UFSC-CTC-INE

INE5384 - Estruturas de Dados

# Ordenação de Dados (III)

Prof. Ronaldo S. Mello

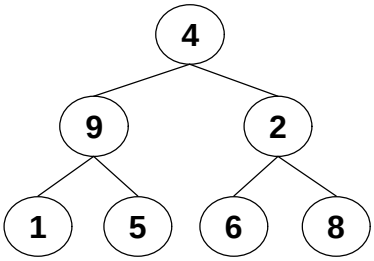
2002/2

## *HeapSort*

- *HeapSort* também é um método de seleção
  - ordena através de sucessivas seleções do elemento correto a ser posicionado em um segmento ordenado
- O *HeapSort* utiliza um *heap binário* para manter o próximo elemento a ser selecionado
  - *heap binário*: árvore binária mantida na forma de vetor
  - o *heap* é gerado e mantido no próprio vetor a ser ordenado (no segmento não-ordenado)

# *Heap* Binário - Exemplo

| 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|
| 4 | 9 | 2 | 1 | 5 | 6 | 8 |

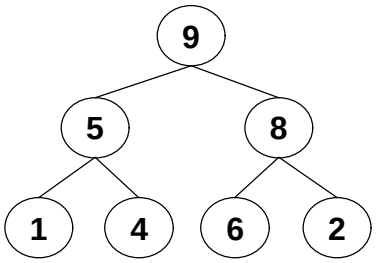



- raiz da árvore: primeira posição do vetor
- filhos de um nodo na posição *i*: posições *2i* e *2i + 1*
- pai de um nodo na posição *i*: posição $\lfloor i / 2 \rfloor$

# *Heap* Binário Máximo

- *Heap* Binário tal que um nodo pai tem valor maior ou igual ao valor dos nodos filhos

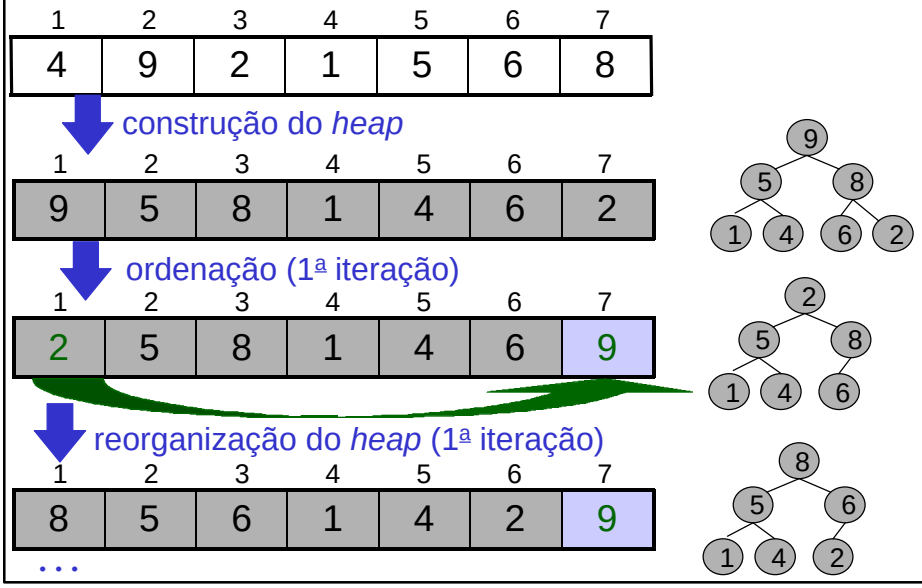
| 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|
| 9 | 5 | 8 | 1 | 4 | 6 | 2 |



# *HeapSort - Funcionamento*

1. Transformação do vetor em um *heap binário máximo* (Construção do *Heap*)
2. Ordenação
   - a cada iteração seleciona-se o maior elemento (na raiz do *heap*) e o adiciona no início de um segmento ordenado
   - após cada seleção de elemento, o *heap* deve ser reorganizado para continuar sendo um *heap binário máximo*

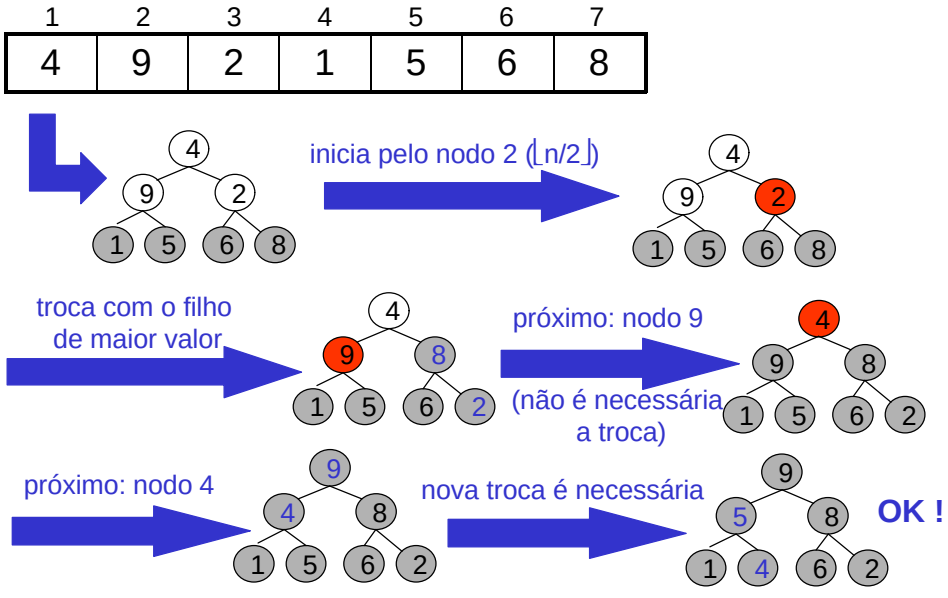
# *HeapSort - Exemplo*



# HeapSort – Construção do Heap

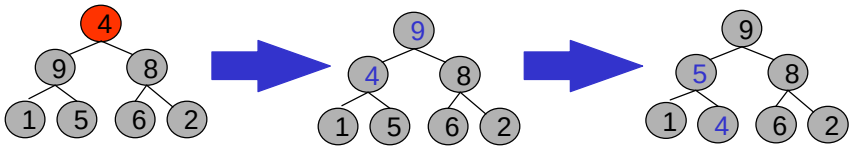
- Cria-se uma árvore binária com os elementos do vetor desordenado
- Transforma a árvore binária em um *heap* binário máximo
  - a transformação ocorre de forma *bottom-up*
    - *inicia-se a partir dos nodos folha e vai executando a transformação em direção a raiz*

# Construção do Heap - Exemplo



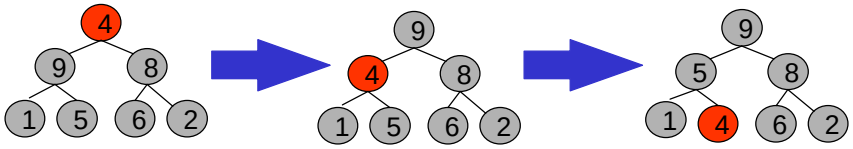
## HeapSort – Ajuste de Elementos

- Método auxiliar responsável pelo ajuste de um elemento no *heap*
  - *método ajustaElemento(posição, vetor.lenght)*
  - *realiza trocas no heap para posicionar corretamente um elemento*
- Exemplo: *ajustaElemento(1, 7)*




## HeapSort – Ajuste de Elementos

- Executa até que o elemento seja transferido para uma posição $i > \lfloor n/2 \rfloor$
  - *após a posição* $\lfloor n/2 \rfloor$*, o elemento já é um nodo folha*




- *Complexidade (pior caso):* ***O(log n)***

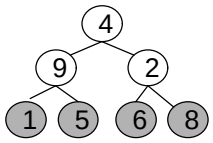
# HeapSort – Construção do Heap

- Executa o método *ajustaElemento* para os elementos do vetor que estão em posições entre [⌊n/2⌋, 1], na seguinte ordem: ⌊n/2⌋, ⌊n/2⌋ - 1, ⌊n/2⌋ - 2, ..., 1
- Qual a complexidade da Construção do *Heap*?
  - A princípio:

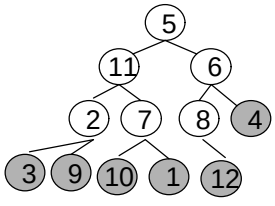
    ⌊n/2⌋ . complexidade(*ajustaElemento*) =
    ⌊n/2⌋ . log n = ***O(n log n)***

# HeapSort – Construção do Heap

- Na prática, a complexidade da Construção do *Heap* é ***O(n)***
  - número de comparações está sempre dentro do intervalo *(n, 2n)*




**n = 7**
máx. comparações para os nodos 2 e 9 = 2
máx. comparações para o nodo 4 = 4
**Total comparações = 2.2 + 4 = 8**

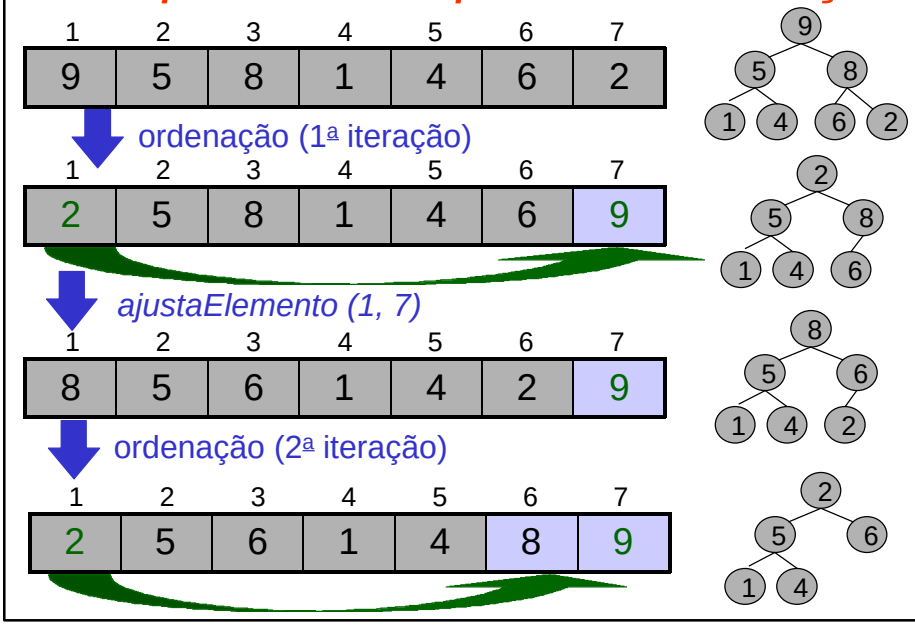



**n = 12**
máx. comparações para os nodos 2, 7 e 8 = 2
máx. comparações para os nodos 11 e 6 = 4
máx. comparações para o nodo 5 = 6
**Total comparações = 3.2 + 2.4 + 6 = 20**

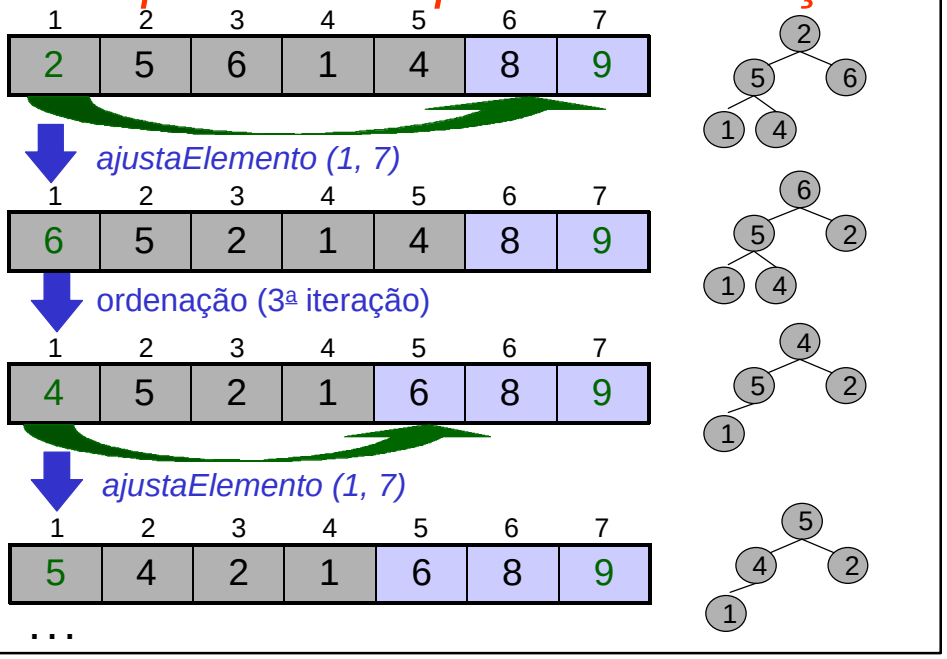
# HeapSort – Etapa de Ordenação

- A cada iteração seleciona o maior elemento do *heap* (sempre está na primeira posição) e o troca com o elemento no final do segmento não-ordenado
- Após a troca, o novo elemento raiz do *heap* deve ser ajustado (deve-se chamar *ajustaElemento* para o nodo raiz)
- O processo termina quando o *heap* tiver somente 1 elemento (vetor ordenado)

# HeapSort – Etapa de Ordenação



# HeapSort – Etapa de Ordenação




# HeapSort - Complexidade

- Etapas:
  1. Construção do *Heap* → ***O(n)***
  2. Ordenação
     - *Troca*(raiz, final do segmento não-ordenado) → ***O(1)***
     - *AjustaElemento*(raiz) → ***O(log n)***

  Executa n-1 vezes → (n-1). log n → ***O(n log n)***

- *Complexidade (em qualquer caso):*

  ***O(n log n)***

# *Algoritmos de Ordenação por Seleção*

| | *SelectionSort* | *HeapSort* |
|---|---|---|
| Pior caso | $O(n^2)$ | $O(n \log n)$ |
| Caso médio | $O(n^2)$ | $O(n \log n)$ |
| Melhor caso | $O(n^2)$ | $O(n \log n)$ |

# *HeapSort X QuickSort*

| | *QuickSort* | *HeapSort* |
|---|---|---|
| Pior caso | $O(n^2)$ | $O(n \log n)$ |
| Caso médio | $O(n \log n)$ | $O(n \log n)$ |
| Melhor caso | $O(n \log n)$ | $O(n \log n)$ |

- *QuickSort* é ruim no pior caso (ocorre raramente...)
- Na prática, *QuickSort* tem um desempenho melhor que o *HeapSort*, considerando que a sua média de iterações é menor (proporcional a *log n*)

# Exercícios

- Por quê a construção do *heap* inicial não poderia iniciar da raiz para as folhas? Mostre um contra-exemplo para provar que tal sugestão não funciona!
- Implementar os seguintes métodos para a classe *OrdenadorHeapSort*:
  - *ajustaElemento(posição inteiro, comprimento inteiro)*
  - *constróiHeap()* (definição do *heap* inicial)
  - *ordena()*