Análise de algoritmos

Heapsort

Prof. Flávio Rogério Uber

Autor: Prof. Marco Aurélio Lopes Barbosa (UEM/DIN)

Conteúdo

Introdução

A estrutura de dados heap
- Definição
- Manutenção da propriedade de heap
- A construção de um heap

O algoritmo heapsort

Exercícios

Referências

Introdução

- Heapsort (ordenação por monte) é um algoritmo de ordenação
- Características
  - O tempo de execução é $O(n \lg n)$ (como a ordenação por intercalação)
  - Ordenação local (como a ordenação por inserção)
  - Usa uma estrutura de dados chamada heap

A estrutura de dados heap

- A estrutura de dados **heap** (binário) é um array que pode ser visto como uma árvore binária praticamente completa
  - Cada nó da árvore corresponde ao elemento do array que armazena o valor do nó
  - A árvore está preenchida em todos os níveis, exceto talvez no nível mais baixo, que é preenchido a partir da esquerda

A estrutura de dados heap

- A estrutura de dados **heap** (binário) é um array que pode ser visto como uma árvore binária praticamente completa
  - Cada nó da árvore corresponde ao elemento do array que armazena o valor do nó
  - A árvore está preenchida em todos os níveis, exceto talvez no nível mais baixo, que é preenchido a partir da esquerda
- Um array `A` que representa um heap tem dois atributos
  - `A.comprimento` que é o número de elementos do array
  - `A.tamanho-do-heap` que é o número de elementos no heap armazenado em `A` (`A.tamanho-do-heap` $\leq$ `A.comprimento`)

A estrutura de dados heap

- A estrutura de dados **heap** (binário) é um array que pode ser visto como uma árvore binária praticamente completa
  - Cada nó da árvore corresponde ao elemento do array que armazena o valor do nó
  - A árvore está preenchida em todos os níveis, exceto talvez no nível mais baixo, que é preenchido a partir da esquerda
- Um array `A` que representa um heap tem dois atributos
  - `A.comprimento` que é o número de elementos do array
  - `A.tamanho-do-heap` que é o número de elementos no heap armazenado em `A` (`A.tamanho-do-heap` $\leq$ `A.comprimento`)
- A raiz da árvore é `A[1]`

A estrutura de dados heap

- A estrutura de dados **heap** (binário) é um array que pode ser visto como uma árvore binária praticamente completa
  - Cada nó da árvore corresponde ao elemento do array que armazena o valor do nó
  - A árvore está preenchida em todos os níveis, exceto talvez no nível mais baixo, que é preenchido a partir da esquerda
- Um array `A` que representa um heap tem dois atributos
  - `A.comprimento` que é o número de elementos do array
  - `A.tamanho-do-heap` que é o número de elementos no heap armazenado em `A` (`A.tamanho-do-heap` $\leq$ `A.comprimento`)
- A raiz da árvore é `A[1]`
- Dado o índice *i* de um nó, os índice de seu pai, do filho a esquerda e do filho a direita podem ser calculados da forma
  - `parent(i) =` $\lfloor$`i/2`$\rfloor$
  - `left(i) = 2i`
  - `right(i) = 2i + 1`

Exemplo

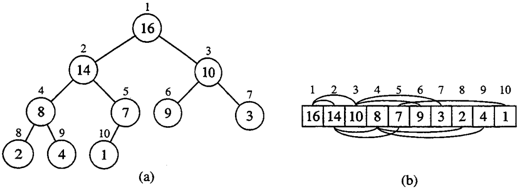


FIGURA 6.1 Um heap máximo visto como (a) uma árvore binária e (b) um arranjo. O número dentro do círculo em cada nó na árvore é o valor armazenado nesse nó. O número acima de um nó é o índice correspondente no arranjo. Acima e abaixo do arranjo encontramos linhas mostrando relacionamentos pai-filho;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

A estrutura de dados heap

- Existem dois tipos de heap
  - heap máximo
  - heap mínimo

A estrutura de dados heap

- Existem dois tipos de heap
  - heap máximo
  - heap mínimo
- Em ambos os tipos, os valores nos nós satisfazem uma **propriedade de heap**

A estrutura de dados heap

- Existem dois tipos de heap
  - heap máximo
  - heap mínimo
- Em ambos os tipos, os valores nos nós satisfazem uma **propriedade de heap**
  - Em um heap máximo, a **propriedade de heap máximo** é que, para todo nó *i* diferente da raiz `A[parent(i)]` $\geq$ `A[i]`

A estrutura de dados heap

- Existem dois tipos de heap
  - heap máximo
  - heap mínimo
- Em ambos os tipos, os valores nos nós satisfazem uma **propriedade de heap**
  - Em um heap máximo, a **propriedade de heap máximo** é que, para todo nó *i* diferente da raiz `A[parent(i)]` $\geq$ `A[i]`
  - Em um heap mínimo, a **propriedade de heap mínimo** é que, para todo nó *i* diferente da raiz `A[parent(i)]` $\leq$ `A[i]`

A estrutura de dados heap

- Existem dois tipos de heap
  - heap máximo
  - heap mínimo
- Em ambos os tipos, os valores nos nós satisfazem uma **propriedade de heap**
  - Em um heap máximo, a **propriedade de heap máximo** é que, para todo nó i diferente da raiz `A[parent(i)]` $\geq$ `A[i]`
  - Em um heap mínimo, a **propriedade de heap mínimo** é que, para todo nó i diferente da raiz `A[parent(i)]` $\leq$ `A[i]`
- Visualizando o heap como uma árvore, definimos
  - a **altura** de um nó como o número de arestas no caminho descendente simples mais longo deste o o nó até uma folha
  - a **altura do heap** como a altura de sua raiz
  - a altura de um heap é $\Theta(\lg n)$

Operações sobre heap

- Algumas operações sobre heap
  - `max-heapify`, executado no tempo $O(\lg n)$, é a chave para manter a propriedade de heap máximo
  - `build-max-heap`, executado em tempo linear, produz um heap a partir de um array de entrada não ordenado
  - `heapsort`, executado no tempo $O(n \lg n)$, ordena um array localmente

Manutenção da propriedade de heap

- A função `max-heapify` recebe como parâmetro um array `A` e um índice `i`, e requer que
  - As árvores binárias com raízes em `left(i)` e `right(i)` sejam heaps máximos
- `A[i]` pode ser menor que seus filhos
- A função `max-heapify` deixa que o valor `A[i]` "flutue para baixo", de maneira que a subárvore com raiz no índice `i` se torne um heap

Exemplo

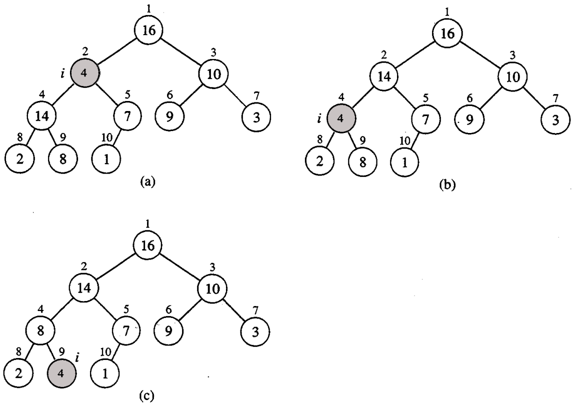


FIGURA 6.2 A ação de MAX-HEAPIFY(A, 2), onde *tamanho-do-heap*[A] = 10. **(a)** A configuração inicial, com $A[2]$ no nó $i = 2$, violando a propriedade de heap máximo, pois ele não é maior que ambos os filhos. A propriedade de heap máximo é restabelecida para o nó 2 em **(b)** pela troca de $A[2]$ por $A[4]$, 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

O algoritmo max-heapify

```
max-heapify(A, i)
l = left(i)
r = rigth(i)
if l <= A.tamanho-do-heap e A[l] > A[i] then
  maior = l
else
  maior = i
if r <= A.tamanho-do-heap e A[r] > A[maior] then
  maior = r
if maior != i then
  troca(A[i], A[maior])
  max-heapify(A, maior)
```

Análise do tempo de execução do max-heapify

- Tempo $\Theta(1)$ para corrigir os relacionamentos entre os elementos `A[i]`, `A[left[i]]` e `A[right(i)]`
- Tempo para executar `max-heapify` em uma subárvore com raiz em dos filhos do nó `i`
- As subárvores de cada filho têm tamanho máximo igual a $2n/3$ - ocorre quando o último nível da árvore está metade cheia

Análise do tempo de execução do max-heapify

- Tempo $\Theta(1)$ para corrigir os relacionamentos entre os elementos `A[i]`, `A[left[i]]` e `A[right(i)]`
- Tempo para executar `max-heapify` em uma subárvore com raiz em dos filhos do nó `i`
- As subárvores de cada filho têm tamanho máximo igual a $2n/3$ - ocorre quando o último nível da árvore está metade cheia
- Portanto, o tempo total de execução pode ser descrito pela recorrência $T(n) \leq T(2n/3) + \Theta(1)$

Análise do tempo de execução do max-heapify

- Tempo $\Theta(1)$ para corrigir os relacionamentos entre os elementos `A[i]`, `A[left[i]]` e `A[right(i)]`
- Tempo para executar `max-heapify` em uma subárvore com raiz em dos filhos do nó `i`
- As subárvores de cada filho têm tamanho máximo igual a $2n/3$ - ocorre quando o último nível da árvore está metade cheia
- Portanto, o tempo total de execução pode ser descrito pela recorrência $T(n) \leq T(2n/3) + \Theta(1)$
- Pelo caso 2 do teorema mestre $T(n) = O(\lg n)$

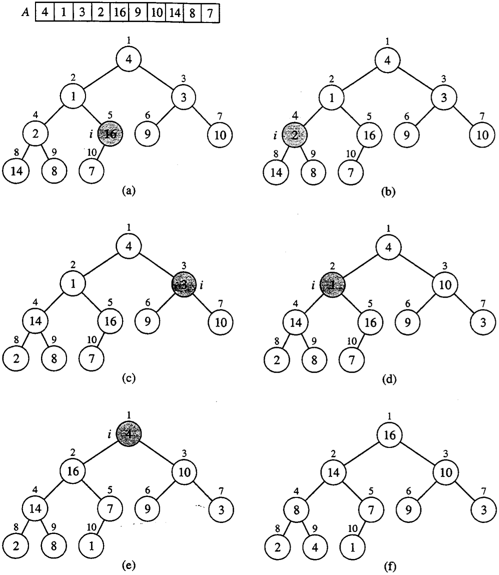
Análise do tempo de execução do max-heapify

- Tempo $\Theta(1)$ para corrigir os relacionamentos entre os elementos `A[i]`, `A[left[i]]` e `A[right(i)]`
- Tempo para executar `max-heapify` em uma subárvore com raiz em dos filhos do nó `i`
- As subárvores de cada filho têm tamanho máximo igual a $2n/3$ - ocorre quando o último nível da árvore está metade cheia
- Portanto, o tempo total de execução pode ser descrito pela recorrência $T(n) \leq T(2n/3) + \Theta(1)$
- Pelo caso 2 do teorema mestre $T(n) = O(\lg n)$
- Também podemos expressar o tempo de execução de `max-heapify` em um nó de altura h como $O(h)$

A construção de um heap

- O procedimento `max-heapify` pode ser usado de baixo para cima para converter um array `A[1..n]` em um heap máximo
- Os elementos no subarray `A`$[(\lfloor n/2 \rfloor + 1)..n]$ são folhas, e cada um é um heap máximo
- O procedimento `build-max-heap` percorre os nós restantes da árvore e executa `max-heapify` sobre cada um

Exemplo do funcionamento do `build-max-heap`



A construção de um heap

```
build-max-heap(A)
A.tamanho-do-heap = A.comprimento
for i = piso(A.comprimento / 2) downto 1
  max-heapify(A, i)
```

Corretude do `build-max-heap`

- `build-max-heap` mantém o seguinte invariante de laço
  - No começo de cada iteração do laço das linhas 2 e 3, cada nó $i + 1, i + 2, \ldots, n$ é a raíz de um heap máximo

Corretude do `build-max-heap`

- `build-max-heap` mantém o seguinte invariante de laço
  - No começo de cada iteração do laço das linhas 2 e 3, cada nó $i+1, i+2, \dots, n$ é a raíz de um heap máximo
- Vamos mostrar que esta invariante é verdadeira e como consequência vamos concluir que `build-max-heap` é correto

Corretude do `build-max-heap`

- `build-max-heap` mantém o seguinte invariante de laço
  - No começo de cada iteração do laço das linhas 2 e 3, cada nó $i+1, i+2, \ldots, n$ é a raíz de um heap máximo
- Vamos mostrar que esta invariante é verdadeira e como consequência vamos concluir que `build-max-heap` é correto
  - **Inicialização**: antes da primeira linha $i = \lfloor n/2 \rfloor$ e cada nó $\lfloor n/2 \rfloor + 1, \lfloor n/2 \rfloor + 2, \ldots, n$ é uma folha, e portanto é a raiz de um heap máximo

Corretude do `build-max-heap`

- `build-max-heap` mantém o seguinte invariante de laço
  - No começo de cada iteração do laço das linhas 2 e 3, cada nó $i + 1, i + 2, \ldots, n$ é a raíz de um heap máximo
- Vamos mostrar que esta invariante é verdadeira e como consequência vamos concluir que `build-max-heap` é correto
  - **Inicialização**: antes da primeira linha $i = \lfloor n/2 \rfloor$ e cada nó $\lfloor n/2 \rfloor + 1, \lfloor n/2 \rfloor + 2, \ldots, n$ é uma folha, e portanto é a raiz de um heap máximo
  - **Manutenção**: os filhos de i tem um número maior que i e pelo invariante são raízes de heaps máximos. Esta é a condição exigida para que a chamada `max-hepify(A, i)` torne i a raiz de um heap máximo. Decrementar i restabelece a invariante para a próxima iteração

Corretude do `build-max-heap`

- `build-max-heap` mantém o seguinte invariante de laço
  - No começo de cada iteração do laço das linhas 2 e 3, cada nó $i+1, i+2, \ldots, n$ é a raíz de um heap máximo
- Vamos mostrar que esta invariante é verdadeira e como consequência vamos concluir que `build-max-heap` é correto
  - **Inicialização**: antes da primeira linha $i = \lfloor n/2 \rfloor$ e cada nó $\lfloor n/2 \rfloor + 1, \lfloor n/2 \rfloor + 2, \ldots, n$ é uma folha, e portanto é a raiz de um heap máximo
  - **Manutenção**: os filhos de i tem um número maior que i e pelo invariante são raízes de heaps máximos. Esta é a condição exigida para que a chamada `max-hepify(A, i)` torne i a raiz de um heap máximo. Decrementar i restabelece a invariante para a próxima iteração
  - **Término**: $i = 0$, pela invariante de laço $1, 2, \ldots, n$ são raizes de um heap máximo, particularmente o nó 1 é uma raíz

Análise do build-max-heap

- Limite superior simples
  - Cada chamada de `max-heapify` custa $O(\lg n)$ e existem $O(n)$ chamadas, portanto, o tempo de execução é $O(n \lg n)$.

Análise do build-max-heap

- Limite restrito
  - O tempo de execução de `max-heapify` varia com a altura da árvore, a altura da maioria dos nós é pequena
  - Um heap de n elementos tem altura $\lfloor \lg n \rfloor$ e no máximo $\lceil n/2^{h+1} \rceil$ nós de altura h

Análise do build-max-heap

- Limite restrito
  - O tempo de execução de `max-heapify` varia com a altura da árvore, a altura da maioria dos nós é pequena
  - Um heap de n elementos tem altura $\lfloor \lg n \rfloor$ e no máximo $\lceil n/2^{h+1} \rceil$ nós de altura h
  - Logo, podemos expressar o tempo de execução do `build-max-heap` como

$$
\begin{aligned}
T(n) &= \sum_{h=0}^{\lfloor \lg n \rfloor} \left\lceil \frac{n}{2^{h+1}} \right\rceil O(h) \\
&= O\left(\frac{n}{2} \sum_{h=0}^{\lfloor \lg n \rfloor} \frac{h}{2^h} \right) \\
&= O\left(n \sum_{h=0}^{\infty} h \left(\frac{1}{2} \right)^h \right)
\end{aligned}
$$

Análise do build-max-heap

- Limite restrito
  - Obtemos que

$$T(n) = O\left(n\sum_{h=0}^{\infty} h\left(\frac{1}{2}\right)^{h}\right)$$

Análise do build-max-heap

- Limite restrito
  - Obtemos que

$$T(n) = O\left(n \sum_{h=0}^{\infty} h \left(\frac{1}{2}\right)^h\right)$$

  - Usando a fórmula $\sum_{k=1}^{\infty} kx^k = \frac{x}{(1-x)^2}$ com $x = \frac{1}{2}$, obtemos

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

Análise do build-max-heap

- Limite restrito
  - Obtemos que

$$T(n) = O\left(n \sum_{h=0}^{\infty} h \left(\frac{1}{2}\right)^h\right)$$

  - Usando a fórmula $\sum_{k=1}^{\infty} kx^k = \frac{x}{(1-x)^2}$ com $x = \frac{1}{2}$, obtemos

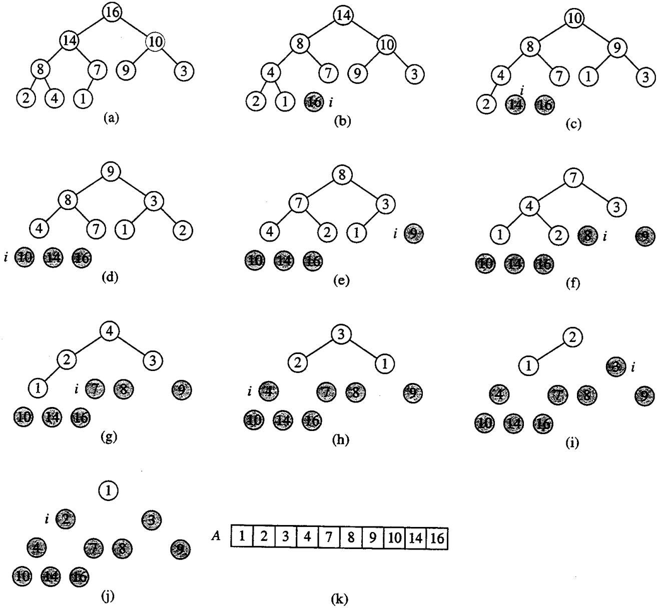
$$\sum_{h=0}^{\infty} h \left(\frac{1}{2}\right)^h = \frac{1/2}{(1-1/2)^2} = 2$$

  - Portanto, o tempo de execução de `build-max-heap` é $T(n) = O(n.2) = O(n)$

O algoritmo heapsort

- Construir um heap, usando a função `build-max-heap`
- Trocar o elemento A[1] com A[n], e atualiza o tamanho do heap para $n - 1$
- Corrigir o heap com a função `max-heapify` e repetir o processo

Exemplo do funcionamento do heapsort



O algoritmo heapsort

```
heapsort(A)
build-max-heap(A)
for i = A.comprimento downto 2
  troca(A[1], A[i])
  A.tamanho-do-heap = A.tamanho-do-heap - 1
  max-heapify(A, 1)
```

Análise do heapsort

- A chamada a `build-max-heap` demora $O(n)$
- O procedimento `max-heapify` demora $O(\lg n)$ e é chamado $n - 1$
- Portanto, o tempo de execução do `heapsort` é $O(n \lg n)$

Exercícios

- Exercícios 6.1-1 a 6.1-7
- Exercícios 6.2-1 a 6.2-6
- Exercícios 6.3-1 a 6.3-3
- Exercícios 6.4-1 a 6.4-3

Referências

- Thomas H. Cormen et al. Introdução a Algoritmos. 2ª edição em português. Capítulo 6.