Algoritmos e Estrutura de Dados

Aula 5 – Algoritmos de Ordenamento: Heap Sort

Prof. Tiago A. E. Ferreira

Algoritmos de Ordenamento

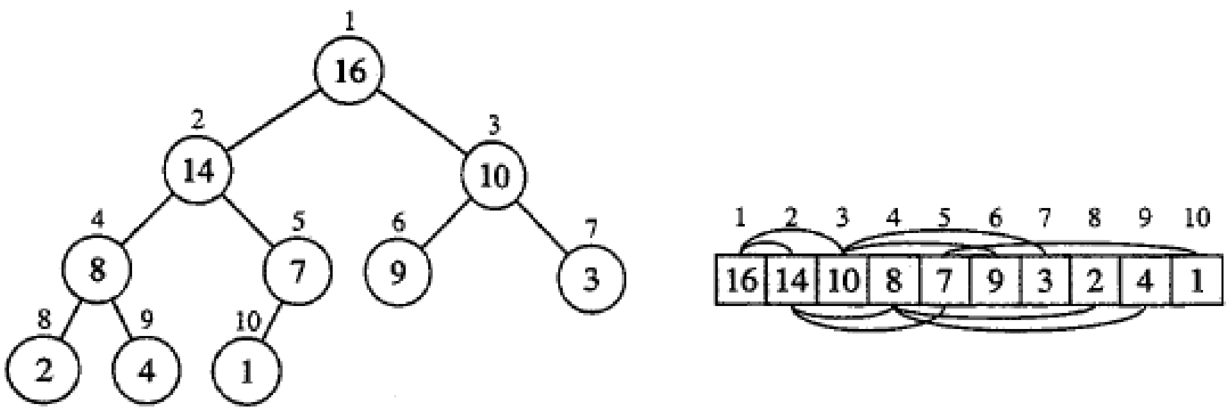
- Os algoritmos de ordenamento são aplicados para a organização de listas de elementos quaisquer em ordem crescente ou decrescente
- Existem vários algoritmos para este fim:
  - Insert Sort
  - Merge Sort
  - Heap Sort
  - Quick Sort
  - Etc...

Algoritmo Heap Sort

- O algoritmo ***Heap Sort*** utiliza o melhor dos algoritmos de ordenamento já estudados até agora:
  - ***Inserction Sort***:
    - Utiliza um ordenamento local
  - ***Merge Sort***:
    - Apenas um número constante de elementos é mantido fora do arranjo de entrada para ser ordenado (dividir para conquistar)

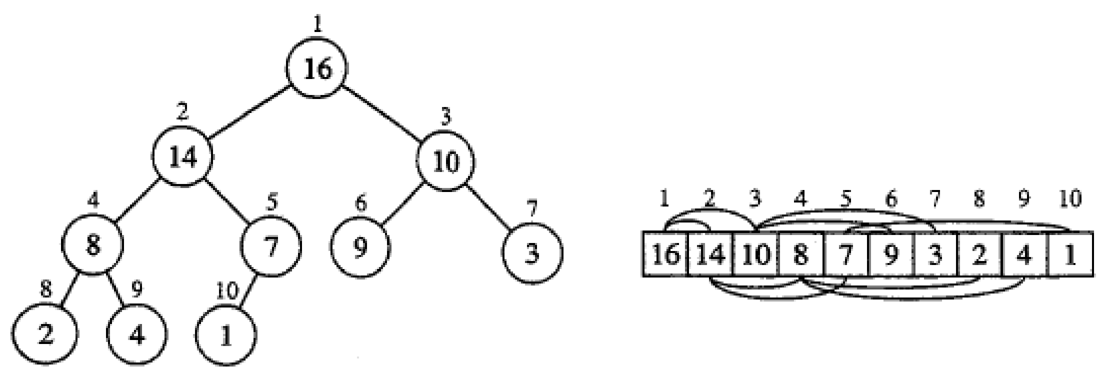
Algoritmo Heap Sort

- O algoritmo ***Heap Sort*** utiliza uma estrutura de dados chamada ***Heap***.
  - Um ***Heap*** é um objeto arranjo que pode ser visto como uma árvore binária incompleta
    - Cada Nó corresponde a um elemento do arranjo



Estrutura Heap

- O Arranjo representa o ***Heap***
  - Este tem dois atributos:
    - **Comprimento[A]**: número de elementos do arranjo A
    - **Tamanho-do-Heap[A]**: número de elementos do Heap arranjado dentro do arranjo A.
  - O arranjo **A[1..comprimento[A]]** pode ter vários elementos
    - Porém, apenas os elementos até o índice **tamanho-do-heap[A]** (≤ **compriemnto[A]**) farão parte do ***heap***



Estrutura Heap

- O primeiro elemento do arranjo é o nó raiz
- Para um nó **i,** é possível verificar:
  - O nó Pai: **PARENT**(i)
    return $\lfloor i/2 \rfloor$
  - O nó Filho Esquerdo: **LEFT**(i)
    return $2i$
  - O nó Filho Direito: **RIGHT**(i)
    return $2i + 1$
    - Estas três operações, na grande maioria dos processadores podem ser executadas em uma única instrução (custo em **tempo constante**)

Estrutura Heap

- Existem dois tipos de heap binários:
  - **Heap máximo**
    - **Propriedade**: para todo elemento diferente da raiz tem-se:
      - **A[Parent[A]] ≥ A[i]**
      - Ou seja, o valor de um nó é no **máximo** o valor do seu pai (o maior elemento está na raiz)
  - **Heap mínimo**
    - **Propridade**: para todo elemento diferente da raiz tem-se:
      - **A[Parent[A]] ≤ A[i]**
      - Ou seja, o valor de um nó é no **mínimo** o valor do seu pai (o menor elemento está na raiz)

Estrutura Heap

- Visualizando um Heap como uma árvore, é possível definir:
  - **Altura de um nó**: número de arestas no caminho descendente simples mais longo do nó até uma folha
  - **Altura do heap**: é a altura do nó raiz
  - **Altura máxima**: para um heap binário de **n elementos**, a altura máxima será **Θ(lgn)**
  - **Custo em tempo:** de forma geral, as operações com o heap terão custo em tempo proporcionais a sua altura, logo **T(n) = O(lgn)**

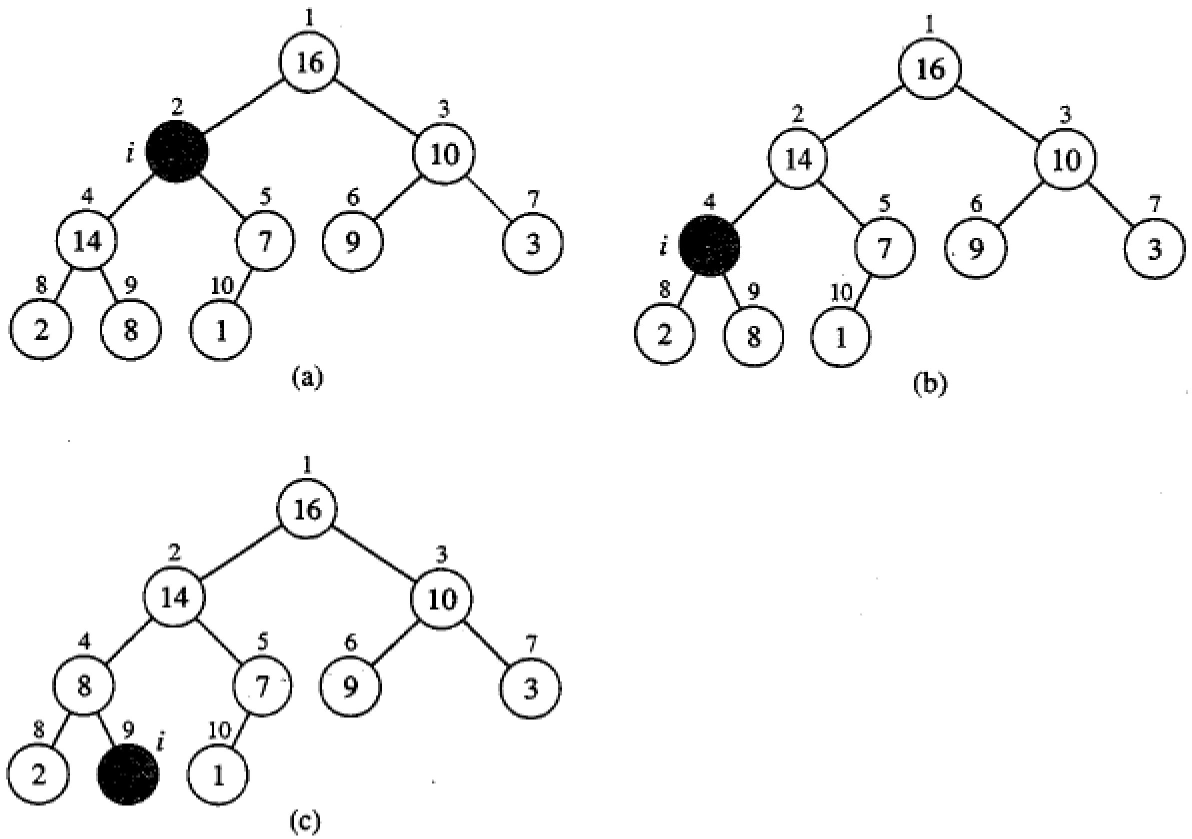
Função MAX-HEAPIFY

- Executado em **O(lgn)**, é a função chave para a manutenção da propriedade do heap máximo
  - **Recebe:** A – Arranjo; i – índice.
  - **Supõe:** árvores com raizes **left[i]** e **right[i]** são heaps máximos, mas **A[i]** não necessariamente.
  - **Objetivo:** “Flutuar” **A[i]** para grantir um heap máximo

```
MAX-HEAPIFY(A, i)
l ← LEFT(i)
r ← RIGHT(i)
if l ≤ tamanho-do-heap[A] e A[l] > A[i]
   then maior ← l
   else maior ← i
if r ≤ tamanho-do-heap[A] e A[r] > A[maior]
   then maior ← r
if maior ≠ i
   then trocar A[i] ↔ A[maior]
        MAX-HEAPIFY(A, maior)
```

Função MAX-HEAPIFY

- Suponha o nó A[2]=4. Se chamarmos a função MAX-HEAPIFY(A,2)



Função MAX-HEAPIFY

- No pior caso, o custo em tempo irá seguir a recorrência:
  - **$T(n) = T(2n/3) + \Theta(1)$**
  - Que segundo o teorema mestre, **$T(n) = O(\lg n)$**

Construindo um Heap Máximo

- É possível utilizar a função MAX-HEAPIFY para construirmos um heap máximo
  - Dado um arranjo, **A[1..n]** é possível mostrar que **A[(⌊n/2⌋+1) .. n]** serão todos folhas (exercício!)
    - Assim, todos estes elementos serão heaps de um elemento que poderão ser utilizados para começarmos a construir o heap máximo
- Dada a função BUILD-MAX-HEAP:

```
BUILD-MAX-HEAP(A)
tamanho-do-heap[A] ← comprimento[A]
for i ← ⌊comprimento[A]/2⌋ downto 1
    do MAX-HEAPIFY(A, i)
```

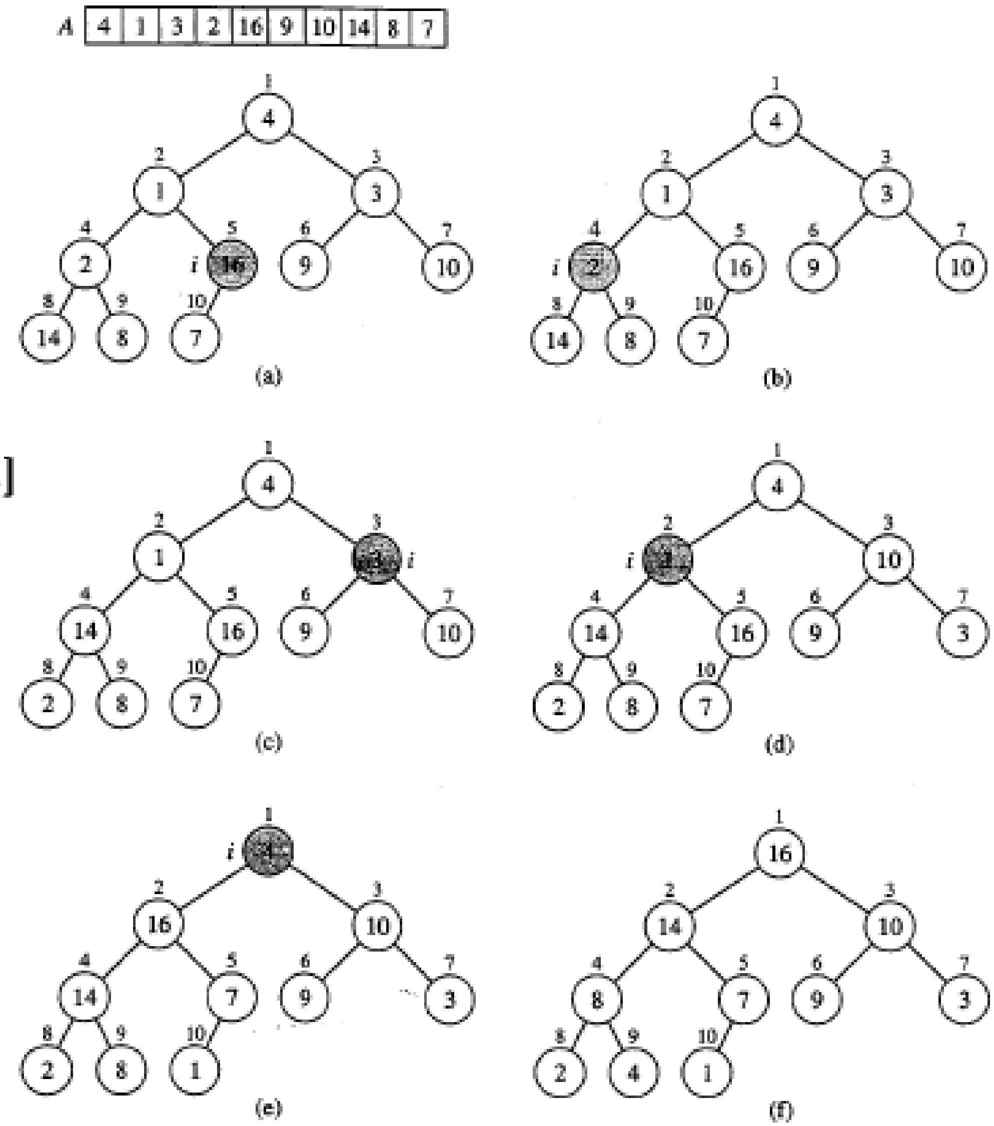
Exemplo: Construindo um Heap Máximo

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*)



Custos

- Um heap com **n** elemento terá **altura** $\lfloor \mathbf{lgn} \rfloor$
  - Porém, é possível mostrar (exercício) que para uma altura **b**, existirá no máximo $\lceil \mathbf{n/2^{b+1}} \rceil$ elementos
  - A função MAX-HEAPIFY terá um custo **O(b)** para uma altura **b**
  - Assim, o custo para a função BUILD-MAX-HEAP será:

$$\sum_{b=0}^{\lfloor \lg n \rfloor} \left\lceil \frac{n}{2^{b+1}} \right\rceil O(b) = O\left(n \sum_{b=0}^{\lfloor \lg n \rfloor} \frac{b}{2^b} \right)$$

- **Mas,** $\sum_{b=0}^{\infty} \frac{b}{2^b} = \frac{1/2}{(1-1/2)^2} = 2$ **, logo,** $O\left(n \sum_{b=0}^{\lfloor \lg n \rfloor} \frac{b}{2^b} \right) = O\left(n \sum_{b=0}^{\infty} \frac{b}{2^b} \right) = O(n)$
- Assim é possível construir um heap máximo em **tempo linear**!

Algoritmo HeapSort

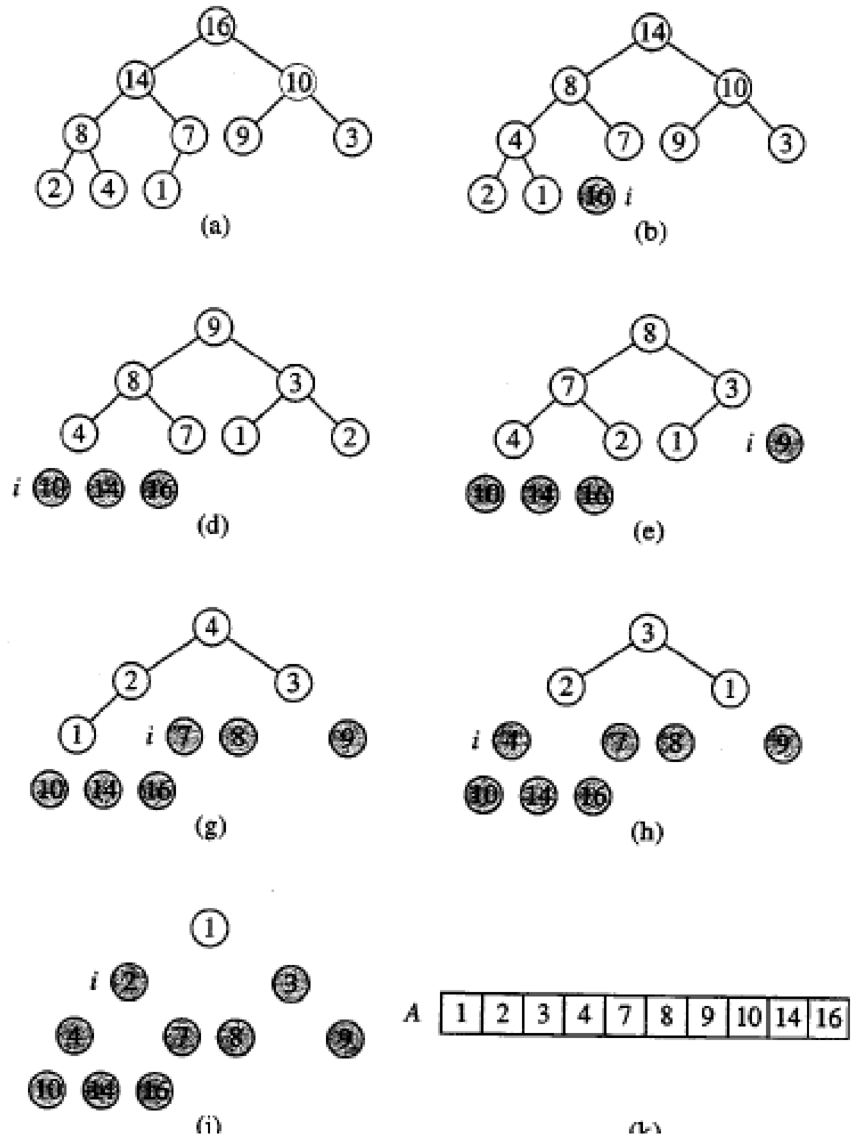
- Dado um arranjo A[1..n] é crido um heap máximo.
  - Como o maior elemento está da raiz (A[1]), troca-se este elemento como o último elemento do arranjo (A[n]) , diminuindo-se o tamanho do heap de 1.
  - O com o novo arranjo A[1..(n-1)] refaz-se o mesmo procedimento.

```
HEAPSORT(A)
BUILD-MAX-HEAP(A)
for i ← comprimento[A] downto 2
    do trocar A[1] ↔ A[i]
        tamanho-do-heap[A] ← tamanho-do-heap[A] – 1
        MAX-HEAPIFY(A, 1)
```

Custo do HeapSort

- Cada chamada da função BUILD-MAX-HEAP tem custo **O(n)**.
  - Esta função é chamada **1** vezes!
- Cada chamada da função MAX-HEAPIFY tem custo **O(lgn)**
  - Esta função é chamda **(n-1)** vezes!
- O custo total do algoritmo HeapSort será:
  - **O(nlgn)**

Exemplo



Exercício:

- Implemente o HeapSort em Python
- Resolva as questões do livro Texto
  - 6.1-2
  - 6.1-7
  - 6.3-3