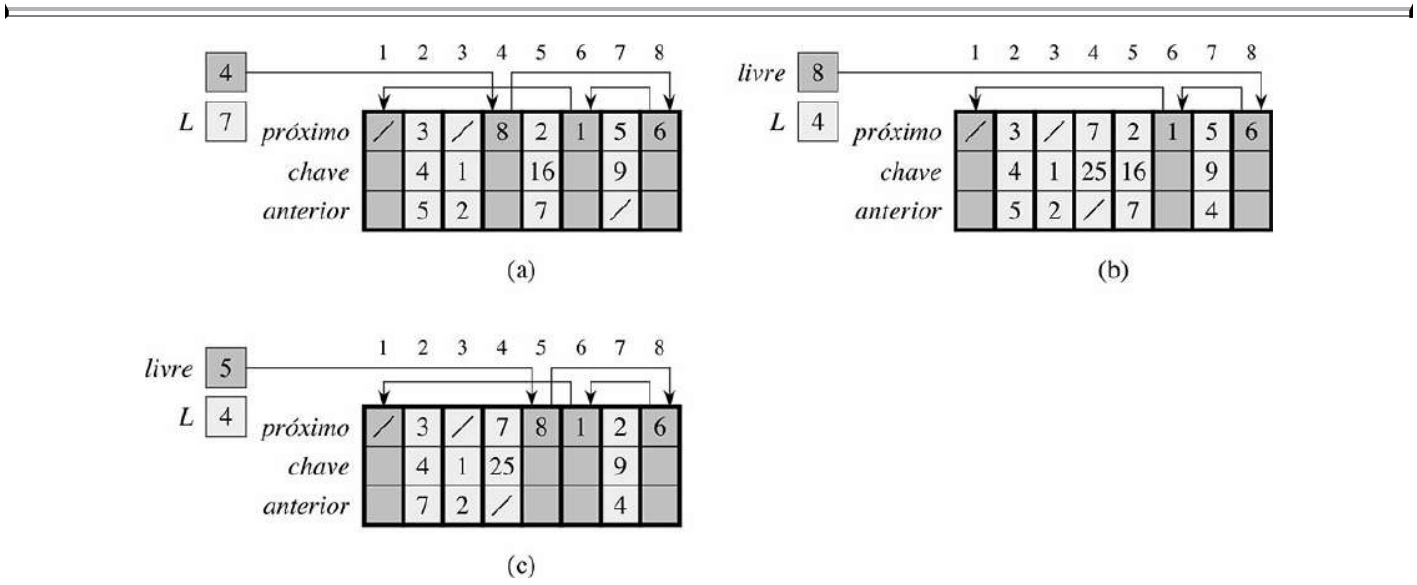


**Figura 10.7** O efeito dos procedimentos ALLOCATE-OBJECT e FREE-OBJECT. (**a**) A lista da Figura 10.5 (sombreada em tom mais claro) e uma lista livre (sombreada em tom mais escuro). As setas mostram a estrutura da lista livre. (**b**) O resultado da chamada ALLOCATE-OBJECT( ) (que retorna o *índice 4*), que define *chave*4 como 25, e chama LIST-INSERT($L$, 4). O novo início da lista livre é o objeto 8, que era *próximo*4 na lista livre. (**c**) Após executar LIST-DELETE($L$, 5), chamamos FREE-OBJECTS(5). O objeto 5 se torna o novo início da lista livre, seguido pelo objeto 8 na lista livre.

Podemos usar uma implementação de lista das operações de pilhas PUSH e POP, para implementar os procedimentos para alocar e liberar objetos, respectivamente. Consideramos que a variável global *livre* usada nos procedimentos a seguir, aponta para o primeiro elemento da lista livre.

```
ALLOCATE-OBJECT( )
1   if livre == NIL
2       error "out of space"
3   else x = livre
4       livre = x.próximo
5       return x

FREE-OBJECT(x)
1   x.próximo = livre
2   livre = x
```

A lista livre contém, inicialmente, todos os $n$ objetos não alocados. Assim que a lista livre é esgotada, o procedimento ALLOCATE-OBJECT sinaliza um erro. Podemos até mesmo atender a várias listas ligadas com apenas uma única lista livre. A Figura 10.8 mostra duas listas ligadas e uma lista livre entrelaçadas por meio de arranjos *chave*, *próximo* e *anterior*.

Os dois procedimentos são executados no tempo $O(1)$, o que os torna bastante práticos. Podemos modificá-los de modo que funcionem para qualquer coleção homogênea de objetos permitindo que qualquer um dos atributos no objeto aja como um atributo *próximo* na lista livre.

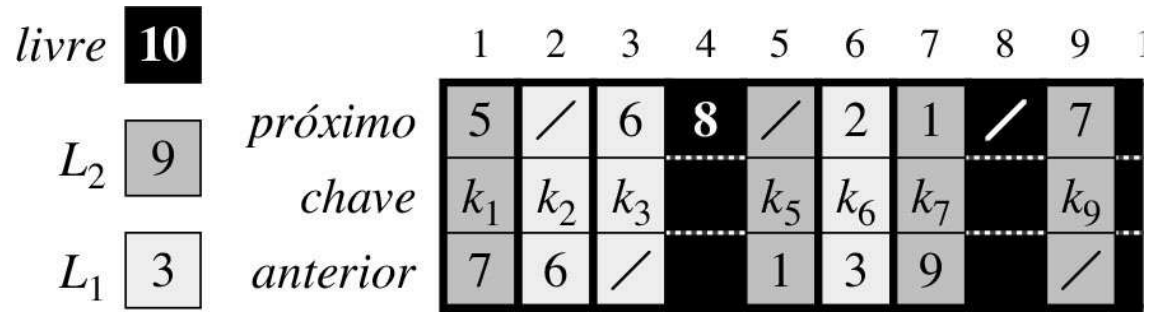


**Figura 10.8** Duas listas ligadas, $L_1$ (sombreada em tom mais claro) e $L_2$ (sombreada em tom mais escuro), e uma lista livre (em negro) entrelaçada.

## Exercícios

***10.3-1*** Trace um quadro da sequência $\langle 13, 4, 8, 19, 5, 11 \rangle$ armazenada como uma lista duplamente ligada utilizando a representação de vários arranjos. Faça o mesmo para a representação de um único arranjo.

***10.3-2*** Escreva os procedimentos ALLOCATE-OBJECT e FREE-OBJECT para uma coleção homogênea de objetos implementada pela representação de um único arranjo.

***10.3-3*** Por que não precisamos definir ou redefinir os atributos *anterior* de objetos na implementação dos procedimentos ALLOCATE-OBJECT e FREE-OBJECT?

***10.3-4*** Muitas vezes, é desejável manter todos os elementos de uma lista duplamente ligada de forma compacta no armazenamento usando, por exemplo, as primeiras $m$ posições do índice na representação de vários arranjos. (Esse é o caso em um ambiente de computação de memória virtual paginada.) Explique como implementar os procedimentos ALLOCATE-OBJECT e FREE-OBJECT de modo que a representação seja compacta. Suponha que não existem ponteiros para elementos da lista ligada fora da própria lista. (*Sugestão*: Use a implementação de uma pilha em arranjo.)

***10.3-5*** Seja $L$ uma lista duplamente ligada de comprimento $n$ armazenada em arranjos *chave*, *anterior* e *próximo* de comprimento $m$. Suponha que esses arranjos sejam gerenciados por procedimentos ALLOCATE-OBJECT e FREE-OBJECT que mantêm uma lista livre duplamente ligada $F$. Suponha ainda que, dos $m$ itens, exatamente $n$ estejam na lista $L$ e $m - n$ na lista livre. Escreva um procedimento COMPACTIFY-LIST($L$, $F$) que, dadas a lista $L$ e a lista livre $F$, desloque os itens em $L$ de modo que ocupem as posições de arranjo 1, 2, ..., $n$ e ajuste a lista livre $F$ para que ela permaneça correta, ocupando as posições de arranjo $m + 1$, $m + 2$, ..., $n$. O tempo de execução do seu procedimento deve ser $Q(m)$, e ele deve utilizar somente uma quantidade constante de espaço extra. Justifique que seu procedimento está correto.

## 10.4 REPRESENTAÇÃO DE ÁRVORES ENRAIZADAS

Os métodos para representar listas dados na seção anterior se estendem a qualquer estrutura de dados homogênea. Nesta seção, examinaremos especificamente o problema da representação de árvores enraizadas por estruturas de dados ligadas. Primeiro, veremos as árvores binárias e depois apresentaremos um método para árvores enraizadas nas quais os nós podem ter um número arbitrário de filhos.

Representamos cada nó de uma árvore por um objeto. Como no caso das listas ligadas, supomos que cada nó contém um atributo *chave*. Os atributos de interesse restantes são ponteiros para outros nós e variam de acordo com o tipo de árvore.

### Árvores binárias

A Figura 10.9 mostra como usamos os atributos $p$, *esquerdo* e *direito* para armazenar ponteiros para o pai, o filho da esquerda e o filho da direita de cada nó em uma árvore binária $T$. Se $x.p = \text{NIL}$, então $x$ é a raiz. Se o nó $x$ não tem nenhum filho à esquerda, então $x.esquerdo = \text{NIL}$, e o mesmo ocorre para o filho à direita. A raiz da árvore $T$ inteira é apontada pelo atributo $T.raiz$. Se $T.raiz = \text{NIL}$, então a árvore é vazia.

### Árvores enraizadas com ramificações ilimitadas

Podemos estender o esquema para representar uma árvore binária a qualquer classe de árvores na qual o número de filhos de cada nó seja no máximo alguma constante $k$: substituímos os atributos *esquerdo* e *direito* por $filho_1$, $filho_2$, ..., $filho_k$. Esse esquema deixa de funcionar quando o número de filhos de um nó é ilimitado, já que não sabemos quantos atributos (arranjos na representação de vários arranjos) devemos alocar antecipadamente. Além disso, ainda que o número de filhos $k$ seja limitado por uma constante grande, mas a maioria dos nós tenha um número pequeno de filhos, é possível que desperdicemos grande quantidade de memória.

Felizmente, existe um esquema inteligente para representar árvores com números arbitrários de filhos. Tal esquema tem a vantagem de utilizar somente o espaço $O(n)$ para qualquer árvore enraizada de $n$ nós. A ***representação filho da esquerda, irmão da direita*** aparece na Figura 10.10. Como antes, cada nó contém um ponteiro pai $p$, e $T.raiz$ aponta para a raiz da árvore $T$. Contudo, em vez de ter um ponteiro para cada um de seus filhos, cada nó $x$ tem somente dois ponteiros: