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:

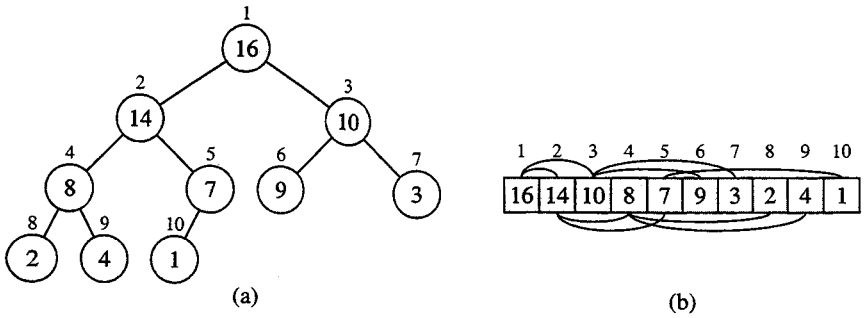

FIGURA 6.1 Um heap máximo visto como (a) uma árvore binária e (b) um arranjo. O número dentro do círculo em cada nó na árvore é o valor armazenado nesse nó. O número acima de um nó é o índice correspondente no arranjo. Acima e abaixo do arranjo encontramos linhas mostrando relacionamentos pai-filho; 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)
```

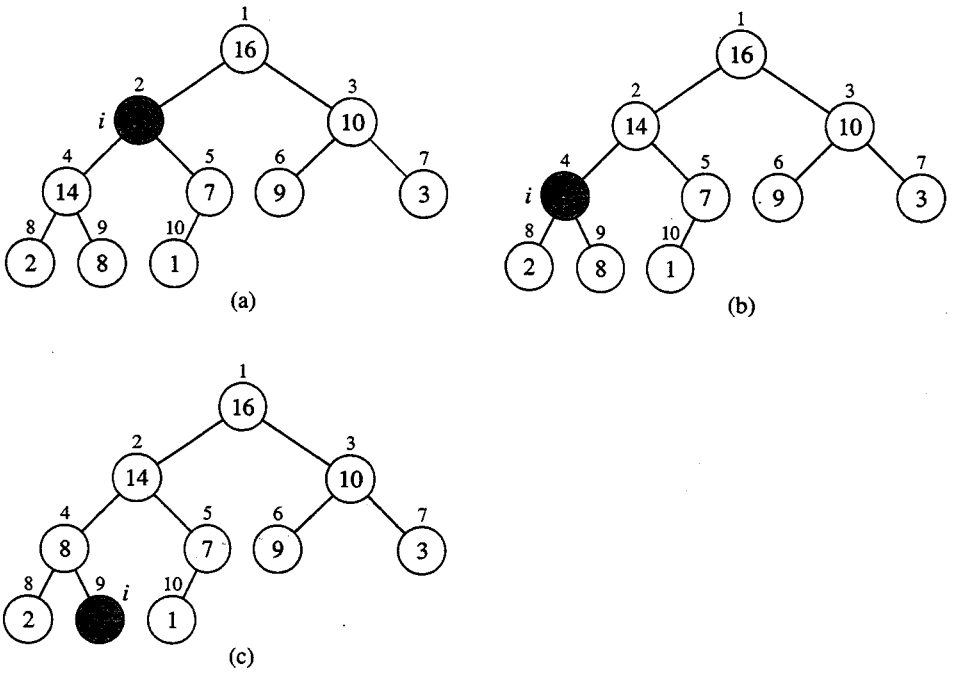

FIGURA 6.2 A ação de MAX-HEAPIFY(*A*, 2), onde *tamanho-do-heap*[*A*] = 10. (**a**) A configuração inicial, com *A*[2] no nó *i* = 2, violando a propriedade de heap máximo, pois ele não é maior que ambos os filhos. A propriedade de heap máximo é restabelecida para o nó 2 em (**b**) pela troca de *A*[2] 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.