Parte II

Ordenação e estatísticas de ordem

Introdução

Esta parte apresenta vários algoritmos que resolvem o ***problema de ordenação*** a seguir:

Entrada: Uma seqüência de n números $\langle a_1, a_2, ..., a_n \rangle$.

Saída: Uma permutação (reordenação) $\langle a'_1, a'_2, ..., a'_n \rangle$ da seqüência de entrada, tal que $a'_1 \leq a'_2 \leq ... \leq a'_n$.

A seqüência de entrada normalmente é um arranjo de n elementos, embora possa ser representada de algum outro modo, como uma lista ligada.

A estrutura dos dados

Na prática, os números a serem ordenados raramente são valores isolados. Em geral, cada um deles faz parte de uma coleção de dados chamada ***registro***. Cada registro contém uma ***chave***, que é o valor a ser ordenado, e o restante do registro consiste em ***dados satélite***, que quase sempre são transportados junto com a chave. Na prática, quando um algoritmo de ordenação permuta as chaves, ele também deve permutar os dados satélite. Se cada registro inclui uma grande quantidade de dados satélite, muitas vezes permutamos um arranjo de ponteiros para os registros em lugar dos próprios registros, a fim de minimizar a movimentação de dados.

De certo modo, são esses detalhes de implementação que distinguem um algoritmo de um programa completamente implementado. O fato de ordenarmos números individuais ou grandes registros que contêm números é irrelevante para o *método* pelo qual um procedimento de ordenação determina a seqüência ordenada. Desse modo, quando nos concentramos no problema de ordenação, em geral supomos que a entrada consiste apenas em números. A tradução de um algoritmo para ordenação de números em um programa para ordenação de registros é conceitualmente direta, embora em uma situação específica de engenharia possam surgir outras sutilezas que fazem da tarefa real de programação um desafio.

Por que ordenar?

Muitos cientistas de computação consideram a ordenação o problema mais fundamental no estudo de algoritmos. Há várias razões:

- Às vezes, a necessidade de ordenar informações é inerente a uma aplicação. Por exemplo, para preparar os extratos de clientes, os bancos precisam ordenar os cheques pelo número do cheque.
- Os algoritmos freqüentemente usam a ordenação como uma sub-rotina chave. Por exemplo, um programa que apresenta objetos gráficos dispostos em camadas uns sobre os outros talvez tenha de ordenar os objetos de acordo com uma relação "acima", de forma a poder desenhar esses objetos de baixo para cima. Neste texto, veremos numerosos algoritmos que utilizam a ordenação como uma sub-rotina.
- Existe uma ampla variedade de algoritmos de ordenação, e eles empregam um rico conjunto de técnicas. De fato, muitas técnicas importantes usadas ao longo do projeto de algoritmos são representadas no corpo de algoritmos de ordenação que foram desenvolvidos ao longo dos anos. Desse modo, a ordenação também é um problema de interesse histórico.
- A ordenação é um problema para o qual podemos demonstrar um limite inferior não trivial (como faremos no Capítulo 8). Nossos melhores limites superiores correspondem ao limite inferior assintoticamente, e assim sabemos que nossos algoritmos de ordenação são assintoticamente ótimos. Além disso, podemos usar o limite inferior da ordenação com a finalidade de demonstrar limites inferiores para alguns outros problemas.
- Muitas questões de engenharia surgem quando se implementam algoritmos de ordenação. O programa de ordenação mais rápido para uma determinada situação pode depender de muitos fatores, como o conhecimento anterior a respeito das chaves e dos dados satélite, da hierarquia de memória (caches e memória virtual) do computador host e do ambiente de software. Muitas dessas questões são mais bem tratadas no nível algorítmico, em vez de ser necessário "mexer" no código.

Algoritmos de ordenação

Introduzimos no Capítulo 2 dois algoritmos para ordenação de n números reais. A ordenação de inserção leva o tempo $\Theta(n^2)$ no pior caso. Porém, pelo fato de seus loops internos serem compactos, ela é um rápido algoritmo de ordenação local para pequenos tamanhos de entrada. (Lembre-se de que um algoritmo de ordenação efetua a ordenação ***local*** se somente um número constante de elementos do arranjo de entrada sempre são armazenados fora do arranjo.) A ordenação por intercalação tem um tempo assintótico de execução melhor, $\Theta(n \lg n)$, mas o procedimento MERGE que ela utiliza não opera no local.

Nesta parte, apresentaremos mais dois algoritmos que ordenam números reais arbitrários. O heapsort, apresentado no Capítulo 6, efetua a ordenação de n números localmente, no tempo $O(n \lg n)$. Ele usa uma importante estrutura de dados, chamada heap (monte), com a qual também podemos implementar uma fila de prioridades.

O quicksort, no Capítulo 7, também ordena n números localmente, mas seu tempo de execução no pior caso é $\Theta(n^2)$. Porém, seu tempo de execução no caso médio é $\Theta(n \lg n)$, e ele em geral supera o heapsort na prática. Como a ordenação por inserção, o quicksort tem um código compacto, e assim o fator constante oculto em seu tempo de execução é pequeno. Ele é um algoritmo popular para ordenação de grandes arranjos de entrada.

A ordenação por inserção, a ordenação por intercalação, o heapsort e o quicksort são todos ordenações por comparação: eles determinam a seqüência ordenada de um arranjo de entrada pela comparação dos elementos. O Capítulo 8 começa introduzindo o modelo de árvore de de-

cisão, a fim de estudar as limitações de desempenho de ordenações por comparação. Usando esse modelo, provamos um limite inferior igual a $\Omega(n \lg n)$ no tempo de execução do pior caso de qualquer ordenação por comparação sobre n entradas, mostrando assim que o heapsort e a ordenação por intercalação são ordenações por comparação assintoticamente ótimas.

Em seguida, o Capítulo 8 mostra que poderemos superar esse limite inferior $\Omega(n \lg n)$ se for possível reunir informações sobre a seqüência ordenada da entrada por outros meios além da comparação dos elementos. Por exemplo, o algoritmo de ordenação por contagem pressupõe que os números da entrada estão no conjunto $\{1, 2, ..., k\}$. Usando a indexação de arranjos como uma ferramenta para determinar a ordem relativa, a ordenação por contagem pode ordenar n números no tempo $(k + n)$. Desse modo, quando $k = O(n)$, a ordenação por contagem é executada em um tempo linear no tamanho do arranjo de entrada. Um algoritmo relacionado, a radix sort (ordenação da raiz), pode ser usado para estender o intervalo da ordenação por contagem. Se houver n inteiros para ordenar, cada inteiro tiver d dígitos e cada dígito estiver no conjunto $\{1, 2, ..., k\}$, a radix sort poderá ordenar os números em um tempo $O(d(n + k))$. Quando d é uma constante e k é $O(n)$, a radix sort é executada em tempo linear. Um terceiro algoritmo, bucket sort (ordenação por balde), requer o conhecimento da distribuição probabilística dos números no arranjo de entrada. Ele pode ordenar n números reais distribuídos uniformemente no intervalo meio aberto $[0, 1)$ no tempo do caso médio $O(n)$.

Estatísticas de ordem

A i-ésima estatística de ordem de um conjunto de n números é o i-ésimo menor número no conjunto. É claro que uma pessoa pode selecionar a i-ésima estatística de ordem, ordenando a entrada e indexando o i-ésimo elemento da saída. Sem quaisquer suposições a respeito da distribuição da entrada, esse método é executado no tempo $\Omega(n \lg n)$, como mostra o limite inferior demonstrado no Capítulo 8.

No Capítulo 9, mostramos que é possível encontrar o i-ésimo menor elemento no tempo $O(n)$, mesmo quando os elementos são números reais arbitrários. Apresentamos um algoritmo com pseudocódigo compacto que é executado no tempo $\Theta(n^2)$ no pior caso, mas em tempo linear no caso médio. Também fornecemos um algoritmo mais complicado que é executado em um tempo $O(n)$ no pior caso.

Experiência necessária

Embora a maioria das seções desta parte não dependa de conceitos matemáticos difíceis, algumas seções exigem uma certa sofisticação matemática. Em particular, as análises do caso médio do quicksort, do bucket sort e do algoritmo de estatística de ordem utilizam a probabilidade, o que revisamos no Apêndice C; o material sobre a análise probabilística e os algoritmos aleatórios é estudado no Capítulo 5. A análise do algoritmo de tempo linear do pior caso para estatísticas de ordem envolve matemática um pouco mais sofisticada que as análises do pior caso desta parte.

Capítulo 6

Heapsort

Neste capítulo, introduzimos outro algoritmo de ordenação. Como a ordenação por intercalação, mas diferente da ordenação por inserção, o tempo de execução do heapsort é $O(n \lg n)$. Como a ordenação por inserção, mas diferente da ordenação por intercalação, o heapsort ordena localmente: apenas um número constante de elementos do arranjo é armazenado fora do arranjo de entrada em qualquer instante. Desse modo, o heapsort combina os melhores atributos dos dois algoritmos de ordenação que já discutimos.

O heapsort também introduz outra técnica de projeto de algoritmos: o uso de uma estrutura de dados, nesse caso uma estrutura que chamamos "heap" (ou "monte") para gerenciar informações durante a execução do algoritmo. A estrutura de dados heap não é útil apenas para o heapsort (ou ordenação por heap); ela também cria uma eficiente fila de prioridades. A estrutura de dados heap reaparecerá em algoritmos de capítulos posteriores.

Observamos que o termo "heap" foi cunhado originalmente no contexto do heapsort, mas, desde então, ele passou a se referir ao "espaço para armazenamento do lixo coletado" como o espaço proporcionado pelas linguagens de programação Lisp e Java. Nossa estrutura de dados heap *não* é um espaço para armazenamento do lixo coletado e, sempre que mencionarmos heaps neste livro, estaremos fazendo referência à estrutura de dados definida neste capítulo.

6.1 Heaps

A estrutura de dados ***heap*** (***binário***) é um objeto arranjo que pode ser visto como uma árvore binária praticamente completa (ver Seção B.5.3), como mostra a Figura 6.1. Cada nó da árvore corresponde a um elemento do arranjo que armazena o valor no nó. A árvore está completamente preenchida em todos os níveis, exceto talvez no nível mais baixo, que é preenchido a partir da esquerda até certo ponto. Um arranjo A que representa um heap é um objeto com dois atributos: *comprimento*$[A]$, que é o número de elementos no arranjo, e *tamanho-do-heap*$[A]$, o número de elementos no heap armazenado dentro do arranjo A. Ou seja, embora $A[1 .. \textit{comprimento}[A]]$ possa conter números válidos, nenhum elemento além de $A[\textit{tamanho-do-heap}[A]]$, onde $\textit{tamanho-do-heap}[A] \leq \textit{comprimento}[A]$, é um elemento do heap. A raiz da árvore é $A[1]$ e, dado o índice i de um nó, os índices de seu pai PARENT(i), do filho da esquerda LEFT(i) e do filho da direita RIGHT(i) podem ser calculados de modo simples:

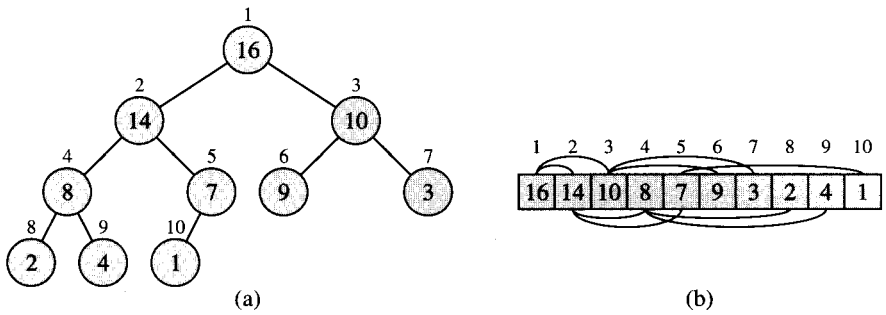

FIGURA 6.1 Um heap máximo visto como (a) uma árvore binária e (b) um arranjo. O número dentro do círculo em cada nó na árvore é o valor armazenado nesse nó. O número acima de um nó é o índice correspondente no arranjo. Acima e abaixo do arranjo encontramos linhas mostrando relacionamentos pai-filho; os pais estão sempre à esquerda de seus filhos. A árvore tem altura três; o nó no índice 4 (com o valor 8) tem altura um

```
PARENT(i)
 return ⌊i/2⌋

LEFT(i)
 return 2i

RIGHT(i)
 return 2i + 1
```

Na maioria dos computadores, o procedimento LEFT pode calcular $2i$ em uma única instrução, simplesmente deslocando a representação binária de i uma posição de bit para a esquerda. De modo semelhante, o procedimento RIGHT pode calcular rapidamente $2i + 1$ deslocando a representação binária de i uma posição de bit para a esquerda e inserindo 1 como valor do bit de baixa ordem. O procedimento PARENT pode calcular $\lfloor i/2 \rfloor$ deslocando i uma posição de bit para a direita. Em uma boa implementação de heapsort, esses três procedimentos são executados freqüentemente como "macros" ou como procedimentos "em linha".

Existem dois tipos de heaps binários: heaps máximos e heaps mínimos. Em ambos os tipos, os valores nos nós satisfazem a uma ***propriedade de heap***, cujos detalhes específicos dependem do tipo de heap. Em um ***heap máximo***, a ***propriedade de heap máximo*** é que, para todo nó i diferente da raiz,

$$A[\text{PARENT}(i)] \geq A[i] ,$$

isto é, o valor de um nó é no máximo o valor de seu pai. Desse modo, o maior elemento em um heap máximo é armazenado na raiz, e a subárvore que tem raiz em um nó contém valores menores que o próprio nó. Um ***heap mínimo*** é organizado de modo oposto; a ***propriedade de heap mínimo*** é que, para todo nó i diferente da raiz,

$$A[\text{PARENT}(i)] \leq A[i] .$$

O menor elemento em um heap mínimo está na raiz.

Para o algoritmo de heapsort, usamos heaps máximos. Heaps mínimos são comumente empregados em filas de prioridades, que discutimos na Seção 6.5. Seremos precisos ao especificar se necessitamos de um heap máximo ou de um heap mínimo para qualquer aplicação específica e, quando as propriedades se aplicarem tanto a heaps máximos quanto a heaps mínimos, simplesmente usaremos o termo "heap".

Visualizando um heap como uma árvore, definimos a ***altura*** de um nó em um heap como o número de arestas no caminho descendente simples mais longo desde o nó até uma folha, e definimos a altura do heap como a altura de sua raiz. Tendo em vista que um heap de n elementos é baseado em uma árvore binária completa, sua altura é $\Theta(\lg n)$ (ver Exercício 6.1-2). Veremos que as operações básicas sobre heaps são executadas em um tempo máximo proporcional à altura da árvore, e assim demoram um tempo $O(\lg n)$. O restante deste capítulo apresenta alguns procedimentos básicos e mostra como eles são usados em um algoritmo de ordenação e em uma estrutura de dados de fila de prioridades.

- O procedimento MAX-HEAPIFY, executado no tempo $O(\lg n)$, é a chave para manter a propriedade de heap máximo (6.1).
- O procedimento BUILD-MAX-HEAP, executado em tempo linear, produz um heap a partir de um arranjo de entrada não ordenado.
- O procedimento HEAPSORT, executado no tempo $O(n \lg n)$, ordena um arranjo localmente.
- Os procedimentos MAX-HEAP-INSERT, HEAP-EXTRACT-MAX, HEAP-INCREASE-KEY e HEAP-MAXIMUM, executados no tempo $O(\lg n)$, permitem que a estrutura de dados heap seja utilizada como uma fila de prioridades.

Exercícios

6.1-1
Quais são os números mínimo e máximo de elementos em um heap de altura h?

6.1-2
Mostre que um heap de n elementos tem altura $\lfloor \lg n \rfloor$.

6.1-3
Mostre que, em qualquer subárvore de um heap máximo, a raiz da subárvore contém o maior valor que ocorre em qualquer lugar nessa subárvore.

6.1-4
Onde em um heap máximo o menor elemento poderia residir, supondo-se que todos os elementos sejam distintos?

6.1-5
Um arranjo que está em seqüência ordenada é um heap mínimo?

6.1-6
A seqüência $\langle 23, 17, 14, 6, 13, 10, 1, 5, 7, 12 \rangle$ é um heap máximo?

6.1-7
Mostre que, com a representação de arranjo para armazenar um heap de n elementos, as folhas são os nós indexados por $\lfloor n/2 \rfloor + 1, \lfloor n/2 \rfloor + 2, \ldots, n$.

6.2 Manutenção da propriedade de heap

MAX-HEAPIFY é uma sub-rotina importante para manipulação de heaps máximos. Suas entradas são um arranjo A e um índice i para o arranjo. Quando MAX-HEAPIFY é chamado, supomos que as árvores binárias com raízes em LEFT(i) e RIGHT(i) são heaps máximos, mas que $A[i]$ pode ser menor que seus filhos, violando assim a propriedade de heap máximo. A função de MAX-HEAPIFY é deixar que o valor em $A[i]$ "flutue para baixo" no heap máximo, de tal forma que a subárvore com raiz no índice i se torne um heap máximo.

```
MAX-HEAPIFY(A, i)
 1 l ← LEFT(i)
 2 r ← RIGHT(i)
 3 if l ≤ tamanho-do-heap[A] e A[l] > A[i]
 4    then maior ← l
 5    else maior ← i
 6 if r ≤ tamanho-do-heap[A] e A[r] > A[maior]
 7    then maior ← r
 8 if maior ≠ i
 9    then trocar A[i] ↔ A[maior]
10         MAX-HEAPIFY(A, maior)
```

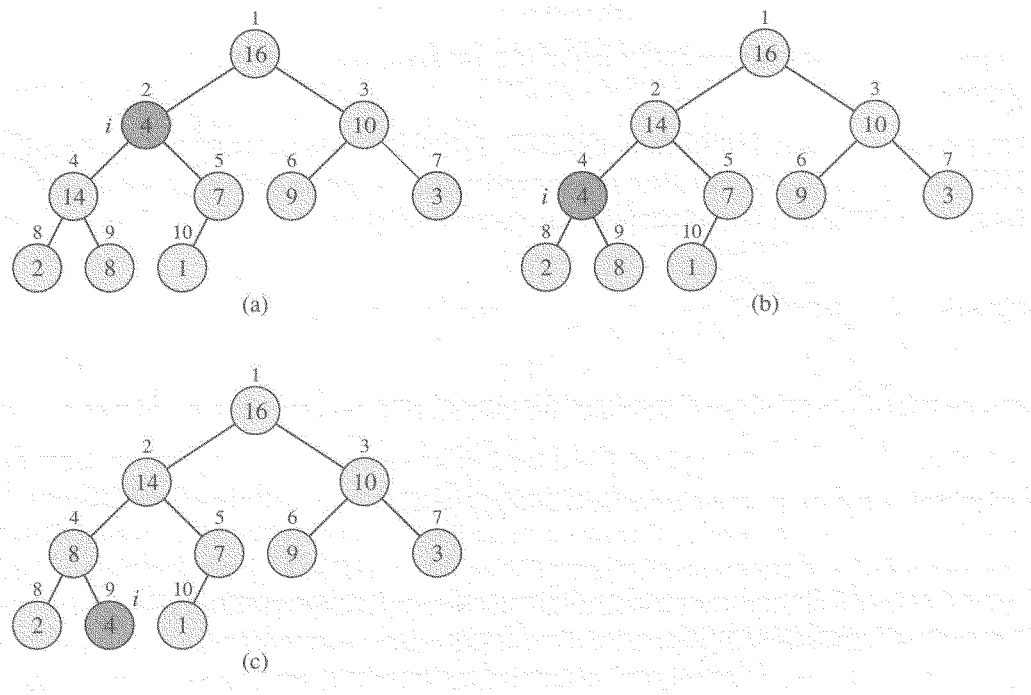

FIGURA 6.2 A ação de MAX-HEAPIFY(*A*, 2), onde *tamanho-do-heap*[*A*] = 10. (**a**) A configuração inicial, com *A*[2] no nó *i* = 2, violando a propriedade de heap máximo, pois ele não é maior que ambos os filhos. A propriedade de heap máximo é restabelecida para o nó 2 em (**b**) pela troca de *A*[2] por *A*[4], o que destrói a propriedade de heap máximo para o nó 4. A chamada recursiva MAX-HEAPIFY(*A*, 4) agora define *i* = 4. Após a troca de *A*[4] por *A*[9], como mostramos em (**c**), o nó 4 é corrigido, e a chamada recursiva a MAX-HEAPIFY(*A*, 9) não produz nenhuma mudança adicional na estrutura de dados

A Figura 6.2 ilustra a ação de MAX-HEAPIFY. Em cada passo, o maior entre os elementos *A*[*i*], *A*[LEFT(*i*)] e *A*[RIGHT(*i*)] é determinado, e seu índice é armazenado em *maior*. Se *A*[*i*] é maior, então a subárvore com raiz no nó *i* é um heap máximo e o procedimento termina. Caso contrário, um dos dois filhos tem o maior elemento, e *A*[*i*] é trocado por *A*[*maior*], o que faz o nó *i* e seus filhos satisfazerem à propriedade de heap máximo. Porém, agora o nó indexado por *maior* tem o valor original *A*[*i*] e, desse modo, a subárvore com raiz em *maior* pode violar a propriedade de heap máximo. Em conseqüência disso, MAX-HEAPIFY deve ser chamado recursivamente nessa subárvore.

O tempo de execução de MAX-HEAPIFY em uma subárvore de tamanho n com raiz em um dado nó *i* é o tempo $\Theta(1)$ para corrigir os relacionamentos entre os elementos *A*[*i*], *A*[LEFT(*i*)] e *A*[RIGHT(*i*)], mais o tempo para executar MAX-HEAPIFY em uma subárvore com raiz em um dos filhos do nó *i*. As subárvores de cada filho têm tamanho máximo igual a $2n/3$ – o pior caso ocorre quando a última linha da árvore está exatamente metade cheia – e o tempo de execução de MAX-HEAPIFY pode então ser descrito pela recorrência

$T(n) \leq T(2n/3) + \Theta(1)$.

A solução para essa recorrência, de acordo com o caso 2 do teorema mestre (Teorema 4.1), é $T(n) = O(\lg n)$. Como alternativa, podemos caracterizar o tempo de execução de MAX-HEAPIFY em um nó de altura h como $O(h)$.

Exercícios

6.2-1
Usando a Figura 6.2 como modelo, ilustre a operação de MAX-HEAPIFY(A, 3) sobre o arranjo $A = \langle 27, 17, 3, 16, 13, 10, 1, 5, 7, 12, 4, 8, 9, 0 \rangle$.

6.2-2
Começando com o procedimento MAX-HEAPIFY, escreva pseudocódigo para o procedimento MIN-HEAPIFY(A, i), que executa a manipulação correspondente sobre um heap mínimo. Como o tempo de execução de MIN-HEAPIFY se compara ao de MAX-HEAPIFY?

6.2-3
Qual é o efeito de chamar MAX-HEAPIFY(A, i) quando o elemento $A[i]$ é maior que seus filhos?

6.2-4
Qual é o efeito de chamar MAX-HEAPIFY(A, i) para $i >$ *tamanho-do-heap*$[A]/2$?

6.2-5
O código de MAX-HEAPIFY é bastante eficiente em termos de fatores constantes, exceto possivelmente para a chamada recursiva da linha 10, que poderia fazer alguns compiladores produzirem um código ineficiente. Escreva um MAX-HEAPIFY eficiente que use uma construção de controle iterativa (um loop) em lugar da recursão.

6.2-6
Mostre que o tempo de execução do pior caso de MAX-HEAPIFY sobre um heap de tamanho n é $\Omega(\lg n)$. (*Sugestão*: Para um heap com n nós, forneça valores de nós que façam MAX-HEAPIFY ser chamado recursivamente em todo nó sobre um caminho desde a raiz até uma folha.)

6.3 A construção de um heap

Podemos usar o procedimento MAX-HEAPIFY de baixo para cima, a fim de converter um arranjo $A[1..n]$, onde $n =$ *comprimento*$[A]$, em um heap máximo. Pelo Exercício 6.1-7, os elementos no subarranjo $A[(\lfloor n/2 \rfloor + 1)..n]$ são todos folhas da árvore, e então cada um deles é um heap de 1 elemento com o qual podemos começar. O procedimento BUILD-MAX-HEAP percorre os nós restantes da árvore e executa MAX-HEAPIFY sobre cada um.

```
BUILD-MAX-HEAP(A)
1 tamanho-do-heap[A] ← comprimento[A]
2 for i ← ⌊comprimento[A]/2⌋ downto 1
3     do MAX-HEAPIFY(A, i)
```

A Figura 6.3 ilustra um exemplo da ação de BUILD-MAX-HEAP.

Para mostrar por que BUILD-MAX-HEAP funciona corretamente, usamos o seguinte loop invariante:

> No começo de cada iteração do loop **for** das linhas 2 e 3, cada nó $i + 1, i + 2, ..., n$ é a raiz de um heap máximo.

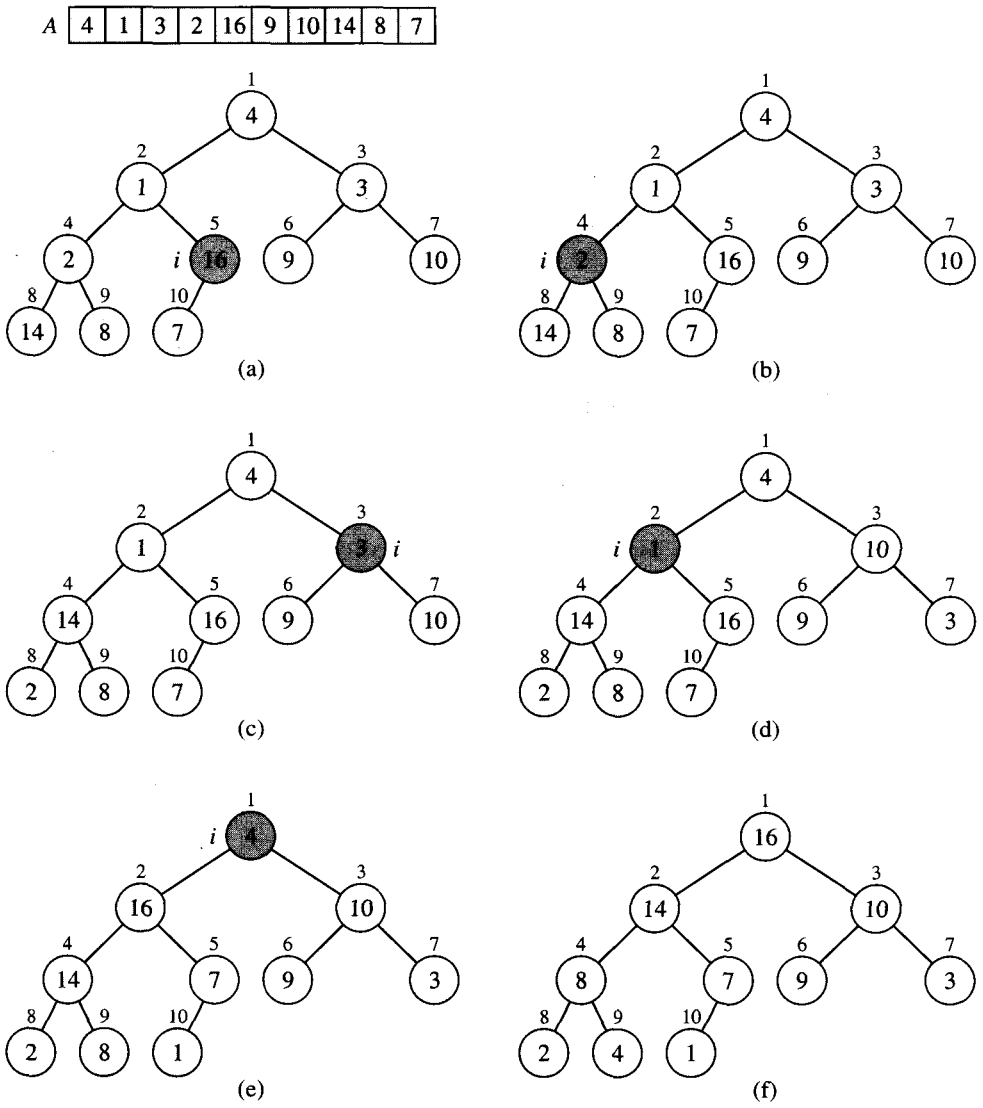

FIGURA 6.3 A operação de BUILD-MAX-HEAP, mostrando a estrutura de dados antes da chamada a MAX-HEAPIFY na linha 3 de BUILD-MAX-HEAP. (**a**) Um arranjo de entrada de 10 elementos A e a árvore binária que ele representa. A figura mostra que o índice de loop i se refere ao nó 5 antes da chamada MAX-HEAPIFY(A, i). (**b**) A estrutura de dados que resulta. O índice de loop i para a próxima iteração aponta para o nó 4. (**c**)–(**e**) Iterações subseqüentes do loop **for** em BUILD-MAX-HEAP. Observe que, sempre que MAX-HEAPIFY é chamado em um nó, as duas subárvores desse nó são ambas heaps máximos. (**f**) O heap máximo após o término de BUILD-MAX-HEAP

Precisamos mostrar que esse invariante é verdade antes da primeira iteração do loop, que cada iteração do loop mantém o invariante e que o invariante fornece uma propriedade útil para mostrar a correção quando o loop termina.

Inicialização: Antes da primeira iteração do loop, $i = \lfloor n/2 \rfloor$. Cada nó $\lfloor n/2 \rfloor + 1, \lfloor n/2 \rfloor + 2, \ldots, n$ é uma folha, e é portanto a raiz de um heap máximo trivial.

Manutenção: Para ver que cada iteração mantém o loop invariante, observe que os filhos do nó i são numerados com valores mais altos que i. Assim, pelo loop invariante, ambos são raízes de heaps máximos. Essa é precisamente a condição exigida para a chamada MAX-HEAPIFY(A, i) para tornar o nó i a raiz de um heap máximo. Além disso, a chamada a MAX-HEAPIFY preserva a propriedade de que os nós $i + 1, i + 2, \ldots, n$ são todos raízes de heaps máximos. Decrementar i na atualização do loop **for** restabelece o loop invariante para a próxima iteração.

Término: No término, $i = 0$. Pelo loop invariante, cada nó 1, 2, ..., n é a raiz de um heap máximo. Em particular, o nó 1 é uma raiz.

Podemos calcular um limite superior simples sobre o tempo de execução de BUILD-MAX-HEAP como a seguir. Cada chamada a MAX-HEAPIFY custa o tempo $O(\lg n)$, e existem $O(n)$ dessas chamadas. Desse modo, o tempo de execução é $O(n \lg n)$. Esse limite superior, embora correto, não é assintoticamente restrito.

Podemos derivar um limite mais restrito observando que o tempo de execução de MAX-HEAPIFY sobre um nó varia com a altura do nó na árvore, e as alturas na maioria dos nós são pequenas. Nossa análise mais restrita se baseia nas propriedades de que um heap de n elementos tem altura $\lfloor \lg n \rfloor$ (ver Exercício 6.1-2) e no máximo $\lceil n/2^{h+1} \rceil$ nós de qualquer altura h (ver Exercício 6.3-3).

O tempo exigido por MAX-HEAPIFY quando é chamado em um nó de altura h é $O(h)$; assim, podemos expressar o custo total de BUILD-MAX-HEAP por

$$\sum_{h=0}^{\lfloor \lg n \rfloor} \left\lceil \frac{n}{2^{h+1}} \right\rceil O(h) = O\left(n \sum_{h=0}^{\lfloor \lg n \rfloor} \frac{h}{2^h} \right).$$

O último somatório pode ser calculado substituindo-se $x = 1/2$ na fórmula (A.8), o que produz

$$\sum_{h=0}^{\infty} \frac{h}{2^h} = \frac{1/2}{(1-1/2)^2}$$

$$= 2 .$$

Desse modo, o tempo de execução de BUILD-MAX-HEAP pode ser limitado como

$$O\left(n \sum_{h=0}^{\lfloor \lg n \rfloor} \frac{h}{2^h} \right) = O\left(n \sum_{h=0}^{\infty} \frac{h}{2^h} \right)$$

$$= O(n) .$$

Conseqüentemente, podemos construir um heap máximo a partir de um arranjo não ordenado em tempo linear.

Podemos construir um heap mínimo pelo procedimento BUILD-MIN-HEAP, que é igual a BUILD-MAX-HEAP, mas tem a chamada a MAX-HEAPIFY na linha 3 substituída por uma chamada a MIN-HEAPFY (ver Exercício 6.2-2). BUILD-MIN-HEAP produz um heap mínimo a partir de um arranjo linear não ordenado em tempo linear.

Exercícios

6.3-1

Usando a Figura 6.3 como modelo, ilustre a operação de BUILD-MAX-HEAP sobre o arranjo $A = \langle 5, 3, 17, 10, 84, 19, 6, 22, 9 \rangle$.

6.3-2

Por que queremos que o índice de loop i na linha 2 de BUILD-MAX-HEAP diminua de $\lfloor comprimento[A]/2 \rfloor$ até 1, em vez de aumentar de 1 até $\lfloor comprimento[A]/2 \rfloor$?

6.3-3

Mostre que existem no máximo $\lceil n/2^{h+1} \rceil$ nós de altura h em qualquer heap de n elementos.

6.4 O algoritmo heapsort

O algoritmo heapsort (ordenação por monte) começa usando BUILD-MAX-HEAP para construir um heap no arranjo de entrada $A[1..n]$, onde $n = comprimento[A]$. Tendo em vista que o elemento máximo do arranjo está armazenado na raiz $A[1]$, ele pode ser colocado em sua posição final correta, trocando-se esse elemento por $A[n]$. Se agora "descartarmos" o nó n do heap (diminuindo *tamanho-do-heap*$[A]$), observaremos que $A[1 .. (n-1)]$ pode ser facilmente transformado em um heap máximo. Os filhos da raiz continuam sendo heaps máximos, mas o novo elemento raiz pode violar a propriedade de heap máximo. Porém, tudo o que é necessário para restabelecer a propriedade de heap máximo é uma chamada a MAX-HEAPIFY(A, 1), que deixa um heap máximo em $A[1 .. (n-1)]$. Então, o algoritmo heapsort repete esse processo para o heap de tamanho $n-1$, descendo até um heap de tamanho 2. (Veja no Exercício 6.4-2 um loop invariante preciso.)

```
HEAPSORT(A)
1 BUILD-MAX-HEAP(A)
2 for i ← comprimento[A] downto 2
3    do trocar A[1] ↔ A[i]
4       tamanho-do-heap[A] ← tamanho-do-heap[A] - 1
5       MAX-HEAPIFY(A, 1)
```

A Figura 6.4 mostra um exemplo da operação de heapsort depois do heap máximo ser inicialmente construído. Cada heap máximo é mostrado no princípio de uma iteração do loop **for** das linhas 2 a 5.

O procedimento HEAPSORT demora o tempo $O(n \lg n)$, pois a chamada a BUILD-MAX-HEAP demora o tempo $O(n)$, e cada uma das $n-1$ chamadas a MAX-HEAPIFY demora o tempo $O(\lg n)$.

Exercícios

6.4-1

Usando a Figura 6.4 como modelo, ilustre a operação de HEAPSORT sobre o arranjo $A = \langle 5, 13, 2, 25, 7, 17, 20, 8, 4 \rangle$.

6.4-2

Discuta a correção de HEAPSORT usando o loop invariante a seguir:

> No início de cada iteração do loop **for** das linhas 2 a 5, o subarranjo $A[1 .. i]$ é um heap máximo contendo os i menores elementos de $A[1 .. n]$, e o subarranjo $A[i+1 .. n]$ contém os $n-i$ maiores elementos de $A[1 .. n]$, ordenados.

6.4-3

Qual é o tempo de execução de heapsort sobre um arranjo A de comprimento n que já está ordenado em ordem crescente? E em ordem decrescente?

6.4-4

Mostre que o tempo de execução do pior caso de heapsort é $\Omega(n \lg n)$.

6.4-5

Mostre que, quando todos os elementos são distintos, o tempo de execução do melhor caso de heapsort é $\Omega(n \lg n)$.

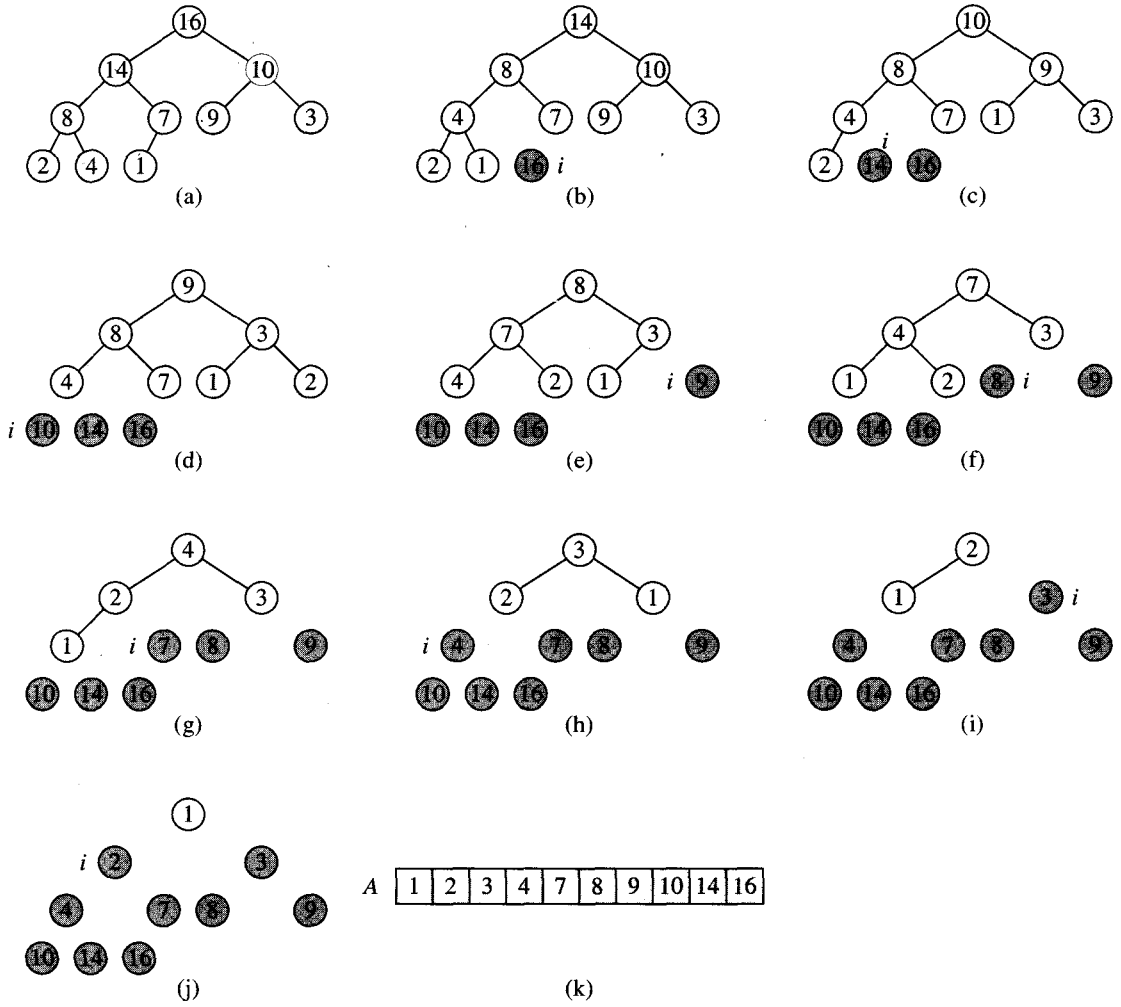

FIGURA 6.4 A operação de HEAPSORT. (**a**) A estrutura de dados heap máximo, logo após ter sido construída por BUILD-MAX-HEAP. (**b**)–(**j**) O heap máximo logo após cada chamada de MAX-HEAPIFY na linha 5. O valor de i nesse instante é mostrado. Apenas os nós levemente sombreados permanecem no heap. (**k**) O arranjo ordenado resultante A

6.5 Filas de prioridades

O heapsort é um algoritmo excelente, mas uma boa implementação de quicksort, apresentado no Capítulo 7, normalmente o supera na prática. Não obstante, a estrutura de dados de heap propriamente dita tem uma utilidade enorme. Nesta seção, apresentaremos uma das aplicações mais populares de um heap: seu uso como uma fila de prioridades eficiente. Como ocorre no caso dos heaps, existem dois tipos de filas de prioridades: as filas de prioridade máxima e as filas de prioridade mínima. Focalizaremos aqui a implementação das filas de prioridade máxima, que por sua vez se baseiam em heaps máximos; o Exercício 6.5-3 lhe pede para escrever os procedimentos correspondentes a filas de prioridade mínima.

Uma ***fila de prioridades*** é uma estrutura de dados para manutenção de um conjunto S de elementos, cada qual com um valor associado chamado ***chave***. Uma fila de prioridade máxima admite as operações a seguir.

INSERT(S, x) insere o elemento x no conjunto S. Essa operação poderia ser escrita como $S \leftarrow S \cup \{x\}$.

MAXIMUM(S) retorna o elemento de S com a maior chave.

EXTRACT-MAX(S) remove e retorna o elemento de S com a maior chave.

INCREASE-KEY(S, x, k) aumenta o valor da chave do elemento x para o novo valor k, que se presume ser pelo menos tão grande quanto o valor da chave atual de x.

Uma aplicação de filas de prioridade máxima é programar trabalhos em um computador compartilhado. A fila de prioridade máxima mantém o controle dos trabalhos a serem executados e de suas prioridades relativas. Quando um trabalho termina ou é interrompido, o trabalho de prioridade mais alta é selecionado dentre os trabalhos pendentes, com o uso de EXTRACT-MAX. Um novo trabalho pode ser adicionado à fila em qualquer instante, com a utilização de INSERT.

Como alternativa, uma ***fila de prioridade mínima*** admite as operações INSERT, MINIMUM, EXTRACT-MIN e DECREASE-KEY. Uma fila de prioridade mínima pode ser usada em um simulador orientado a eventos. Os itens na fila são eventos a serem simulados, cada qual com um tempo de ocorrência associado que serve como sua chave. Os eventos devem ser simulados em ordem de seu momento de ocorrência, porque a simulação de um evento pode provocar outros eventos a serem simulados no futuro. O programa de simulação utiliza EXTRACT-MIN em cada etapa para escolher o próximo evento a simular. À medida que novos eventos são produzidos, eles são inseridos na fila de prioridade mínima com o uso de INSERT. Veremos outros usos de filas de prioridade mínima destacando a operação DECREASE-KEY, nos Capítulos 23 e 24.

Não surpreende que possamos usar um heap para implementar uma fila de prioridades. Em uma dada aplicação, como a programação de trabalhos ou a simulação orientada a eventos, os elementos de uma fila de prioridades correspondem a objetos na aplicação. Freqüentemente é necessário determinar que objeto da aplicação corresponde a um dado elemento de fila de prioridades e vice-versa. Então, quando um heap é usado para implementar uma fila de prioridades, com freqüência precisamos armazenar um ***descritor*** para o objeto da aplicação correspondente em cada elemento do heap. A constituição exata do descritor (isto é, um ponteiro, um inteiro etc.) depende da aplicação. De modo semelhante, precisamos armazenar um descritor para o elemento do heap correspondente em cada objeto da aplicação. Aqui, o descritor em geral seria um índice de arranjo. Como os elementos do heap mudam de posições dentro do arranjo durante operações de heap, uma implementação real, ao reposicionar um elemento do heap, também teria de atualizar o índice do arranjo no objeto da aplicação correspondente. Tendo em vista que os detalhes de acesso a objetos de aplicações dependem muito da aplicação e de sua implementação, não os examinaremos aqui, exceto por observar que na prática esses descritores precisam ser mantidos corretamente.

Agora descreveremos como implementar as operações de uma fila de prioridade máxima. O procedimento HEAP-MAXIMUM implementa a operação MAXIMUM no tempo $\Theta(1)$.

```
HEAP-MAXIMUM(A)
1 return A[1]
```

O procedimento HEAP-EXTRACT-MAX implementa a operação EXTRACT-MAX. Ele é semelhante ao corpo do loop **for** (linhas 3 a 5) do procedimento HEAPSORT:

```
HEAP-EXTRACT-MAX(A)
1 if tamanho-do-heap[A] < 1
2    then error "heap underflow"
3 max ← A[1]
4 A[1] ← A[tamanho-do-heap[A]]
5 tamanho-do-heap[A] ← tamanho-do-heap[A] - 1
6 MAX-HEAPIFY(A, 1)
7 return max
```

O tempo de execução de HEAP-EXTRACT-MAX é $O(\lg n)$, pois ele executa apenas uma porção constante do trabalho sobre o tempo $O(\lg n)$ para MAX-HEAPIFY.

O procedimento HEAP-INCREASE-KEY implementa a operação INCREASE-KEY. O elemento da fila de prioridades cuja chave deve ser aumentada é identificado por um índice i no arranjo. Primeiro, o procedimento atualiza a chave do elemento $A[i]$ para seu novo valor. Em seguida, como o aumento da chave de $A[i]$ pode violar a propriedade de heap máximo, o procedimento, de um modo que é uma reminiscência do loop de inserção (linhas 5 a 7) de INSERTION-SORT da Seção 2.1, percorre um caminho desde esse nó em direção à raiz, até encontrar um lugar apropriado para o elemento recém-aumentado. Durante essa travessia, ele compara repetidamente um elemento a seu pai, permutando suas chaves e continuando se a chave do elemento é maior, e terminando se a chave do elemento é menor, pois a propriedade de heap máximo agora é válida. (Veja no Exercício 6.5-5 um loop invariante preciso.)

```
HEAP-INCREASE-KEY(A, i, chave)
1 if chave < A[i]
2    then error "nova chave é menor que chave atual"
3 A[i] ← chave
4 while i > 1 e A[PARENT(i)] < A[i]
5       do troca A[i] ↔ A[PARENT(i)]
6          i ← PARENT(i)
```

A Figura 6.5 mostra um exemplo de uma operação de HEAP-INCREASE-KEY. O tempo de execução de HEAP-INCREASE-KEY sobre um heap de n elementos é $O(\lg n)$, pois o caminho traçado desde o nó atualizado na linha 3 até a raiz tem o comprimento $O(\lg n)$.

O procedimento MAX-HEAP-INSERT implementa a operação INSERT. Ele toma como uma entrada a chave do novo elemento a ser inserido no heap máximo A. Primeiro, o procedimento expande o heap máximo, adicionando à árvore uma nova folha cuja chave é $-\infty$. Em seguida, ele chama HEAP-INCREASE-KEY para definir a chave desse novo nó com seu valor correto e manter a propriedade de heap máximo.

```
MAX-HEAP-INSERT(A, chave)
1 tamanho-do-heap[A] ← tamanho-do-heap[A] + 1
2 A[tamanho-do-heap[A]] ← −∞
3 HEAP-INCREASE-KEY(A, tamanho-do-heap[A], chave)
```

O tempo de execução de MAX-HEAP-INSERT sobre um heap de n elementos é $O(\lg n)$.

Em resumo, um heap pode admitir qualquer operação de fila de prioridades em um conjunto de tamanho n no tempo $O(\lg n)$.

Exercícios

6.5-1

Ilustre a operação de HEAP-EXTRACT-MAX sobre o heap $A = \langle 15, 13, 9, 5, 12, 8, 7, 4, 0, 6, 2, 1\rangle$.

6.5-2

Ilustre a operação de MAX-HEAP-INSERT(A, 10) sobre o heap $A = \langle 15, 13, 9, 5, 12, 8, 7, 4, 0, 6, 2, 1\rangle$. Use o heap da Figura 6.5 como modelo para a chamada de HEAP-INCREASE-KEY.

6.5-3

Escreva pseudocódigo para os procedimentos HEAP-MINIMUM, HEAP-EXTRACT-MIN, HEAP-DECREASE-KEY e MIN-HEAP-INSERT que implementam uma fila de prioridade mínima com um heap mínimo.

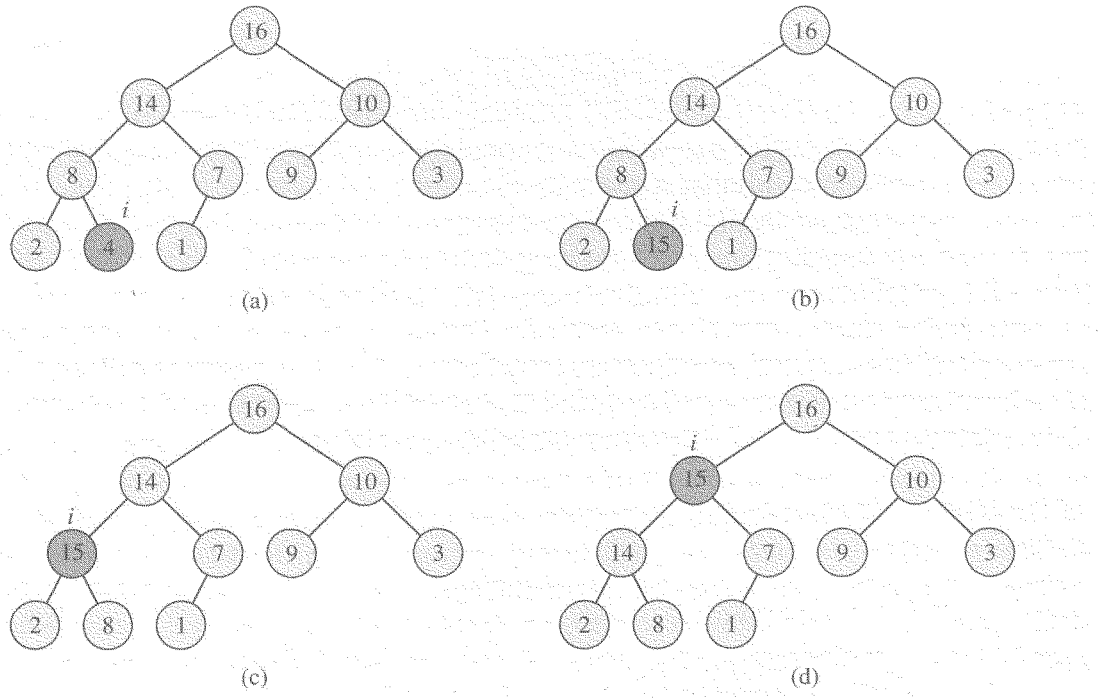

FIGURA 6.5 A operação de HEAP-INCREASE-KEY. (**a**) O heap máximo da Figura 6.4(a) com um nó cujo índice é i, fortemente sombreado. (**b**) Esse nó tem sua chave aumentada para 15. (**c**) Depois de uma iteração do loop **while** das linhas 4 a 6, o nó e seu pai trocaram chaves, e o índice i sobe para o pai. (**d**) O heap máximo depois de mais uma iteração do loop **while**. Nesse momento, $A[\text{PARENT}(i)] \geq A[i]$. Agora, a propriedade de heap máximo é válida, e o procedimento termina

6.5-4

Por que nos preocupamos em definir a chave do nó inserido como $-\infty$ na linha 2 de MAX-HEAP-INSERT quando a nossa próxima ação é aumentar sua chave para o valor desejado?

6.5-5

Demonstre a correção de HEAP-INCREASE-KEY usando este loop invariante:

> No começo de cada iteração do loop **while** das linhas 4 a 6, o arranjo $A[1 .. \textit{tamanho-do-heap}[A]]$ satisfaz à propriedade de heap máximo, a não ser pelo fato de que é possível haver uma violação: $A[i]$ pode ser maior que $A[\text{PARENT}(i)]$.

6.5-6

Mostre como implementar uma fila de primeiro a entrar, primeiro a sair com uma fila de prioridades. Mostre como implementar uma pilha com uma fila de prioridades. (Filas e pilhas são definidas na Seção 10.1.)

6.5-7

A operação HEAP-DELETE(A, i) elimina o item do nó i do heap A. Forneça uma implementação de HEAP-DELETE que seja executada no tempo $O(\lg n)$ para um heap máximo de n elementos.

6.5-8

Forneça um algoritmo de tempo $O(n \lg k)$ para intercalar k listas ordenadas em uma única lista ordenada, onde n é o número total de elementos em todas as listas de entrada. (*Sugestão*: Use um heap mínimo para fazer a intercalação de k modos.)

Problemas

6-1 Construção de um heap com o uso de inserção

O procedimento BUILD-MAX-HEAP na Seção 6.3 pode ser implementado pelo uso repetido de MAX-HEAP-INSERT para inserir os elementos no heap. Considere a implementação a seguir:

```
BUILD-MAX-HEAP'(A)
1 tamanho-do-heap[A] ← 1
2 for i ← 2 to comprimento[A]
3     do MAX-HEAP-INSERT(A, A[i])
```

a. Os procedimentos BUILD-MAX-HEAP e BUILD-MAX-HEAP' sempre criam o mesmo heap quando são executados sobre o mesmo arranjo de entrada? Prove que eles o fazem, ou então forneça um contra-exemplo.

b. Mostre que no pior caso, BUILD-MAX-HEAP' exige o tempo $\Theta(n \lg n)$ para construir um heap de n elementos.

6-2 Análise de heaps d-ários

Um ***heap d-ário*** é semelhante a um heap binário, mas (com uma única exceção possível) nós que não são de folhas têm d filhos em vez de dois filhos.

a. Como você representaria um heap d-ário em um arranjo?

b. Qual é a altura de um heap d-ário de n elementos em termos de n e d?

c. Dê uma implementação eficiente de EXTRACT-MAX em um heap máximo d-ário. Analise seu tempo de execução em termos de d e n.

d. Forneça uma implementação eficiente de INSERT em um heap máximo d-ário. Analise seu tempo de execução em termos de d e n.

e. Dê uma implementação eficiente de HEAP-INCREASE-KEY(A, i, k), que primeiro configura $A[i] \leftarrow \max(A[i], k)$ e depois atualiza adequadamente a estrutura de heap máximo d-ário. Analise seu tempo de execução em termos de d e n.

6-3 Quadros de Young

Um ***quadro de Young*** $m \times n$ é uma matriz $m \times n$ *tal que* as entradas de cada linha estão em seqüência ordenada da esquerda para a direita, e as entradas de cada coluna estão em seqüência ordenada de cima para baixo. Algumas das entradas de um quadro de Young podem ser ∞, o que tratamos como elementos inexistentes. Desse modo, um quadro de Young pode ser usado para conter $r \leq mn$ números finitos.

a. Trace um quadro de Young 4×4 contendo os elementos $\{9, 16, 3, 2, 4, 8, 5, 14, 12\}$.

b. Demonstre que um quadro de Young Y $m \times n$ é vazio se $Y[1, 1] = \infty$. Demonstre que Y é completo (contém mn elementos) se $Y[m, n] < \infty$.

c. Forneça um algoritmo para implementar EXTRACT-MIN em um quadro de Young $m \times n$ não vazio que funcione no tempo $O(m+n)$. Seu algoritmo deve usar uma sub-rotina recursiva que solucione um problema $m \times n$ resolvendo recursivamente um subproblema $(m-1) \times n$ ou $m \times (n-1)$. (*Sugestão*: Pense em MAX-HEAPIFY.) Defina $T(p)$, onde $p = m + n$, como o tempo de execução máximo de EXTRACT-MIN em qualquer quadro de Young $m \times n$. Forneça e resolva uma recorrência para $T(p)$ que produza o limite de tempo $O(m + n)$.

d. Mostre como inserir um novo elemento em um quadro de Young $m \times n$ não completo no tempo $O(m + n)$.

e. Sem usar nenhum outro método de ordenação como uma sub-rotina, mostre como utilizar um quadro de Young $n \times n$ para ordenar n^2 números no tempo $O(n^3)$.

f. Forneça um algoritmo de tempo $O(m + n)$ para determinar se um dado número está armazenado em um determinado quadro de Young $m \times n$.

Notas do capítulo

O algoritmo heapsort foi criado por Williams [316], que também descreveu como implementar uma fila de prioridades com um heap. O procedimento BUILD-MAX-HEAP foi sugerido por Floyd [90].

Usamos heaps mínimos para implementar filas de prioridade mínima nos Capítulos 16, 23 e 24. Também damos uma implementação com limites de tempo melhorados para certas operações nos Capítulos 19 e 20.

Implementações mais rápidas de filas de prioridades são possíveis para dados inteiros. Uma estrutura de dados criada por van Emde Boas [301] admite as operações MINIMUM, MAXIMUM, INSERT, DELETE, SEARCH, EXTRACT-MIN, EXTRACT-MAX, PREDECESSOR e SUCCESSOR, no tempo de pior caso $O(\lg \lg C)$, sujeito à restrição de que o universo de chaves é o conjunto $\{1, 2, \ldots, C\}$. Se os dados são inteiros de b bits e a memória do computador consiste em palavras de b bits endereçáveis, Fredman e Willard [99] mostraram como implementar MINIMUM no tempo $O(1)$, e INSERT e EXTRACT-MIN no tempo $O(\sqrt{\lg n})$. Thorup [299] melhorou o limite $O(\sqrt{\lg n})$ para o tempo $O((\lg \lg n)^2)$. Esse limite usa uma quantidade de espaço ilimitada em n, mas pode ser implementado em espaço linear com o uso de hash aleatório.

Um caso especial importante de filas de prioridades ocorre quando a seqüência de operações de EXTRACT-MIN é ***monotônica*** ou ***monótona***, ou seja, os valores retornados por operações sucessivas de EXTRACT-MIN são monotonicamente crescentes com o tempo. Esse caso surge em várias aplicações importantes, como o algoritmo de caminhos mais curtos de origem única de Dijkstra, discutido no Capítulo 24, e na simulação de eventos discretos. Para o algoritmo de Dijkstra é particularmente importante que a operação DECREASE-KEY seja implementada de modo eficiente.

No caso monotônico, se os dados são inteiros no intervalo $1, 2, \ldots, C$, Ahuja, Melhorn, Orlin e Tarjan [8] descrevem como implementar EXTRACT-MIN e INSERT no tempo amortizado $O(\lg C)$ (consulte o Capítulo 17 para obter mais informações sobre análise amortizada) e DECREASE-KEY no tempo $O(1)$, usando uma estrutura de dados chamada heap de raiz. O limite $O(\lg C)$ pode ser melhorado para $O(\sqrt{\lg C})$ com o uso de heaps de Fibonacci (consulte o Capítulo 20) em conjunto com heaps de raiz. O limite foi melhorado ainda mais para o tempo esperado $O(\lg^{1/3 + \varepsilon} C)$ por Cherkassky, Goldberg e Silverstein [58], que combinam a estrutura de baldes em vários níveis de Denardo e Fox [72] com o heap de Thorup mencionado antes. Raman [256] melhorou mais ainda esses resultados para obter um limite de $O(\min(\lg^{1/4 + \varepsilon} C, \lg^{1/3 + \varepsilon} n))$, para qualquer $\varepsilon > 0$. Discussões mais detalhadas desses resultados podem ser encontradas em trabalhos de Raman [256] e Thorup [299].

Capítulo 7

Quicksort

O quicksort (ordenação rápida) é um algoritmo de ordenação cujo tempo de execução do pior caso é $\Theta(n^2)$ sobre um arranjo de entrada de n números. Apesar desse tempo de execução lento no pior caso, o quicksort com freqüência é a melhor opção prática para ordenação, devido a sua notável eficiência na média: seu tempo de execução esperado é $\Theta(n \lg n)$, e os fatores constantes ocultos na notação $\Theta(n \lg n)$ são bastante pequenos. Ele também apresenta a vantagem da ordenação local (ver Capítulo 2) e funciona bem até mesmo em ambientes de memória virtual.

A Seção 7.1 descreve o algoritmo e uma sub-rotina importante usada pelo quicksort para particionamento. Pelo fato do comportamento de quicksort ser complexo, começaremos com uma discussão intuitiva de seu desempenho na Seção 7.2 e adiaremos sua análise precisa até o final do capítulo. A Seção 7.3 apresenta uma versão de quicksort que utiliza a amostragem aleatória. Esse algoritmo tem um bom tempo de execução no caso médio, e nenhuma entrada específica induz seu comportamento do pior caso. O algoritmo aleatório é analisado na Seção 7.4, onde mostraremos que ele é executado no tempo $\Theta(n^2)$ no pior caso e no tempo $O(n \lg n)$ em média.

7.1 Descrição do quicksort

O quicksort, como a ordenação por intercalação, se baseia no paradigma de dividir e conquistar introduzido na Seção 2.3.1. Aqui está o processo de dividir e conquistar em três passos para ordenar um subarranjo típico $A[p \,..\, r]$.

Dividir: O arranjo $A[p \,..\, r]$ é particionado (reorganizado) em dois subarranjos (possivelmente vazios) $A[p \,..\, q-1]$ e $A[q+1 \,..\, r]$ tais que cada elemento de $A[p \,..\, q-1]$ é menor que ou igual a $A[q]$ que, por sua vez, é igual ou menor a cada elemento de $A[q+1 \,..\, r]$. O índice q é calculado como parte desse procedimento de particionamento.

Conquistar: Os dois subarranjos $A[p \,..\, q-1]$ e $A[q+1 \,..\, r]$ são ordenados por chamadas recursivas a quicksort.

Combinar: Como os subarranjos são ordenados localmente, não é necessário nenhum trabalho para combiná-los: o arranjo $A[p \,..\, r]$ inteiro agora está ordenado.

O procedimento a seguir implementa o quicksort.

```
QUICKSORT(A, p, r)
1 if p < r
2    then q ← PARTITION(A, p, r)
3         QUICKSORT(A, p, q − 1)
4         QUICKSORT(A, q + 1, r)
```

Para ordenar um arranjo A inteiro, a chamada inicial é QUICKSORT(A, 1, *comprimento*[A]).

Particionamento do arranjo

A chave para o algoritmo é o procedimento PARTITION, que reorganiza o subarranjo $A[p\,..\,r]$ localmente.

```
PARTITION(A, p, r)
1 x ← A[r]
2 i ← p − 1
3 for j ← p to r − 1
4     do if A[j] ≤ x
5           then i ← i + 1
6                trocar A[i] ← A[j]
7 trocar A[i + 1] ↔ A[r]
8 return i + 1
```

A Figura 7.1 mostra a operação de PARTITION sobre um arranjo de 8 elementos. PARTITION sempre seleciona um elemento $x = A[r]$ como um elemento ***pivô*** ao redor do qual será feito o particionamento do subarranjo $A[p\,..\,r]$. À medida que o procedimento é executado, o arranjo é particionado em quatro regiões (possivelmente vazias). No início de cada iteração do loop **for** nas linhas 3 a 6, cada região satisfaz a certas propriedades, que podemos enunciar como um loop invariante:

No início de cada iteração do loop das linhas 3 a 6, para qualquer índice de arranjo k,

1. Se $p \le k \le i$, então $A[k] \le x$.
2. Se $i + 1 \le k \le j - 1$, então $A[k] > x$.
3. Se $k = r$, então $A[k] = x$.

A Figura 7.2 resume essa estrutura. Os índices entre j e $r - 1$ não são cobertos por quaisquer dos três casos, e os valores nessas entradas não têm nenhum relacionamento particular para o pivô x.

Precisamos mostrar que esse loop invariante é verdadeiro antes da primeira iteração, que cada iteração do loop mantém o invariante e que o invariante fornece uma propriedade útil para mostrar a correção quando o loop termina.

Inicialização: Antes da primeira iteração do loop, $i = p - 1$ e $j = p$. Não há nenhum valor entre p e i, e nenhum valor entre $i + 1$ e $j - 1$; assim, as duas primeiras condições do loop invariante são satisfeitas de forma trivial. A atribuição na linha 1 satisfaz à terceira condição.

Manutenção: Como mostra a Figura 7.3, existem dois casos a considerar, dependendo do resultado do teste na linha 4. A Figura 7.3(a) mostra o que acontece quando $A[j] > x$; a única ação no loop é incrementar j. Depois que j é incrementado, a condição 2 é válida para $A[j - 1]$ e todas as outras entradas permanecem inalteradas.

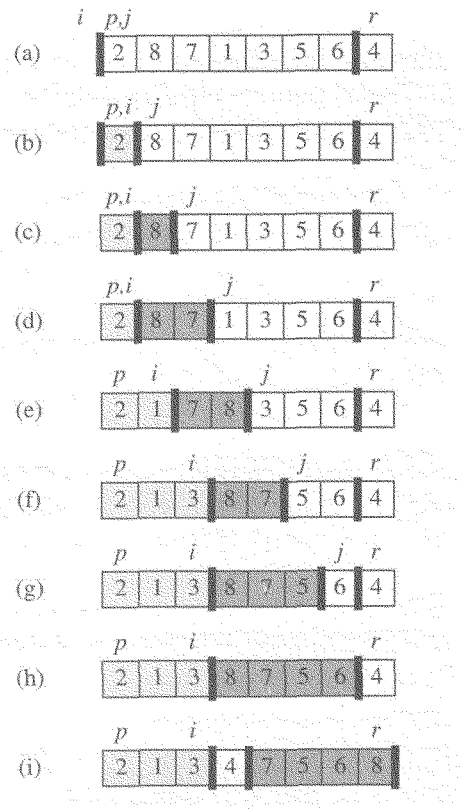

FIGURA 7.1 A operação de PARTITION sobre um exemplo de arranjo. Os elementos do arranjo ligeiramente sombreados estão todos na primeira partição com valores não maiores que x. Elementos fortemente sombreados estão na segunda partição com valores maiores que x. Os elementos não sombreados ainda não foram inseridos em uma das duas primeiras partições, e o elemento final branco é o pivô. (**a**) O arranjo inicial e as configurações de variáveis. Nenhum dos elementos foi inserido em qualquer das duas primeiras partições. (**b**) O valor 2 é "trocado com ele mesmo" e inserido na partição de valores menores. (**c**)–(**d**) Os valores 8 e 7 foram acrescentados à partição de valores maiores. (**e**) Os valores 1 e 8 são permutados, e a partição menor cresce. (**f**) Os valores 3 e 8 são permutados, e a partição menor cresce. (**g**)–(**h**) A partição maior cresce até incluir 5 e 6 e o loop termina. (**i**) Nas linhas 7 e 8, o elemento pivô é permutado de forma a residir entre as duas partições

A Figura 7.3(b) mostra o que acontece quando $A[j] \leq x$; i é incrementado, $A[i]$ e $A[j]$ são permutados, e então j é incrementado. Por causa da troca, agora temos que $A[i] \leq x$, e a condição 1 é satisfeita. De modo semelhante, também temos que $A[j-1] > x$, pois o item que foi trocado em $A[j-1]$ é, pelo loop invariante, maior que x.

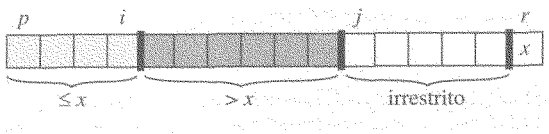

FIGURA 7.2 As quatro regiões mantidas pelo procedimento PARTITION em um subarranjo $A[p \,..\, r]$. Os valores em $A[p \,..\, i]$ são todos menores que ou iguais a x, os valores em $A[i+1 \,..\, j-1]$ são todos maiores que x, e $A[r] = x$. Os valores em $A[j \,..\, r-1]$ podem ser quaisquer valores

Término: No término, $j = r$. Então, toda entrada no arranjo está em um dos três conjuntos descritos pelo invariante, e particionamos os valores no arranjo em três conjuntos: os que são menores que ou iguais a x, os maiores que x, e um conjunto unitário contendo x.

As duas linhas finais de PARTITION movem o elemento pivô para seu lugar no meio do arranjo, permutando-o com o elemento mais à esquerda que é maior que x. A saída de PARTITION agora satisfaz às especificações dadas para a etapa de dividir.

O tempo de execução de PARTITION sobre o subarranjo $A[p\,..\,r]$ é $\Theta(n)$, onde $n = r - p + 1$ (ver Exercício 7.1-3).

Exercícios

7.1-1

Usando a Figura 7. 1 como modelo, ilustre a operação de PARTITION sobre o arranjo $A = \langle 13, 19, 9, 5, 12, 8, 7, 4, 11, 2, 6, 21\rangle$.

7.1-2

Que valor de q PARTITION retorna quando todos os elementos no arranjo $A[p\,..\,r]$ têm o mesmo valor? Modifique PARTITION de forma que $q = (p + r)/2$ quando todos os elementos no arranjo $A[p\,..\,r]$ têm o mesmo valor.

7.1-3

Forneça um breve argumento mostrando que o tempo de execução de PARTITION sobre um subarranjo de tamanho n é $\Theta(n)$.

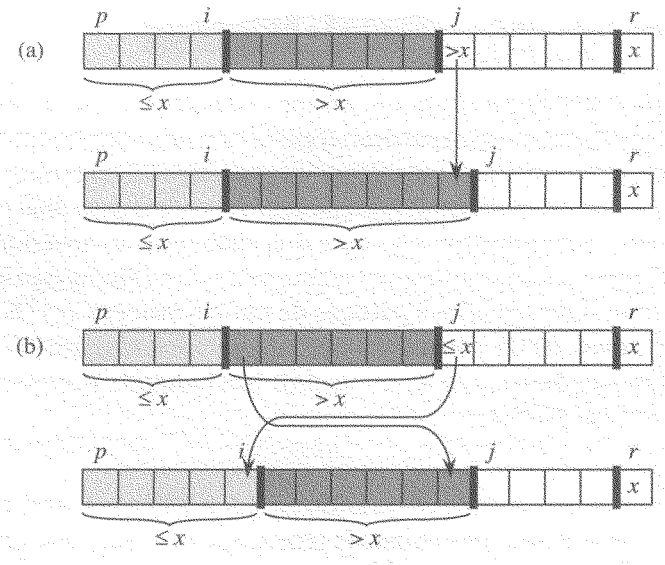

FIGURA 7.3 Os dois casos para uma iteração do procedimento PARTITION. (**a**) Se $A[j] > x$, a única ação é incrementar j, o que mantém o loop invariante. (**b**) Se $A[j] \leq x$, o índice i é incrementado, $A[i]$ e $A[j]$ são permutados, e então j é incrementado. Novamente, o loop invariante é mantido

7.1-4

De que maneira você modificaria QUICKSORT para fazer a ordenação em ordem não crescente?

7.2 O desempenho de quicksort

O tempo de execução de quicksort depende do fato de o particionamento ser balanceado ou não balanceado, e isso por sua vez depende de quais elementos são usados para particionar. Se o particionamento é balanceado, o algoritmo é executado assintoticamente tão rápido quanto a ordenação por intercalação. Contudo, se o particionamento é não balanceado, ele pode ser executado assintoticamente de forma tão lenta quanto a ordenação por inserção. Nesta seção, investigaremos informalmente como o quicksort é executado sob as hipóteses de particionamento balanceado e particionamento não balanceado.

Particionamento no pior caso

O comportamento do pior caso para o quicksort ocorre quando a rotina de particionamento produz um subproblema com $n - 1$ elementos e um com 0 elementos. (Essa afirmativa é demonstrada na Seção 7.4.1.) Vamos supor que esse particionamento não balanceado surge em cada chamada recursiva. O particionamento custa o tempo $\Theta(n)$. Tendo em vista que a chamada recursiva sobre um arranjo de tamanho 0 simplesmente retorna, $T(0) = \Theta(1)$, e a recorrência para o tempo de execução é

$$\begin{aligned} T(n) &= T(n-1) + T(0) + \Theta(n) \\ &= T(n-1) + \Theta(n) . \end{aligned}$$

Intuitivamente, se somarmos os custos incorridos a cada nível da recursão, conseguimos uma série aritmética (equação (A.2)), que tem o valor $\Theta(n^2)$. Na realidade, é simples usar o método de substituição para provar que a recorrência $T(n) = T(n-1) + \Theta(n)$ tem a solução $T(n) = \Theta(n^2)$. (Veja o Exercício 7.2-1.)

Portanto, se o particionamento é não balanceado de modo máximo em cada nível recursivo do algoritmo, o tempo de execução é $\Theta(n^2)$. Por conseguinte, o tempo de execução do pior caso do quicksort não é melhor que o da ordenação por inserção. Além disso, o tempo de execução $\Theta(n^2)$ ocorre quando o arranjo de entrada já está completamente ordenado – uma situação comum na qual a ordenação por inserção é executada no tempo $O(n)$.

Particionamento do melhor caso

Na divisão mais uniforme possível, PARTITION produz dois subproblemas, cada um de tamanho não maior que $n/2$, pois um tem tamanho $\lfloor n/2 \rfloor$ e o outro tem tamanho $\lceil n/2 \rceil - 1$. Nesse caso, o quicksort é executado com muito maior rapidez. A recorrência pelo tempo de execução é então

$$T(n) \leq 2T(n/2) + \Theta(n)$$

que, pelo caso 2 do teorema mestre (Teorema 4.1) tem a solução $T(n) = O(n \lg n)$. Desse modo, o balanceamento equilibrado dos dois lados da partição em cada nível da recursão produz um algoritmo assintoticamente mais rápido.

Particionamento balanceado

O tempo de execução do caso médio de quicksort é muito mais próximo do melhor caso que do pior caso, como mostrarão as análises da Seção 7.4. A chave para compreender por que isso poderia ser verdade é entender como o equilíbrio do particionamento se reflete na recorrência que descreve o tempo de execução.

Por exemplo, suponha que o algoritmo de particionamento sempre produza uma divisão proporcional de 9 para 1, que a princípio parece bastante desequilibrada. Então, obtemos a recorrência

$$T(n) \leq T(9n/10) + T(n/10) + cn$$

no tempo de execução de quicksort, onde incluímos explicitamente a constante c oculta no termo $\Theta(n)$. A Figura 7.4 mostra a árvore de recursão para essa recorrência. Note que todo nível da árvore tem custo cn, até ser alcançada uma condição limite à profundidade $\log_{10} n = \Theta(\lg n)$, e então os níveis têm o custo máximo cn. A recursão termina na profundidade $\log_{10/9} n = \Theta(\lg n)$. O custo total do quicksort é portanto $O(n \lg n)$. Desse modo, com uma divisão na proporção de

9 para 1 em todo nível de recursão, o que intuitivamente parece bastante desequilibrado, o quicksort é executado no tempo $O(n \lg n)$ – assintoticamente o mesmo tempo que teríamos se a divisão fosse feita exatamente ao meio. De fato, até mesmo uma divisão de 99 para 1 produz um tempo de execução igual a $O(n \lg n)$. A razão é que qualquer divisão de proporcionalidade *constante* produz uma árvore de recursão de profundidade $\Theta(\lg n)$, onde o custo em cada nível é $O(n)$. Portanto, o tempo de execução será então $O(n \lg n)$ sempre que a divisão tiver proporcionalidade constante.

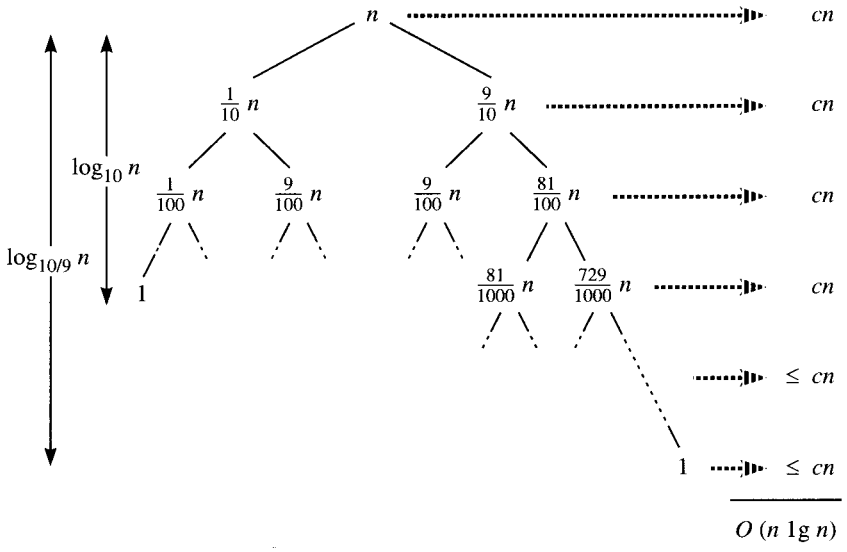

FIGURA 7.4 Uma árvore de recursão para QUICKSORT, na qual PARTITION sempre produz uma divisão de 9 para 1, resultando no tempo de execução $O(n \lg n)$. Os nós mostram tamanhos de subproblemas, com custos por nível à direita. Os custos por nível incluem a constante c implícita no termo $\Theta(n)$

Intuição para o caso médio

Para desenvolver uma noção clara do caso médio de quicksort, devemos fazer uma suposição sobre a freqüência com que esperamos encontrar as várias entradas. O comportamento de quicksort é determinado pela ordenação relativa dos valores nos elementos do arranjo dados como entrada, e não pelos valores específicos no arranjo. Como em nossa análise probabilística do problema da contratação na Seção 5.2, iremos supor por enquanto que todas as permutações dos números de entrada são igualmente prováveis.

Quando executamos o quicksort sobre um arranjo de entrada aleatório, é improvável que o particionamento sempre ocorra do mesmo modo em todo nível, como nossa análise informal pressupôs. Esperamos que algumas divisões sejam razoavelmente bem equilibradas e que algumas sejam bastante desequilibradas. Por exemplo, o Exercício 7.2-6 lhe pede para mostrar que, em cerca de 80% do tempo, PARTITION produz uma divisão mais equilibrada que 9 para 1, e mais ou menos em 20% do tempo ele produz uma divisão menos equilibrada que 9 para 1.

No caso médio, PARTITION produz uma mistura de divisões "boas" e "ruins". Em uma árvore de recursão para uma execução do caso médio de PARTITION, as divisões boas e ruins estão distribuídas aleatoriamente ao longo da árvore. Porém, suponha para fins de intuição, que as divisões boas e ruins alternem seus níveis na árvore, e que as divisões boas sejam divisões do melhor caso e as divisões ruins sejam divisões do pior caso. A Figura 7.5(a) mostra as divisões em dois níveis consecutivos na árvore de recursão. Na raiz da árvore, o custo é n para particionamento e os subarranjos produzidos têm tamanhos $n - 1$ e 1: o pior caso. No nível seguinte, o subarranjo de tamanho $n - 1$ é particionado no melhor caso em dois subarranjos de tamanho $(n - 1)/2 - 1$ e $(n - 1)/2$. Vamos supor que o custo da condição limite seja 1 para o subarranjo de tamanho 0.

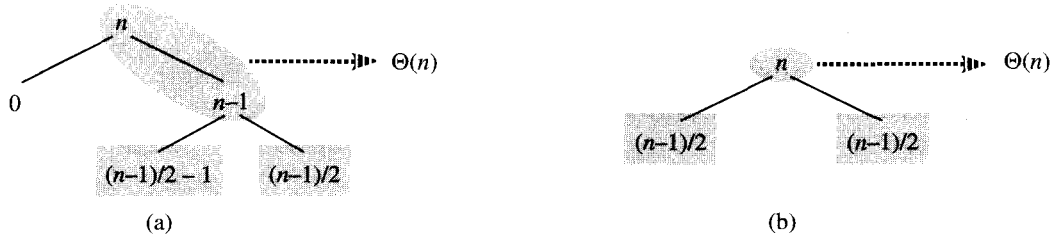

FIGURA 7.5 **(a)** Dois níveis de uma árvore de recursão para quicksort. O particionamento na raiz custa n e produz uma divisão "ruim": dois subarranjos de tamanhos 0 e $n-1$. O particionamento do subarranjo de tamanho $n-1$ custa $n-1$ e produz uma divisão "boa": subarranjos de tamanho $(n-1)/2-1$ e $(n-1)/2$. **(b)** Um único nível de uma árvore de recursão que está muito bem equilibrada. Em ambas as partes, o custo de particionamento para os subproblemas mostrados com sombreamento elíptico é $\Theta(n)$. Ainda assim, os subproblemas que restam para serem resolvidos em (a), mostrados com sombreamento retangular, não são maiores que os subproblemas correspondentes que restam para serem resolvidos em (b)

A combinação da divisão ruim seguida pela divisão boa produz três subarranjos de tamanhos 0, $(n-1)/2-1$ e $(n-1)/2$, a um custo de particionamento combinado de $\Theta(n) + \Theta(n-1) = \Theta(n)$. Certamente, essa situação não é pior que a da Figura 7.5(b), ou seja, um único nível de particionamento que produz dois subarranjos de tamanho $(n-1)/2$, ao custo $\Theta(n)$. Ainda assim, essa última situação é equilibrada! Intuitivamente, o custo $\Theta(n-1)$ da divisão ruim pode ser absorvido no custo $\Theta(n)$ da divisão boa, e a divisão resultante é boa. Desse modo, o tempo de execução do quicksort, quando os níveis se alternam entre divisões boas e ruins, é semelhante ao tempo de execução para divisões boas sozinhas: ainda $O(n \lg n)$, mas com uma constante ligeiramente maior oculta pela notação de O. Faremos uma análise rigorosa do caso médio na Seção 7.4.2.

Exercícios

7.2-1

Use o método de substituição para provar que a recorrência $T(n) = T(n-1) + \Theta(n)$ tem a solução $T(n) = \Theta(n^2)$, como afirmamos no início da Seção 7.2.

7.2-2

Qual é o tempo de execução de QUICKSORT quando todos os elementos do arranjo A têm o mesmo valor?

7.2-3

Mostre que o tempo de execução de QUICKSORT é $\Theta(n^2)$ quando o arranjo A contém elementos distintos e está ordenado em ordem decrescente.

7.2-4

Os bancos freqüentemente registram transações em uma conta na ordem dos horários das transações, mas muitas pessoas gostam de receber seus extratos bancários com os cheques relacionados na ordem de número do cheque. Em geral, as pessoas preenchem seus cheques na ordem do número do cheque, e os comerciantes normalmente os descontam com uma presteza razoável. Portanto, o problema de converter a ordenação pela hora da transação na ordenação pelo número do cheque é o problema de ordenar uma entrada quase ordenada. Demonstre que o procedimento INSERTION-SORT tenderia a superar o procedimento QUICKSORT nesse problema.

7.2-5

Suponha que as divisões em todo nível de quicksort estejam na proporção $1-\alpha$ para α, onde $0 < \alpha \leq 1/2$ é uma constante. Mostre que a profundidade mínima de uma folha na árvore de recursão é aproximadamente $-\lg n/\lg \alpha$ e a profundidade máxima é aproximadamente $-\lg n/\lg(1-\alpha)$. (Não se preocupe com o arredondamento até inteiro.)

7.2-6 ⋆

Demonstre que, para qualquer constante $0 < \alpha \leq 1/2$, a probabilidade de que, em um arranjo de entradas aleatórias, PARTITION produza uma divisão mais equilibrada que $1 - \alpha$ para α é aproximadamente $1 - 2\alpha$.

7.3 Uma versão aleatória de quicksort

Na exploração do comportamento do caso médio de quicksort, fizemos uma suposição de que todas as permutações dos números de entrada são igualmente prováveis. Porém, em uma situação de engenharia nem sempre podemos esperar que ela se mantenha válida. (Ver Exercício 7.2-4.) Como vimos na Seção 5.3, às vezes adicionamos um caráter aleatório a um algoritmo para obter bom desempenho no caso médio sobre todas as entradas. Muitas pessoas consideram a versão aleatória resultante de quicksort o algoritmo de ordenação preferido para entrada grandes o suficiente.

Na Seção 5.3, tornamos nosso algoritmo aleatório permutando explicitamente a entrada. Também poderíamos fazer isso para quicksort, mas uma técnica de aleatoriedade diferente, chamada amostragem aleatória, produz uma análise mais simples. Em vez de sempre usar $A[r]$ como pivô, usaremos um elemento escolhido ao acaso a partir do subarranjo $A[p \,..\, r]$. Faremos isso permutando o elemento $A[r]$ com um elemento escolhido ao acaso de $A[p \,..\, r]$. Essa modificação, em que fazemos a amostragem aleatória do intervalo $p, \ldots, r$, assegura que o elemento pivô $x = A[r]$ tem a mesma probabilidade de ser qualquer um dos $r - p + 1$ elementos do subarranjo. Como o elemento pivô é escolhido ao acaso, esperamos que a divisão do arranjo de entrada seja razoavelmente bem equilibrada na média.

As mudanças em PARTITION e QUICKSORT são pequenas. No novo procedimento de partição, simplesmente implementamos a troca antes do particionamento real:

```
RANDOMIZED-PARTITION(A, p, r)
1 i ← RANDOM(p, r)
2 trocar A[p] ↔ A[i]
3 return PARTITION(A, p, r)
```

O novo quicksort chama RANDOMIZED-PARTITION em lugar de PARTITION:

```
RANDOMIZED-QUICKSORT(A, p, r)
1 if p < r
2   then q ← RANDOMIZED-PARTITION(A, p, r)
3        RANDOMIZED-QUICKSORT(A, p, q − 1)
4        RANDOMIZED-QUICKSORT(A, q + 1, r)
```

Analisaremos esse algoritmo na próxima seção.

Exercícios

7.3-1

Por que analisamos o desempenho do caso médio de um algoritmo aleatório e não seu desempenho no pior caso?

7.3-2

Durante a execução do procedimento RANDOMIZED-QUICKSORT, quantas chamadas são feitas ao gerador de números aleatórios RANDOM no pior caso? E no melhor caso? Dê a resposta em termos de notação Θ.

7.4 Análise de quicksort

A Seção 7.2 forneceu alguma intuição sobre o comportamento do pior caso do quicksort e sobre o motivo pelo qual esperamos que ele funcione rapidamente. Nesta seção, analisaremos o comportamento do quicksort de forma mais rigorosa. Começaremos com uma análise do pior caso, que se aplica a QUICKSORT ou a RANDOMIZED-QUICKSORT, e concluímos com uma análise do caso médio de RANDOMIZED-QUICKSORT.

7.4.1 Análise do pior caso

Vimos na Seção 7.2 que uma divisão do pior caso em todo nível de recursão do quicksort produz um tempo de execução igual a $\Theta(n^2)$ que, intuitivamente, é o tempo de execução do pior caso do algoritmo. Agora, vamos provar essa afirmação.

Usando o método de substituição (ver Seção 4.1), podemos mostrar que o tempo de execução de quicksort é $O(n^2)$. Seja $T(n)$ o tempo no pior caso para o procedimento QUICKSORT sobre uma entrada de tamanho n. Então, temos a recorrência

$$T(n) = \max_{0 \le q \le n-1} (T(q) + T(n-q-1)) + \Theta(n) , \tag{7.1}$$

onde o parâmetro q varia de 0 a $n-1$, porque o procedimento PARTITION produz dois subproblemas com tamanho total $n-1$. Supomos que $T(n) \le cn^2$ para alguma constante c. Pela substituição dessa suposição na recorrência (7.1), obtemos

$$\begin{aligned} T(n) &= \max_{0 \le q \le n-1} (cq^2 + c(n-q-1)^2) + \Theta(n) \\ &= c \cdot \max_{0 \le q \le n-1} (q^2 + (n-q-1)^2) + \Theta(n) . \end{aligned}$$

A expressão $q^2 + (n-q-1)^2$ alcança um máximo sobre o intervalo $0 \le q \le n-1$ do parâmetro em um dos pontos extremos, como pode ser visto pelo fato da segunda derivada da expressão com relação a q ser positiva (ver Exercício 7.4-3). Essa observação nos dá o limite $\max_{0 \le q \le n-1} (q^2 + (n-q-1)^2) \le (n-1)^2 = n^2 - 2n + 1$. Continuando com nossa definição do limite de $T(n)$, obtemos

$$\begin{aligned} T(n) &\le cn^2 - c(2n-1) + \Theta(n) \\ &\le cn^2 , \end{aligned}$$

pois podemos escolher a constante c grande o suficiente para que o termo $c(2n-1)$ domine o termo $\Theta(n)$. Portanto, $T(n) = O(n^2)$. Vimos na Seção 7.2 um caso específico em que quicksort demora o tempo $\Omega(n^2)$: quando o particionamento é desequilibrado. Como alternativa, o Exercício 7.4-1 lhe pede para mostrar que a recorrência (7.1) tem uma solução $T(n) = \Omega(n^2)$. Desse modo, o tempo de execução (no pior caso) de quicksort é $\Theta(n^2)$.

7.4.2 Tempo de execução esperado

Já fornecemos um argumento intuitivo sobre o motivo pelo qual o tempo de execução do caso médio de RANDOMIZED-QUICKSORT é $O(n \lg n)$: se, em cada nível de recursão, a divisão induzida por RANDOMIZED-PARTITION colocar qualquer fração constante dos elementos em um lado da partição, então a árvore de recursão terá a profundidade $\Theta(\lg n)$, e o trabalho $O(n)$ será

executado em cada nível. Ainda que sejam adicionados novos níveis com a divisão mais desequilibrada possível entre esses níveis, o tempo total continuará a ser $O(n \lg n)$. Podemos analisar o tempo de execução esperado de RANDOMIZED-QUICKSORT com precisão, compreendendo primeiro como o procedimento de particionamento opera, e depois usando essa compreensão para derivar um limite $O(n \lg n)$ sobre o tempo de execução esperado. Esse limite superior no tempo de execução esperado, combinado com o limite no melhor caso $\Theta(n \lg n)$ que vimos na Seção 7.2, resulta em um tempo de execução esperado $\Theta(n \lg n)$.

Tempo de execução e comparações

O tempo de execução de QUICKSORT é dominado pelo tempo gasto no procedimento PARTITION. Toda vez que o procedimento PARTITION é chamado, um elemento pivô é selecionado, e esse elemento nunca é incluído em quaisquer chamadas recursivas futuras a QUICKSORT e PARTITION. Desse modo, pode haver no máximo n chamadas a PARTITION durante a execução inteira do algoritmo de quicksort. Uma chamada a PARTITION demora o tempo $O(1)$ mais um período de tempo proporcional ao número de iterações do loop **for** das linhas 3 a 6. Cada iteração desse loop **for** executa uma comparação na linha 4, comparando o elemento pivô a outro elemento do arranjo A. Assim, se pudermos contar o número total de vezes que a linha 4 é executada, poderemos limitar o tempo total gasto no loop **for** durante toda a execução de QUICKSORT.

Lema 7.1
Seja X o número de comparações executadas na linha 4 de PARTITION por toda a execução de QUICKSORT sobre um arranjo de n elementos. Então, o tempo de execução de QUICKSORT é $O(n + X)$.

Prova Pela discussão anterior, existem n chamadas a PARTITION, cada uma das quais faz uma proporção constante do trabalho e depois executa o loop **for** um certo número de vezes. Cada iteração do loop **for** executa a linha 4. ■

Portanto, nossa meta é calcular X, o número total de comparações executadas em todas as chamadas a PARTITION. Não tentaremos analisar quantas comparações são feitas em *cada* chamada a PARTITION. Em vez disso, derivaremos um limite global sobre o número total de comparações. Para fazê-lo, devemos reconhecer quando o algoritmo compara dois elementos do arranjo e quando ele não o faz. Para facilitar a análise, renomeamos os elementos do arranjo A como $z_1, z_2, \ldots, z_n$, com z_i sendo o i-ésimo menor elemento. Também definimos o conjunto $Z_{ij} = \{z_i, z_{i+1}, \ldots, z_j\}$ como o conjunto de elementos entre z_i e z_j, inclusive.

Quando o algoritmo compara z_i e z_j? Para responder a essa pergunta, primeiro observamos que cada par de elementos é comparado no máximo uma vez. Por quê? Os elementos são comparados apenas ao elemento pivô e, depois que uma chamada específica de PARTITION termina, o elemento pivô usado nessa chamada nunca é comparado novamente a quaisquer outros elementos.

Nossa análise utiliza variáveis indicadoras aleatórias (consulte a Seção 5.2). Definimos

$$X_{ij} = \mathrm{I}\{z_i \text{ é comparado a } z_j\},$$

onde estamos considerando se a comparação tem lugar em qualquer instante durante a execução do algoritmo, não apenas durante uma iteração ou uma chamada de PARTITION. Tendo em vista que cada par é comparado no máximo uma vez, podemos caracterizar facilmente o número total de comparações executadas pelo algoritmo:

$$X = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} X_{ij} .$$

Tomando as expectativas em ambos os lados, e depois usando a linearidade de expectativa e o Lema 5.1, obtemos

$$\begin{aligned} \mathrm{E}[X] &= \mathrm{E}\left[\sum_{i=1}^{n-1}\sum_{j=i+1}^{n} X_{ij}\right] \\ &= \sum_{i=1}^{n-1}\sum_{j=i+1}^{n} \mathrm{E}[X_{ij}] \\ &= \sum_{i=1}^{n-1}\sum_{j=i+1}^{n} \Pr\{z_i \text{ é comparado a } z_j\} . \end{aligned} \tag{7.2}$$

Resta calcular Pr $\{z_i$ é comparado a $z_j\}$. Nossa análise parte do princípio de que cada pivô é escolhido ao acaso e de forma independente.

É útil imaginar quando dois itens *não* são comparados. Considere uma entrada para quicksort dos números 1 a 10 (em qualquer ordem) e suponha que o primeiro elemento pivô seja 7. Então, a primeira chamada a PARTITION separa os números em dois conjuntos: {1, 2, 3, 4, 5, 6} e {8, 9, 10}. Fazendo-se isso, o elemento pivô 7 é comparado a todos os outros elementos, mas nenhum número do primeiro conjunto (por exemplo, 2) é ou jamais será comparado a qualquer número do segundo conjunto (por exemplo, 9).

Em geral, uma vez que um pivô x é escolhido com $z_i < x < z_j$, sabemos que z_i e z_j não podem ser comparados em qualquer momento subseqüente. Se, por outro lado, z_i for escolhido como um pivô antes de qualquer outro item em Z_{ij}, então z_i será comparado a cada item em Z_{ij}, exceto ele mesmo. De modo semelhante, se z_j for escolhido como pivô antes de qualquer outro item em Z_{ij}, então z_j será comparado a cada item em Z_{ij}, exceto ele próprio. Em nosso exemplo, os valores 7 e 9 são comparados porque 7 é o primeiro item de $Z_{7,9}$ a ser escolhido como pivô. Em contraste, 2 e 9 nunca serão comparados, porque o primeiro elemento pivô escolhido de $Z_{2,9}$ é 7. Desse modo, z_i e z_j são comparados se e somente se o primeiro elemento a ser escolhido como pivô de Z_{ij} é z_i ou z_j.

Agora, calculamos a probabilidade de que esse evento ocorra. Antes do ponto em que um elemento de Z_{ij} é escolhido como pivô, todo o conjunto Z_{ij} está reunido na mesma partição. Por conseguinte, qualquer elemento de Z_{ij} tem igual probabilidade de ser o primeiro escolhido como pivô. Pelo fato do conjunto Z_{ij} ter $j - i + 1$ elementos e tendo em vista que os pivôs são escolhidos ao acaso e de forma independente, a probabilidade de qualquer elemento dado ser o primeiro escolhido como pivô é $1/(j - i + 1)$. Desse modo, temos

$$\begin{aligned} \Pr\{z_i \text{ é comparado a } z_j\} &= \Pr\{z_i \text{ ou } z_j \text{ é o primeiro pivô escolhido de } Z_{ij}\} \\ &= \Pr\{z_i \text{ é o primeiro pivô escolhido de } Z_{ij}\} \\ &\quad + \Pr\{z_j \text{ é o primeiro pivô escolhido de } Z_{ij}\} \\ &= \frac{1}{j-i+1} + \frac{1}{j-i+1} \\ &= \frac{2}{j-i+1} . \end{aligned} \tag{7.3}$$

A segunda linha se segue porque os dois eventos são mutuamente exclusivos. Combinando as equações (7.2) e (7.3), obtemos

$$\mathrm{E}[X] = \sum_{i=1}^{n-1}\sum_{j=i+1}^{n} \frac{2}{j-i+1} .$$

Podemos avaliar essa soma usando uma troca de variáveis ($k = j - i$) e o limite sobre a série harmônica na equação (A.7):

$$\begin{aligned}
\mathrm{E}[X] &= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{2}{j-i+1} \\
&= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{2}{k+1} \\
&< \sum_{i=1}^{n-1} \sum_{k=1}^{n} \frac{2}{k} \\
&= \sum_{i=1}^{n-1} O(\lg n) \\
&= O(n \lg n) . \qquad (7.4)
\end{aligned}$$

Desse modo concluímos que, usando-se RANDOMIZED-PARTITION, o tempo de execução esperado de quicksort é $O(n \lg n)$.

Exercícios

7.4-1
Mostre que, na recorrência

$$T(n) = \max_{0 \le q \le n-1} (T(q) + T(n-q-1)) + \Theta(n) ,$$

$T(n) = \Omega(n^2)$.

7.4-2
Mostre que o tempo de execução do quicksort no melhor caso é $\Omega(n \lg n)$.

7.4-3
Mostre que $q^2 + (n-q-1)^2$ alcança um máximo sobre $q = 0, 1, \ldots, n-1$ quando $q = 0$ ou $q = n-1$.

7.4-4
Mostre que o tempo de execução esperado do procedimento RANDOMIZED-QUICKSORT é $\Omega(n \lg n)$.

7.4-5
O tempo de execução do quicksort pode ser melhorado na prática, aproveitando-se o tempo de execução muito pequeno da ordenação por inserção quando sua entrada se encontra "quase" ordenada. Quando o quicksort for chamado em um subarranjo com menos de k elementos, deixe-o simplesmente retornar sem ordenar o subarranjo. Após o retorno da chamada de alto nível a quicksort, execute a ordenação por inserção sobre o arranjo inteiro, a fim de concluir o processo de ordenação. Mostre que esse algoritmo de ordenação é executado no tempo esperado $O(nk + n \lg(n/k))$. Como k deve ser escolhido, tanto na teoria quanto na prática?

7.4-6 ★
Considere a modificação do procedimento PARTITION pela escolha aleatória de três elementos do arranjo A e pelo particionamento sobre sua mediana (o valor médio dos três elementos). Faça a aproximação da probabilidade de se obter na pior das hipóteses uma divisão de α para $(1 - \alpha)$, como uma função de α no intervalo $0 < \alpha < 1$.

Problemas

7-1 Correção da partição de Hoare

A versão de PARTITION dada neste capítulo não é o algoritmo de particionamento original. Aqui está o algoritmo de partição original, devido a T. Hoare:

```
HOARE-PARTITION(A, p, r)
 1 x ← A[p]
 2 i ← p − 1
 3 j ← r + 1
 4 while TRUE
 5     do repeat j ← j − 1
 6          until A[j] ≤ x
 7        repeat i ← i + 1
 8          until A[i] ≥ x
 9        if i < j
10          then trocar A[i] ↔ A[j]
11          else return j
```

a. Demonstre a operação de HOARE-PARTITION sobre o arranjo $A = \langle 13, 19, 9, 5, 12, 8, 7, 4, 11, 2, 6, 21\rangle$, mostrando os valores do arranjo e os valores auxiliares depois de cada iteração do loop **for** das linhas 4 a 11.

As três perguntas seguintes lhe pedem para apresentar um argumento cuidadoso de que o procedimento HOARE-PARTITION é correto. Prove que:

b. Os índices i e j são tais que nunca acessamos um elemento de A fora do subarranjo $A[p..r]$.

c. Quando HOARE-PARTITION termina, ele retorna um valor j tal que $p \leq j < r$.

d. Todo elemento de $A[p..j]$ é menor que ou igual a todo elemento de $A[j+1..r]$ quando HOARE-PARTITION termina.

O procedimento PARTITION da Seção 7.1 separa o valor do pivô (originalmente em $A[r]$) das duas partições que ele forma. Por outro lado, o procedimento HOARE-PARTITION sempre insere o valor do pivô (originalmente em $A[p]$) em uma das duas partições $A[p..j]$ e $A[j+1..r]$. Como $p \leq j < r$, essa divisão é sempre não trivial.

e. Reescreva o procedimento QUICKSORT para usar HOARE-PARTITION.

7-2 Análise alternativa de quicksort

Uma análise alternativa do tempo de execução de quicksort aleatório se concentra no tempo de execução esperado de cada chamada recursiva individual a QUICKSORT, em vez de se ocupar do número de comparações executadas.

a. Demonstre que, dado um arranjo de tamanho n, a probabilidade de que qualquer elemento específico seja escolhido como pivô é $1/n$. Use isso para definir variáveis indicadoras aleatórias $X_i = \mathrm{I}\{i\text{-ésimo menor elemento é escolhido como pivô}\}$. Qual é $\mathrm{E}[X_i]$?

b. Seja $T(n)$ uma variável aleatória denotando o tempo de execução de quicksort sobre um arranjo de tamanho n. Demonstre que

$$\mathrm{E}[T(n)] = \mathrm{E}\left[\sum_{q=1}^{n} X_q (T(q-1) + T(n-q) + \Theta(n))\right]. \tag{7.5}$$

c. Mostre que a equação (7.5) é simplificada para

$$\mathrm{E}[T(n)] = \frac{2}{n}\sum_{q=0}^{n-1}\mathrm{E}[T(q)] + \Theta(n) . \tag{7.6}$$

d. Mostre que

$$\sum_{k=1}^{n-1} k \lg k \leq \frac{1}{2}n^2 \lg n - \frac{1}{8}n^2 . \tag{7.7}$$

(*Sugestão*: Divida o somatório em duas partes, uma para $k = 1, 2, \ldots, \lceil n/2 \rceil - 1$ e uma para $k = \lceil n/2 \rceil, \ldots, n - 1$.)

e. Usando o limite da equação (7.7), mostre que a recorrência na equação (7.6) tem a solução $\mathrm{E}[T(n)] = \Theta(n \lg n)$. (*Sugestão*: Mostre, por substituição, que $\mathrm{E}[T(n)] \leq an \log n - bn$ para algumas constantes positivas a e b.)

7-3 Ordenação do pateta

Os professores Howard, Fine e Howard propuseram o seguinte algoritmo de ordenação "elegante":

```
STOOGE-SORT(A, i, j)
1 if A[i] > A[j]
2    then trocar A[i] ↔ A[j]
3 if i + 1 ≥ j
4    then return
5 k ← ⌊(j − i + 1)/3⌋          ▷ Arredonda para menos.
6 STOOGE-SORT(A, i, j − k)     ▷ Primeiros dois terços.
7 STOOGE-SORT(A, i + k, j)     ▷ Últimos dois terços.
8 STOOGE-SORT(A, i, j − k)     ▷ Primeiros dois terços novamente.
```

a. Mostre que, se $n = comprimento[A]$, então STOOGE-SORT(A, 1, $comprimento[A]$) ordena corretamente o arranjo de entrada $A[1 .. n]$.

b. Forneça uma recorrência para o tempo de execução no pior caso de STOOGE-SORT e um limite assintótico restrito (notação Θ) sobre o tempo de execução no pior caso.

c. Compare o tempo de execução no pior caso de STOOGE-SORT com o da ordenação por inserção, da ordenação por intercalação, de heapsort e de quicksort. Os professores merecem a estabilidade no emprego?

7-4 Profundidade de pilha para quicksort

O algoritmo QUICKSORT da Seção 7.1 contém duas chamadas recursivas a ele próprio. Após a chamada a PARTITION, o subarranjo da esquerda é ordenado recursivamente, e depois o subarranjo da direita é ordenado recursivamente. A segunda chamada recursiva em QUICKSORT não é realmente necessária; ela pode ser evitada pelo uso de uma estrutura de controle iterativa. Essa técnica, chamada ***recursão do final***, é automaticamente fornecida por bons compiladores. Considere a versão de quicksort a seguir, que simula a recursão do final.

```
QUICKSORT'(A, p, r)
1 while p < r
2    do ▷ Particiona e ordena o subarranjo esquerdo
3       q ← PARTITION(A, p, r)
4       QUICKSORT'(A, p, q − 1)
5       p ← q + 1
```

a. Mostre que QUICKSORT'(A, 1, *comprimento*[A]) ordena corretamente o arranjo A.

Os compiladores normalmente executam procedimentos recursivos usando uma ***pilha*** que contém informações pertinentes, inclusive os valores de parâmetros, para cada chamada recursiva. As informações para a chamada mais recente estão na parte superior da pilha, e as informações para a chamada inicial encontram-se na parte inferior. Quando um procedimento é invocado, suas informações são ***empurradas*** sobre a pilha; quando ele termina, suas informações são ***extraídas***. Tendo em vista nossa suposição de que os parâmetros de arranjos são na realidade representados por ponteiros, as informações correspondentes a cada chamada de procedimento na pilha exigem o espaço de pilha $O(1)$. A ***profundidade de pilha*** é a quantidade máxima de espaço da pilha usado em qualquer instante durante uma computação.

b. Descreva um cenário no qual a profundidade de pilha de QUICKSORT' é $\Theta(n)$ sobre um arranjo de entrada de n elementos.

c. Modifique o código de QUICKSORT' de tal modo que a profundidade de pilha no pior caso seja $\Theta(\lg n)$. Mantenha o tempo de execução esperado $O(n \lg n)$ do algoritmo.

7-5 Partição de mediana de 3

Um modo de melhorar o procedimento RANDOMIZED-QUICKSORT é criar uma partição em torno de um elemento x escolhido com maior cuidado que a simples escolha de um elemento aleatório do subarranjo. Uma abordagem comum é o método da ***mediana de 3***: escolha x como a mediana (o elemento intermediário) de um conjunto de 3 elementos selecionados aleatoriamente a partir do subarranjo. Para esse problema, vamos supor que os elementos no arranjo de entrada $A[1 .. n]$ sejam distintos e que $n \geq 3$. Denotamos o arranjo de saída ordenado por $A'[1.. n]$. Usando o método da mediana de 3 para escolher o elemento pivô x, defina $p_i = \Pr\{x = A'[i]\}$.

a. Dê uma fórmula exata para p_i como uma função de n e i para $i = 2, 3, ..., n - 1$. (Observe que $p_1 = p_n = 0$.)

b. Por qual valor aumentamos a probabilidade de escolher $x = A'[\lfloor (n + 1)/2 \rfloor]$, a mediana de $A[1.. n]$, em comparação com a implementação comum? Suponha que $n \to \infty$ e forneça a razão de limitação dessas probabilidades.

c. Se definimos uma "boa" divisão com o significado de escolher o pivô como $x = A'[i]$, onde $n/3 \leq i \leq 2n/3$, por qual quantidade aumentamos a probabilidade de se obter uma boa divisão em comparação com a implementação comum? (*Sugestão*: Faça uma aproximação da soma por uma integral.)

d. Mostre que, no tempo de execução $\Omega(n \lg n)$ de quicksort, o método da mediana de 3 só afeta o fator constante.

7-6 Ordenação nebulosa de intervalos

Considere um problema de ordenação no qual os números não são conhecidos exatamente. Em vez disso, para cada número, conhecemos um intervalo sobre a linha real a que ele pertence. Ou seja, temos n intervalos fechados da forma $[a_i, b_i]$, onde $a_i \leq b_i$. O objetivo é fazer a ***ordenação nebulosa*** desses intervalos, isto é, produzir uma permutação $\langle i_1, i_2, ..., i_n \rangle$ dos intervalos tal que exista $c_j \in [a_{i_j}, b_{i_j}]$ que satisfaça a $c_1 \leq c_2 \leq \cdots \leq c_n$.

a. Projete um algoritmo para ordenação do pateta de n intervalos. Seu algoritmo devia ter a estrutura geral de um algoritmo que faz o quicksort das extremidades esquerdas (os valores a_i), mas deve tirar proveito da sobreposição de intervalos para melhorar o tempo de execução. (À medida que os intervalos se sobrepõem cada vez mais, o problema de fazer a ordenação do pateta dos intervalos fica cada vez mais fácil. Seu algoritmo deve aproveitar tal sobreposição, desde que ela exista.)

b. Demonstre que seu algoritmo é executado no tempo esperado $\Theta(n \lg n)$ em geral, mas funciona no tempo esperado $\Theta(n)$ quando todos os intervalos se sobrepõem (isto é, quando existe um valor x tal que $x \in [a_i, b_i]$ para todo i). O algoritmo não deve verificar esse caso de forma explícita; em vez disso, seu desempenho deve melhorar naturalmente à medida que aumentar a proporção de sobreposição.

Notas do capítulo

O procedimento quicksort foi criado por Hoare [147]; a versão de Hoare aparece no Problema 7-1. O procedimento PARTITION dado na Seção 7.1 se deve a N. Lomuto. A análise da Seção 7.4 se deve a Avrim Blum. Sedgewick [268] e Bentley [40] fornecem uma boa referência sobre os detalhes de implementação e como eles são importantes.

McIlroy [216] mostrou como engenheiro um "adversário matador" que produz um arranjo sobre o qual virtualmente qualquer implementação de quicksort demora o tempo $\Theta(n^2)$. Se a implementação é aleatória, o adversário produz o arranjo depois de ver as escolhas ao acaso do algoritmo de quicksort.