ENGENHARIA INFORMÁTICA E DE COMPUTADORES

Algoritmos e Estruturas de Dados

(parte 6 – Amontoados Binários)

2º Semestre 2022/2023

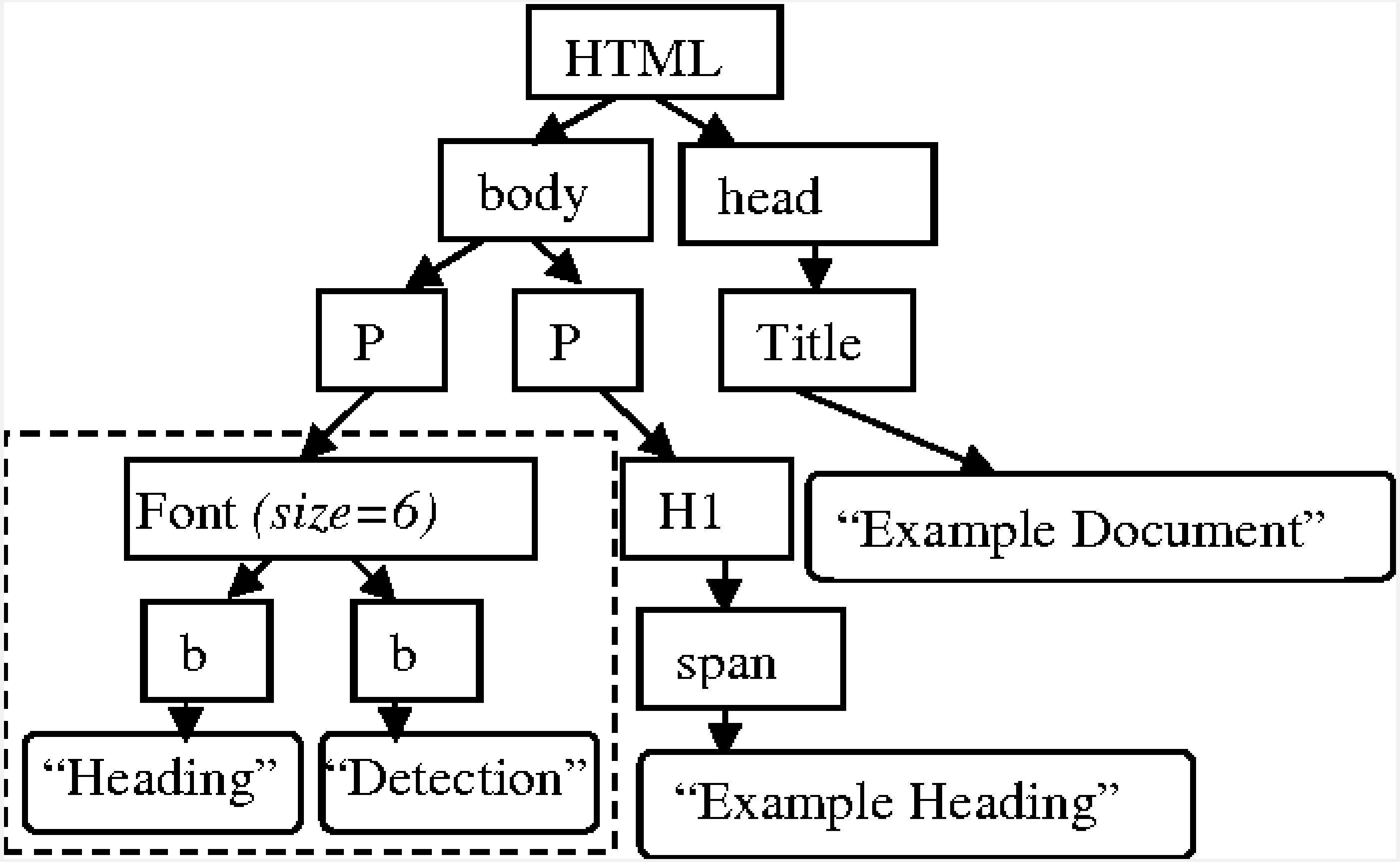
Instituto Superior de Engenharia de Lisboa

Paula Graça

- Introdução
  - Existem vários tipos de estruturas hierárquicas

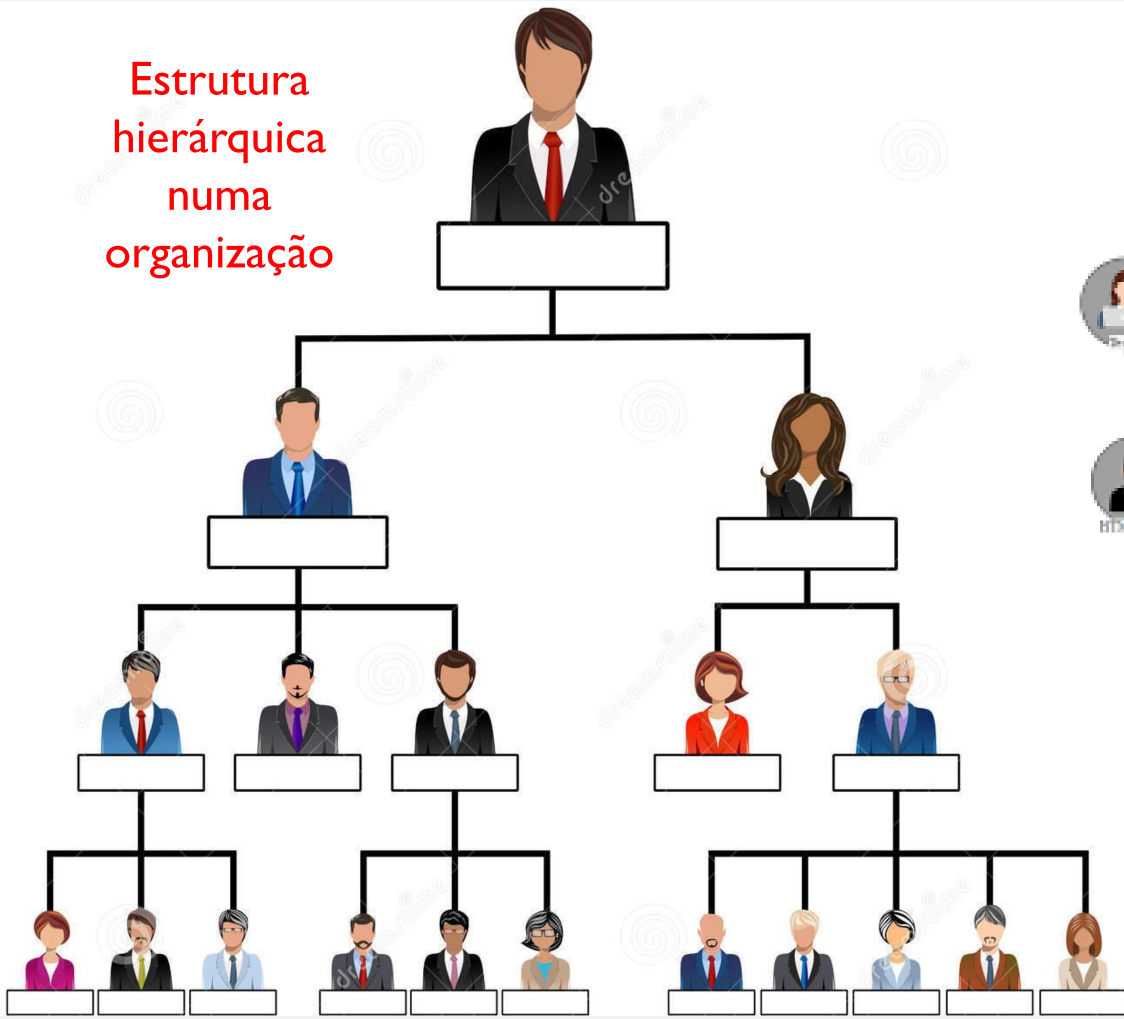
    - Árvores Livres
    - Árvores com Raiz
    - Árvores Ordenadas N-árias
    - Árvores Binárias
    - Árvores Binárias de Pesquisa
    - Amontoados Binários (*Heaps*)

- Estrutura hierárquica de um documento Web

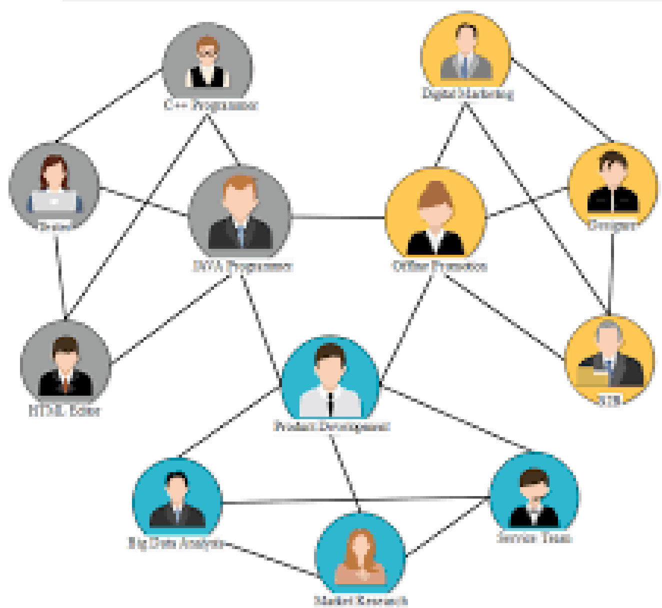


ÁRVORES COM RAIZ

Estrutura hierárquica numa organização

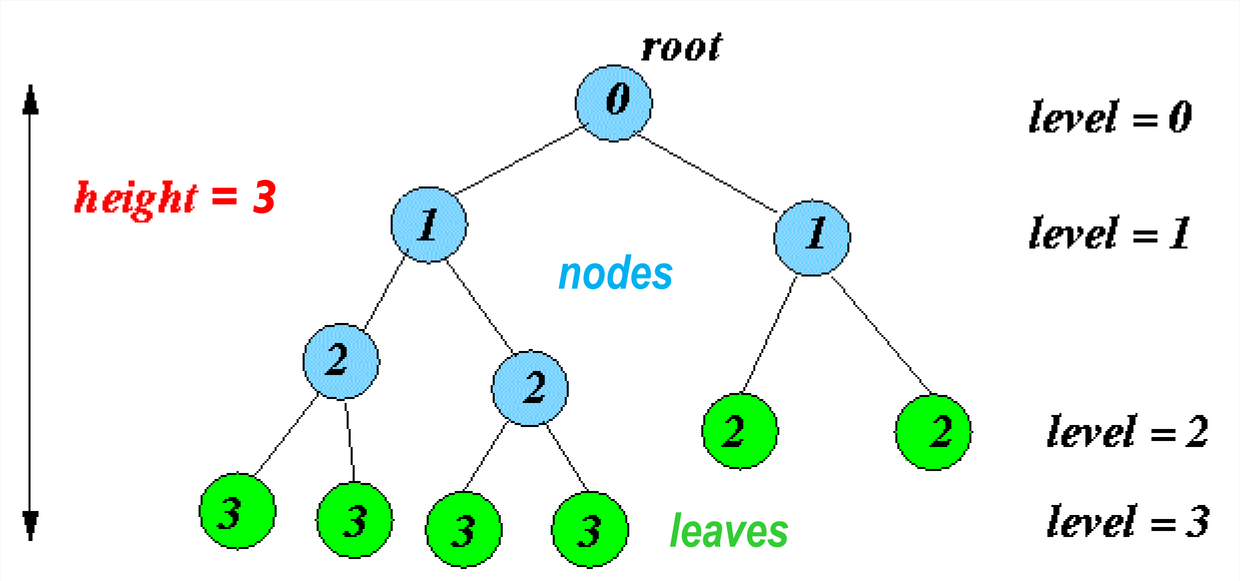


Estrutura não hierárquica



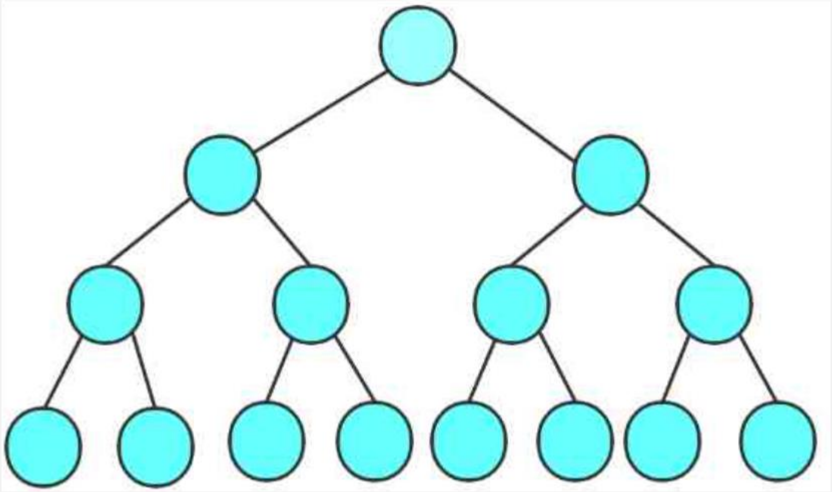
ÁRVORES BINÁRIAS

- Uma árvore binária tem vários níveis (*levels*) de nós (*nodes*) desde a raiz (*root*) até às folhas (*leaves*)
- A raiz está no nível 0
- As folhas são nós sem descendentes
- A altura (*heigh*) de uma árvore é o maior dos níveis das folhas (uma árvore só com a raiz tem altura 0)



ÁRVORES BINÁRIAS

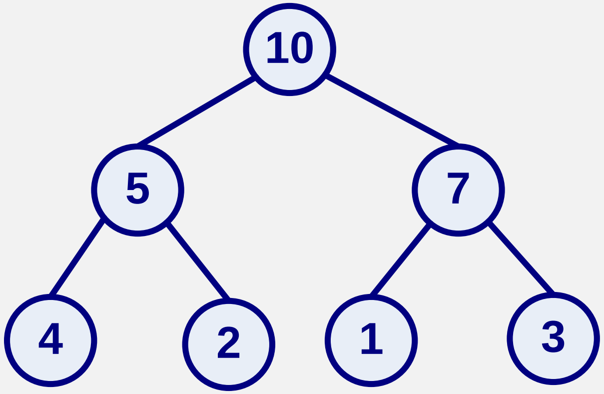
- Definições
  - Uma árvore binária diz-se completa se estiver totalmente preenchida, ou seja,
    - Se todas as folhas estiverem no mesmo nível
    - Se todos os nós, exceto as folhas, tiverem todos os filhos



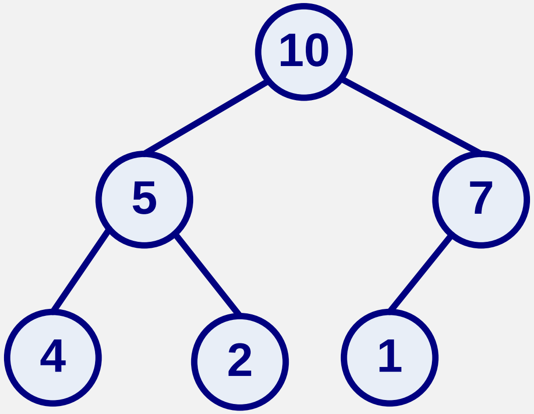
HEAPS

- Um *heap* (amontoado binário)
  - E uma estrutura de dados parcialmente ordenada, representada numa árvore binária completa ou quase completa

Completa



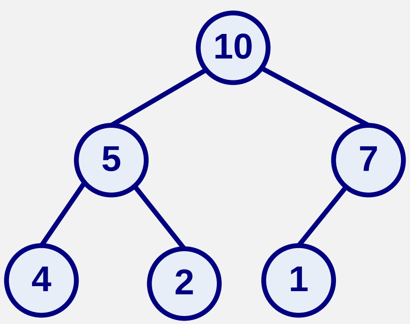
Quase completa



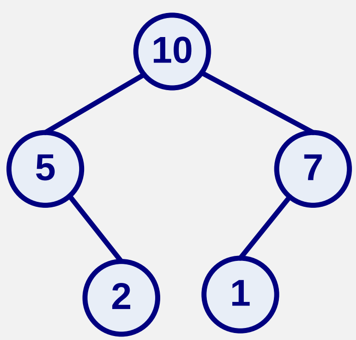
HEAP - REPRESENTAÇÃO

- Um *heap* pode ser representado através de uma árvore binária, em que cada nó contém um valor (chave) e obedece às seguintes duas propriedades:

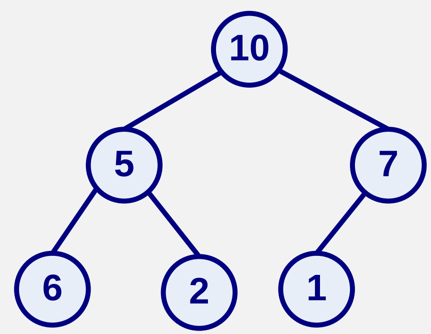
  1. ESTRUTURAL - A árvore está completa ou quase completa. No caso de ser quase completa, as folhas terão que estar preenchidas da esquerda para a direita
  2. DE ORDENAÇÃO – O valor de cada um dos nós é maior ou igual que o valor dos nós dos seus filhos



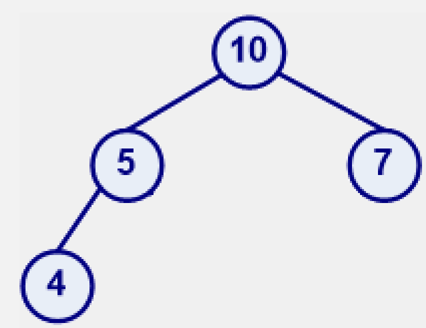
heap



não é *heap* (1)



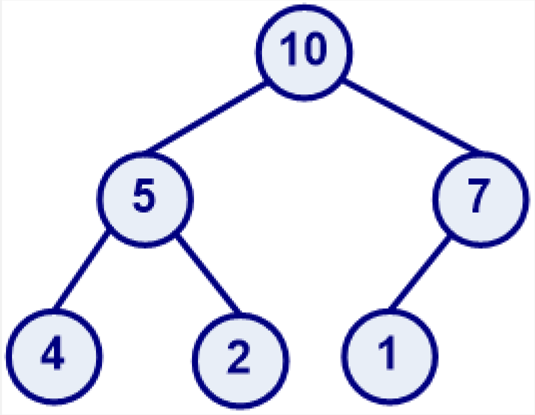
não é *heap* (2)

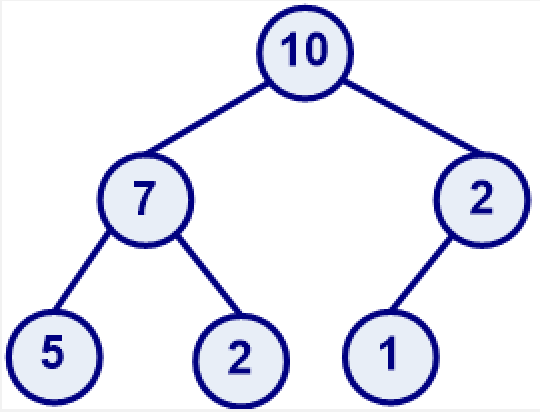


heap

HEAP - ORDENAÇÃO

- Os valores dos nós de um *heap* estão ordenados de cima para baixo (*top-down*), ou seja, a sequência de valores de qualquer caminho desde a raiz a uma folha é decrescente
  - Valores menores ou iguais, caso existam valores iguais
- Dentro do mesmo nível, da esquerda para a direita, não existe nenhuma ordenação definida dos valores





HEAP - DEFINIÇÃO

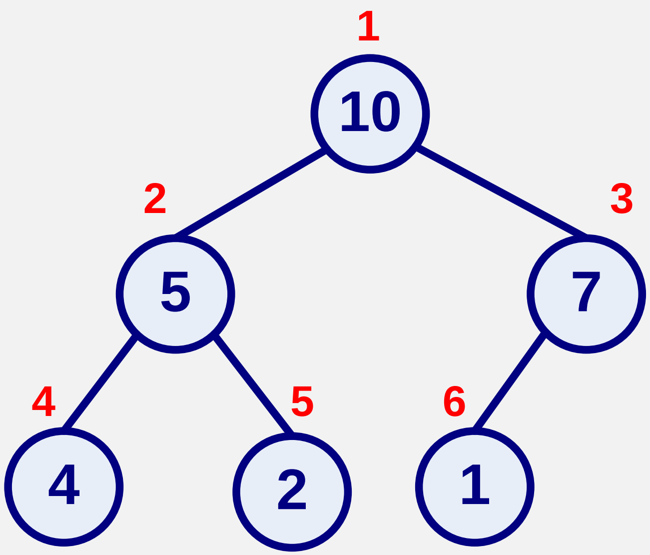
1. Um *heap* é uma árvore binária completa ou quase completa, com n nós cuja altura h é $\lfloor \lg n \rfloor$

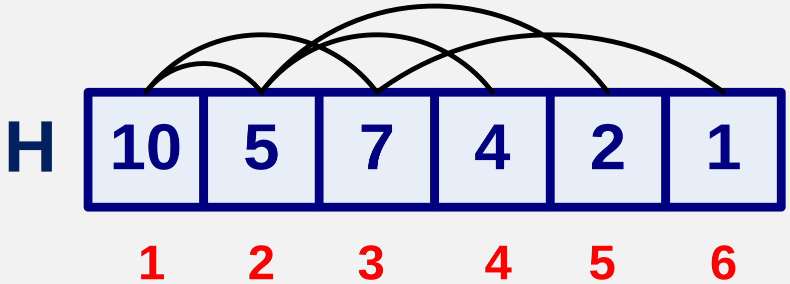
2. A raiz do *heap* contém sempre o maior elemento

3. Qualquer nó do *heap* incluindo todos os seus descendentes, é também um *heap*

4. Um *heap* pode ser implementado num *array* em que os elementos são organizados de cima para baixo e da esquerda para a direita

HEAP - ESTRUTURA

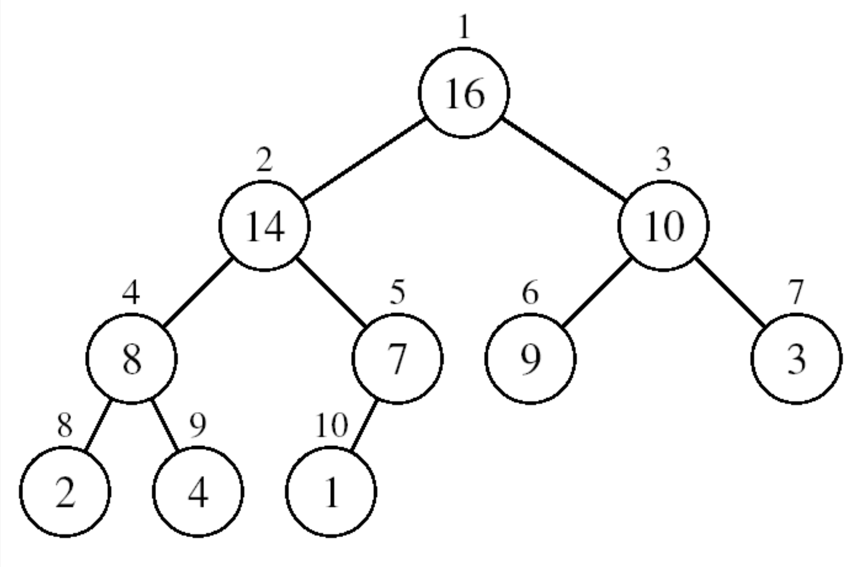


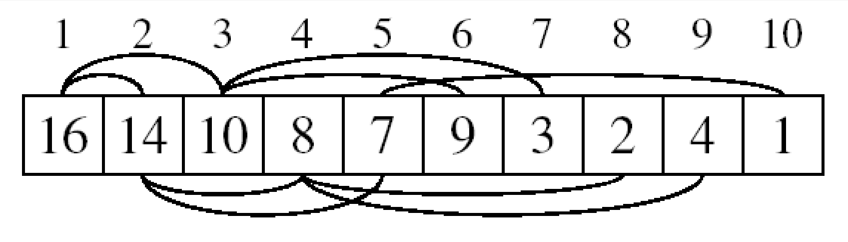


H = *array* de elementos

H.heap-size = nº de elementos do *array*

HEAP - EXEMPLO





n = 10

h = $\lfloor \lg 10 \rfloor = 3$

Altura nó[4] = 1

- A altura de um nó no *heap* é o número de arestas desde o nó até à folha

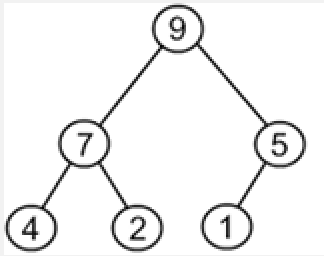
- A altura do *heap* é a altura da raiz

HEAP - ESTRUTURA

- Existem dois tipos de *heaps*
  - Max-heap e Min-heap

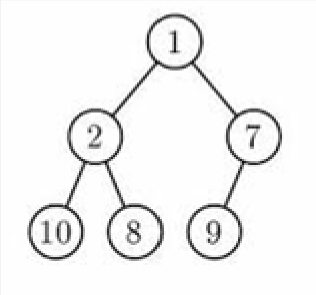
- Em ambos os casos o valor dos nós satisfaz a propriedade de ordenação
  - max-heap
    - Para todo o nó diferente da raiz

      H[Parent (i)] >= H[i]



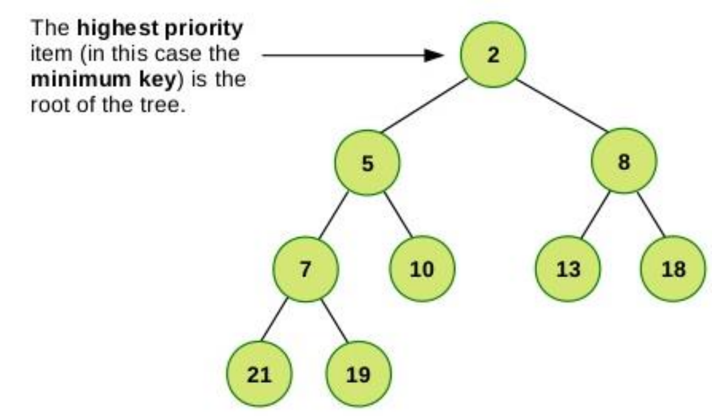
  - min-heap
    - Para todo o nó diferente da raiz

      H[Parent (i)] <= H[i]



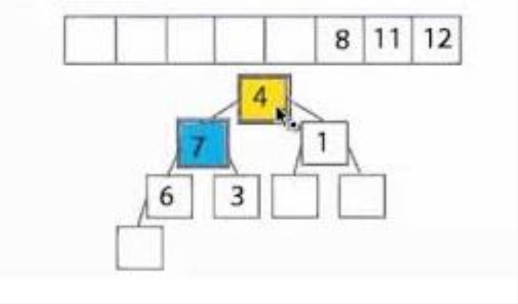
HEAPS - OBJETIVO

- Os *heaps* são estruturas de dados "inteligentes", cuja organização:
  - É especialmente usada para a implementação de filas prioritárias (*priority queues*)



min-heap

max-heap



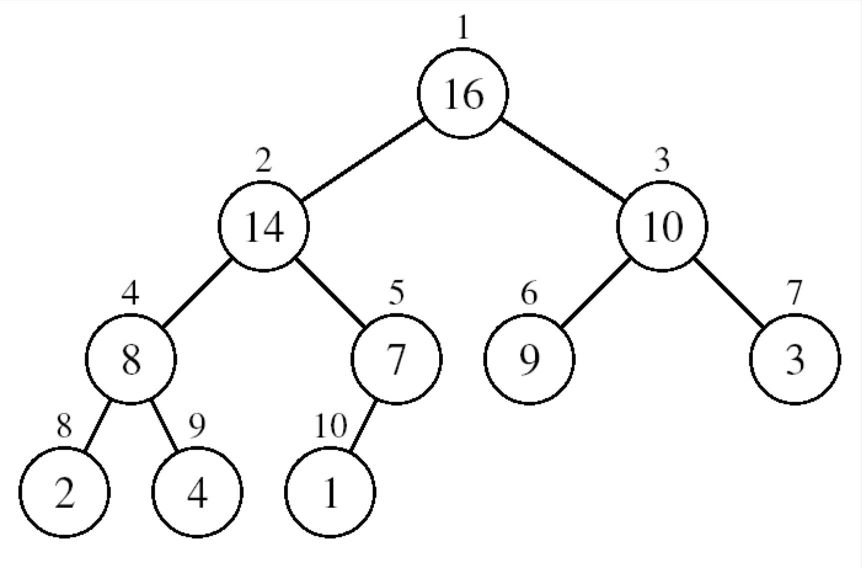
  - É também usado como estrutura de dados para um importante algoritmo de ordenação (*heap sort*)

HEAPS - OPERAÇÕES BASE

```
Parent (i)
      return i/2

Left (i)
      return 2i

Right (i)
      return 2i + 1
```



Parent (4) = 4/2 = 2
Left (4) = 2 * 4 = 8
Right (4) = 2 * 4 + 1 = 9

MIN-HEAP - OPERAÇÕES

- Mantém/Repõe a propriedade de ordenação do *min-heap* após a alteração de um elemento x para um valor maior
  - Min-Heapify

- Decrementa o valor do elemento x para o novo valor k, que é assumido ser menor ou igual ao valor de x
  - Decrease-Key

- Constrói um min-heap a partir de um *array* desordenado
  - Build-Min-Heap

MAX-HEAP - OPERAÇÕES

- Mantém/Repõe a propriedade de ordenação do *max-heap* após a alteração de um elemento x para um valor menor
  - Max-Heapify

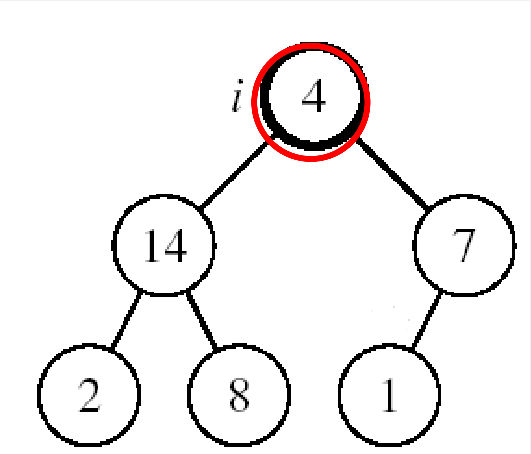
- Incrementa o valor do elemento x para o novo valor k, que é assumido ser maior ou igual ao valor de x
  - Increase-Key

- Constrói um max-heap a partir de um *array* desordenado
  - Build-Max-Heap

- Ordena uma lista de elementos a partir de um max-heap
  - Heap-Sort

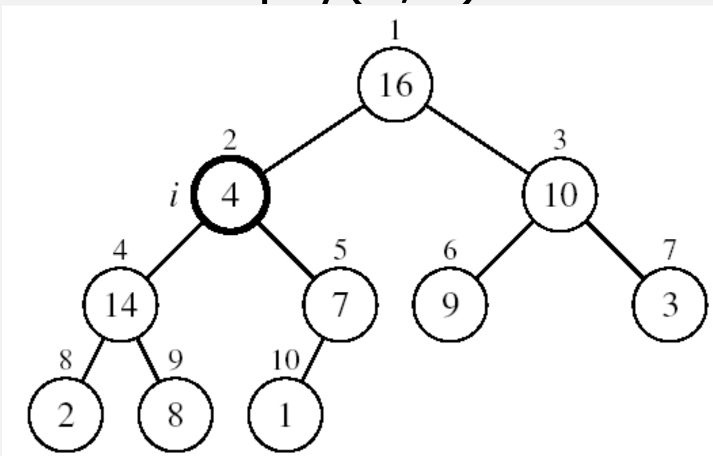
MAX-HEAPIFY

- Quando num determinado nó, a propriedade de ordenação do heap é violada, i.e., o seu valor é alterado para um valor menor
  - O Max-Heapify repõe a ordenação do *heap* a partir desse nó
    - Troca-se o valor do nó com o maior valor dos seu filhos
    - Desce-se um nível na árvore
    - Repete-se o processo até que o nó não tenha valor inferior ao filho ou até chegar a uma folha



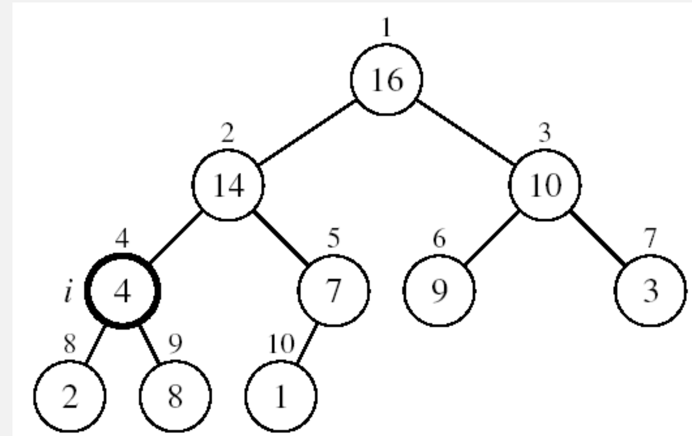
MAX-HEAPIFY - EXEMPLO

Max-Heapify(H, 2)



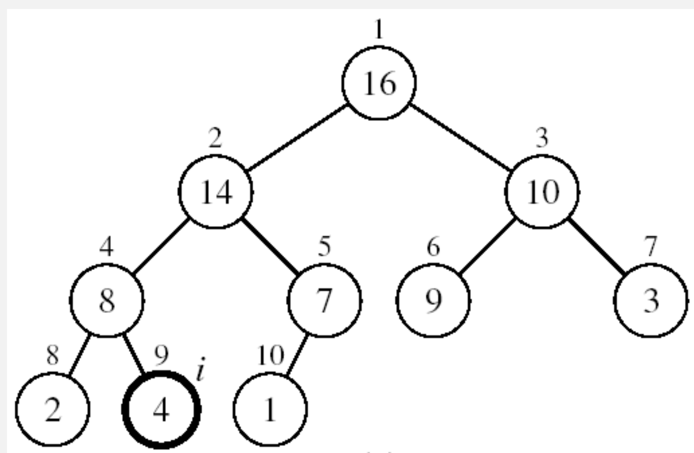
H[2] viola a propriedade do *heap*

H[2] ↔ H[4]



H[4] viola a propriedade do *heap*

H[4] ↔ H[9]



propriedade do *heap* reposta

MAX-HEAPIFY - ALGORITMO

- O Max-Heapify repõe a propriedade de ordenação de um max-heap
- O seu tempo de execução é *O* (lg n)

```
Max-Heapify (H, i)
    l = left(i)
    r = right(i)
    if  l ≤ H.heap-size and H[l] > H[i]
        largest = l
    else largest = i
    if  r ≤ H.heap-size and H[r] > H[largest]
        largest = r
    if  largest ≠ i // se i não é o índice do maior, continua
        exchange H[i] with H[largest]
        Max-Heapify(H, largest)
```

MAX-HEAPIFY - ANÁLISE

- Tempo de execução do Max-Heapify numa subárvore de dimensão n, com raiz num dado nó i

  - $O(1)$ para determinar a relação entre o valor do nó H[i] e os seus filhos H[left(i)] e H[right(i)]

  - Tempo para executar Max-Heapify numa sub-árvore de raiz num dos filhos do nó corrente

- Quantas chamadas recursivas a Max-Heapify podem existir numa subárvore de dimensão n?

MAX-HEAPIFY - ANÁLISE

- Melhor caso 
  - Não é feita a troca do valor do nó H[i] com um dos seus descendentes, pois tem um valor superior

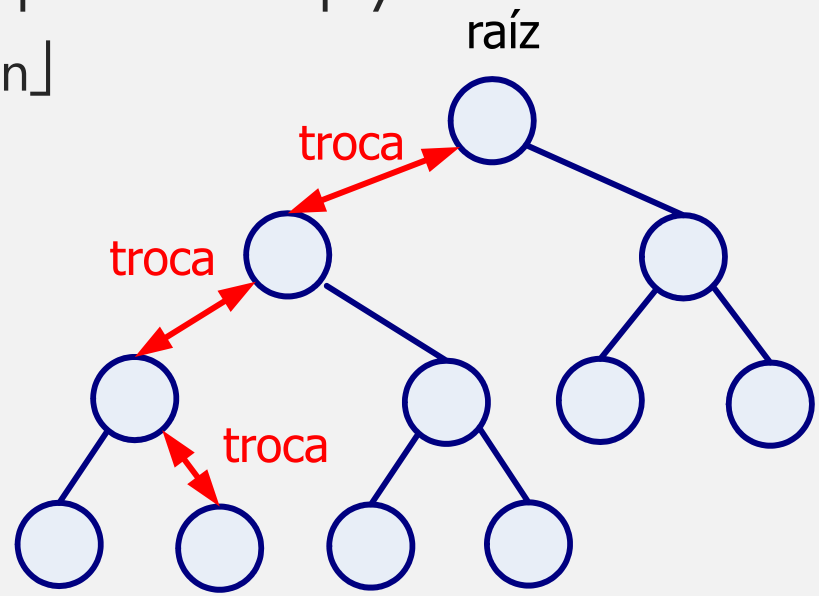
    $T_{best}(n) = O\ (1)$

- Pior caso 
  - Ocorre quando se faz a troca do valor de todos os nós, iniciando na raiz H[1] até uma folha no nível máximo da árvore

MAX-HEAPIFY - ANÁLISE

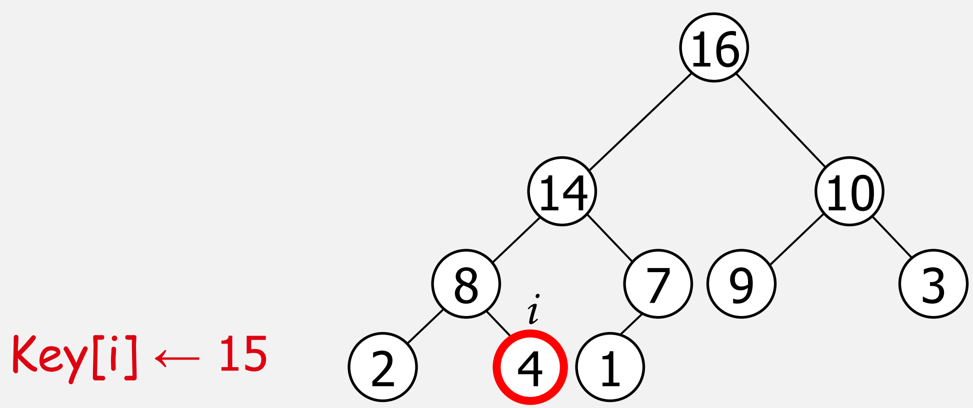
- Pior caso (cont.)
  - Uma árvore binária completa tem
    - Nº níveis = $\lfloor \lg n \rfloor + 1$
    - Altura h = $\lfloor \lg n \rfloor$
  - Podem ser executados pelo Max-Heapify
    - Nº níveis – 1 = $\lfloor \lg n \rfloor$

$T(n) = O\ (\lg n)$

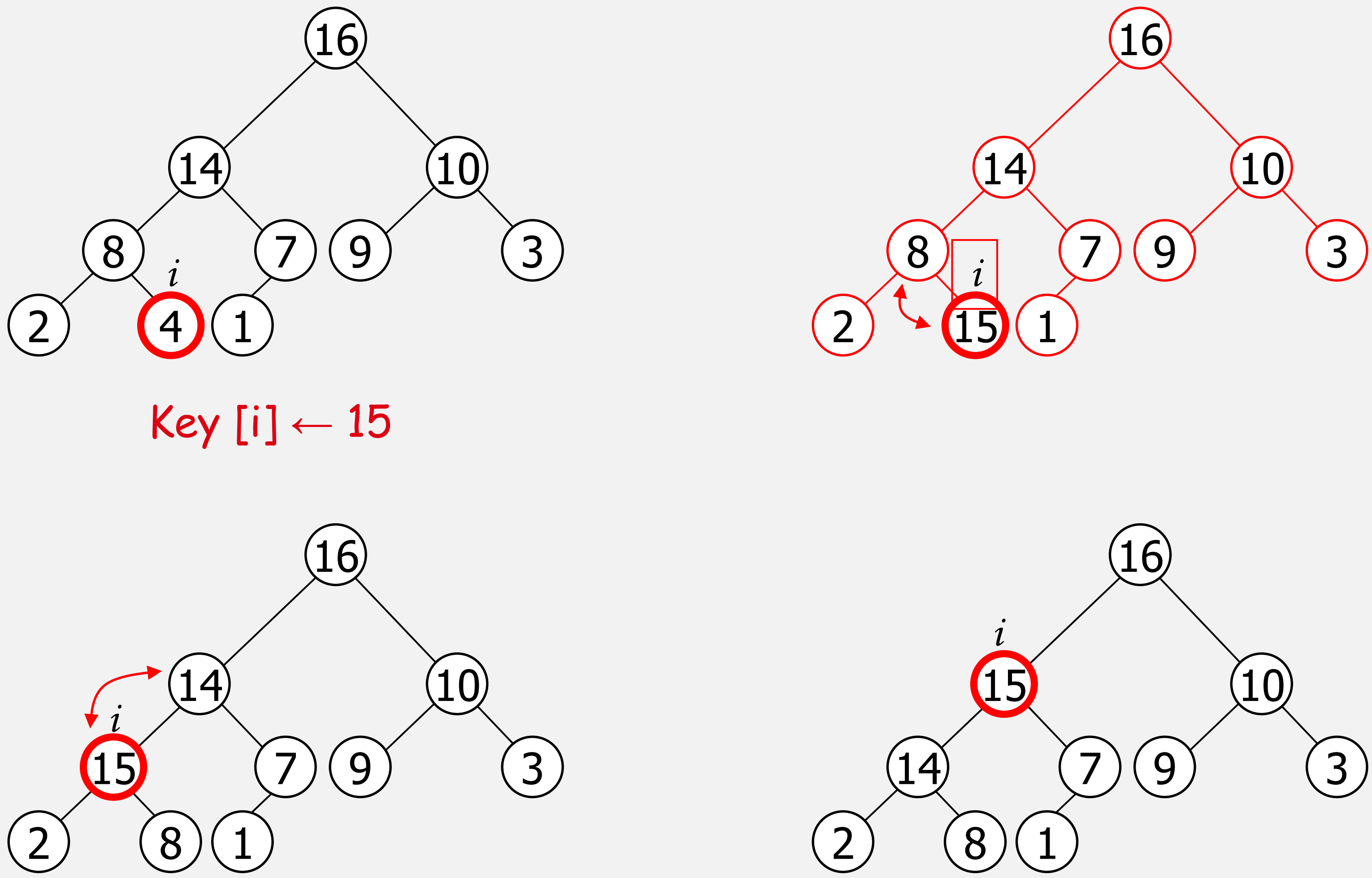


HEAP-INCREASE-KEY

- Heap-Increase-Key - Incrementa a prioridade de um elemento i do *heap*
  - Incrementa o valor de H[i] para o novo valor > H[i]
  - Se for violada a propriedade de ordenação do *max-heap*
    - Percorre-se um caminho desde o nó em direcção à raiz, trocando a chave do nó corrente com a do pai enquanto esta for menor



HEAP-INCREASE-KEY - EXEMPLO

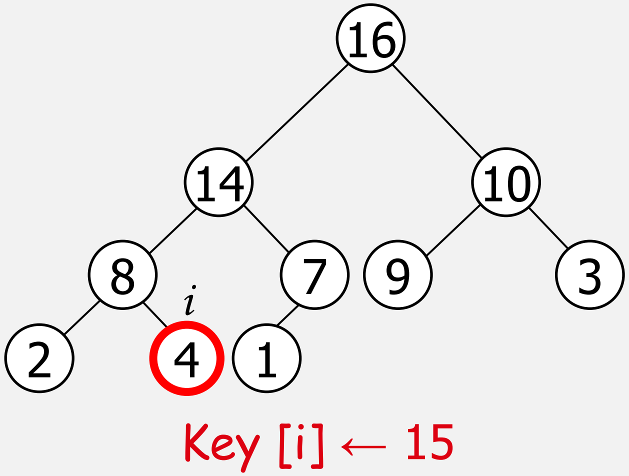


HEAP-INCREASE-KEY - ALGORITMO

Assumindo a operação:

```
H[i] = key        // em que key >= H[i]

Heap-Increase-Key (H, i)
    while i > 1 and H[Parent(i)] < H[i]
        exchange H[i] with H[Parent(i)]
        i = Parent(i)
```



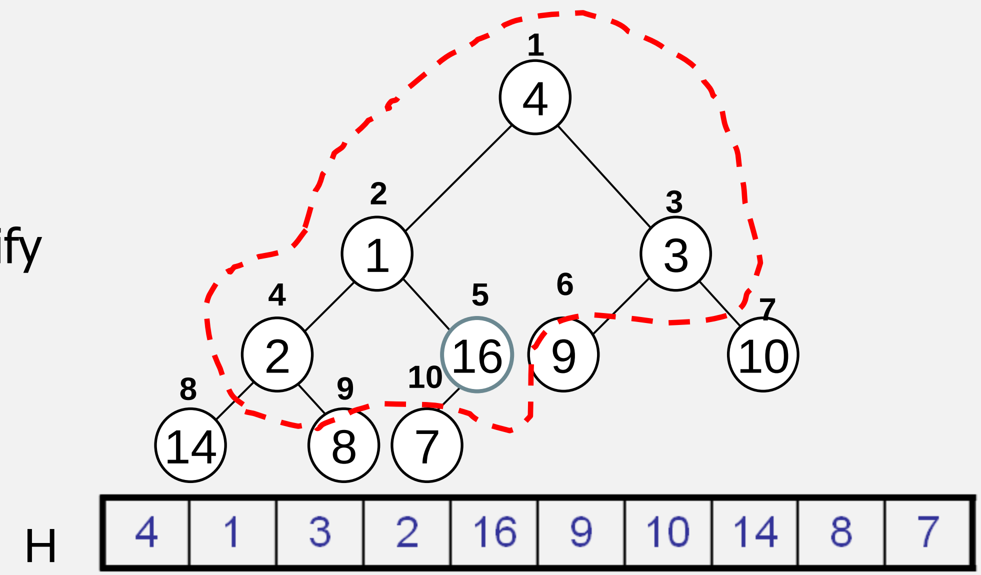
Tempo de execução: O(lg n)

BUILD-MAX-HEAP

- Permite construir um *max-heap*. Este é construído de baixo para cima (*botton-up*) de forma a converter todo o *array* H[1 .. n] num *max-heap*
  - Converte-se o *array* H[1 .. n] num *max-heap*

    (n = length[H])
  - Os elementos do *subarray* H[(n/2+1) .. n] são folhas
  - Aplica-se Max-Heapify aos elementos entre 1 e n/2

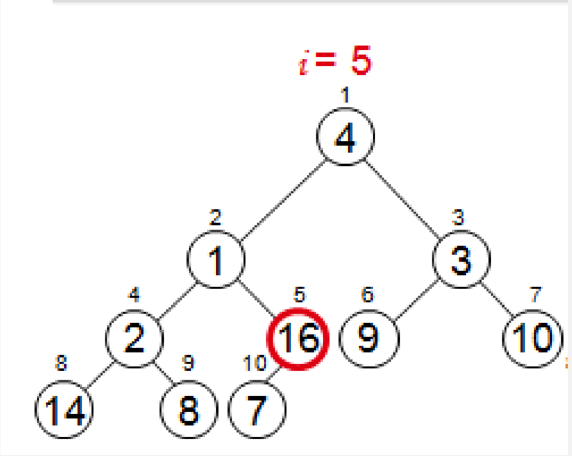
H[1 .. 10]
H[6 .. 10] – folhas
H[1 .. 5] – aplica-se Max-Heapify



BUILD-MAX-HEAP - EXEMPLO

H

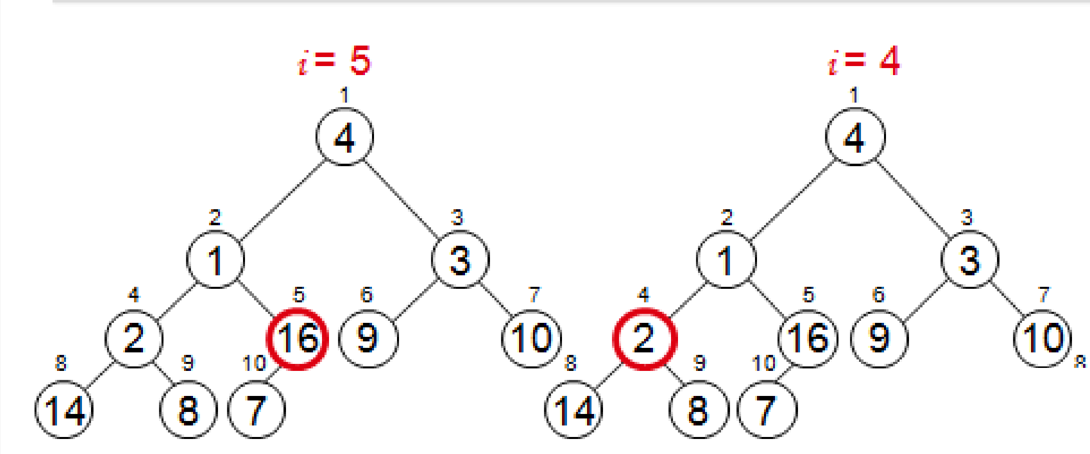
4	1	3	2	16	9	10	14	8	7
1	2	3	4	5	6	7	8	9	10



BUILD-MAX-HEAP - EXEMPLO

H

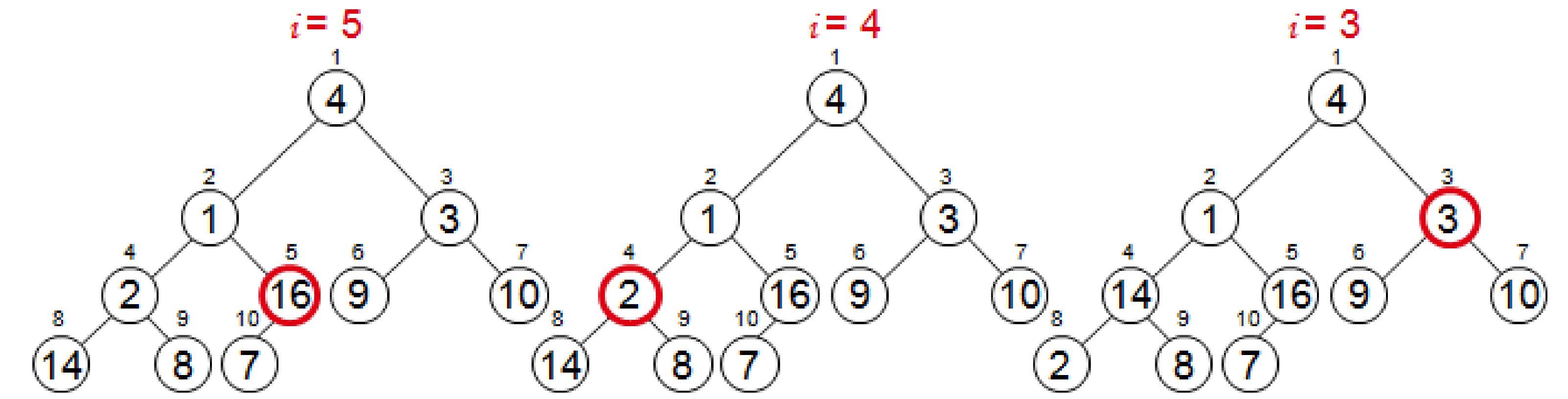
1	2	3	4	5	6	7	8	9	10
4	**1**	**3**	**2**	**16**	**9**	**10**	**14**	**8**	**7**



BUILD-MAX-HEAP - EXEMPLO

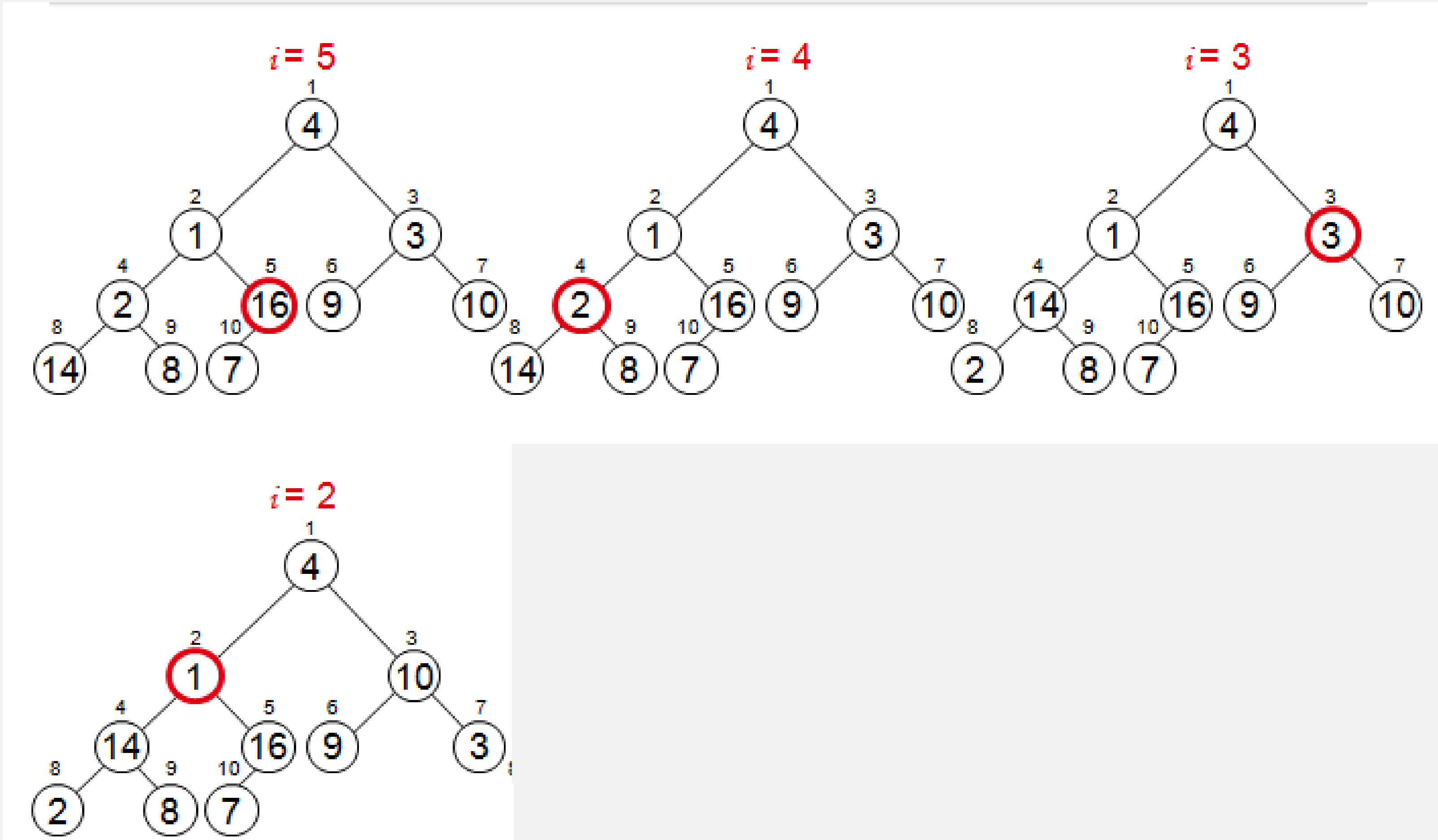
H

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



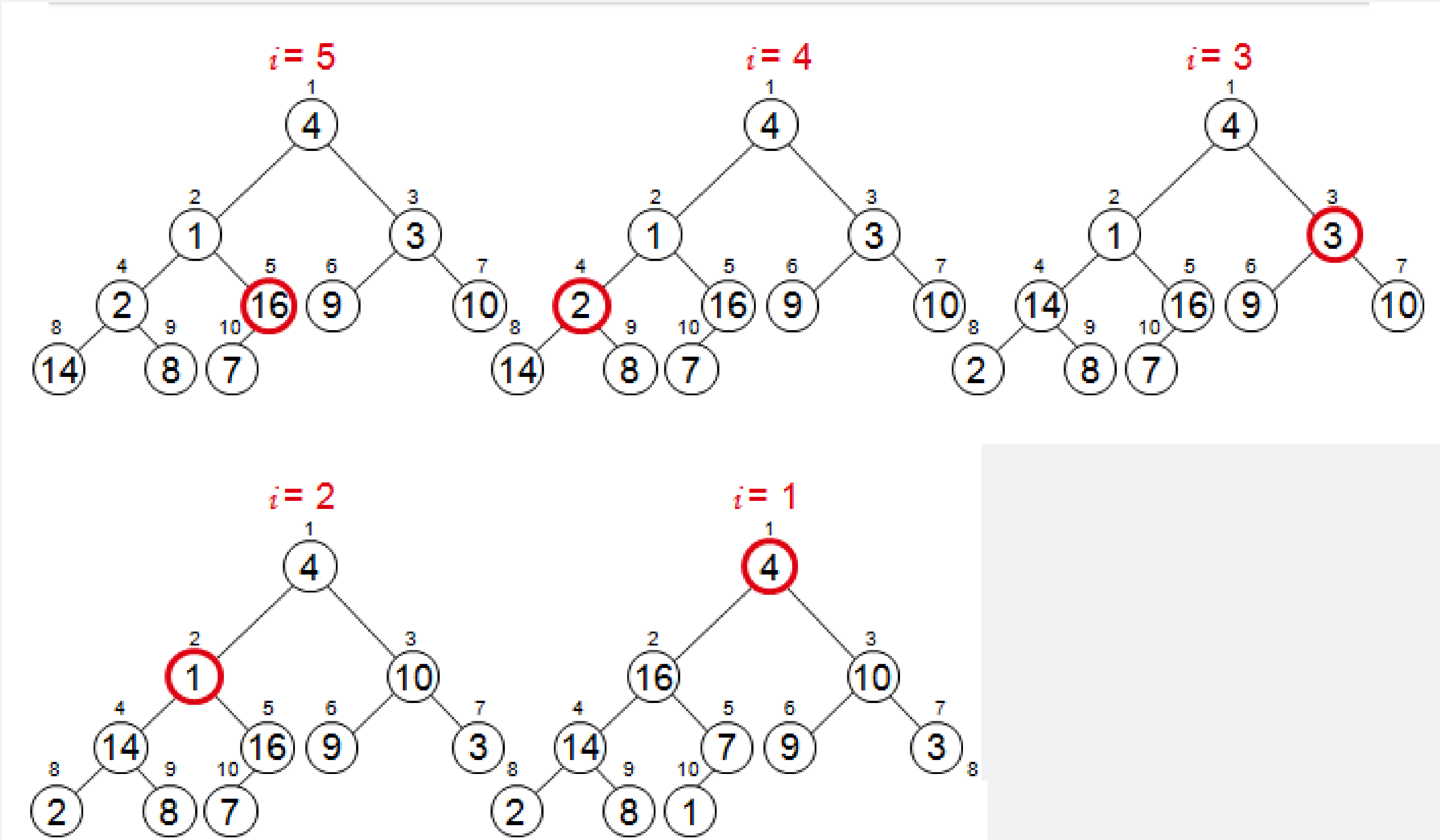
BUILD-MAX-HEAP - EXEMPLO

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



BUILD-MAX-HEAP - EXEMPLO

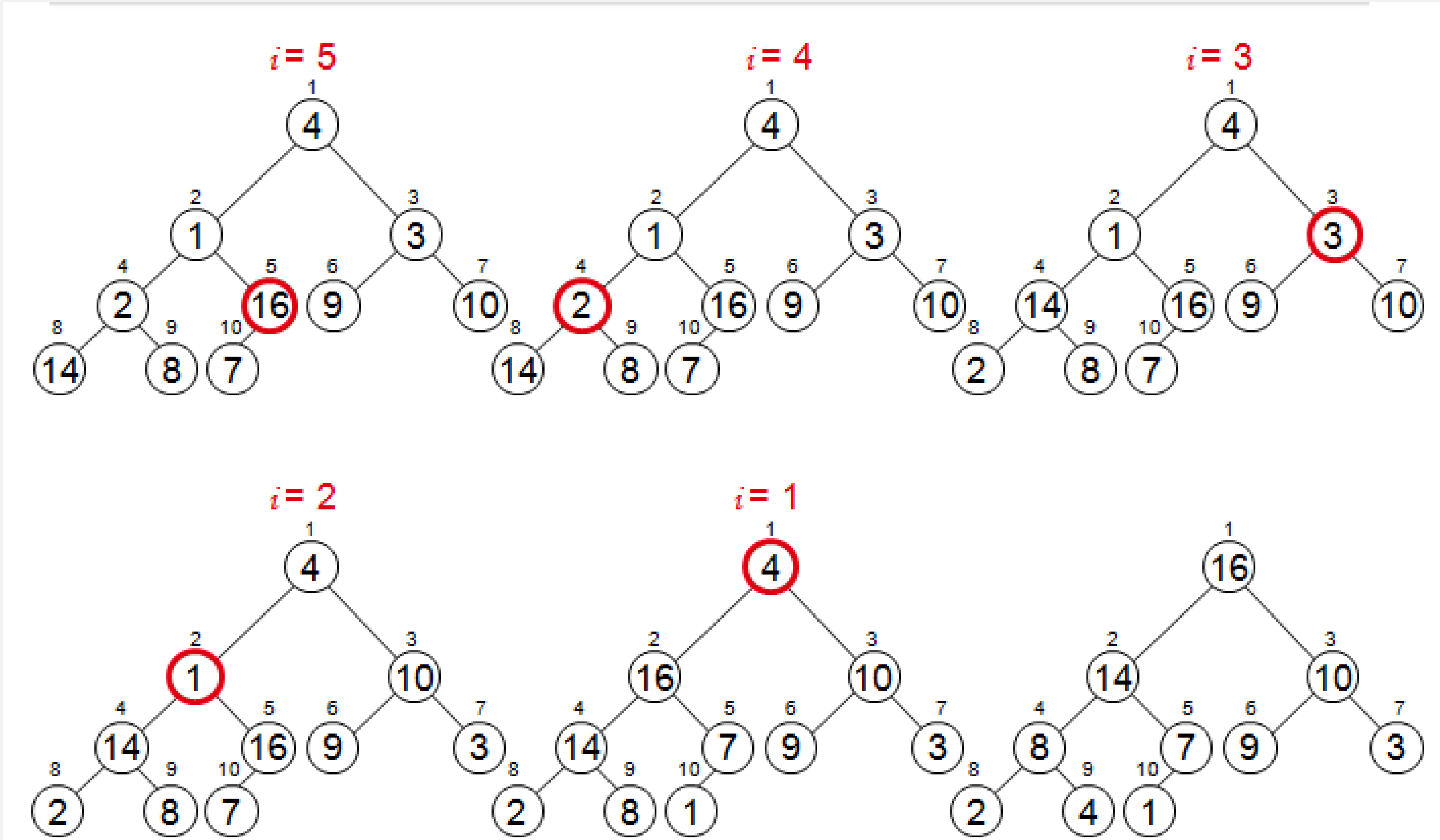
H	**4**	**1**	**3**	**2**	**16**	**9**	**10**	**14**	**8**	**7**
	1	2	3	4	5	6	7	8	9	10



BUILD-MAX-HEAP - EXEMPLO

H

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



BUILD-MAX-HEAP - ALGORITMO

- A função Build-Max-Heap constrói um *max-heap* com tempo de execução pertencente a *O* (n)

```
Build-Max-Heap (H, n)
    H.heap-size = H.lenght
    for  i = n/2 downto 1
        Max-Heapify(H, i)
```

BUILD-MAX-HEAP - ANÁLISE

- Cada chamada a Max-Heapify tem custo O (lg n)

  pois o custo é de O (h) = O (log n)

- O Max-Heapify é executado n/2 vezes, tendo custo:

  O (n/2) = O (½.n) = ½.O (n) = O (n)

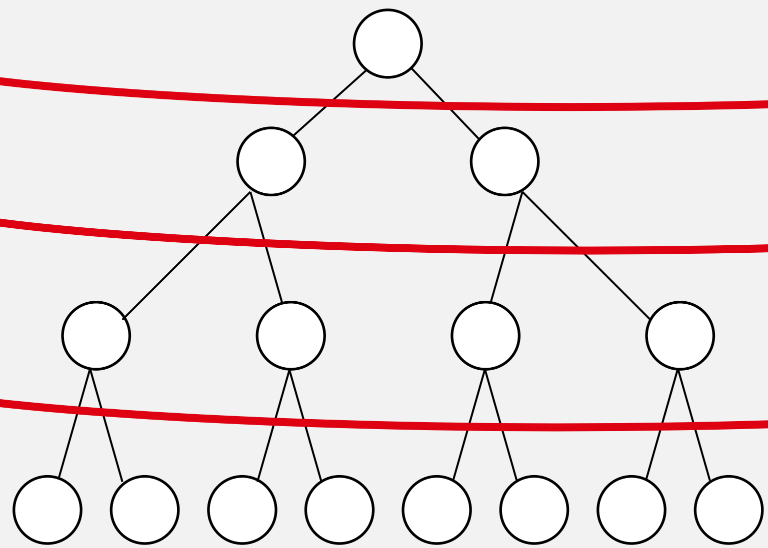
  ⇒ Custo total do Build-Max-Heap é O (lg n) * O (n)=O (n lg n)

- Contudo, este não é um limite assimptótico superior preciso, pois muitas das chamadas a Max-Heapify são em subárvores com dimensão < n

BUILD-MAX-HEAP - ANÁLISE

- Custo total do Build-Max-Heap : n * lg n = n * h
  - Então o custo do nível i do *heap* é dado por $n_i * h_i$
  - Sendo o total do Build-Max-Heap de árvore completa $\Rightarrow T(n) = \sum_{i=0}^{h} n_i h_i$

height h_i	level i	nodes n_i
$h_0 = 3$	i = 0	2^0
$h_1 = 2$	i = 1	2^1
$h_2 = 1$	i = 2	2^2
$h_3 = 0$	i = 3	2^3



$h_i = h - i$ (altura do heap no nível i)

$n_i = 2^i$ (nº nós no nível i)

BUILD-MAX-HEAP - ANÁLISE

$$T(n) = \sum_{i=0}^{h} n_i h_i$$

Custo total do Build-Max-Heap

$$= \sum_{i=0}^{h} 2^i (h-i)$$

Substituição pelos valores de n_i and h_i

$$= \sum_{i=0}^{h} \frac{2^h (h-i)}{2^h 2^{-i}} = \sum_{i=0}^{h} \frac{h-i}{2^{h-i}} 2^h$$

Multiplicação por 2^h do nominador e denominador e escrita de 2^i como $\frac{1}{2^{-i}}$

$$= \sum_{k=0}^{h} \frac{k}{2^k} 2^h = 2^h \sum_{k=0}^{h} \frac{k}{2^k}$$

Alteração das variáveis: k = h – i e passando a constante para fora

$$n \sum_{k=0}^{h} \frac{k}{2^k} \leq n \sum_{k=0}^{\infty} \frac{k}{2^k}$$

Substituindo $2^h = n$ pois h = lg n

BUILD-MAX-HEAP - ANÁLISE

Considerando que: $$\boxed{n\sum_{k=0}^{h}\frac{k}{2^k} \le n\sum_{k=0}^{\infty}\frac{k}{2^k}}$$

Sendo conhecida a seguinte fórmula: $$\boxed{\sum_{k=0}^{\infty} kx^k = \frac{x}{(1-x)^2}}$$

A expressão seguinte obedece à formula para x = 1/2

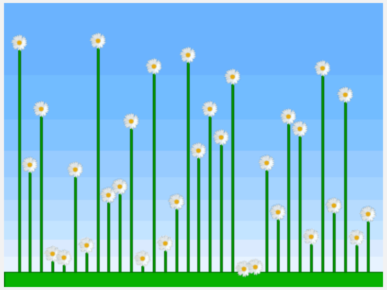
$$\sum_{k=0}^{\infty} k\left(\frac{1}{2}\right)^k = \sum_{k=0}^{\infty}\frac{k}{2^k} = \frac{1/2}{(1-1/2)^2} = \frac{1/2}{1/4} = 2$$

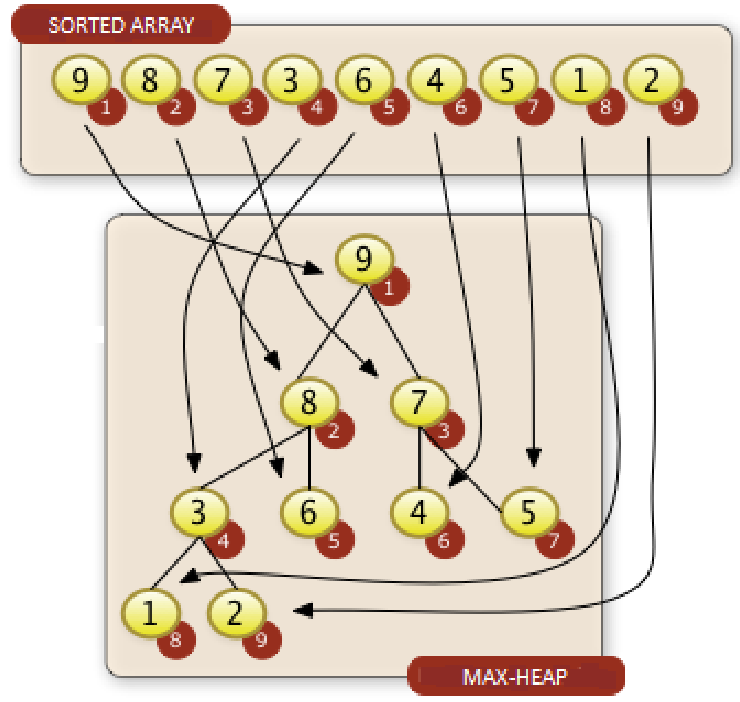
Então: $$n\sum_{k=0}^{h}\frac{k}{2^k} \le 2n$$ $T(n) = O(n)$

HEAP-SORT

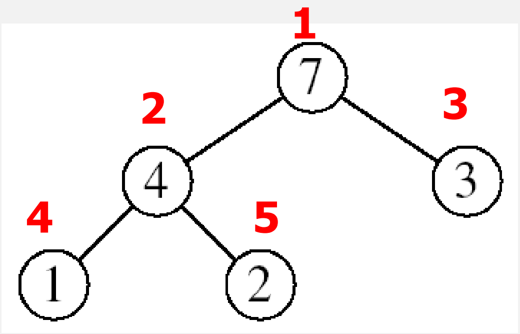
- Ordena um *array* usando um *max-heap* como estrutura de dados
- Utiliza o mesmo *array* (*in place*) para efectuar a ordenação
- O tempo de execução pertence a

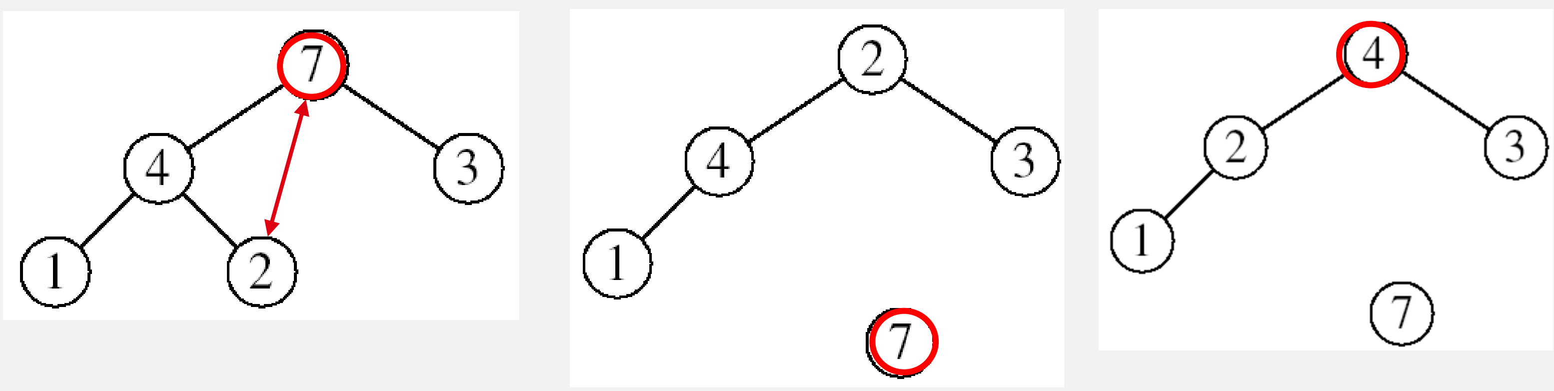
  O (n lg n)





- Troca a raiz (maior valor) com o último nó do *array*
- Descarta o último nó decrementando a dimensão do *heap*
- Executa Max-Heapify na nova raiz, pois a troca pode ter violado a propriedade de ordenação
- Repete-se o processo até o heap ter dimensão 1

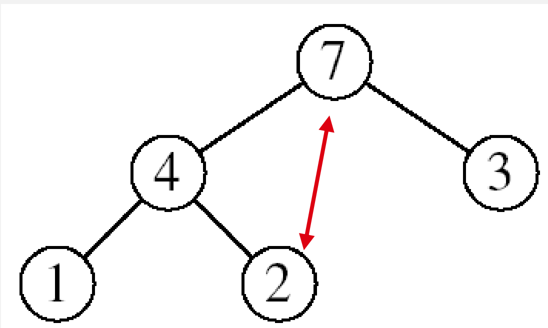




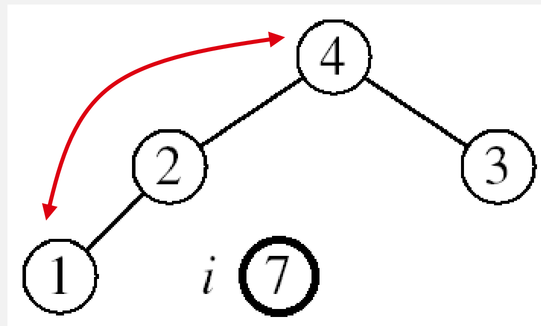
Max-Heapify(H, 1)

HEAP-SORT - EXEMPLO

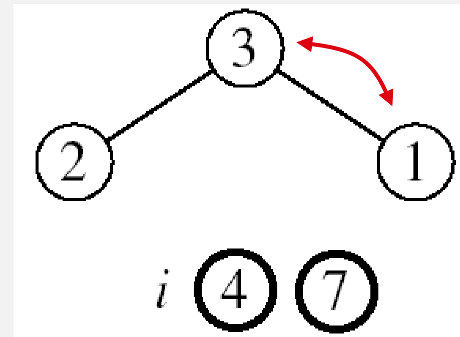
H=[7, 4, 3, 1, 2]



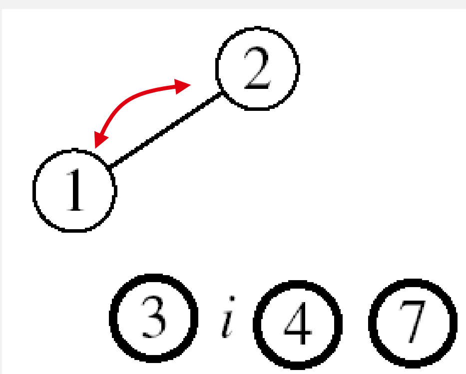
Max-Heapify(H, 1)



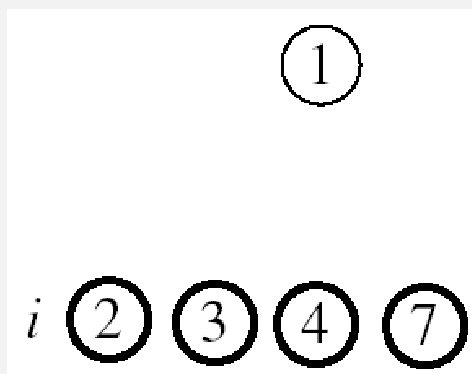
Max-Heapify(H, 1)



Max-Heapify(H, 1)



Max-Heapify(H, 1)



Max-Heapify(H, 1)

H=[1, 2, 3, 4, 7]

HEAP-SORT - ALGORITMO

- A função Heap-Sort constrói um *max-heap* e ordena-o

```
Heap-Sort (H, n)
    Build-Max-Heap(H, n)
    for  i  = n downto 2
        exchange H[1] with H[i]
        H.heap-size = H.heap-size - 1
        Max-Heapify(H, 1)
```

HEAP-SORT - ANÁLISE

```
Heap-Sort (H, n)
    Build-Max-Heap(H, n)
    for  i  = n downto 2
        exchange H[1] with H[i]
        H.heap-size = H.heap-size - 1
        Max-Heapify(H, 1)
```

- *Build-Max-Heap*: O (n)
- *for loop*: n – 1 vezes

    exchange : O (1)

    decrement : O (1)

    Max-Heapify : O (lg n)

$$T(n) = O(n) + O(n \lg n) = O(n \lg n)$$