*10.1-3*
Usando a Figura 10.2 como modelo, ilustre o resultado de cada operação na seqüência ENQUEUE($Q$, 4), ENQUEUE($Q$, 1), ENQUEUE($Q$, 3), DEQUEUE($Q$), ENQUEUE($Q$, 8) e DEQUEUE($Q$) sobre 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 positivo de uma fila.

*10.1-5*
Enquanto uma pilha permite a inserção e a eliminação de elementos em apenas uma extremidade e uma fila permite a inserção em uma extremidade e a eliminação na outra extremidade, uma ***deque*** (double-ended queue, ou fila de extremidade dupla) permite a inserção e a 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 sobre filas.

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

## 10.2 Listas ligadas

Uma ***lista ligada*** é uma estrutura de dados em que os objetos estão organizados em uma ordem linear. Entretanto, diferente 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. As listas ligadas fornecem uma representação simples e flexível para conjuntos dinâmicos, admitindo (embora não necessariamente de modo eficiente) todas as operações listadas na introdução à Parte III, seção "Operações sobre conjuntos dinâmicos".

Como mostra a Figura 10.3, cada elemento de uma ***lista duplamente ligada*** $L$ é um objeto com um campo de *chave* e dois outros campos de ponteiros: *próximo* e *anterior*. O objeto também pode conter outros dados satélite. Sendo dado um elemento $x$ na lista, *próximo*$[x]$ aponta para seu sucessor na lista ligada, e *anterior*$[x]$ aponta para seu predecessor. Se *anterior*$[x]$ = NIL, o elemento $x$ não tem nenhum predecessor e portanto é o primeiro elemento, ou ***início***, da lista. Se *próximo*$[x]$ = NIL, o elemento $x$ não tem nenhum sucessor e assim é o último elemento, ou ***fim***, da lista. Um atributo *início*$[L]$ aponta para o primeiro elemento da lista. Se *início*$[L]$ = 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; 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. Desse modo, a lista pode ser vista 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 pesquisar em uma lista ligada

O procedimento LIST-SEARCH($L$, $k$) encontra o primeiro elemento com a chave $k$ na lista $L$ através de uma pesquisa linear simples, retornando um ponteiro para esse elemento. Se nenhum objeto com a chave $k$ aparecer na lista, então NIL será retornado. 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.

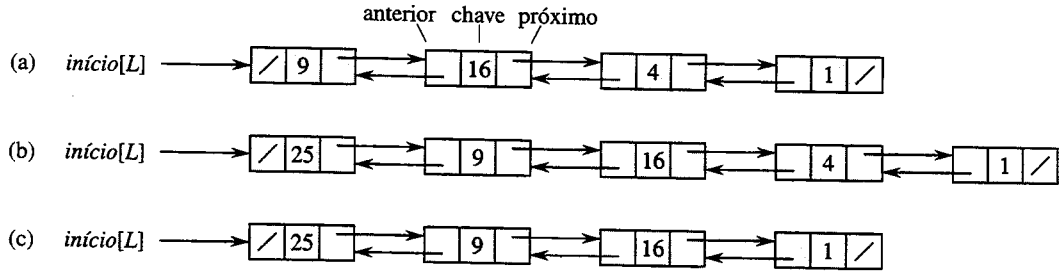


FIGURA 10.3 (a) Uma lista duplamente ligada $L$ representando o conjunto dinâmico $\{1, 4, 9, 16\}$. Cada elemento na lista é um objeto com campos para a chave e ponteiros (mostrados por setas) para os objetos próximo e anterior. O campo *próximo* do fim e o campo *anterior* do início são NIL, indicados por uma barra em diagonal. O atributo *início*[$L$] aponta para o início. (b) Seguindo a execução de LIST-INSERT($L$, $x$), onde *chave*[$x$] = 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 subseqüente LIST-DELETE($L$, $x$), onde $x$ aponta para o objeto com chave 4

```
LIST-SEARCH(L, k)
1 x ← início[L]
2 while x ≠ NIL e chave[x] ≠ k
3     do x ← próximo[x]
4 return x
```

Para pesquisar em uma lista de $n$ objetos, o procedimento LIST-SEARCH demora o tempo $\Theta(n)$ no pior caso, pois pode ter de pesquisar a lista inteira.

## Inserção de elementos em uma lista ligada

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

```
LIST-INSERT(L, x)
1 próximo[x] ← início[L]
2 if início[L] ≠ NIL
3     then anterior[início[L]] ← x
4 início[L] ← x
5 anterior[x] ← NIL
```

O tempo de execução para LIST-INSERT sobre uma lista de $n$ elementos é $O(1)$.

## Eliminação de elementos de 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 "retirar" $x$ da lista, atualizando os ponteiros. Se desejarmos eliminar um elemento com uma determinada chave, deveremos primeiro chamar LIST-SEARCH, a fim de recuperar um ponteiro para o elemento.

```
LIST-DELETE(L, x)
1 if anterior[x] ≠ NIL
2    then próximo[anterior[x]] ← próximo[x]
3    else início[L] ← próximo[x]
4 if próximo[x] ≠ NIL
5    then anterior[próximo[x]] ← anterior[x]
```

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 $\Theta(n)$ no pior caso, porque primeiro devemos chamar LIST-SEARCH.

## Sentinelas

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

```
LIST-DELETE'(L, x)
1 próximo[anterior[x]] ← próximo[x]
2 anterior[próximo[x]] ← anterior[x]
```

Uma ***sentinela*** é um objeto fictício que nos permite simplificar condições limites. Por exemplo, vamos supor que fornecemos com a lista $L$ um objeto *nulo*[$L$] que representa NIL, mas tem todos os campos dos outros elementos da lista. Onde quer que tenhamos uma referência NIL no código da lista, vamos substituí-la por uma referência à sentinela *nulo*[$L$]. Como vemos na Figura 10.4, isso transforma uma lista duplamente ligada normal em uma ***lista circular, duplamente ligada com uma sentinela***, na qual a sentinela *nulo*[$L$] é colocada entre o início e o fim; o campo *próximo*[*nulo*[$L$]] aponta para o início da lista, enquanto *anterior*[*nulo*[$L$]] aponta para o fim. De modo semelhante, tanto o campo *próximo* do fim quanto o campo *anterior* do início apontam para *nulo*[$L$]. Tendo em vista que *próximo*[*nulo*[$L$]] aponta para o início, podemos eliminar totalmente o atributo *início*[$L$], substituindo as referências a ele por referências a *próximo*[*nulo*[$L$]]. Uma lista vazia consiste apenas na sentinela, pois tanto *próximo*[*nulo*[$L$]] quanto *anterior*[*nulo*[$L$]] podem ser definidos como *nulo*[$L$].

O código para LIST-SEARCH permanece o mesmo de antes, mas tem as referências a NIL e *início*[$L$] modificadas do modo especificado antes.

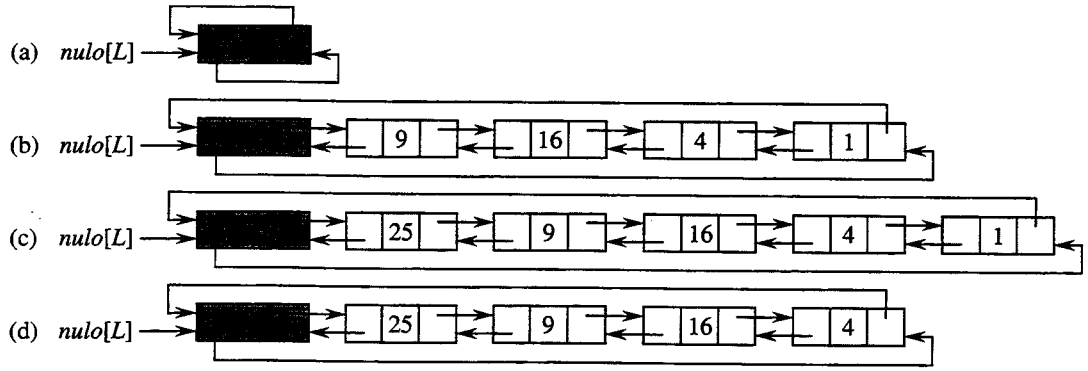


FIGURA 10.4 Uma lista circular duplamente ligada com uma sentinela. A sentinela *nulo*[$L$] aparece entre o início e o fim. O atributo *início*[$L$] não é mais necessário, pois podemos obter acesso ao início da lista por *próximo*[*nulo*[$L$]]. **(a)** Uma lista vazia. **(b)** A lista ligada da Figura 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 *chave*[$x$] = 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 ← próximo[nulo[L]]
2 while x ≠ nulo[L] e chave[x] ≠ k
3     do x ← próximo[x]
4 return x
```

Usamos o procedimento de duas linhas LIST-DELETE' para eliminar um elemento da lista. Utilizamos o procedimento a seguir para inserir um elemento na lista.

```
LIST-INSERT'(L, x)
1 próximo[x] ← próximo[nulo[L]]
2 anterior[próximo[nulo[L]]] ← x
3 próximo[nulo[L]] ← x
4 anterior[x] ← nulo[L]
```

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

As sentinelas raramente reduzem os limites assintóticos de tempo de operações de estrutura de dados, mas podem reduzir fatores constantes. O ganho de se utilizar sentinelas dentro de loops em geral é uma questão de clareza de código, em vez de velocidade; por exemplo, o código da lista ligada é simplificado pelo uso de sentinelas, mas poupamos apenas o tempo $O(1)$ nos procedimentos LIST-INSERT' e LIST-DELETE'. Contudo, em outras situações, o uso de sentinelas ajuda a tornar mais compacto o código em um loop, reduzindo assim o coeficiente de, digamos, $n$ ou $n^2$ no tempo de execução.

As sentinelas não devem ser usadas indiscriminadamente. Se houver muitas listas pequenas, o espaço de armazenamento extra usado por suas sentinelas poderá representar um desperdício significativo de memória. Neste livro, só utilizaremos sentinelas quando elas realmente simplificarem o código.

## Exercícios

### *10.2-1*

A operação sobre conjuntos dinâmicos INSERT pode ser implementada sobre uma lista simplesmente ligada em tempo $O(1)$? E no caso de 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 através 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 de loop no procedimento LIST-SEARCH' exige dois testes: um para $x \neq nulo[L]$ e um para $chave[x] \neq k$. Mostre como eliminar o teste para $x \neq nulo[L]$ 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 sobre conjuntos dinâmicos UNION utiliza como entrada dois conjuntos disjuntos $S_1$ e $S_2$ 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 oferecer suporte a UNION no tempo $O(1)$ usando uma estrutura de dados de lista apropriada.

*10.2-7*

Forneça um procedimento não recursivo de tempo $\Theta(n)$ que inverta uma lista simplesmente ligada de $n$ elementos. O procedimento não deve usar nada mais além do espaço de armazenamento constante necessário para a própria lista.

*10.2-8* ⋆

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

## 10.3 Implementação de ponteiros e objetos

De que modo implementamos ponteiros e objetos em linguagens como Fortran, que não os oferecem? Nesta seção, veremos duas maneiras de implementar estruturas de dados ligadas sem um tipo de dados ponteiro explícito. Sintetizaremos objetos e ponteiros a partir de arranjos e índices de arranjos.

### Uma representação de objetos com vários arranjos

Podemos representar uma coleção de objetos que têm os mesmos campos usando um arranjo para cada campo. 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. Sob 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 segue o objeto com chave 16 na lista ligada. Na Figura 10.5, a chave 4 aparece em $chave[2]$ e a chave 16 aparece em $chave[5]$; assim, temos $próximo[5] = 2$ e $anterior[2] = 5$. Embora a constante NIL apareça no campo *próximo* do fim e no campo *anterior* do início, em geral usamos um inteiro (como 0 ou – 1) que não tem possibilidade de representar um índice real para os arranjos. Uma variável $L$ contém o índice do início da lista.

Em nosso pseudocódigo, temos usado colchetes para denotar tanto a indexação de um arranjo quanto a seleção de um campo (atributo) de um objeto. De qualquer modo, os significados de $chave[x]$, $próximo[x]$ e $anterior[x]$ são consistentes com a prática de implementação.

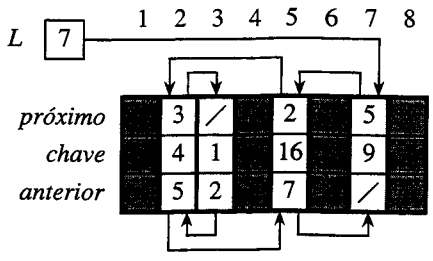


FIGURA 10.5 A lista ligada da Figura 10.3(a) representada pelos arranjos *chave*, *próximo* e *anterior*. Cada fatia vertical dos arranjos representa um objeto único. Os ponteiros armazenados correspondem aos índices do arranjo mostrados na parte superior; as setas mostram como interpretá-los. As posições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