Capítulo 10

Estruturas de dados elementares

Neste capítulo, examinaremos a representação de conjuntos dinâmicos por estruturas de dados simples que usam ponteiros. Embora muitas estruturas de dados complexas possam ser modeladas com a utilização de ponteiros, apresentaremos apenas as estruturas rudimentares: pilhas, filas, listas ligadas e árvores enraizadas. Também discutiremos um método pelo qual objetos e ponteiros podem ser sintetizados a partir de arranjos.

10.1 Pilhas e filas

As pilhas e filas são conjuntos dinâmicos nos quais o elemento removido do conjunto pela operação DELETE é especificado previamente. Em uma ***pilha***, o elemento eliminado do conjunto é o mais recentemente inserido: a pilha implementa uma norma de ***último a entrar, primeiro a sair***, ou ***LIFO*** (last-in, first-out). De modo semelhante, em uma ***fila***, o elemento eliminado é sempre o que esteve no conjunto pelo tempo mais longo: a fila implementa uma norma de ***primeiro a entrar, primeiro a sair***, ou ***FIFO*** (first-in, first-out). Existem vários modos eficientes de implementar pilhas e filas em um computador. Nesta seção, mostraremos como usar um arranjo simples para implementar cada uma dessas estruturas.

Pilhas

A operação INSERT sobre uma pilha é chamada com freqüência PUSH, e a operação DELETE, que não toma um argumento de elemento, é freqüentemente chamada POP. Esses nomes são alusões a pilhas físicas, como as pilhas de pratos usados em restaurantes. A ordem em que os pratos são retirados da pilha é o oposto da ordem em que eles são colocados sobre a pilha e, como conseqüência, apenas o prato do topo está acessível.

Como mostra a Figura 10.1, podemos implementar uma pilha de no máximo n elementos com um arranjo $S[1..n]$. O arranjo tem um atributo $topo[S]$ que realiza a indexação do elemento inserido mais recentemente. A pilha consiste nos elementos $S[1 .. topo[S]]$, onde $S[1]$ é o elemento na parte inferior da pilha e $S[topo[S]]$ é o elemento na parte superior (ou no topo).

Quando $topo[S] = 0$, a pilha não contém nenhum elemento e está ***vazia***. É possível testar se a pilha está vazia, através da operação de consulta STACK-EMPTY. Se uma pilha vazia sofre uma operação de extração, dizemos que a pilha tem um ***estouro negativo***, que é normalmente um erro. Se $topo[S]$ excede n, a pilha tem um ***estouro positivo***. (Em nossa implementação de pseudocódigo, não nos preocuparemos com o estouro de pilhas.)

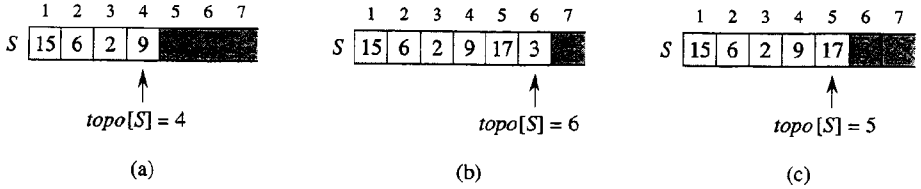

FIGURA 10.1 Uma implementação de arranjo de uma pilha S. Os elementos da pilha só aparecem nas posições levemente sombreadas. (a) A pilha S tem 4 elementos. O elemento do topo é 9. (b) A pilha S após as chamadas PUSH(S, 17) e PUSH(S, 3). (c) A pilha S após a chamada POP(S) retornou o elemento 3, que é o elemento mais recentemente inserido na pilha. Embora o elemento 3 ainda apareça no arranjo, ele não está mais na pilha; o elemento do topo é o elemento 17

Cada uma das operações sobre pilhas pode ser implementada com algumas linhas de código.

```
STACK-EMPTY(S)
1 if topo[S] = 0
2    then return TRUE
3    else return FALSE

PUSH(S, x)
1 topo[S] ← topo[S] + 1
2 S[topo[S]] ← x

POP(S)
1 if STACK-EMPTY(S)
2    then error "underflow"
3    else topo[S] ← topo[S] – 1
4         return S[topo[S] + 1]
```

A Figura 10.1 mostra os efeitos das operações de modificação PUSH (EMPILHAR) e POP (DESEMPILHAR). Cada uma das três operações sobre pilhas demora o tempo $O(1)$.

Filas

Chamamos a operação INSERT sobre uma fila de ENQUEUE (ENFILEIRAR), e também a operação DELETE de DEQUEUE (DESINFILEIRAR); como a operação sobre pilhas POP, DEQUEUE não tem nenhum argumento de elemento. A propriedade FIFO de uma fila faz com que ela opere como uma fileira de pessoas no posto de atendimento da previdência social. A fila tem um ***início*** (ou cabeça) e um ***fim*** (ou cauda). Quando um elemento é colocado na fila, ele ocupa seu lugar no fim da fila, como um aluno recém-chegado que ocupa um lugar no final da fileira. O elemento retirado da fila é sempre aquele que está no início da fila, como o aluno que se encontra no começo da fileira e que esperou por mais tempo. (Felizmente, não temos de nos preocupar com a possibilidade de elementos computacionais "furarem" a fila.)

A Figura 10.2 mostra um modo de implementar uma fila de no máximo $n - 1$ elementos usando um arranjo $Q[1 .. n]$. A fila tem um atributo $início[Q]$ que indexa ou aponta para seu início. O atributo $fim[Q]$ realiza a indexação da próxima posição na qual um elemento recém-chegado será inserido na fila. Os elementos na fila estão nas posições $início[Q]$, $início[Q] + 1$, ..., $fim[Q] - 1$, onde "retornamos", no sentido de que a posição 1 segue imediatamente a posição n em uma ordem circular. Quando $início[Q] = fim[Q]$, a fila está vazia. Inicialmente, temos $início[Q] = fim[Q] = 1$. Quando a fila está vazia, uma tentativa de retirar um elemento da fila provoca o estouro negativo da fila. Quando $início[Q] = fim[Q] + 1$, a fila está cheia, e uma tentativa de colocar um elemento na fila provoca o estouro positivo da fila.

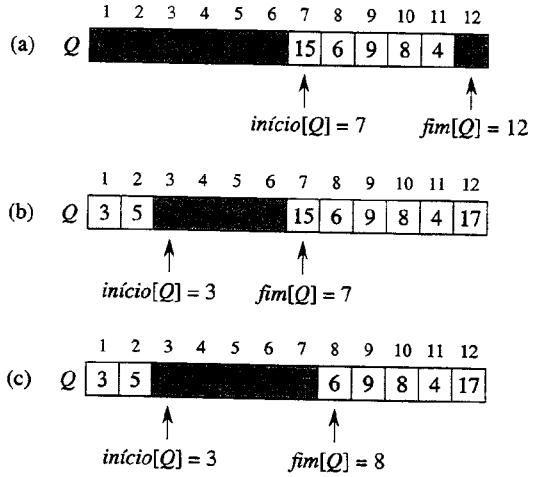

FIGURA 10.2 Uma fila implementada com a utilização de um arranjo $Q[1..12]$. Os elementos da fila aparecem apenas nas posições levemente sombreadas. (a) A fila tem 5 elementos, nas localizações $Q[7..11]$. (b) A configuração da fila depois das chamadas ENQUEUE(Q, 17), ENQUEUE(Q, 3) e ENQUEUE(Q, 5). (c) A configuração da fila depois da chamada DEQUEUE(Q) retorna o valor de chave 15 que se encontrava anteriormente no início da fila. O novo início tem a chave 6

Em nossos procedimentos ENQUEUE e DEQUEUE, a verificação de erros de estouro negativo (underflow) e estouro positivo (overflow) foi omitida. (O Exercício 10.1-4 lhe pede para fornecer o código que efetua a verificação dessas duas condições de erro.)

```
ENQUEUE(Q, x)
1 Q[fim[Q]] ← x
2 if fim[Q] = comprimento[Q]
3    then fim[Q] ← 1
4    else fim[Q] ← fim[Q] + 1
```

```
DEQUEUE(Q)
1 x ← Q[início[Q]]
2 if início[Q] = comprimento[Q]
3    then início[Q] ← 1
4    else início[Q] ← início[Q] + 1
5 return x
```

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 seqüência 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 das pilhas sofra um estouro positivo, 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 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.

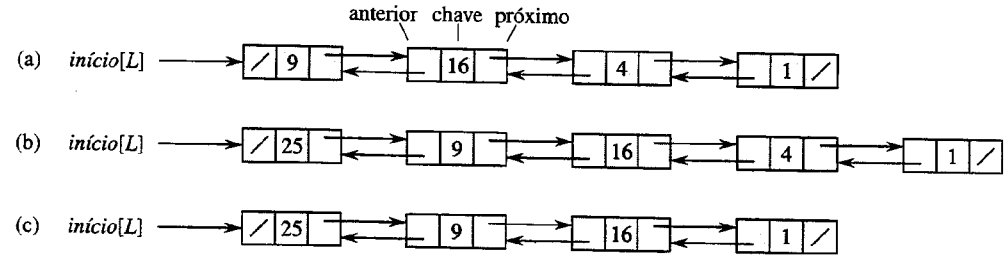

FIGURA 10.3 (a) Uma lista duplamente ligada L representando o conjunto dinâmico $\{1, 4, 9, 16\}$. Cada elemento na lista é um objeto com campos para a chave e ponteiros (mostrados por setas) para os objetos próximo e anterior. O campo *próximo* do fim e o campo *anterior* do início são NIL, indicados 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.

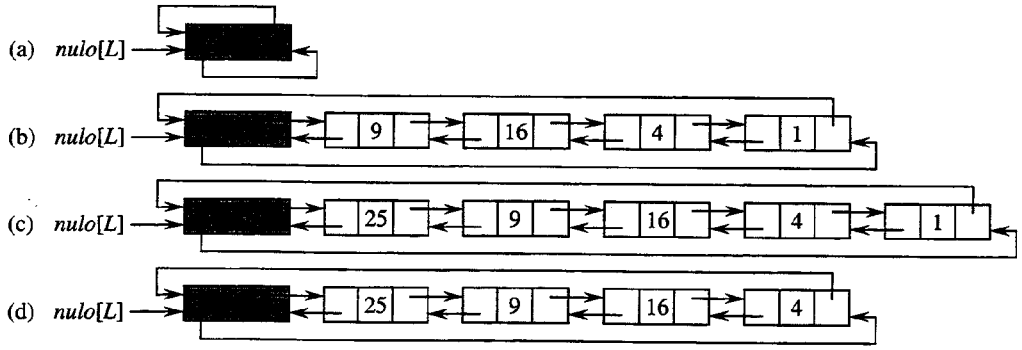

FIGURA 10.4 Uma lista circular duplamente ligada com uma sentinela. A sentinela *nulo*[L] aparece entre o início e o fim. O atributo *início*[L] não é mais necessário, pois podemos obter acesso ao início da lista por *próximo*[*nulo*[L]]. (a) Uma lista vazia. (b) A lista ligada da Figura 10.3(a), com 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.

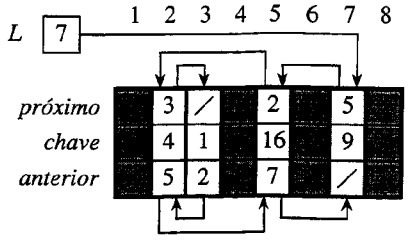

FIGURA 10.5 A lista ligada da Figura 10.3(a) representada pelos arranjos *chave*, *próximo* e *anterior*. Cada fatia vertical dos arranjos representa um objeto único. Os ponteiros armazenados correspondem aos índices do arranjo mostrados na parte superior; as setas mostram como interpretá-los. As posições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

Uma representação de objetos com um único arranjo

As palavras na memória de um computador normalmente são endereçadas por inteiros desde 0 até $M-1$, 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 outras posições de memória dentro do objeto podem ser indexadas pela inclusão de um deslocamento para o ponteiro.

Podemos utilizar a mesma estratégia para implementar objetos em ambientes de programação que não fornecem tipos de dados ponteiro explícito. Por exemplo, a Figura 10.6 mostra como um único arranjo A pode ser usado para armazenar a lista ligada das Figuras 10.3(a) e 10.5. Um objeto ocupa um subarranjo contíguo $A[j \,..\, k]$. Cada campo do objeto corresponde a um deslocamento no intervalo de 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 $A[i+2]$.

A representação de um único arranjo é flexível pelo fato de permitir 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 campos. Tendo em vista que a maioria das estruturas de dados que consideraremos são compostas por elementos homogêneos, será suficiente para nossos propósitos empregar a representação de objetos de vários arranjos.

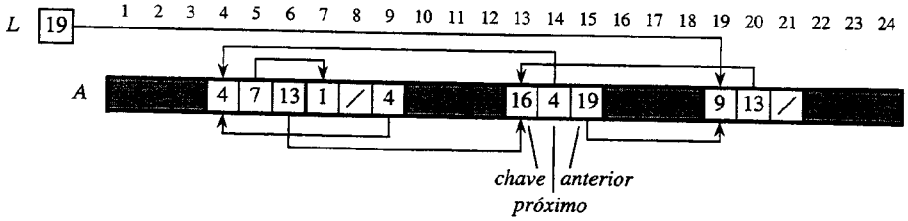

FIGURA 10.6 A lista ligada das Figuras 10.3(a) e 10.5, representada em um único arranjo A. Cada elemento da lista é um objeto que ocupa um subarranjo contíguo de comprimento 3 dentro do arranjo. Os três campos *chave*, *próximo* e *anterior* correspondem aos deslocamentos 0, 1 e 2, respectivamente. Um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

Alocação e liberação de objetos

Para inserir uma chave em um conjunto dinâmico representado por uma lista duplamente ligada, devemos alocar um ponteiro para um objeto atualmente não utilizado na representação da lista ligada. Assim, é útil gerenciar o espaço de armazenamento de objetos não utilizados no momento na representação da lista ligada, de tal modo que um objeto possa ser alocado. Em alguns sistemas, um ***coletor de lixo*** é responsável por determinar quais objetos não serão utilizados. Porém, muitas aplicações são simples o bastante para poderem assumir a responsabilidade pela devolução de um objeto não utilizado a um gerenciador de espaço 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 têm comprimento m e que em algum momento o conjunto dinâmico contém $n \le m$ elementos. Então, n objetos representam elementos que se encontram atualmente no conjunto dinâmico, e os $m-n$ objetos restantes são ***livres***; os objetos livres podem ser usados para representar elementos inseridos no conjunto dinâmico no futuro.

Mantemos os objetos livres em uma lista simplesmente ligada, que chamamos ***lista livre***. A lista livre usa apenas o arranjo *próximo*, que armazena os ponteiros *próximo* dentro da 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 ser 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.

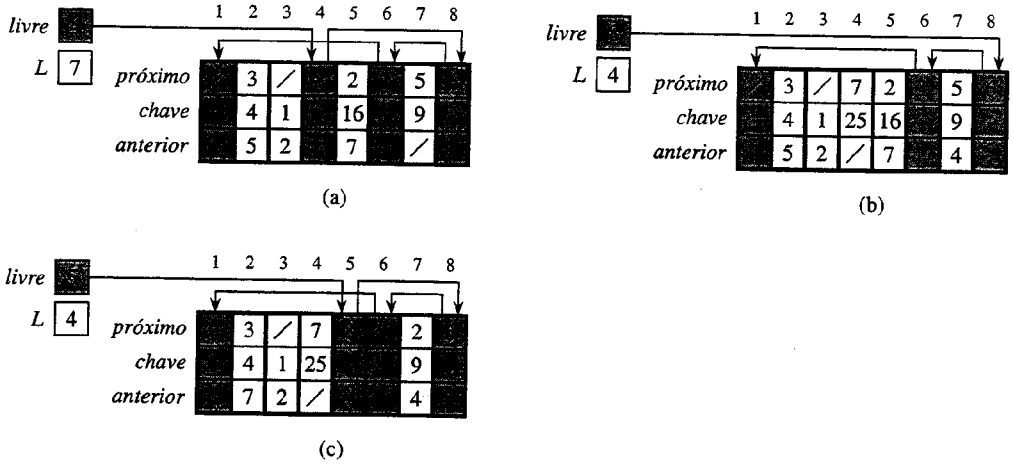

FIGURA 10.7 O efeito dos procedimentos ALLOCATE-OBJECT e FREE-OBJECT. (**a**) A lista da Figura 10.5 (levemente sombreada) e uma lista livre (fortemente sombreada). As setas mostram a estrutura da lista livre. (**b**) O resultado da chamada ALLOCATE-OBJECT( ) (que retorna o *índice* 4), definindo *chave*[4] como 25, e da chamada a LIST-INSERT(L, 4). O novo início da lista livre é o objeto 8, que era *próximo*[4] na lista livre. (**c**) Depois de executar LIST-DELETE(L, 5), chamamos FREE-OBJECT(5). O objeto 5 se torna o novo início da lista livre, seguido pelo objeto 8 na lista livre

A lista livre é uma pilha: o próximo objeto alocado é o último objeto liberado. Podemos usar uma implementação de lista das operações de pilhas PUSH e POP, a fim de implementar os procedimentos para alocar e liberar objetos, respectivamente. Supomos 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    then error "out of space"
3    else x ← livre
4         livre ← próximo[x]
5         return x

FREE-OBJECT(x)
1 próximo[x] ← livre
2 livre ← x
```

A lista livre contém inicialmente todos os n objetos não alocados. Quando a lista livre é esgotada, o procedimento ALLOCATE-OBJECT assinala um erro. É comum o uso de uma única lista livre para servir várias listas ligadas. A Figura 10.8 mostra duas listas ligadas e uma lista livre entrelaçada através dos arranjos *chave*, *próximo* e *anterior*.

Os dois procedimentos são executados no tempo $O(1)$, o que os torna bastante práticos. Eles podem ser modificados com o objetivo de funcionarem para qualquer coleção homogênea de objetos, permitindo que qualquer um dos campos no objeto atue como um campo *próximo* na lista livre.

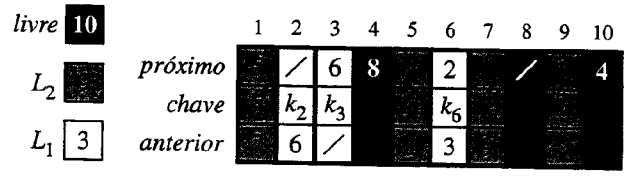

FIGURA 10.8 Duas listas ligadas, L_1 (levemente sombreada) e L_2 (fortemente sombreada), e uma lista livre (escurecida) entrelaçada

Exercícios

10.3-1

Trace um quadro da seqüência $\langle 13, 4, 8, 19, 5, 11 \rangle$ 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 campos *anterior* de objetos na implementação dos procedimentos ALLOCATE-OBJECT e FREE-OBJECT?

10.3-4

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

10.3-5

Seja L uma lista duplamente ligada de comprimento m armazenada nos arranjos *chave*, *anterior* e *próximo* de comprimento n. Suponha que esses arranjos sejam gerenciados pelos procedimentos ALLOCATE-OBJECT e FREE-OBJECT, que mantêm uma lista livre duplamente ligada F. Suponha ainda que, dos n itens, exatamente m itens estejam na lista L e $n - m$ estejam na lista livre. Escreva um procedimento COMPACTIFY-LIST(L, F) que, dada a lista L e a lista livre F, desloque os itens em L de forma que eles ocupem as posições de arranjos $1, 2, \ldots, m$ e ajuste a lista livre F para que ela permaneça correta, ocupando as posições de arranjos $m + 1, m + 2, \ldots, n$. O tempo de execução do seu procedimento deve ser $\Theta(m)$, e ele só deve utilizar uma quantidade constante de espaço extra. Forneça um argumento cuidadoso para justificar a correção do seu procedimento.

10.4 Representação de árvores enraizadas

Os métodos para representação de listas dados na seção anterior se estendem a qualquer estrutura de dados homogêneos. 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, vamos supor que cada nó contém um campo *chave*. Os campos restantes de interesse são ponteiros para outros nós, e eles variam de acordo com o tipo de árvore.

Árvores binárias

Como mostra a Figura 10.9, usamos os campos p, *esquerdo* e *direito* com o objetivo de armazenar ponteiros para o pai, o filho da esquerda e o filho da direita de cada nó em uma árvore binária T. Se $p[x]$ = NIL, então x é a raiz. Se o nó x não tem nenhum filho da esquerda, então *esquerdo*$[x]$ = NIL, e temos uma situação semelhante para o filho da direita. A raiz da árvore T inteira é apontada pelo atributo *raiz*$[T]$. Se *raiz*$[T]$ = NIL, então a árvore é vazia.

Árvores enraizadas com ramificações ilimitadas

O esquema para representar uma árvore binária pode ser estendido a qualquer classe de árvores na qual o número de filhos de cada nó seja no máximo alguma constante k: substituímos os campos *esquerdo* e *direito* por $filho_1, filho_2, \ldots, filho_k$. Esse esquema não funciona mais quando o número de filhos de um nó é ilimitado, pois não sabemos quantos campos (arranjos na representação de vários arranjos) devemos alocar antecipadamente. Além disso, mesmo que o número de filhos k seja limitado por uma constante grande, mas a maioria dos nós tenha um número pequeno de filhos, poderemos desperdiçar uma grande quantidade de memória.

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

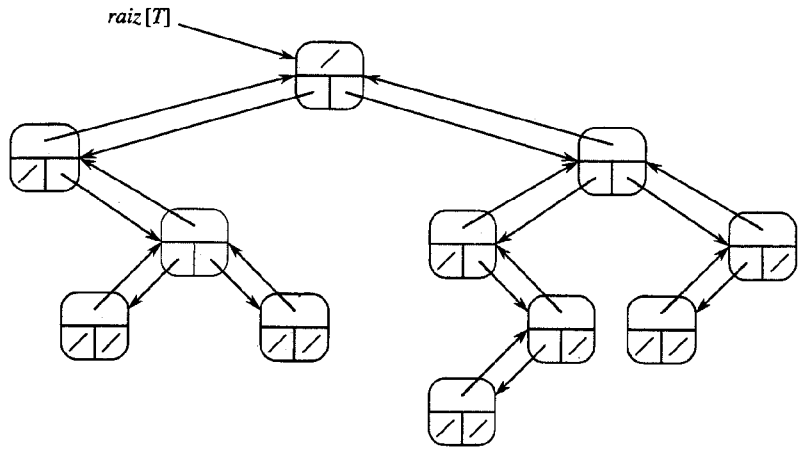

FIGURA 10.9 A representação de uma árvore binária T. Cada nó x tem os campos $p[x]$ (superior), *esquerdo*$[x]$ (inferior esquerdo) e *direito*$[x]$ (inferior direito). Os campos *chave* não são mostrados

1. *Filho da esquerda*$[x]$ aponta para o filho da extremidade esquerda do nó x, e
2. *Irmão da direita*$[x]$ aponta para o irmão de x situado imediatamente à direita.

Se o nó x não tem nenhum filho, então *filho da esquerda*$[x]$ = NIL e, se o nó x é o filho da extremidade direita de seu pai, então *irmão da direita*$[x]$ = NIL.

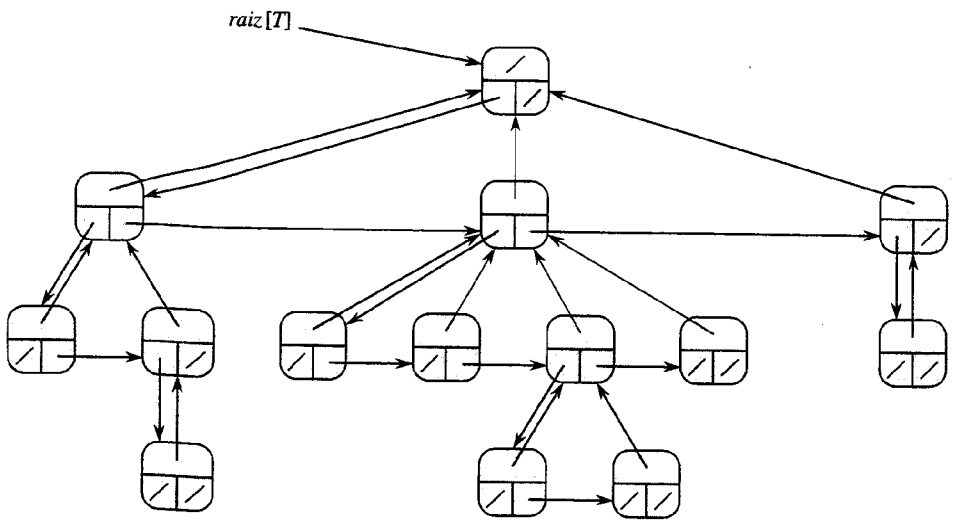

FIGURA 10.10 A representação de filho da esquerda, irmão da direita de uma árvore T. Cada nó x tem campos $p[x]$ (superior), *filho da esquerda*$[x]$ (inferior esquerdo) e *irmão da direita*$[x]$ (inferior direito). As chaves não são mostradas

Outras representações de árvores

Algumas vezes, representamos árvores enraizadas de outras maneiras. Por exemplo, no Capítulo 6, representamos um heap (monte), que se baseia em uma árvore binária completa, por um único arranjo e mais um índice. As árvores que aparecem no Capítulo 21 são atravessadas somente em direção à raiz; assim, apenas os ponteiros superiores estão presentes: não há ponteiros para filhos. Muitos outros esquemas são possíveis. O melhor esquema dependerá da aplicação.

Exercícios

10.4-1

Desenhe a árvore binária enraizada no índice 6 que é representada pelos campos a seguir.

índice	chave	esquerdo	direito
1	12	7	3
2	15	8	NIL
3	4	10	NIL
4	10	5	9
5	2	NIL	NIL
6	18	1	4
7	7	NIL	NIL
8	14	6	2
9	21	NIL	NIL
10	5	NIL	NIL

10.4-2

Escreva um procedimento recursivo de tempo $O(n)$ que, dada uma árvore binária de n nós, imprima a chave de cada nó na árvore.

10.4-3

Escreva um procedimento não recursivo de tempo $O(n)$ que, dada uma árvore binária de n nós, imprima a chave de cada nó na árvore. Use uma pilha como estrutura de dados auxiliar.

10.4-4

Escreva um procedimento de tempo $O(n)$ que imprima todas as chaves de uma árvore enraizada arbitrária com n nós, onde a árvore é armazenada usando a representação de filho da esquerda, irmão da direita.

10.4-5 ⋆

Escreva um procedimento não recursivo de tempo $O(n)$ que, dada uma árvore binária de n nós, imprima a chave de cada nó. Não utilize mais que um espaço extra constante fora da própria árvore e não modifique a árvore, mesmo temporariamente, durante o procedimento.

10.4-6 ⋆

A representação de filho da esquerda, irmão da direita de uma árvore enraizada arbitrária utiliza três ponteiros em cada nó: *filho da esquerda*, *irmão da direita* e *pai*. A partir de qualquer nó, seu pai pode ser alcançado e identificado em tempo constante, e todos os seus filhos podem ser alcançados e identificados em tempo linear no número de filhos. Mostre como usar apenas dois ponteiros e um valor booleano em cada nó para que o pai de um nó ou todos os seus filhos possam ser alcançados e identificados em tempo linear no número de filhos.

Problemas

10-1 Comparações entre listas

Para cada um dos quatro tipos de listas na tabela a seguir, qual é o tempo de execução assintótico no pior caso para cada operação sobre conjuntos dinâmicos listada?

	não ordenada, simplesmente ligada	ordenada, simplesmente ligada	não ordenada, duplamente ligada	ordenada, duplamente ligada
SEARCH(L, k)				
INSERT(L, x)				
DELETE(L, x)				
SUCCESSOR(L, x)				
PREDECESSOR(L, x)				
MINIMUM(L)				
MAXIMUM(L)				

10-2 Heaps intercaláveis com o uso de listas ligadas

Um heap intercalável admite as seguintes operações: MAKE-HEAP (que cria um heap intercalável vazio), INSERT, MINIMUM, EXTRACT-MIN e UNION.[1] Mostre como implementar heaps intercaláveis com o uso de listas ligadas em cada um dos casos a seguir. Procure tornar cada operação tão eficiente quanto possível. Analise o tempo de execução de cada operação em termos do tamanho do(s) conjunto(s) dinâmico(s) sobre o(s) qual(is) é realizada a operação.

[1] Tendo em vista que definimos um heap intercalável para dar suporte a MINIMUM e EXTRACT-MIN, também podemos nos referir a ele como um heap intercalável mínimo. Como outra alternativa, se o heap admitisse MAXIMUM e EXTRACT-MAX, ele seria um heap intercalável máximo.

a. As listas são ordenadas.

b. As listas são não ordenadas.

c. As listas são não ordenadas, e os conjuntos dinâmicos a serem intercalados são disjuntos.

10-3 Pesquisa em uma lista compacta ordenada

O Exercício 10.3-4 perguntou como poderíamos manter compacta uma lista de n elementos nas primeiras n posições de um arranjo. Vamos supor que todas as chaves sejam distintas e que a lista compacta também seja ordenada; ou seja $chave[i] < chave[próximo[i]]$ para todo $i = 1, 2, ..., n$ tal que $próximo[i] \neq$ NIL. Sob essas hipóteses, você mostrará que o algoritmo aleatório a seguir pode ser usado para pesquisar a lista no tempo esperado $O(\sqrt{n})$.

```
COMPACT-LIST-SEARCH(L, n, k)
 1 i ← início[L]
 2 while i ≠ NIL e chave[i] < k
 3       do j ← RANDOM(1, n)
 4          if chave[i] < chave[j] e chave[j] ≤ k
 5             then i ← j
 6                  if chave[i] = k
 7                     then return i
 8          i ← próximo[i]
 9 if i = NIL or chave[i] > k
10    then return NIL
11    else return i
```

Se ignorarmos as linhas 3 a 7 do procedimento, teremos um algoritmo comum para pesquisa em uma lista ligada ordenada, na qual o índice i aponta por sua vez para cada posição da lista. A pesquisa termina uma vez que o índice i "cai" no final da lista ou uma vez que $chave[i] \geq k$. Nesse último caso, se $chave[i] = k$, torna-se claro que encontramos uma chave com o valor k. Se, porém, $chave[i] > k$, isso significa que nunca encontraremos uma chave com o valor k, e então encerrar a pesquisa é a ação correta.

As linhas 3 a 7 tentam saltar à frente, até uma posição j escolhida aleatoriamente. Esse salto será vantajoso se $chave[j]$ for maior que $chave[i]$ e não maior que k; em tal caso, j marcará uma posição na lista pela qual i teria de passar durante uma pesquisa comum na lista. Pelo fato de a lista ser compacta, sabemos que qualquer escolha de j entre 1 e n efetua a indexação de algum objeto na lista, em vez de uma abertura na lista livre.

Em lugar de analisar o desempenho de COMPACT-LIST-SEARCH diretamente, analisaremos um algoritmo relacionado, COMPACT-LIST-SEARCH', que executa dois loops separados. Esse algoritmo utiliza um parâmetro adicional t que determina um limite superior sobre o número de iterações do primeiro loop.

```
COMPACT-LIST-SEARCH'(L, n, k, t)
 1 i ← início[L]
 2 for q ← 1 to t
 3       do j ← RANDOM(1, n)
 4          if chave[i] < chave[j] e chave[j] ≤ k
 5             then i ← j
 6                  if chave[i] = k
 7                     then return i
 8 while i ≠ NIL e chave[i] < k
 9       do i ← próximo[i]
10  if i = NIL or chave[i] > k
11     then return NIL
12     else return i
```

Para comparar a execução dos algoritmos COMPACT-LIST-SEARCH(L, k) e COMPACT-LIST-SEARCH'(L, k, t), vamos supor que a seqüência de inteiros retornados pelas chamadas de RANDOM(1, n) seja a mesma para ambos os algoritmos.

a. Suponha que COMPACT-LIST-SEARCH(L, k) execute t iterações do loop **while** das linhas 2 a 8. Demonstre que COMPACT-LIST-SEARCH'(L, k, t) retorna a mesma resposta e que o número total de iterações dos loops **for** e **while** dentro de COMPACT-LIST-SEARCH' é pelo menos t.

Na chamada COMPACT-LIST-SEARCH'(L, k, t), seja X_t a variável aleatória que descreve a distância na lista ligada (isto é, através da cadeia de ponteiros *próximo*) desde a posição i até a chave desejada k, após terem ocorrido t iterações do loop **for** das linhas 2 a 7.

b. Mostre que o tempo de execução esperado de COMPACT-LIST-SEARCH'(L, k, t) é $O(t + \mathrm{E}[X_t])$.

c. Mostre que $\mathrm{E}[X_t] \leq \sum_{r=1}^{n} (1 - r/n)^t$. (*Sugestão*: Use a equação (C.24).)

d. Mostre que $\sum_{r=0}^{n-1} r^t \leq n^{t+1} / (t + 1)$.

e. Prove que $\mathrm{E}[X_t] \leq n/(t + 1)$.

f. Mostre que COMPACT-LIST-SEARCH'(L, k, t) é executado no tempo esperado $O(t + n/t)$.

g. Conclua que COMPACT-LIST-SEARCH é executado no tempo esperado $O(\sqrt{n})$.

h. Por que supomos que todas as chaves são distintas em COMPACT-LIST-SEARCH? Demonstre que não é obrigatório que saltos aleatórios ajudem assintoticamente quando a lista contém valores de chave repetidos.

Notas do capítulo

Aho, Hopcroft e Ullman [6] e Knuth [182] são excelentes referências sobre estruturas de dados elementares. Muitos outros textos focalizam tanto estruturas de dados básicos quanto sua implementação em uma linguagem de programação particular. Os exemplos desses tipos de livros-texto incluem Goodrich e Tamassia [128], Main [209], Shaffer [273] e Weiss [310, 312, 313]. Gonnet [126] fornece dados experimentais a respeito do desempenho de muitas operações sobre estruturas de dados.

A origem de pilhas e filas como estruturas de dados em ciência da computação não é clara, pois já existiam noções correspondentes em matemática e em práticas comerciais baseadas em papéis antes da introdução dos computadores digitais. Knuth [182] cita A. M. Turing sobre o desenvolvimento de pilhas para o encadeamento de sub-rotinas em 1947.

Estruturas de dados baseadas em ponteiros também parecem ser uma criação popular. De acordo com Knuth, os ponteiros foram usados aparentemente nos primeiros computadores com memórias de tambor. A linguagem A-1, desenvolvida por G. M. Hopper em 1951, representava fórmulas algébricas como árvores binárias. Knuth credita à linguagem IPL-II, desenvolvida em 1956 por A. Newell, J. C. Shaw e H. A. Simon, o reconhecimento da importância e a promoção do uso de ponteiros. Sua linguagem IPL-III, desenvolvida em 1957, incluía operações explícitas sobre pilhas.