AULA 06 – HEAP

Prof. Daniel Kikuti

Universidade Estadual de Maringá

12 de agosto de 2014

Conteúdo

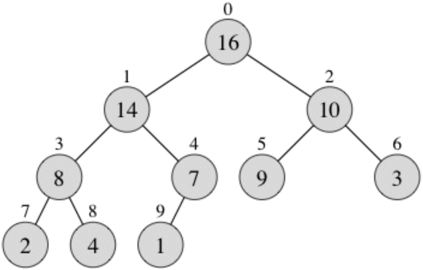
- Estrutura de dados **heap**:
  - Definição.
  - Manutenção.
  - Construção de um heap.
- O algoritmo heapsort.
- Fila de prioridades.
- Exercícios

A estrutura de dados **heap**

- A estrutura de dados **heap** (binário) é um array que pode ser visto como uma árvore binária praticamente completa.
- 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`.

A estrutura de dados **heap**

Exemplo



0	1	2	3	4	5	6	7	8	9
16	14	10	8	7	9	3	2	4	1

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, para todo nó i diferente da raiz `A[parent(i)]` $\geq$ `A[i]`.
  - Em um heap mínimo, para todo nó i diferente da raiz `A[parent(i)]` $\leq$ `A[i]`.
- a **altura** de um nó é o número de arestas no caminho descendente simples mais longo deste nó até uma folha.
- a **altura do heap** é a altura de sua raiz $= \Theta(\lg n)$.

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.

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)]`, 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 $2n/3$ – 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)]`, 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 $2n/3$ – 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)]`, 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 $2n/3$ – 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`

No quadro

Considere o vetor: $A = 3{,}5{,}4{,}2{,}1$.

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)
```

Correção do `build-max-heap`

- Para demonstrar a correção do algoritmo, vamos mostrar que `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 raiz de um heap máximo.

Correção do `build-max-heap`

- Para demonstrar a correção do algoritmo, vamos mostrar que `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 raiz de um heap máximo.
  - **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.

Correção do `build-max-heap`

- Para demonstrar a correção do algoritmo, vamos mostrar que `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 raiz de um heap máximo.

  - **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.

Correção do `build-max-heap`

- Para demonstrar a correção do algoritmo, vamos mostrar que `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 raiz de um heap máximo.

  - **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 raízes de um heap máximo, particularmente o nó 1 é uma raiz.

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

O algoritmo

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

Exemplo

Considere o heap: A= 5,3,4,2,1.

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 $O(n \lg n)$.

Implementação do heap como fila de prioridades

Fila de prioridades

- Mantém um conjunto dinâmico S de elementos.
- Cada elemento do conjunto possui uma chave (valor associado).
- Operações dinâmicas suportadas:
  - `insert(S,x)`: insere o elemento `x` no conjunto `S`.
  - `maximum(S)`: devolve o emento de `S` com a maior chave.
  - `extract-max(S)`: remove e devolve o emento de `S` com a maior chave.
  - `increase-key(S,x,k)`: incrementa o valor da chave do elemento `x` para k. Assume-se que $k \geq x$.

Implementação do heap como fila de prioridades

Encontrar o máximo

- Fácil: devolve a raiz.
- Custo: $\Theta(1)$.

Implementação do heap como fila de prioridades

Encontrar o máximo

- Fácil: devolve a raiz.
- Custo: $\Theta(1)$.

Extrair o máximo

Dado um vetor A:

- Certifique-se de que o heap não está vazio.
- Faça uma cópia da raiz.
- Faça do último nó na árvore a raiz.
- Remonte o heap, com um nó a menos.
- Devolva a cópia do elemento máximo.

Exercício: Descreva em pseudocódigo a ideia acima e analise o tempo de execução?

Implementação do heap como fila de prioridades

Aumentar o valor de uma chave

Dado um conjunto S, o elemento x e o novo valor k:

- Certifique-se que $k \geq x$.
- Atualize o valor de x.
- Percorra a árvore para cima comparando x com seu pai e trocando as chaves se necessário, até a chave ser menor que a chave de seu pai.

Inserir um elemento

Dada uma chave k a ser inserida em S:

- Insira o nó na última posição da árvore (com valor $-\infty$).
- Atualize o valor de $-\infty$ para k usando o procedimento acima.

Exercício: Descreva em pseudo-código e analise a complexidade.

Exercícios

Leitura do capítulo 6 do Livro do Cormen.
Pensar nos exercícios:

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