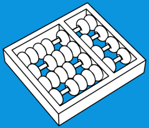
Ordenação e fila de prioridade

MO417 - Complexidade de Algoritmos I

04/24

9



UNICAMP

Santiago Valdés Ravelo
https://ic.unicamp.br/~santiago/
ravelo@unicamp.br

“Prioridade é uma função do contexto.”

Stephen R. Covey.

Ordenação por seleção

Selecionando o próximo menor

Vamos ordenar usando ideia diferente:

- Suponha que o sub-vetor $A[1 \ldots i-1]$ esteja ordenado.
- Também, suponha que $\max A[1 \ldots i-1] \leq \min A[i \ldots n]$.
- Substituímos a posição $A[i]$ pelo mínimo em $A[i \ldots n]$.

Antes de substituir:

1						i				n
20	**25**	**35**	**40**	**44**	**55**	70	80	99	**65**	85

Após substituir:

1						i				n
20	**25**	**35**	**40**	**44**	**55**	**65**	80	99	70	85

Pseudocódigo de Selection-Sort

Algoritmo: $\textsc{Selection-Sort}(A, n)$

para $i \leftarrow 1$ **até** $n - 1$
    $min \leftarrow i$
    para $j \leftarrow i + 1$ **até** n
        se $A[j] < A[min]$
            $min \leftarrow j$
    $A[i] \leftrightarrow A[min]$

Teorema (Invariante)

Ao início de cada iteração:

1. $A[1 \ldots i - 1]$ *está ordenado.*
2. $A[1 \ldots i - 1] \leq A[i \ldots n]$.

Complexidade de Selection-Sort

Selection-Sort(A, n)	Tempo
para $i \leftarrow 1$ **até** $n-1$	$\Theta(n)$
$min \leftarrow i$	$\Theta(n)$
para $j \leftarrow i+1$ **até** n	$\Theta(n^2)$
se $A[j] < A[min]$	$\Theta(n^2)$
$min \leftarrow j$	$\Theta(n^2)$
$A[i] \leftrightarrow A[min]$	$\Theta(n)$

Consumo de tempo no pior caso? $\Theta(n^2)$.

E no melhor caso?

Uma versão alternativa

Podemos reescrever esse algoritmo:

- Ordenamos a partir do final.
- Selecionamos o **MAIOR** remanescente.
- Refatoramos com uma sub-rotina MAXIMUM.

Revendo a complexidade

Algoritmo: $\textsc{Selection-Sort}(A, n)$

```
para i ← n até 2
    max ← MAXIMUM(A, i)
    A[i] ↔ A[max]
```

- Suponha que $\textsc{Maximum}(A, i)$ leva tempo $O(t(i))$.
- Então o tempo total é:

$$T(n) = \sum_{i=2}^{n} O(t(i)) \leq \sum_{i=2}^{n} O(t(n)) = O(n \cdot t(n))$$

- Vamos tentar otimizar a rotina $\textsc{Maximum}(A, i)$.

Heapsort

O algoritmo HEAP-SORT

Vamos estudar a **FILA DE PRIORIDADE**:

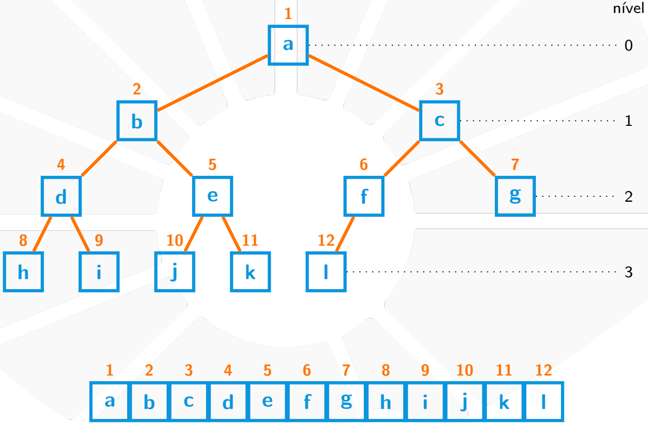
- É uma estrutura de dados também chamada de max-heap.
- Implementa MAXIMUM com tempo $O(\log n)$.
- Usando um heap, podemos ordenar em $O(n \log n)$.
- Esse algoritmo de ordenação é chamado de **heapsort**.

Representando um heap

Um heap é uma **ÁRVORE BINÁRIA** armazenada em vetor $A[1 \dots n]$:

- Filhos de um nó i:
  - O filho esquerdo é $2i$.
  - O filho direito é $2i + 1$.
- Pais:
  - O pai de um nó i é $\left\lfloor \frac{i}{2} \right\rfloor$.
  - O nó 1 não tem pai.
- Folhas:
  - Um nó i é folha se não tiver filhos, i.e., se $2i > n$.
  - As folhas são $\lfloor n/2 \rfloor + 1, \dots, n-1, n$.

Exemplo de um heap



Árvore completa

Um heap é uma árvore **COMPLETA**:

- Cada nível $\ell = 0, 1, 2, \ldots$ tem 2^ℓ nós (a não ser o último).
- Se i está no nível ℓ, o número nós nos níveis anteriores é:

$$2^0 + 2^1 + \cdots + 2^{\ell-1} = 2^\ell - 1$$

.

- Então os nós do nível ℓ são índices:

$$2^\ell, 2^\ell + 1, 2^\ell + 2, \ldots, 2^{\ell+1} - 1$$

.

Árvore completa

Teorema (Nível de um nó)

O nível de um nó de índice i é $\lfloor \log_2 i \rfloor$.

- Se i está no nível ℓ, então:

$$\begin{array}{rcccll} 2^\ell & \leq & i & < & 2^{\ell+1} & \Rightarrow \\ \log_2 2^\ell & \leq & \log_2 i & < & \log_2 2^{\ell+1} & \Rightarrow \\ \ell & \leq & \log_2 i & < & \ell + 1 & \end{array}$$

- Assim, $\ell = \lfloor \log_2 i \rfloor$.

Portanto, a altura da árvore é $\lfloor \log_2 n \rfloor$.

Max-heaps

Definição

Um heap $A[1 \dots n]$ *é chamado de* ***MAX-HEAP*** *se cada nó tiver valor maior que seus filhos, i.e., para cada* i,

- $A[i] \geq A[2i]$ *e* $A[i] \geq A[2i+1]$.

- Essa restrição é a **PROPRIEDADE DE MAX-HEAP**.
- O valor da raiz é um máximo do heap.
- Cada subárvore também é um max-heap.

Consertando um max-heap

- Suponha que as subárvores $2i$ e $2i+1$ são max-heaps.
- Como transformar a subárvore i em um max-heap?

Algoritmo: $\textsc{Max-Heapify}(A, n, i)$

$e \leftarrow 2i$
$d \leftarrow 2i + 1$
maior $\leftarrow i$
se $e \leq n$ **e** $A[e] > A[\text{maior}]$
  maior $\leftarrow e$
se $d \leq n$ **e** $A[d] > A[\text{maior}]$
  maior $\leftarrow d$
se maior $\neq i$
  $A[i] \leftrightarrow A[\text{maior}]$
  $\textsc{Max-Heapify}(A, n, \text{maior})$

Correção de Max-Heapify

Lema

Max-Heapify *transforma a subárvore i em max-heap.*

Ideia para demonstração: indução na altura h do nó i.

- Se $h = 0$, então i é folha e o algoritmo está correto.
- Considere um nó i com altura $h > 0$.
- Suponha que o algoritmo funciona para árvores menores.
  - Antes da linha 8, temos $A[maior] \geq A[i], A[2i], A[2i+1]$.
  - Após a linha 9, temos $A[i] \geq A[2i], A[2i+1]$.
  - Segue que $A[i]$ é máximo no vetor.
  - Pela hipótese de indução, Max-Heapify transforma a subárvore com raiz maior em max-heap.
  - Segue que i é max-heap.

Complexidade de Max-Heapify

Max-Heapify(A, n, i)	Tempo
$e \leftarrow 2i$	$\Theta(1)$
$d \leftarrow 2i + 1$	$\Theta(1)$
maior $\leftarrow i$	$\Theta(1)$
se $e \leq n$ **e** $A[e] > A[i]$	$\Theta(1)$
maior $\leftarrow e$	$O(1)$
se $d \leq n$ **e** $A[d] > A[\text{maior}]$	$\Theta(1)$
maior $\leftarrow d$	$O(1)$
se maior $\neq i$	$\Theta(1)$
$A[i] \leftrightarrow A[\text{maior}]$	$O(1)$
Max-Heapify(A, n, maior)	$T(h-1)$

O tempo de execução é $T(h) = T(h-1) + \Theta(1) = O(h)$.

Construindo um max-heap

Podemos consertar um vetor inteiro:

- Recebemos um vetor $A[1 \dots n]$ desorganizado.
- Cada folha já é um heap.
- Consertamos o penúltimo nível.
- Depois o antepenúltimo.
- Assim por diante.

Algoritmo: $\textsc{Build-Max-Heap}(A, n)$

---

para $i \leftarrow \lfloor n/2 \rfloor$ **até** 1

  $\textsc{Max-Heapify}(A, n, i)$

Análise de BUILD-MAX-HEAP

Algoritmo: BUILD-MAX-HEAP(A, n)

para $i \leftarrow \lfloor n/2 \rfloor$ **até** 1
  MAX-HEAPIFY(A, n, i)

Teorema (Invariante)

No início de cada iteração, $i+1, \ldots, n$ são raízes de max-heaps.

Complexidade

- Uma análise rápida leva a $T(n) = n \cdot O(\log n) = O(n \log n)$.
- Na verdade mostraremos que $T(n)$ é **LINEAR**!

Complexidade de Build-Max-Heap

Análise mais cuidadosa:

- Seja k a altura da árvore inteira:
  - Temos 1 nó de altura k.
  - Temos 2 nós de altura $k - 1$.
  - Temos 4 nós de altura $k - 2$.
  - Assim por diante.
- Uma chamada de Max-Heapify leva tempo $O(h)$.
- Vamos somar os tempos de todas as chamadas.

Complexidade de Build-Max-Heap (cont)

Seja $k = \lfloor \log_2 n \rfloor$ a altura da árvore inteira, então:

$$T(n) \leq \sum_{h=1}^{k} 2^{k-h} \cdot O(h) = O(2^k) \sum_{h=1}^{k} \frac{h}{2^h}.$$

Note que:

$$\begin{array}{rlr} \displaystyle\sum_{h=1}^{k} \frac{h}{2^h} = & \left.\dfrac{1}{2} + \dfrac{1}{2^2} + \dfrac{1}{2^3} + \cdots + \dfrac{1}{2^k}\right\} & 1 - \dfrac{1}{2^k} < 1 \\ & \left.+ \dfrac{1}{2^2} + \dfrac{1}{2^3} + \cdots + \dfrac{1}{2^k}\right\} & \dfrac{1}{2} - \dfrac{1}{2^k} < \dfrac{1}{2} \\ & \left.+ \dfrac{1}{2^3} + \cdots + \dfrac{1}{2^k}\right\} & \dfrac{1}{4} - \dfrac{1}{2^k} < \dfrac{1}{4} \end{array}$$

Ou seja,

$$T(n) \leq O(2^k) \sum_{h=0}^{k-1} \frac{1}{2^h} \leq O(2^k) \cdot \sum_{h=0}^{\infty} \frac{1}{2^h} = O(2^k) \cdot 2 = O(n).$$

O algoritmo Heap-Sort

Algoritmo: Heap-Sort(A, n)

Build-Max-Heap(A, n)
para $m \leftarrow n$ **até** 2
  $A[1] \leftrightarrow A[m]$
  Max-Heapify(A, $m-1$, 1)

Análise de HEAP-SORT

Algoritmo: HEAP-SORT(A, n)

```
BUILD-MAX-HEAP(A, n)
para m ← n até 2
    A[1] ↔ A[m]
    MAX-HEAPIFY(A, m − 1, 1)
```

Lema (Invariantes)

No início de cada iteração vale:

1. $A[1 \ldots m]$ *é um max-heap.*
2. $A[m+1 \ldots n]$ *é crescente.*
3. $A[1 \ldots m] \leq A[m+1]$.

Complexidade de HEAP-SORT

HEAP-SORT(A, n)		Tempo
1	BUILD-MAX-HEAP(A, n)	$\Theta(n)$
2	**para** $m \leftarrow n$ **até** 2	$\Theta(n)$
3	$A[1] \leftrightarrow A[m]$	$\Theta(n)$
4	MAX-HEAPIFY$(A, m-1, 1)$	$n \cdot O(\log n)$

Consumo de tempo no pior caso? $O(n \log n)$.

Fila de prioridades

Filas de prioridades

Definição

Uma ***FILA DE PRIORIDADES*** *é um tipo abstrato de dados que consiste de uma coleção S de itens com prioridades associadas e permite as operações:*

- $\textsc{Maximum}(S)$ *devolve um elemento de maior prioridade.*
- $\textsc{Extract-Max}(S)$ *remove um elemento de maior prioridade.*
- $\textsc{Increase-Key}(S, x, p)$ ***aumenta*** *a prioridade de x para p.*
- $\textsc{Insert}(S, x, p)$ *insere um elemento x com prioridade p.*

Implementação com max-heap

Algoritmo: $\textsc{Heap-Max}(A, n)$

devolva $A[1]$

Complexidade de tempo: $\Theta(1)$.

Algoritmo: $\textsc{Heap-Extract-Max}(A, n)$

$A[1] \leftarrow A[n]$

$n \leftarrow n - 1$

$\textsc{Max-Heapify}\,(A, n, 1)$

Complexidade de tempo: $O(\log n)$.

Implementação com max-heap

Algoritmo: $\textsc{Heap-Increase-Key}(A, i, chave)$

$A[i] \leftarrow chave$
enquanto $i > 1$ **e** $A[\lfloor i/2 \rfloor] < A[i]$
    $A[i] \leftrightarrow A[\lfloor i/2 \rfloor]$
    $i \leftarrow \lfloor i/2 \rfloor$

Complexidade de tempo: $O(\log n)$.

Algoritmo: $\textsc{Max-Heap-Insert}(A, n, chave)$

$n \leftarrow n + 1$
$A[n] \leftarrow -\infty$
$\textsc{Heap-Increase-Key}(A, n, chave)$

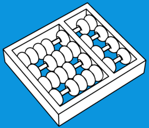
Complexidade de tempo: $O(\log n)$.

Ordenação e fila de prioridade

MO417 - Complexidade de Algoritmos I

04/24

9



UNICAMP

Santiago Valdés Ravelo
https://ic.unicamp.br/~santiago/
ravelo@unicamp.br