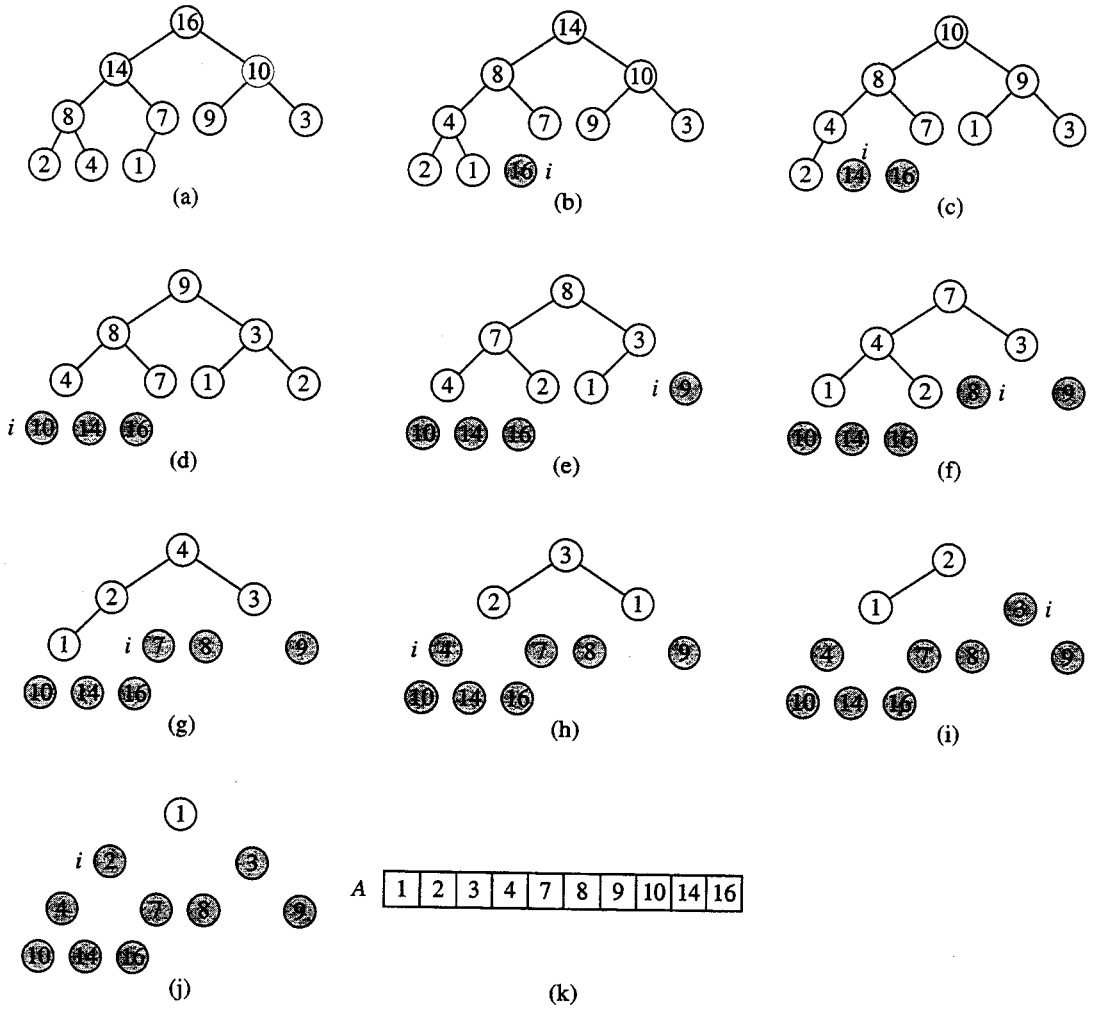

FIGURA 6.4 A operação de HEAPSORT. (a) A estrutura de dados heap máximo, logo após ter sido construída por BUILD-MAX-HEAP. (b)–(j) O heap máximo logo após cada chamada de MAX-HEAPIFY na linha 5. O valor de i nesse instante é mostrado. Apenas os nós levemente sombreados permanecem no heap. (k) O arranjo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

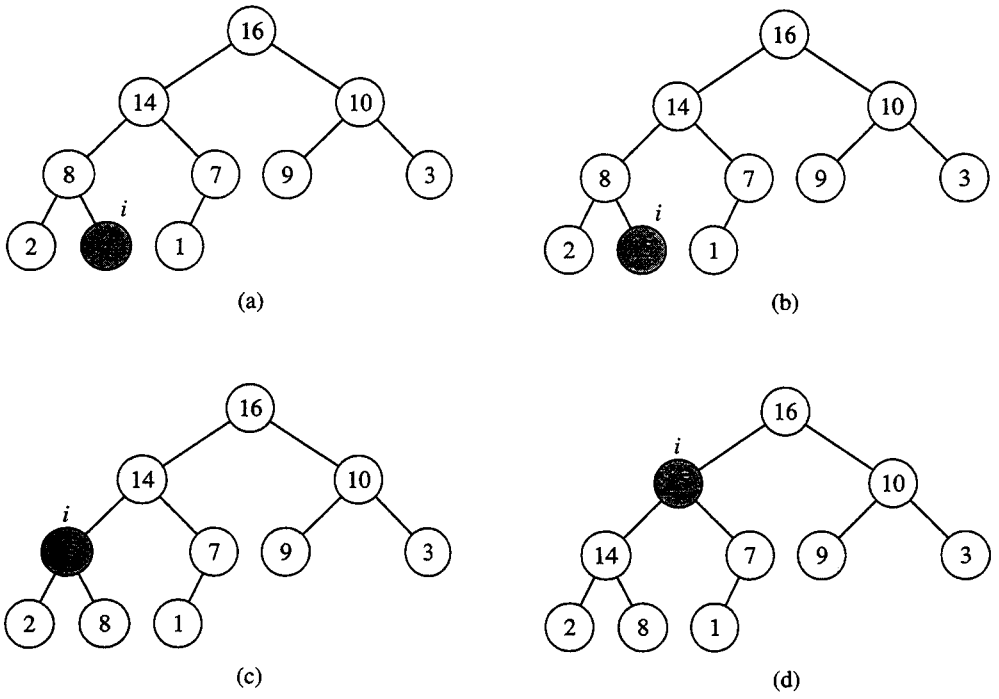

FIGURA 6.5 A operação de HEAP-INCREASE-KEY. (**a**) O heap máximo da Figura 6.4(a) com um nó cujo índice é i, fortemente sombreado. (**b**) Esse nó tem sua chave aumentada para 15. (**c**) Depois de uma iteração do loop **while** das linhas 4 a 6, o nó e seu pai trocaram chaves, e o índice i sobe para 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].