EDCO4B
ESTRUTURAS DE DADOS 2

Aula 07 - Heap Sort

Prof. Rafael G. Mantovani
Prof. Luiz Fernando Carvalho

UTFPR
UNIVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ

Apucarana - PR, Brasil

Universidade Tecnológica Federal do Paraná (UTFPR)
Engenharia de Computação

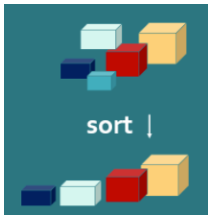
Roteiro

1. Introdução
2. Heaps
3. Heap Sort
4. Exemplo
5. Exercícios
6. Referências

Roteiro

1. Introdução
2. Heaps
3. Heap Sort
4. Exemplo
5. Exercícios
6. Referências

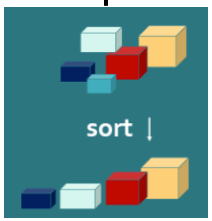
Introdução



Algoritmos de Ordenação

Introdução

Métodos Simples ⟷ Métodos Eficientes



Algoritmos de Ordenação

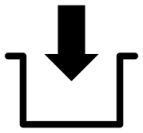
Introdução

Métodos Simples

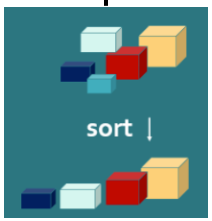
Métodos Eficientes

BubbleSort

InsertionSort



SelectionSort



Algoritmos de Ordenação

HeapSort

MergeSort

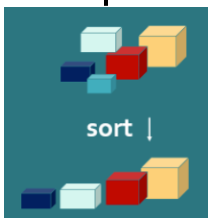
QuickSort

Introdução

Métodos Simples

Métodos Eficientes

OK BubbleSort



Algoritmos de Ordenação

HeapSort

OK InsertionSort

MergeSort

OK

OK SelectionSort

QuickSort OK

Roteiro

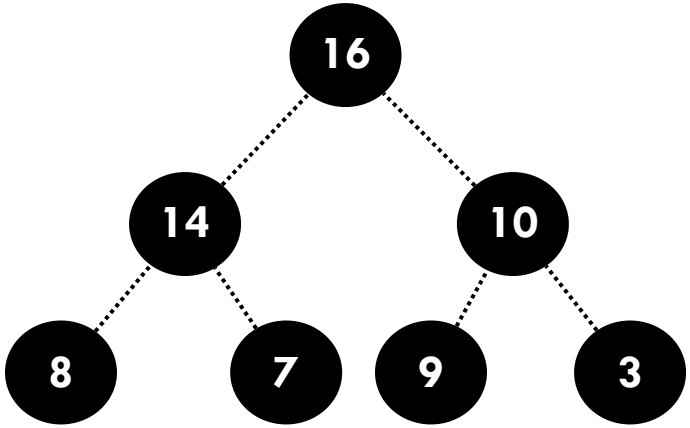
1. Introdução
2. Heaps
3. Heap Sort
4. Exemplo
5. Exercícios
6. Referências

Heap

- **Heap:** estrutura de dados em array que pode ser vista como uma árvore binária quase completa

Heap

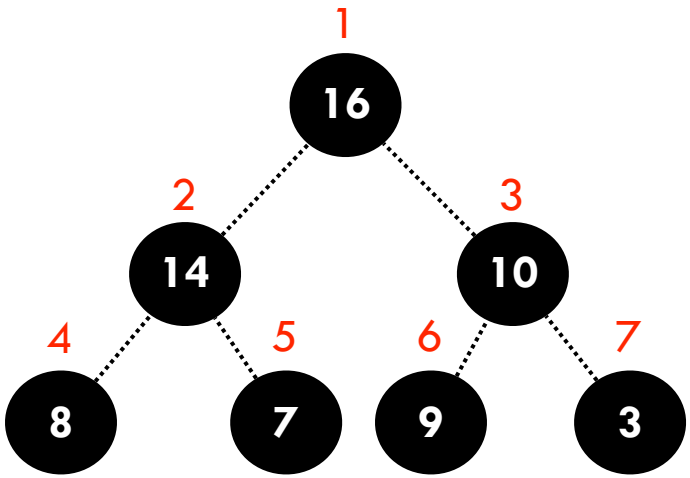
- **Heap:** estrutura de dados em array que pode ser vista como uma árvore binária quase completa



a) heap como árvore binária

Heap

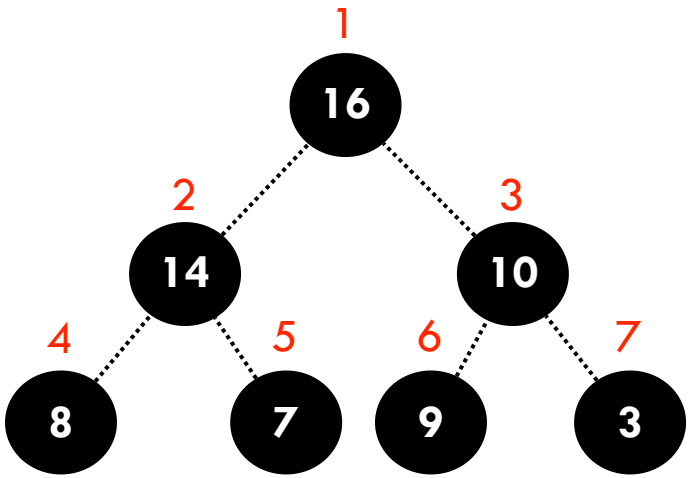
- **Heap:** estrutura de dados em array que pode ser vista como uma árvore binária quase completa



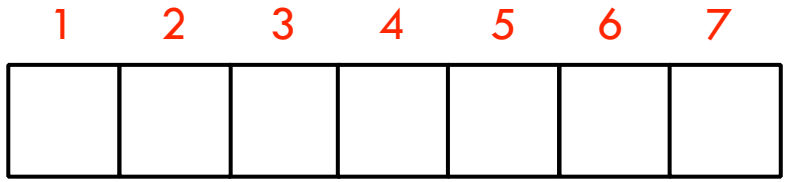
a) heap como árvore binária

Heap

- **Heap:** estrutura de dados em array que pode ser vista como uma árvore binária quase completa



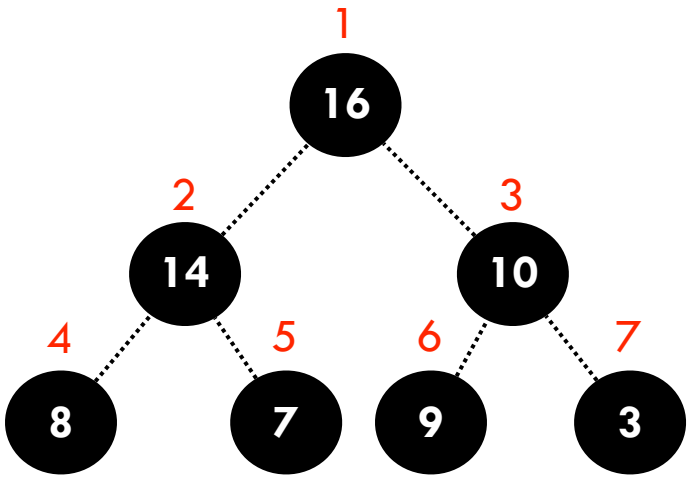
a) heap como árvore binária



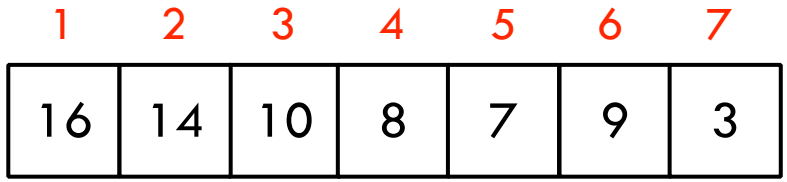
b) heap como array

Heap

- **Heap:** estrutura de dados em array que pode ser vista como uma árvore binária quase completa



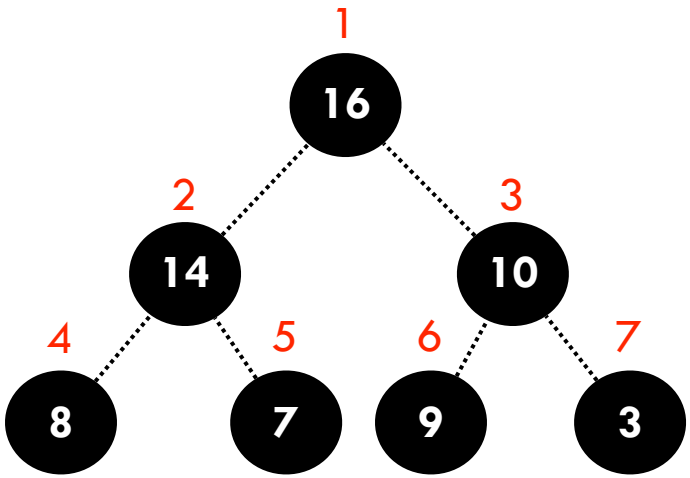
a) heap como árvore binária



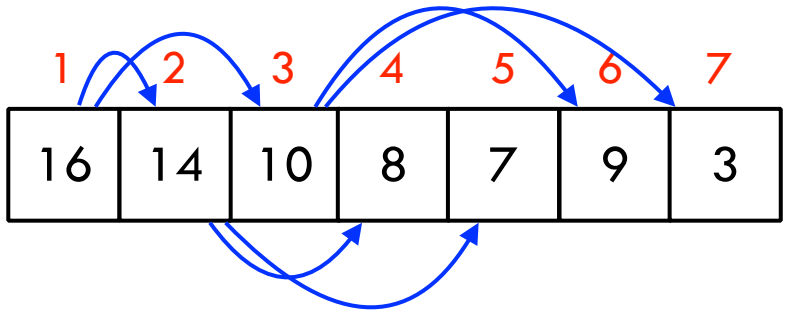
b) heap como array

Heap

- **Heap:** estrutura de dados em array que pode ser vista como uma árvore binária quase completa



a) heap como árvore binária



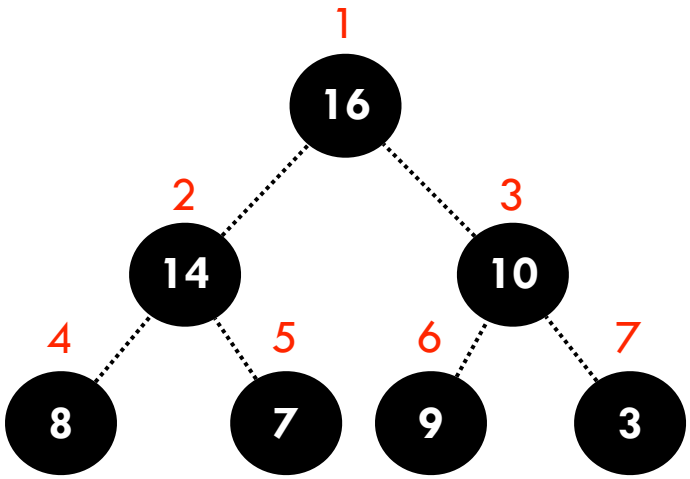
b) heap como array

Heap

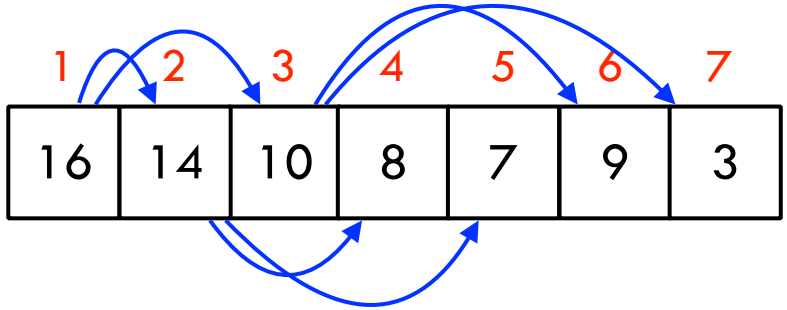
* Pai(i) = floor(i/2)
* Esquerda(i) = 2*i
* Direita(i) = 2*i + 1

Heap

* Pai(i) = floor(i/2)
* Esquerda(i) = 2*i
* Direita(i) = 2*i + 1



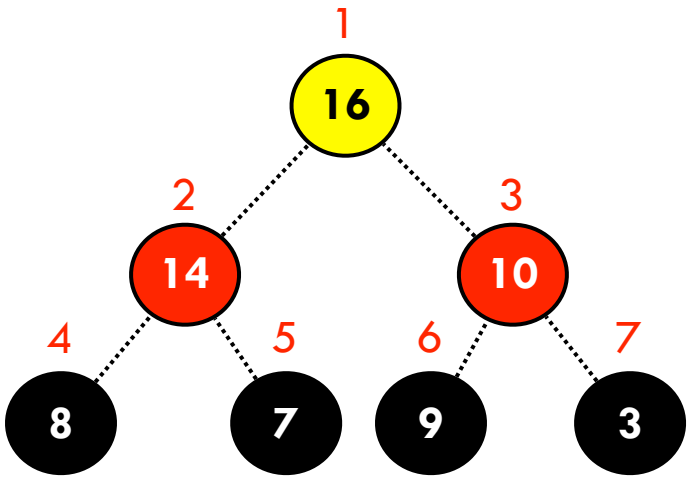
a) heap como árvore binária



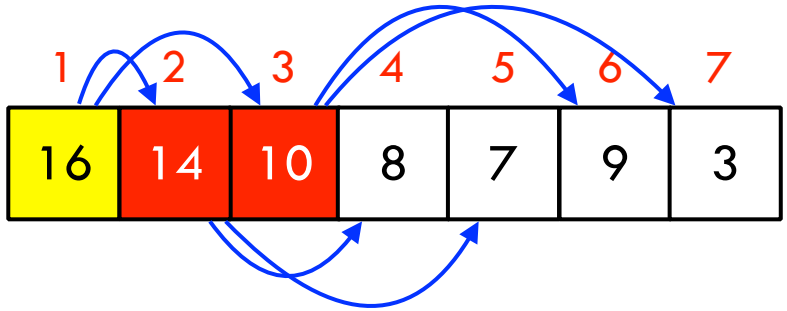
b) heap como array

Heap

* Esquerda(1) = 2*1 = 2
* Direita(1) = 2*1 + 1 = 3



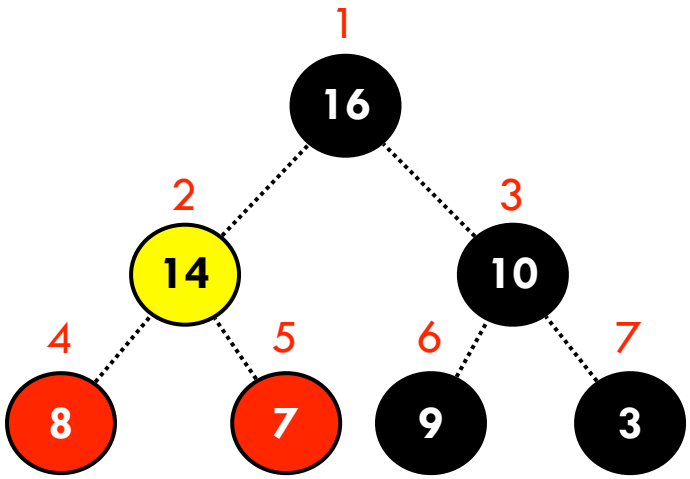
a) heap como árvore binária



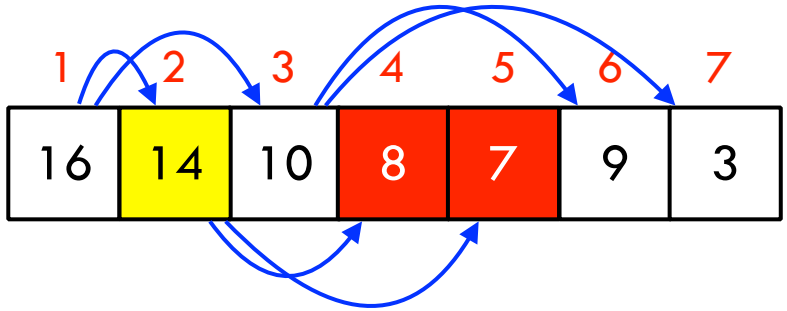
b) heap como array

Heap

* Pai(2) = floor(2/2) =1
* Esquerda(2) = 2*2 =4
* Direita(2) = 2*2 + 1 =5



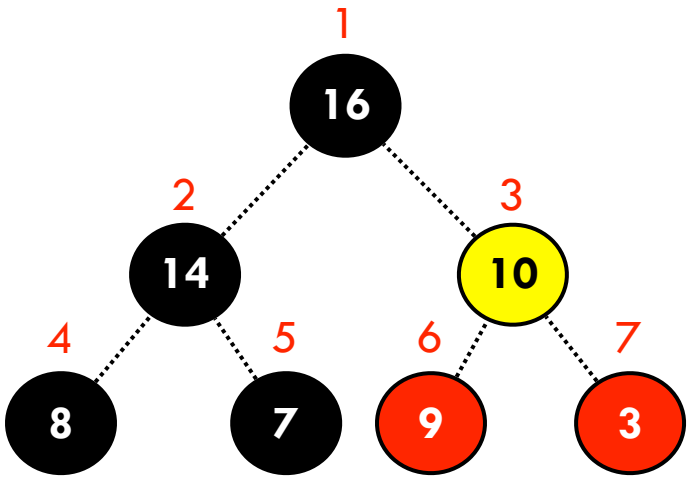
a) heap como árvore binária



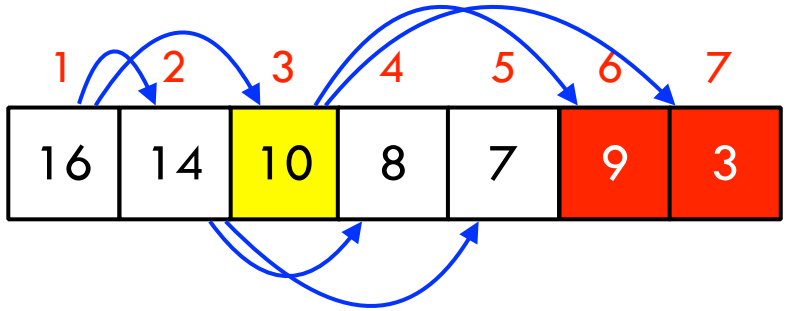
b) heap como array

Heap

* Pai(3) = floor(3/2) =1
* Esquerda(3) = 2*3 =6
* Direita(3) = 2*3 + 1 =7



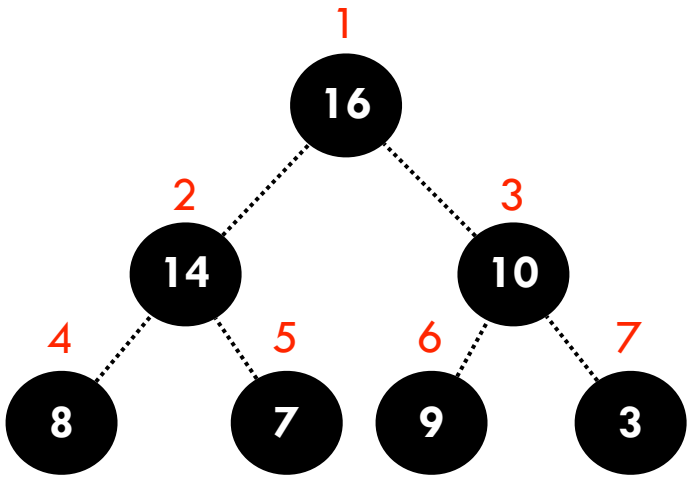
a) heap como árvore binária



b) heap como array

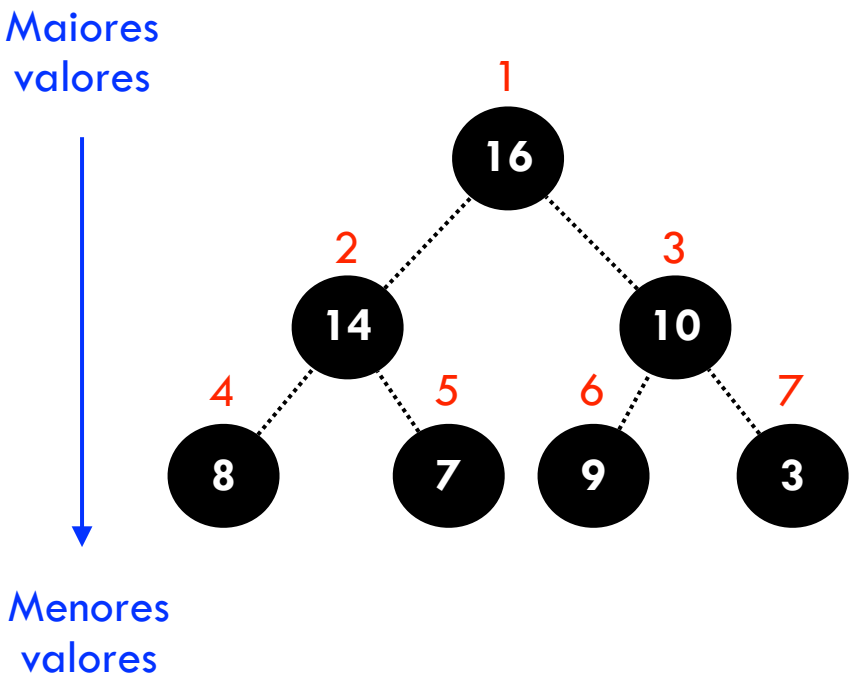
Heap

- **Heap de máximo:**
  - A[Pai(i)] >= A[i]



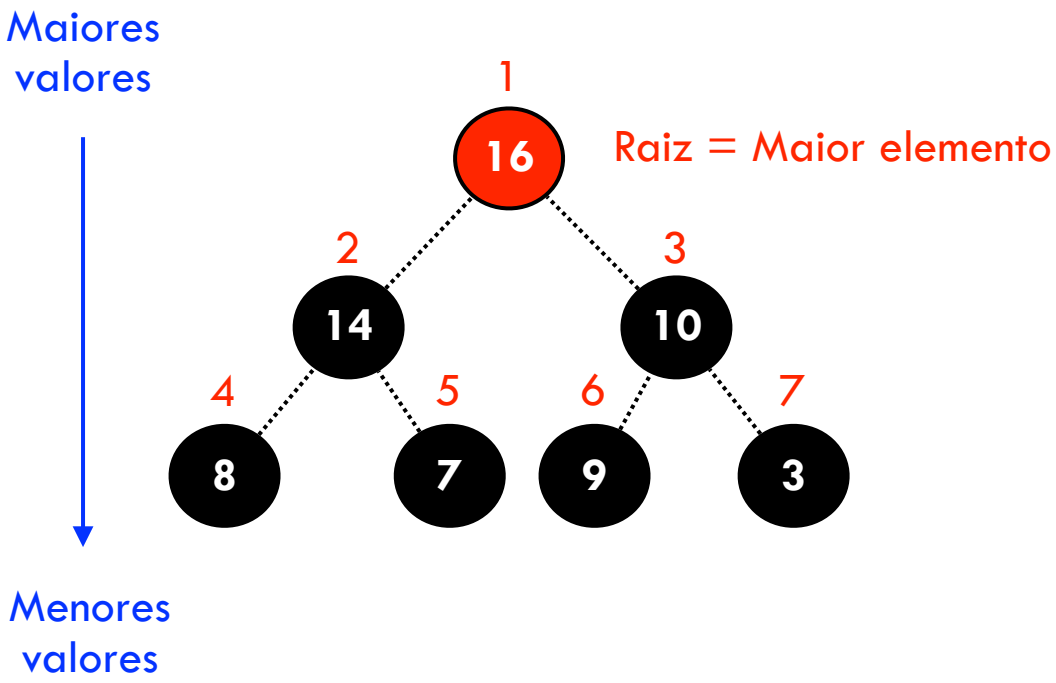
Heap

- **Heap de máximo:**
  - A[Pai(i)] >= A[i]



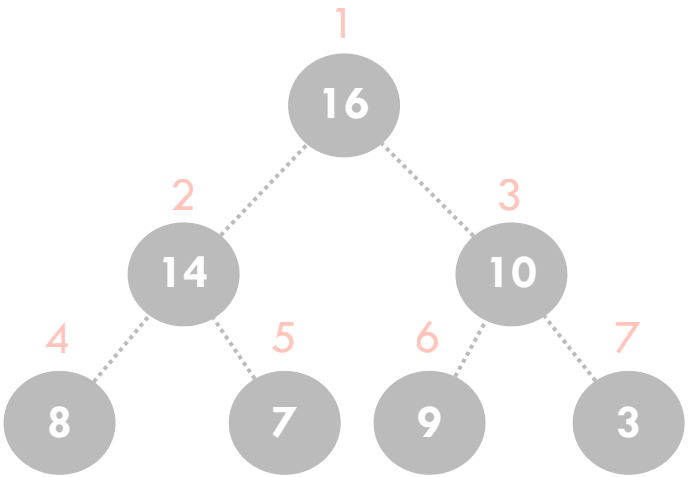
Heap

- **Heap de máximo:**
  - A[Pai(i)] >= A[i]



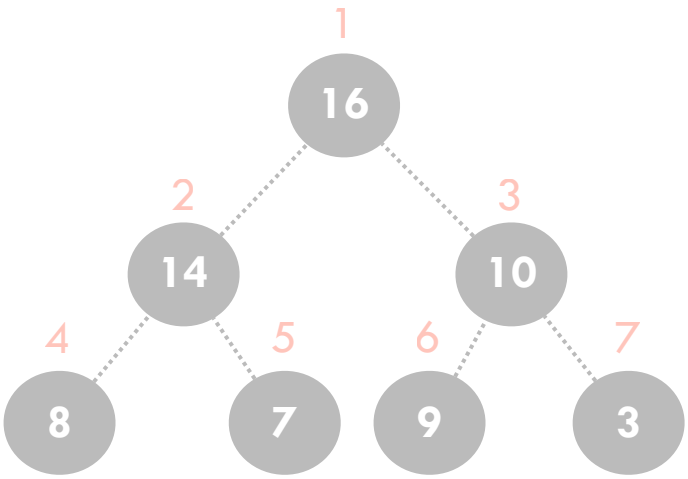
Heap

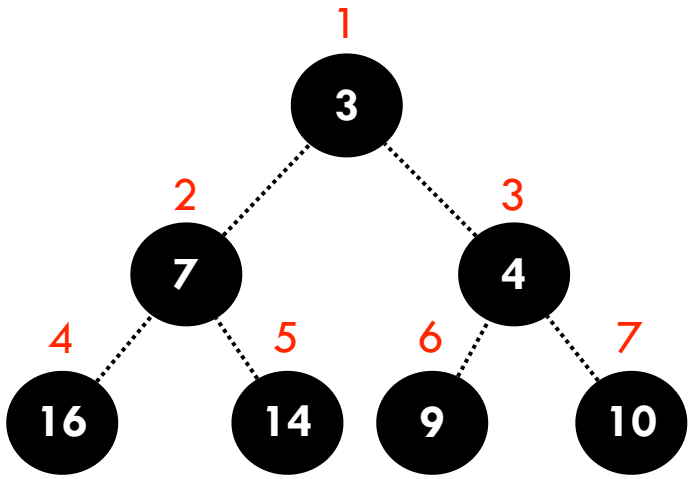
- **Heap de máximo:**
  - A[Pai(i)] >= A[i]



Heap

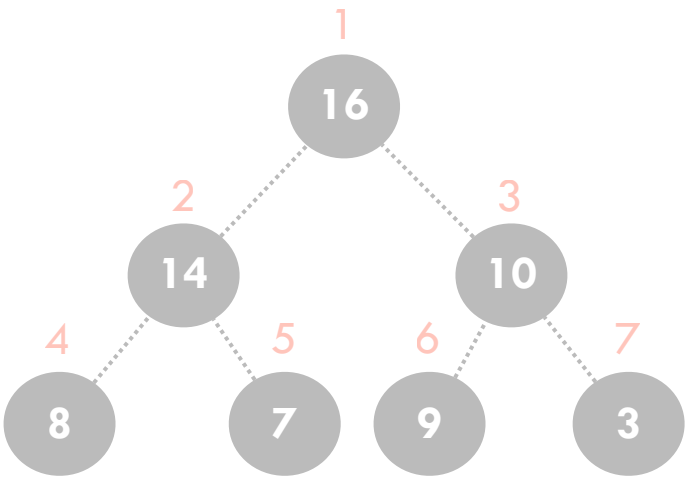
- **Heap de mínimo:**
  - A[Pai(i)] <= A[i]

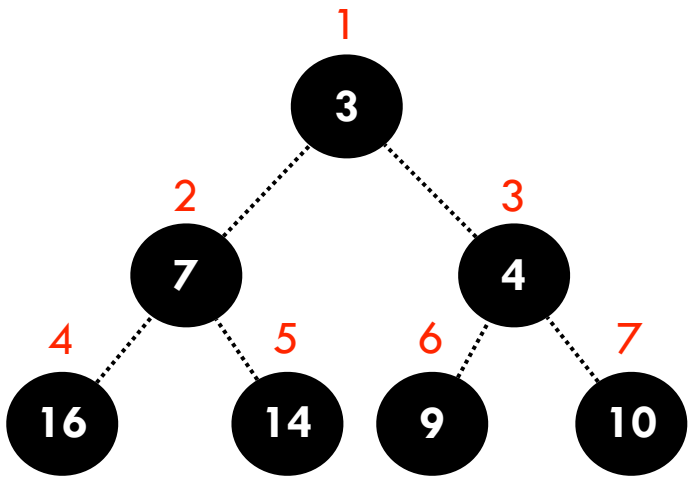




Heap

- **Heap de mínimo:**
  - A[Pai(i)] <= A[i]



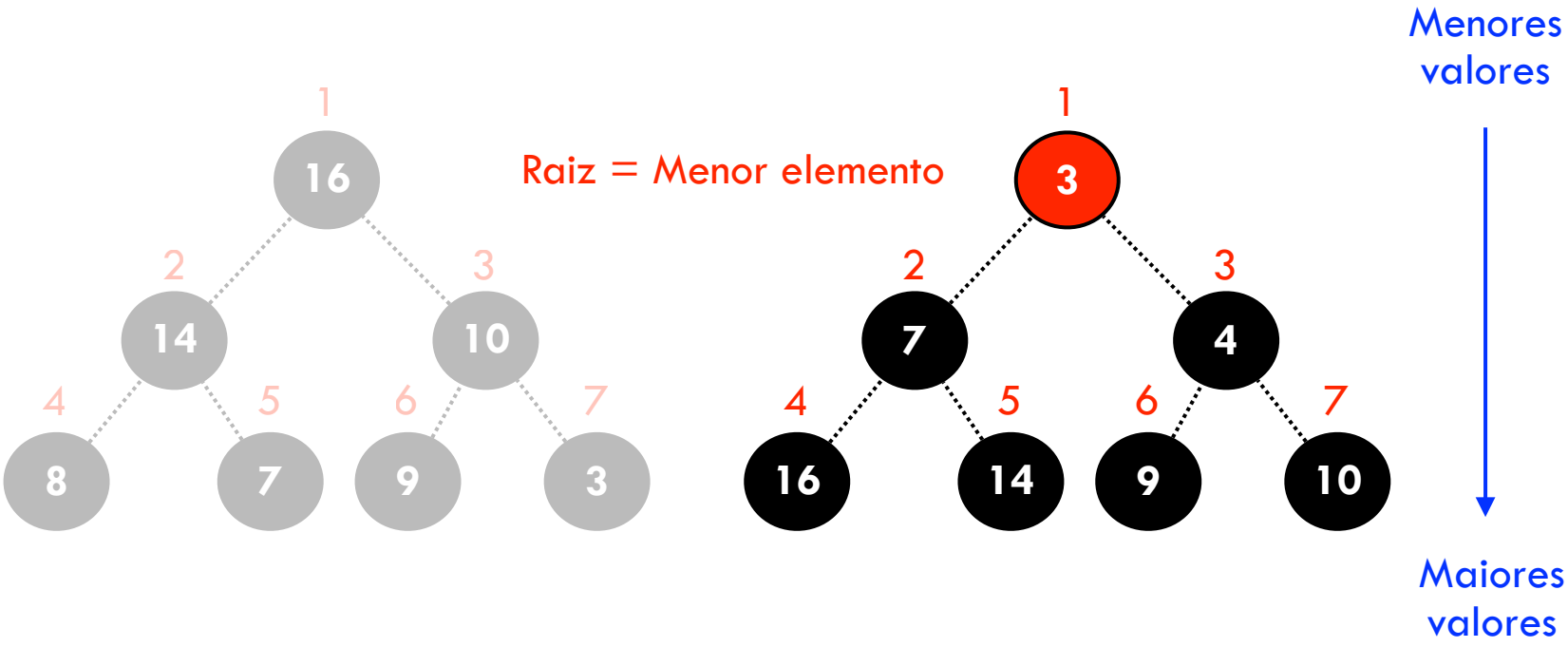


Menores valores

Maiores valores

Heap

- **Heap de mínimo:**
  - A[Pai(i)] <= A[i]



Roteiro

1. Introdução
2. Heaps
3. **Heap Sort**
4. Exemplo
5. Exercícios
6. Referências

Heap Sort

Heap Sort

- **Funcionamento**

* Passos

1. Transformar o heap (vetor) em um max-heap
2. Mover a raiz (maior valor) para o final do vetor (última posição)
3. Reconstrói o heap do vetor restante

Heap Sort

- **Desempenho**

* **melhor caso:** O(N log N) // *chaves distintas*
  O(N) // *chaves iguais*

* **pior caso:** O(N log N)

* **caso médio:** O(N log N)

Heap Sort

- **Desempenho**

```
* melhor caso: O(N log N)  // chaves distintas
               O(N)        // chaves iguais

* pior caso:   O(N log N)

* caso médio:  O(N log N)
```

Obs: Na prática, o HeapSort é mais lento que o QuickSort, exceto no pior caso.

Heap Sort

- **Pseudocódigo**

1. **HEAPSORT:** ordena o vetor usando heap auxiliar

2. **BUILD-MAX-HEAP**: cria um heap de máximo a partir dos valores aleatórios de um vetor

3. **MAX-HEAPIFY**: mantem a propriedade de heap de máximo de um conjunto de valores

Heap Sort

- **Pseudocódigo** (mantém heap)

1. **MAX-HEAPIFY** (**vetor**, i)

- Entradas:
  - **vetor**: vetor de elementos
  - **i**: índice da posição para se criar max-heap

- Além disso, em **vetor** podemos obter:
  - **vetor.tamanho-heap**: tamanho a ser considerado para criar heap
  - **vetor.tamanho**: tamanho do vetor

Heap Sort

MAX-HEAPIFY (vetor, i):

```
1. L = Esquerda ( i )   // 2 * I + 1
2. R = Direita ( i )    // 2 * I + 2
3. MAIOR = i
```

Heap Sort

MAX-HEAPIFY (vetor, i):

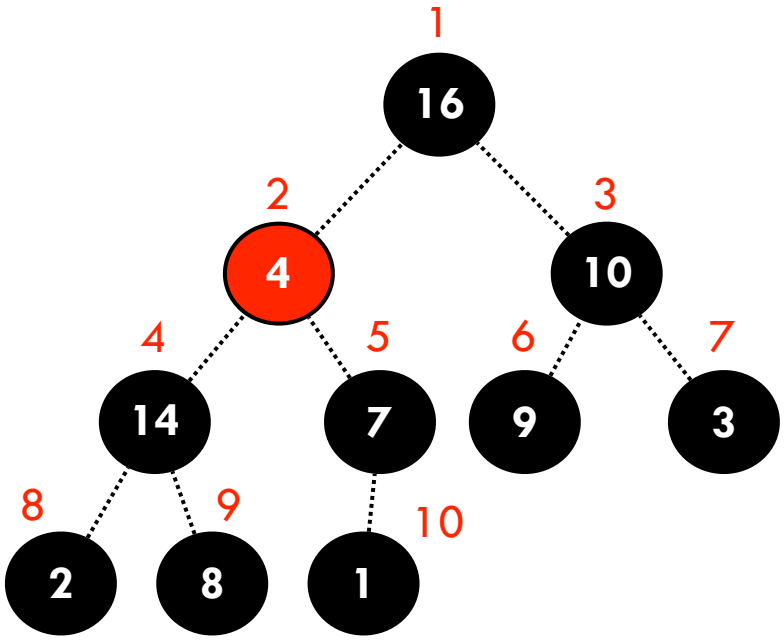
```
1.  L = Esquerda ( i )    // 2 * I + 1
2.  R = Direita ( i )     // 2 * I + 2
3.  MAIOR = i
    //determina qual filho de i é maior (R ou L)
4.  Se ( L <= vetor.tamamho-heap E vetor[ L ] > vetor[ i ]):
5.      MAIOR = L
6.  Se ( R <= vetor.tamamho-heap E vetor[ R ] > vetor[ MAIOR ]):
7.      MAIOR = R
```

Heap Sort

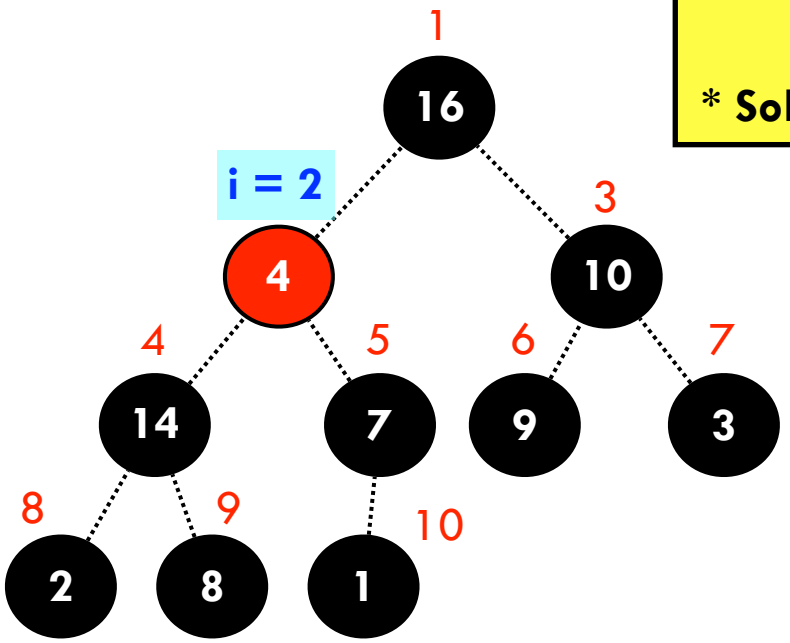
MAX-HEAPIFY (vetor, i):

```
1.  L = Esquerda ( i )   // 2 * I + 1
2.  R = Direita ( i )    // 2 * I + 2
3.  MAIOR = i
    //determina qual filho de i é maior (R ou L)
4.  Se ( L <= vetor.tamamho-heap E vetor[ L ] > vetor[ i ]):
5.      MAIOR = L
6.  Se ( R <= vetor.tamamho-heap E vetor[ R ] > vetor[ MAIOR ]):
7.      MAIOR = R
8.  Se ( MAIOR != i ):    // se algum filho é maior que o pai, trocar elementos
9.      Trocar os elementos das posições vetor[ i ] e vetor [ MAIOR ]
10.     MAX-HEAPIFY (vetor, MAIOR)          // chamar recursivamente
```

MAX-HEAPIFY



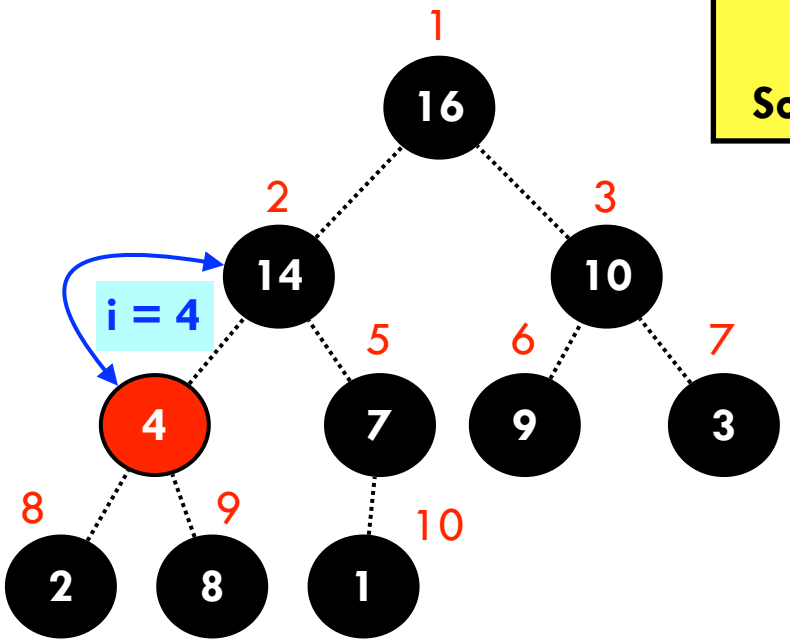
MAX-HEAPIFY



* Configuração inicial (i = 2)
  vetor[2] viola a propriedade de heap máximo:
  vetor[2] < **vetor**[4] e **vetor**[2] < **vetor**[5]

* **Solução:** Trocar(**vetor**[2], **vetor**[4])

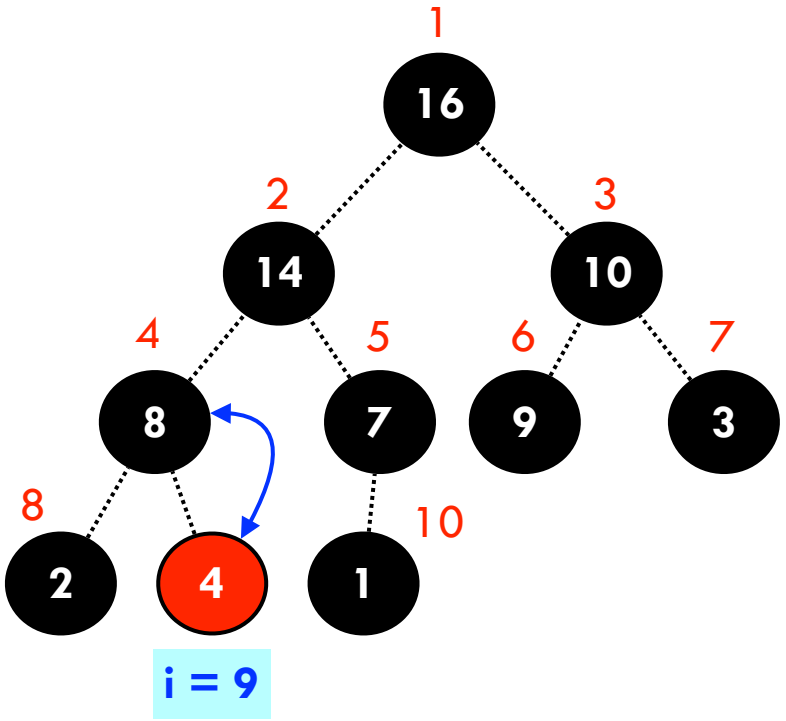
MAX-HEAPIFY



* Configuração atual (i = 4)
  vetor[4] viola a propriedade de heap máximo:
  vetor[4] < **vetor**[9]

  Solução: Trocar(**vetor**[4], V[9])

MAX-HEAPIFY



Tudo certo agora :)

Heap Sort

- **Pseudocódigo** (cria max-heap)

2. **BUILD-MAX-HEAP** (**vetor**)

- Entradas:
  - **vetor:** vetor de elementos
- Usa MAX-HEAPIFY em uma abordagem *bottom up* convertendo o array **vetor** em um max-heap
  - iterar de N/2 ... até 1

Heap Sort

BUILD-MAX-HEAP(vetor):

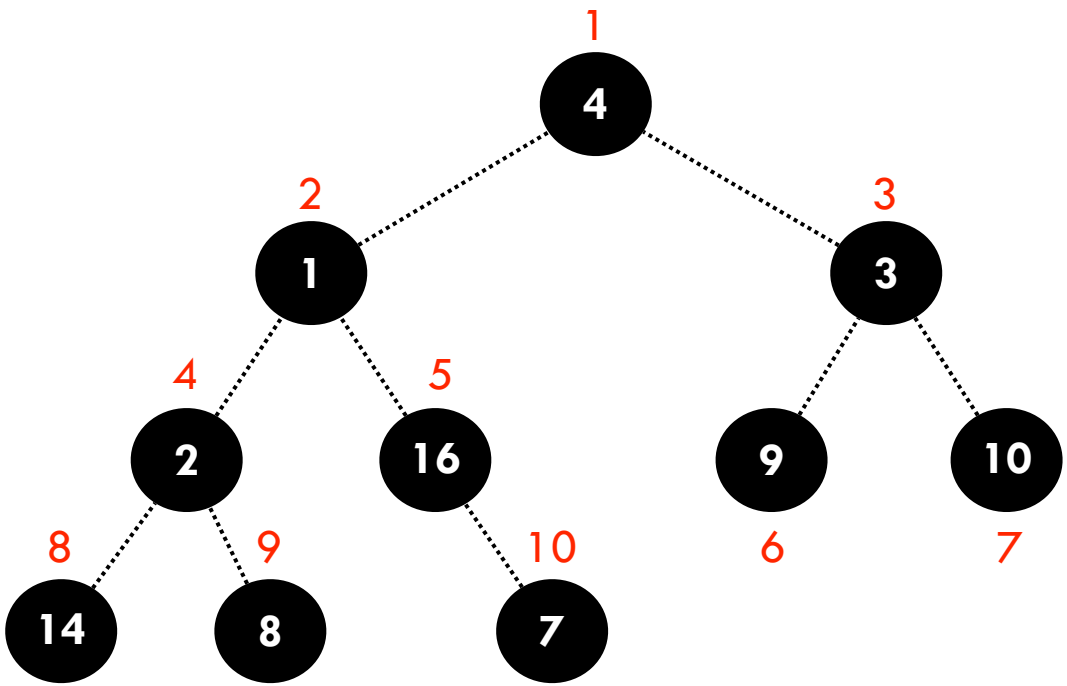
```
1.  vetor.tamanho-heap  = vetor.tamanho

    //cria um heap de máximo "de baixo para cima"
2.  Para i de piso(vetor.tamanho/2) até 1, faça:
        //cria um heap de máximo para cada posição i
3.      MAX-HEAPIFY (vetor, i)
```

BUILD-MAX-HEAP

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

BUILD-MAX-HEAP

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



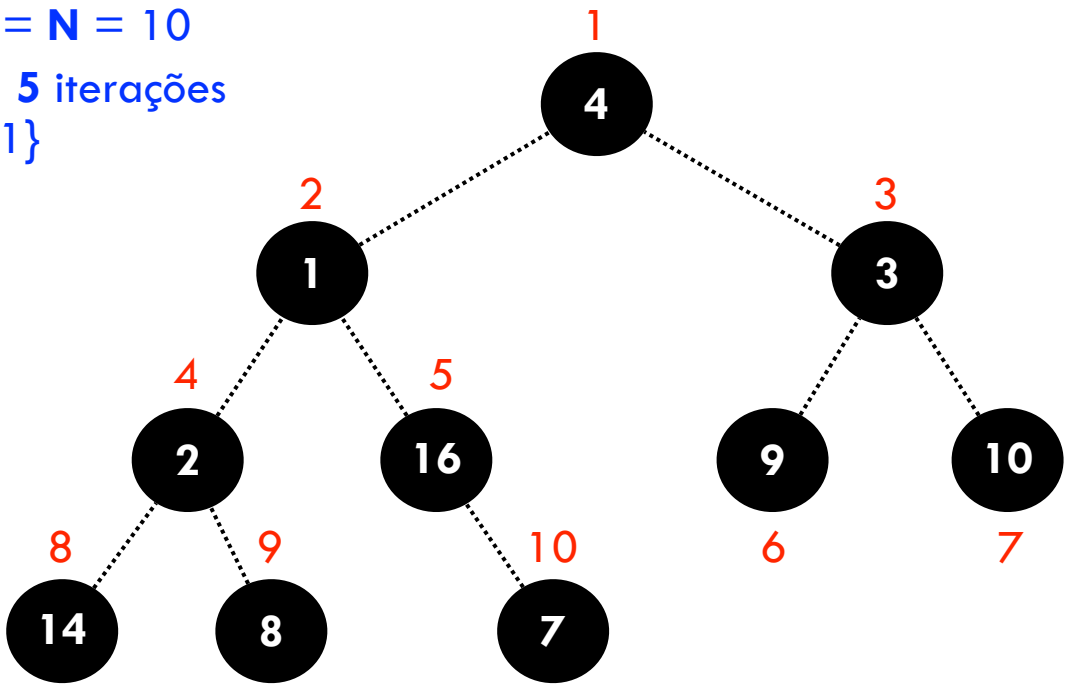
BUILD-MAX-HEAP

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

vetor.tamanho = **N** = 10

N/2 = 10/2 = **5** iterações

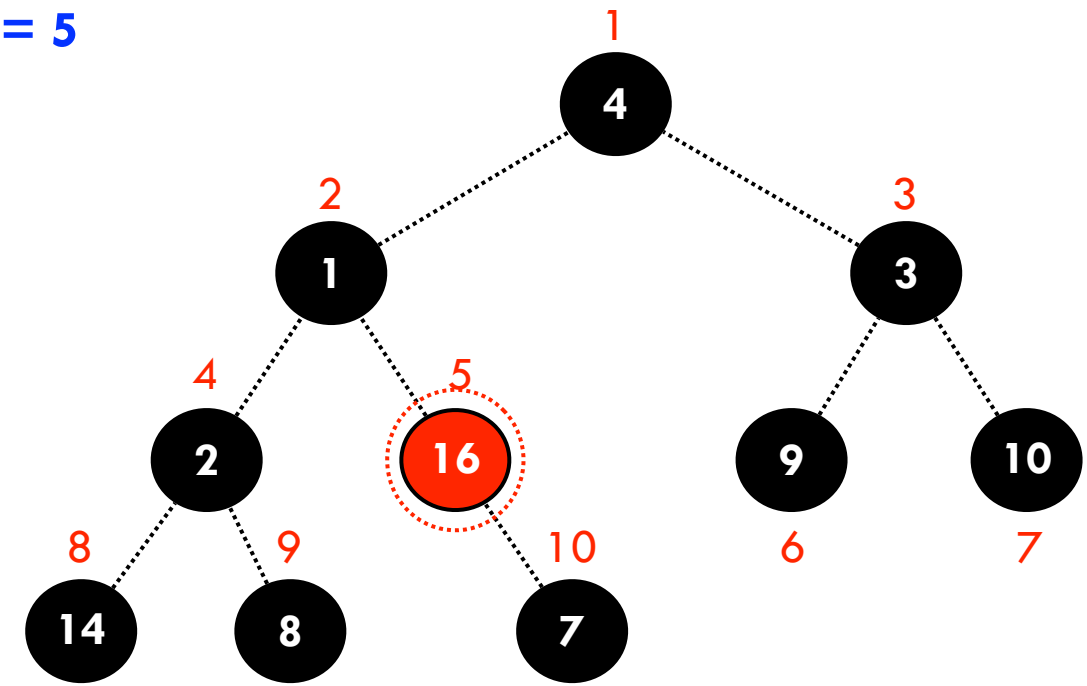
i = {5, 2, 3, 4, 1}



BUILD-MAX-HEAP

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

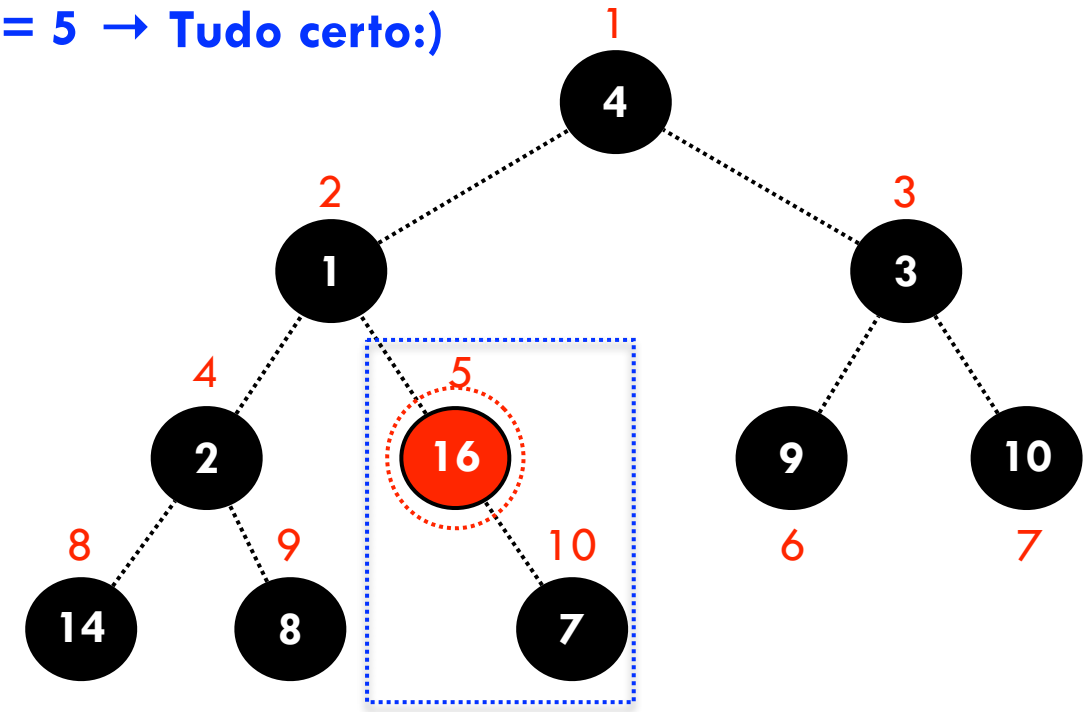
- iteração **i = 5**



BUILD-MAX-HEAP

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

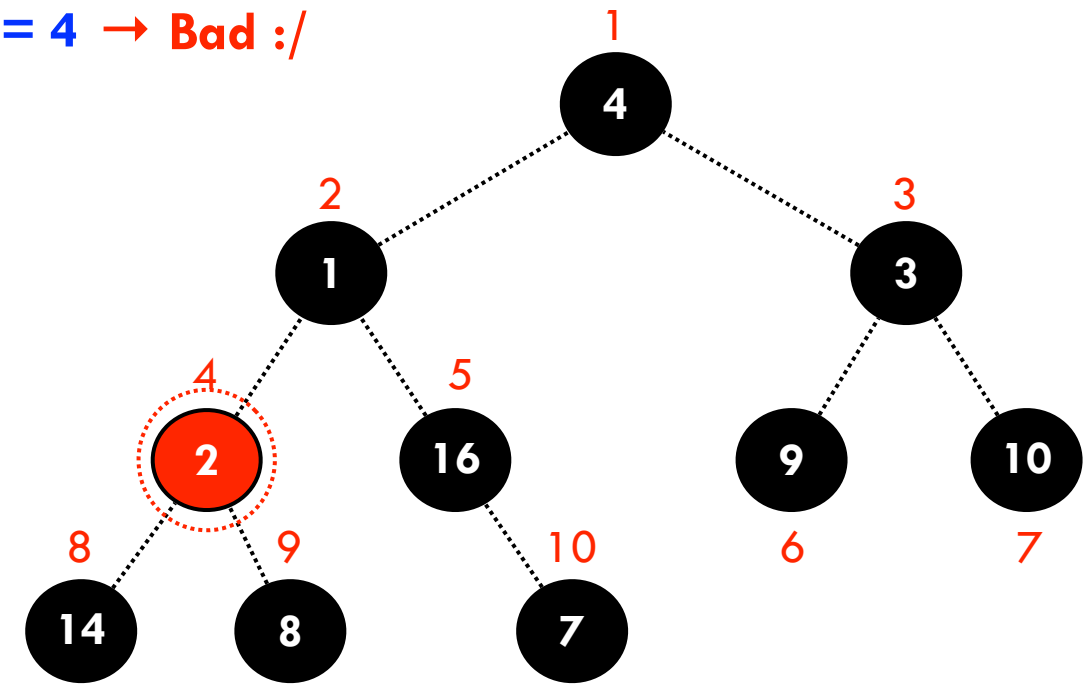
- iteração **i = 5 → Tudo certo:)**



BUILD-MAX-HEAP

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

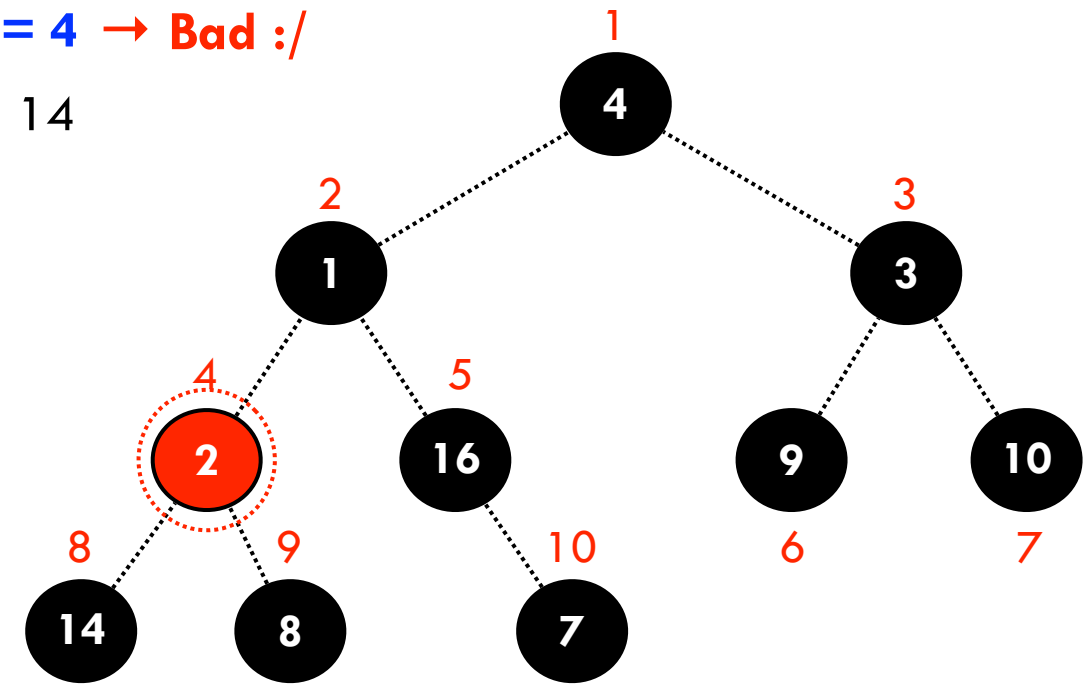
- iteração **i = 4** → **Bad :/**



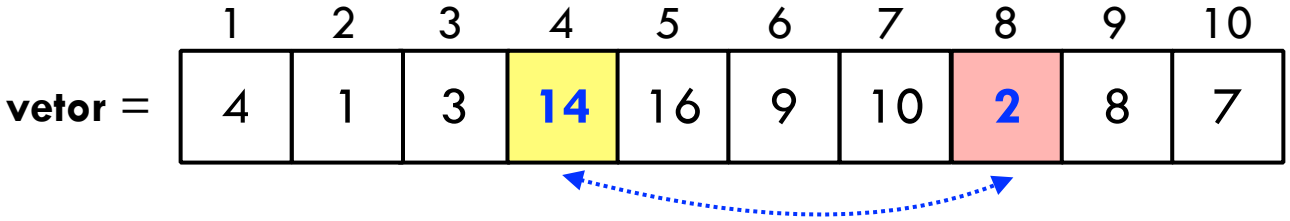
BUILD-MAX-HEAP

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

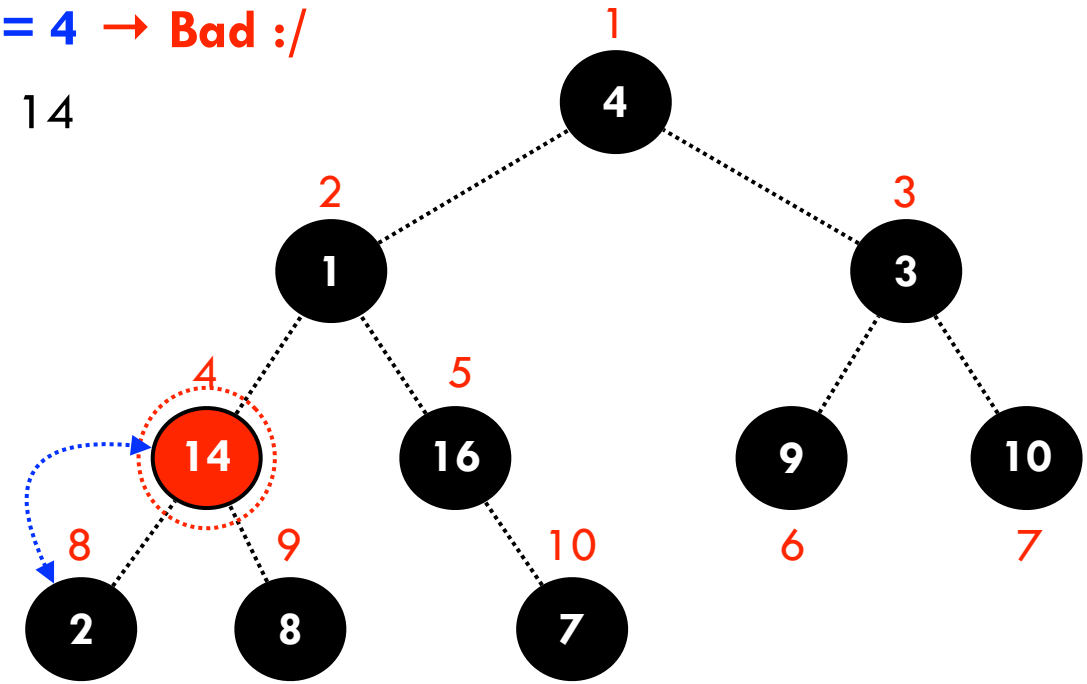
- iteração **i = 4** → **Bad :/**
- Trocar 2 e 14



BUILD-MAX-HEAP



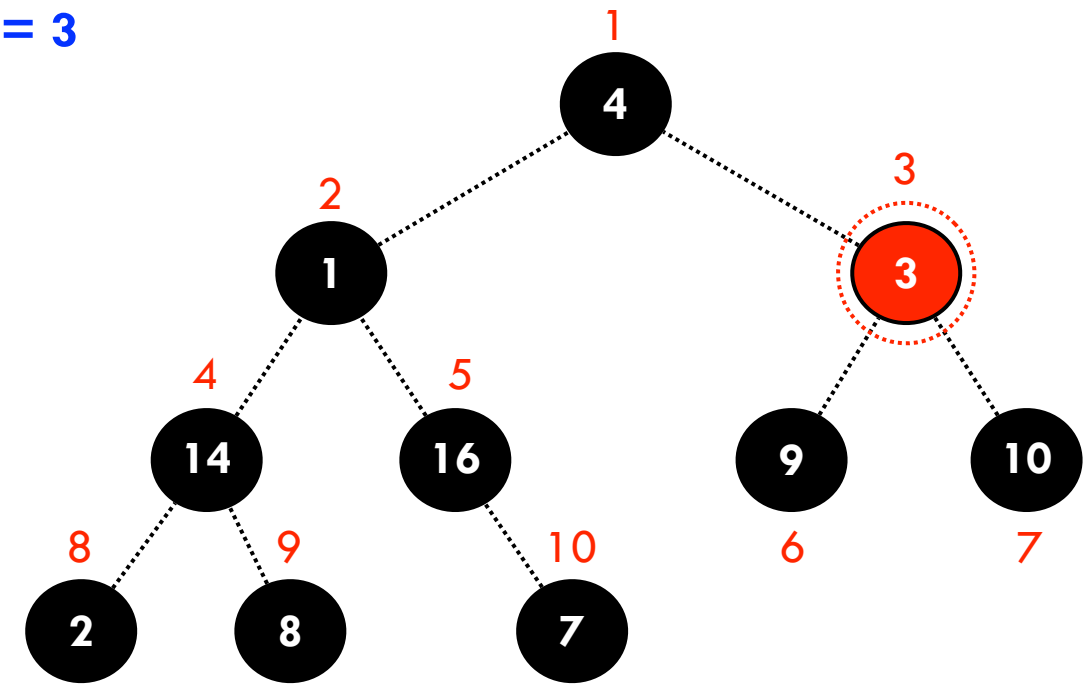
- iteração **i = 4** → **Bad :/**
- Trocar 2 e 14



BUILD-MAX-HEAP

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

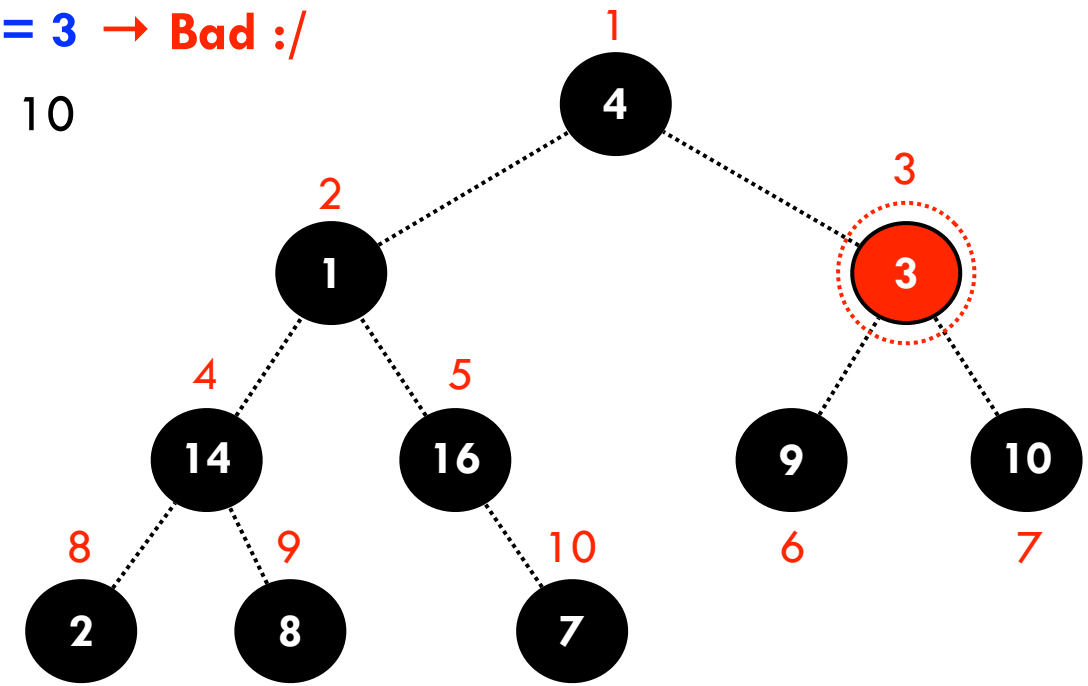
- iteração **i = 3**



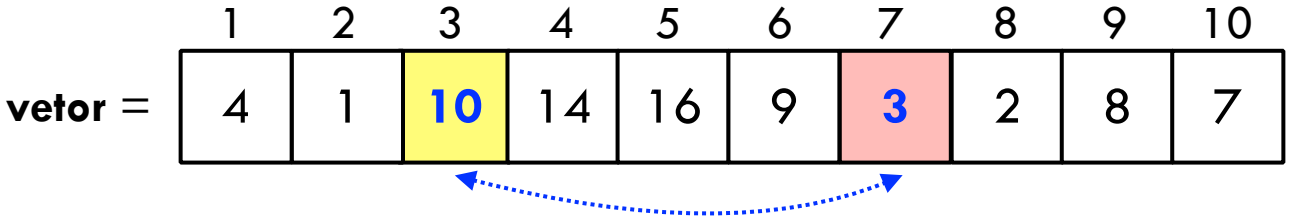
BUILD-MAX-HEAP

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

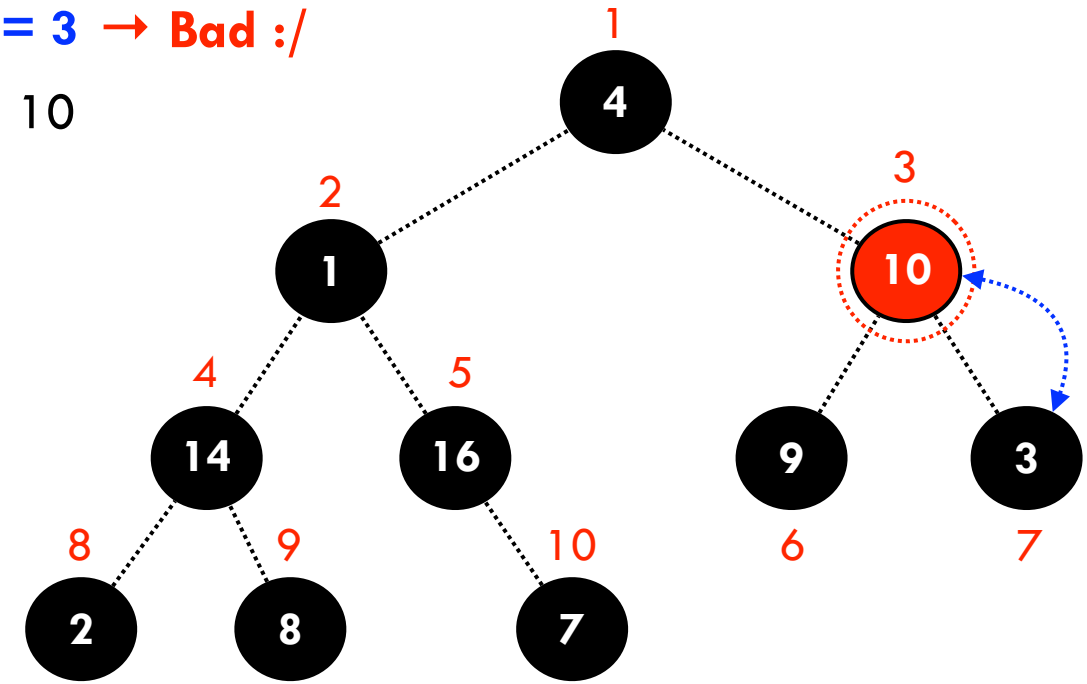
- iteração **i = 3** → **Bad :/**
- Trocar 3 e 10



BUILD-MAX-HEAP



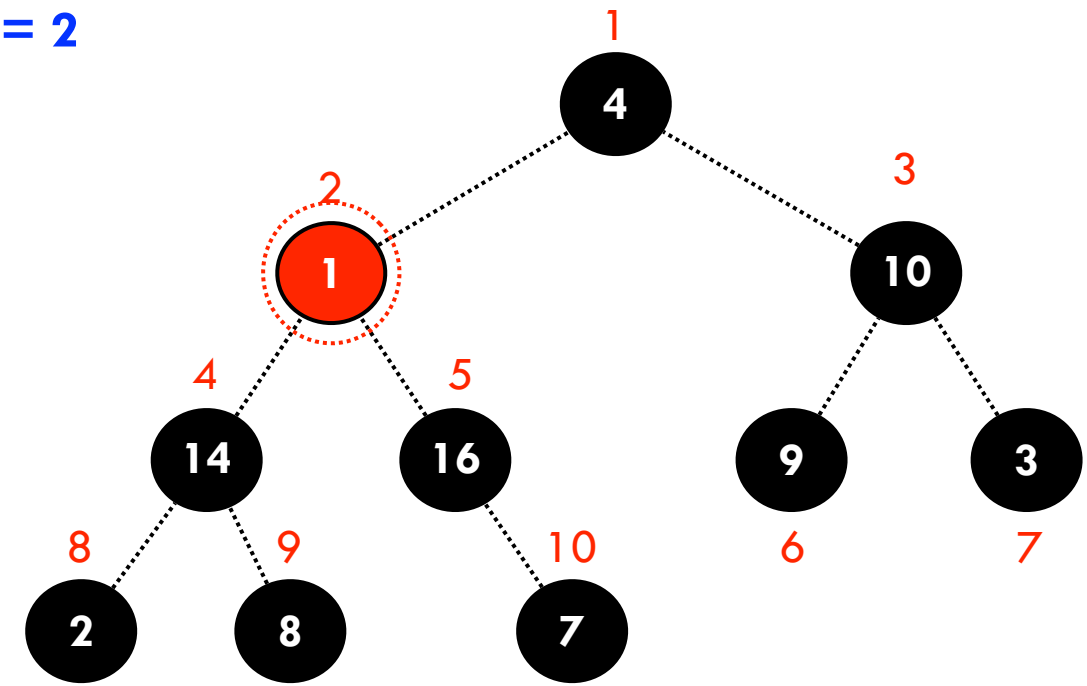
- iteração **i = 3** → **Bad :/**
- Trocar 3 e 10



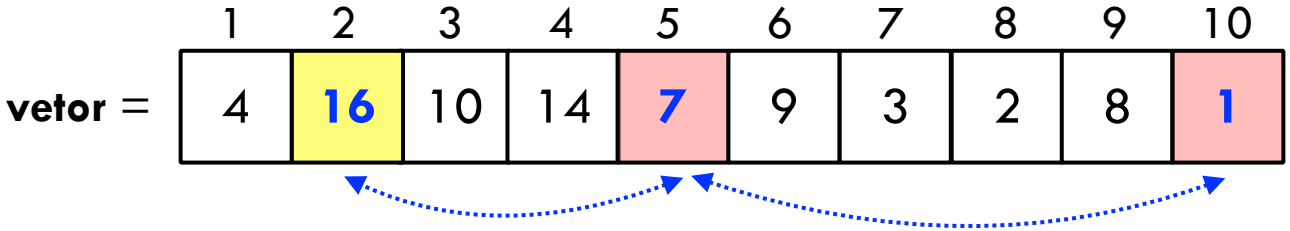
BUILD-MAX-HEAP

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

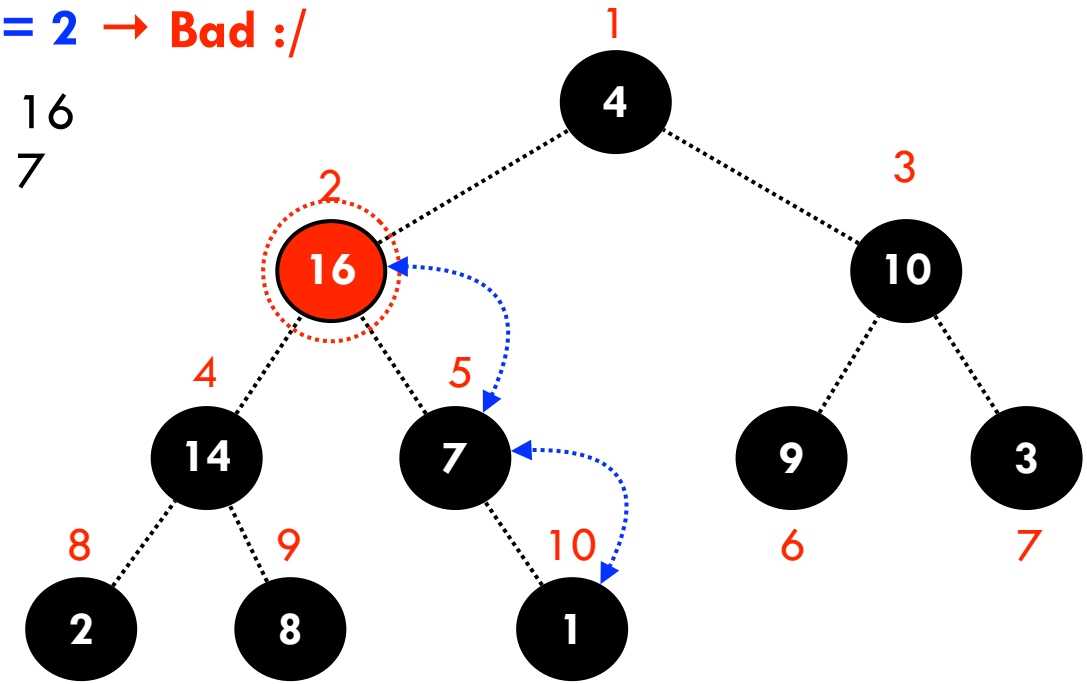
- iteração **i = 2**



BUILD-MAX-HEAP



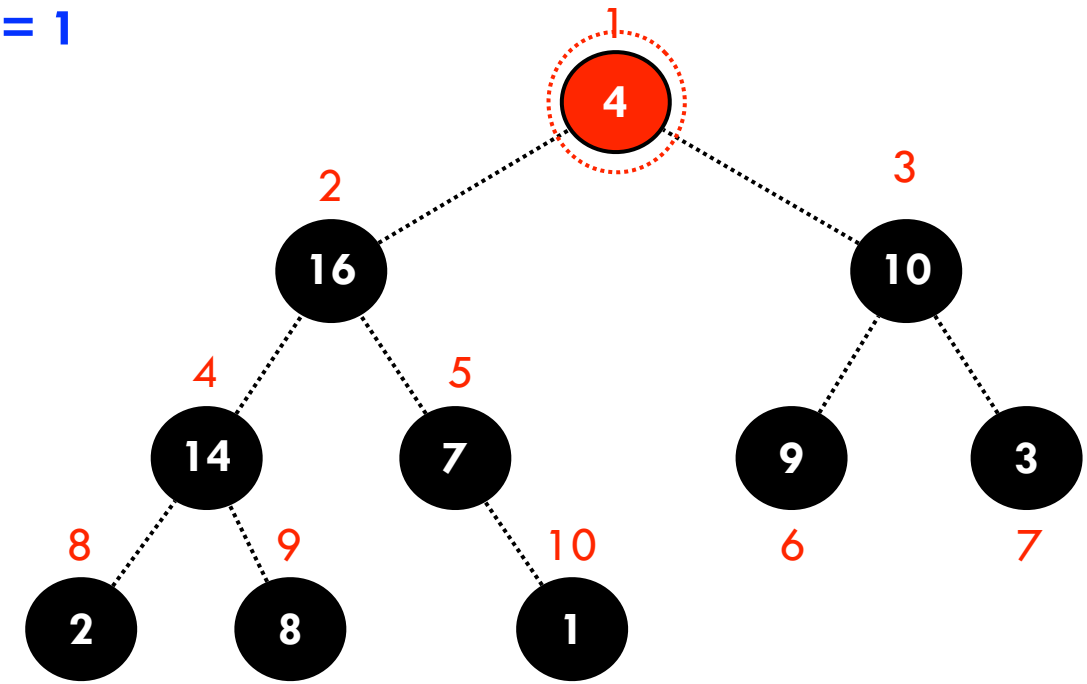
- iteração **i = 2** → **Bad :/**
- Trocar 1 e 16
- Trocar 1 e 7



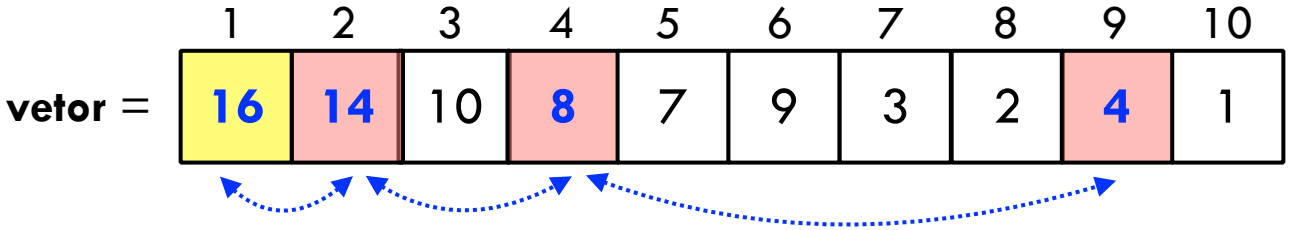
BUILD-MAX-HEAP

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

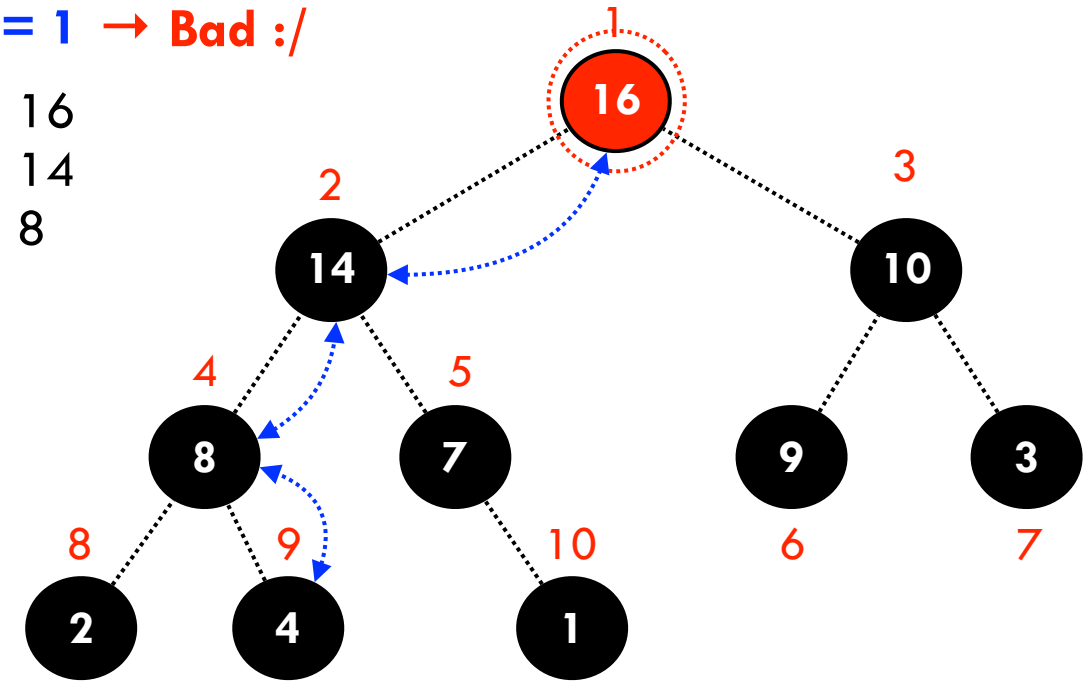
- iteração **i = 1**



BUILD-MAX-HEAP



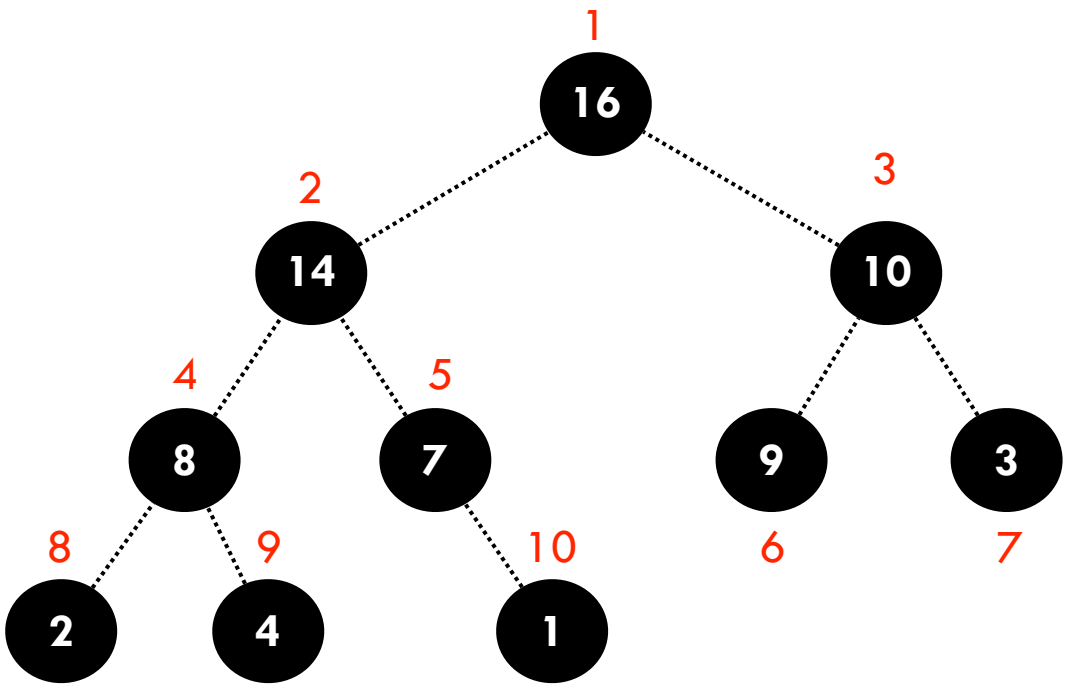
- iteração i = 1 → Bad :/
- Trocar 4 e 16
- Trocar 4 e 14
- Trocar 4 e 8



BUILD-MAX-HEAP

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

- Final!



Heap Sort

HEAPSORT(vetor):

```
//cria um heap de máximo
1. BUILD-MAX-HEAP(vetor)
```

Heap Sort

HEAPSORT(vetor):

```
//cria um heap de máximo
1. BUILD-MAX-HEAP(vetor)
2. Para i de vetor.tamanho até 2, faça:
```

Heap Sort

HEAPSORT(vetor):

```
   //cria um heap de máximo
1. BUILD-MAX-HEAP(vetor)
2. Para i de vetor.tamanho até 2, faça:
       // "remove" o elemento da primeira posição
3.     Trocar os elementos das posições vetor[ 1 ] e vetor [ i ]
4.     vetor.tamanho-heap  = vetor.tamanho-heap - 1
```

Heap Sort

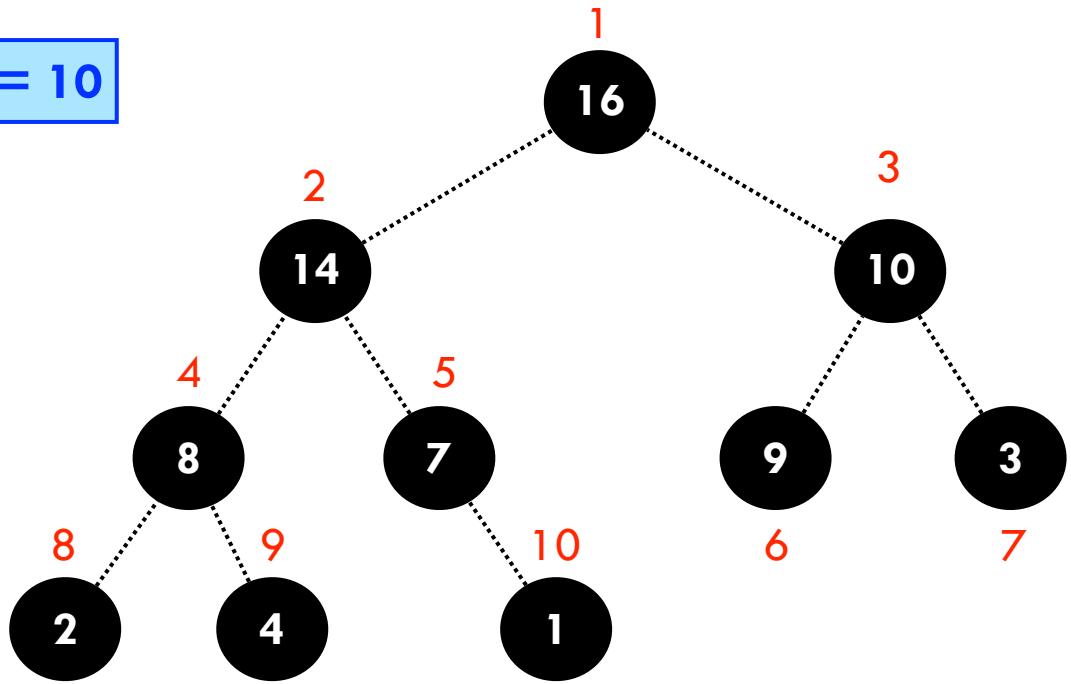
HEAPSORT(vetor):

```
    //cria um heap de máximo
1.  BUILD-MAX-HEAP(vetor)
2.  Para i de vetor.tamanho até 2, faça:
        // "remove" o elemento da primeira posição
3.      Trocar os elementos das posições vetor[ 1 ] e vetor [ i ]
4.      vetor.tamanho-heap  = vetor.tamanho-heap - 1
        // reconstrói o heap
5.      MAX-HEAPIFY (vetor, 1)
```

HEAP SORT

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

- iteração **i = 10**

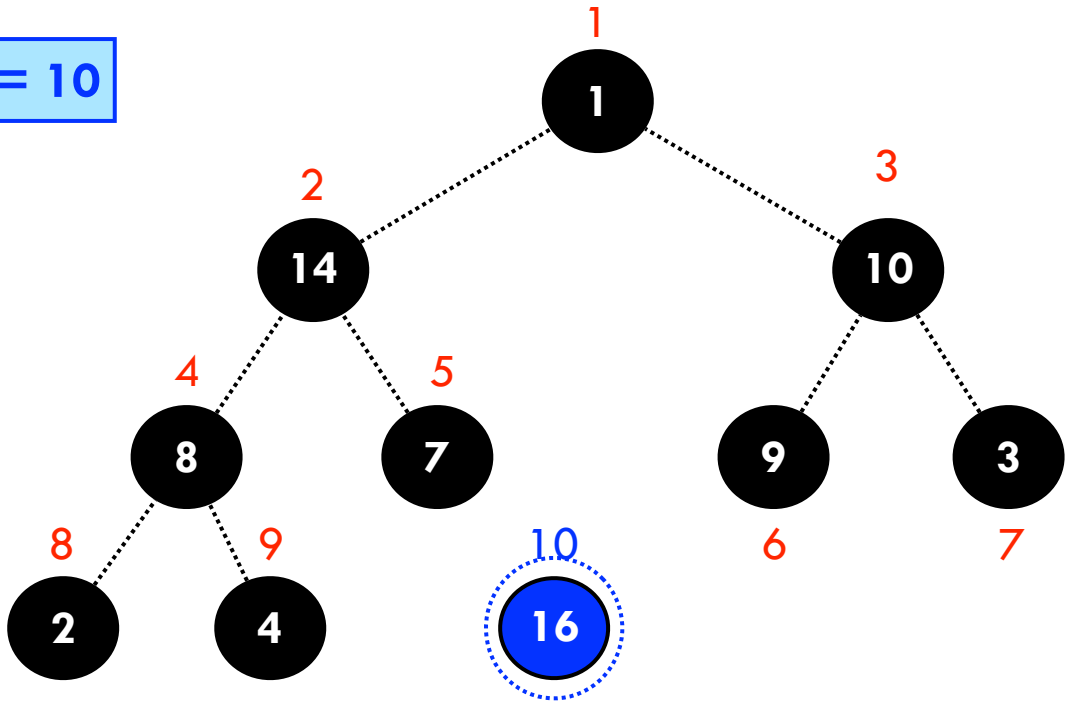


HEAP SORT

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

MAX-HEAPIFY ⟷

- iteração **i = 10**

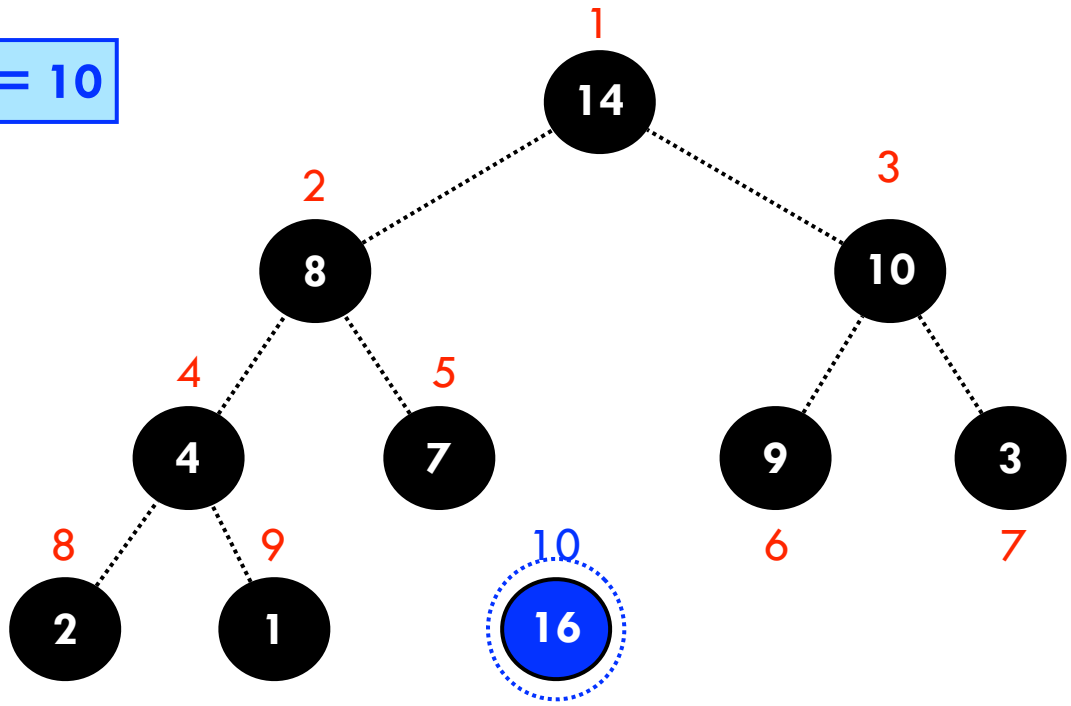


HEAP SORT

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

MAX-HEAPIFY ⟷

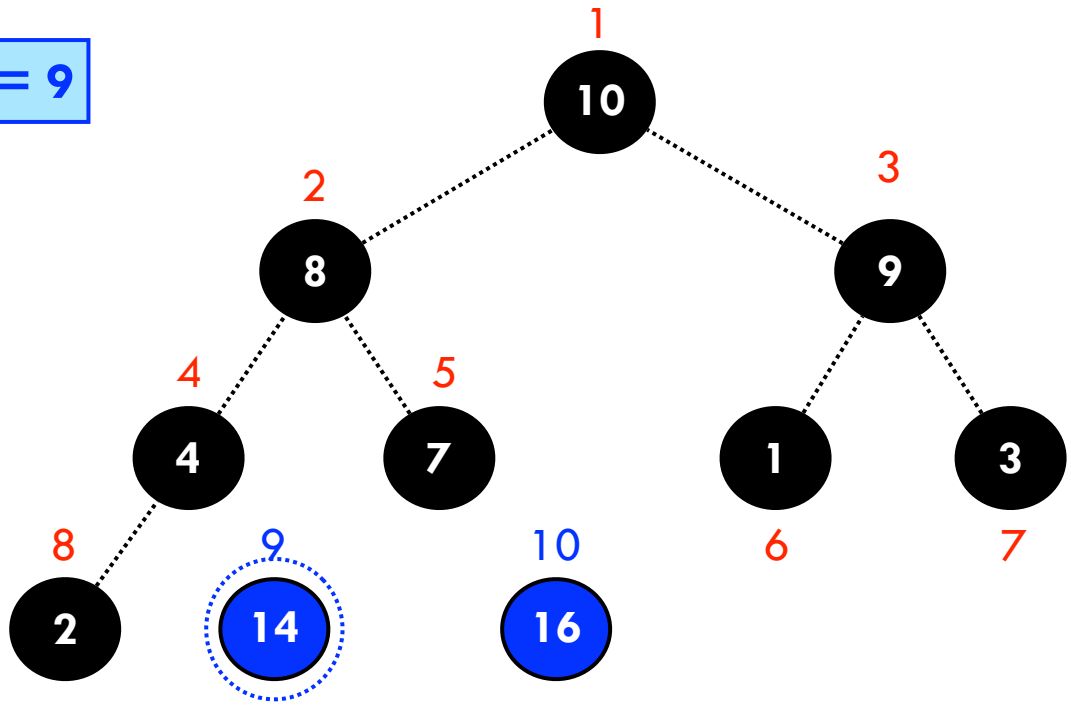
- iteração **i = 10**



HEAP SORT

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

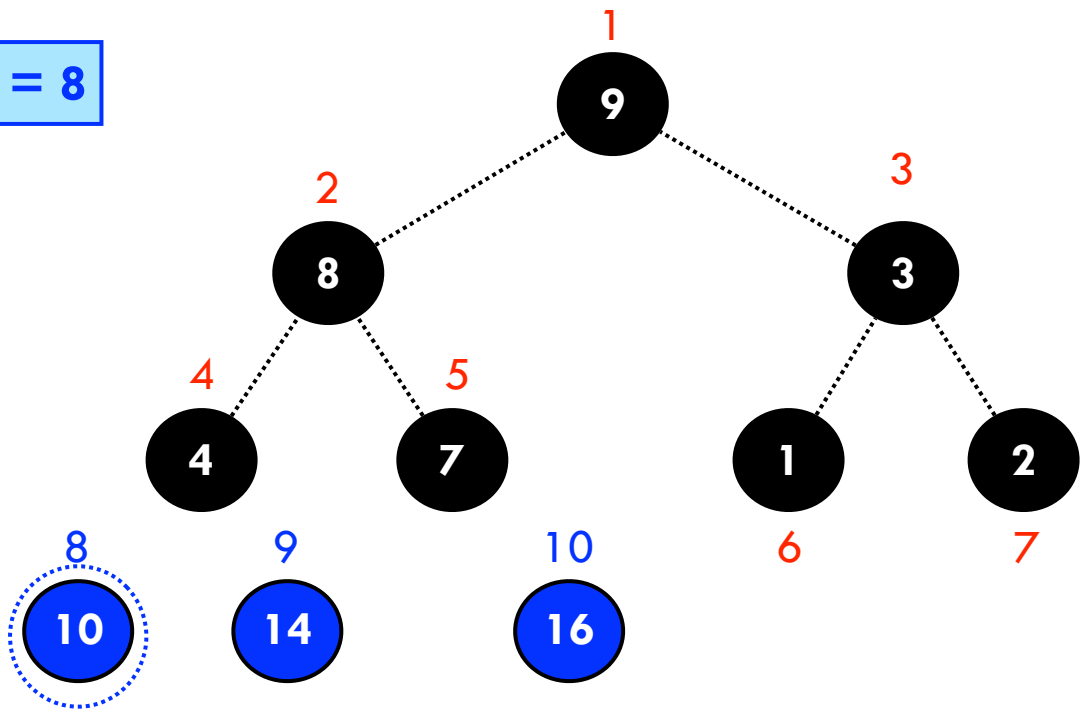
- iteração **i = 9**



HEAP SORT

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

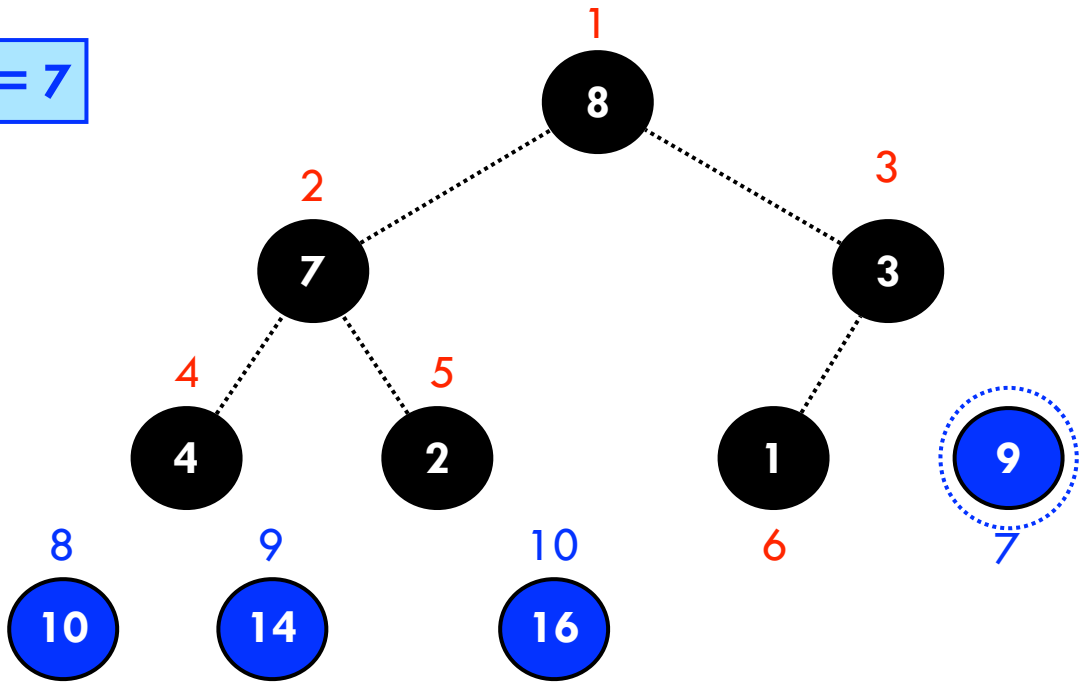
- iteração **i = 8**



HEAP SORT

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

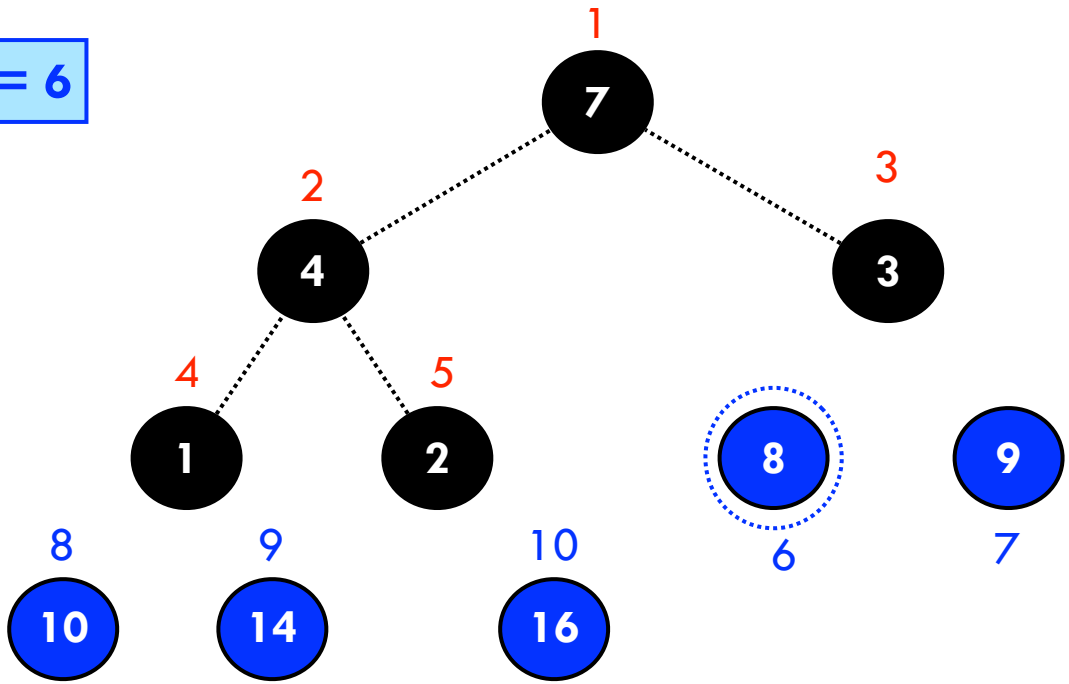
- iteração **i = 7**



HEAP SORT

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

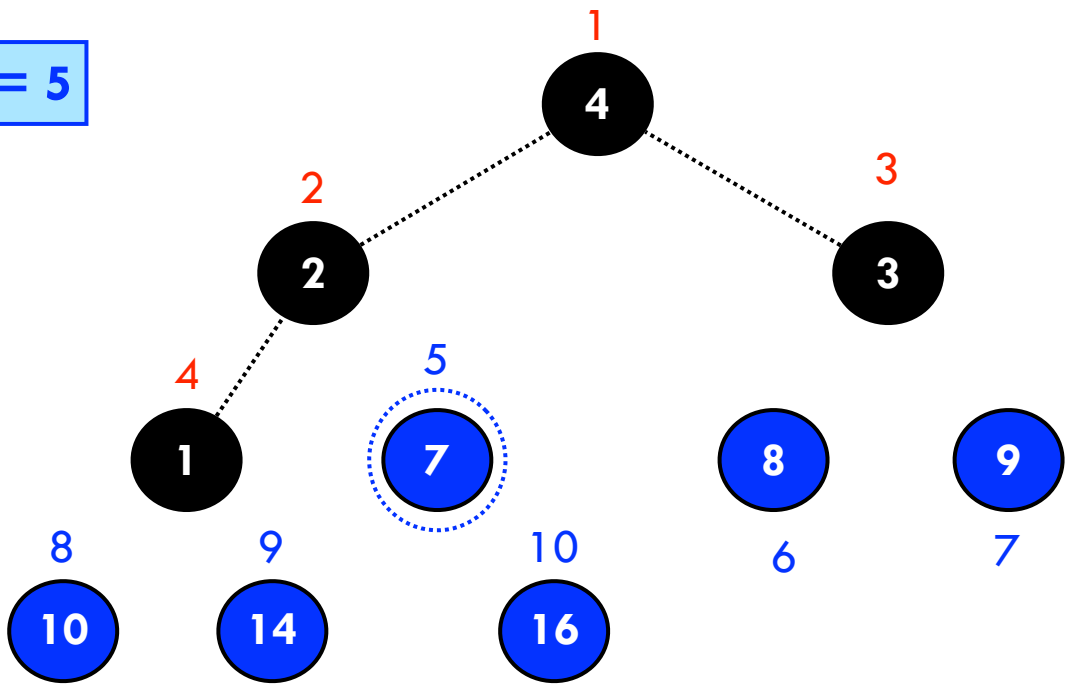
- iteração **i = 6**



HEAP SORT

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

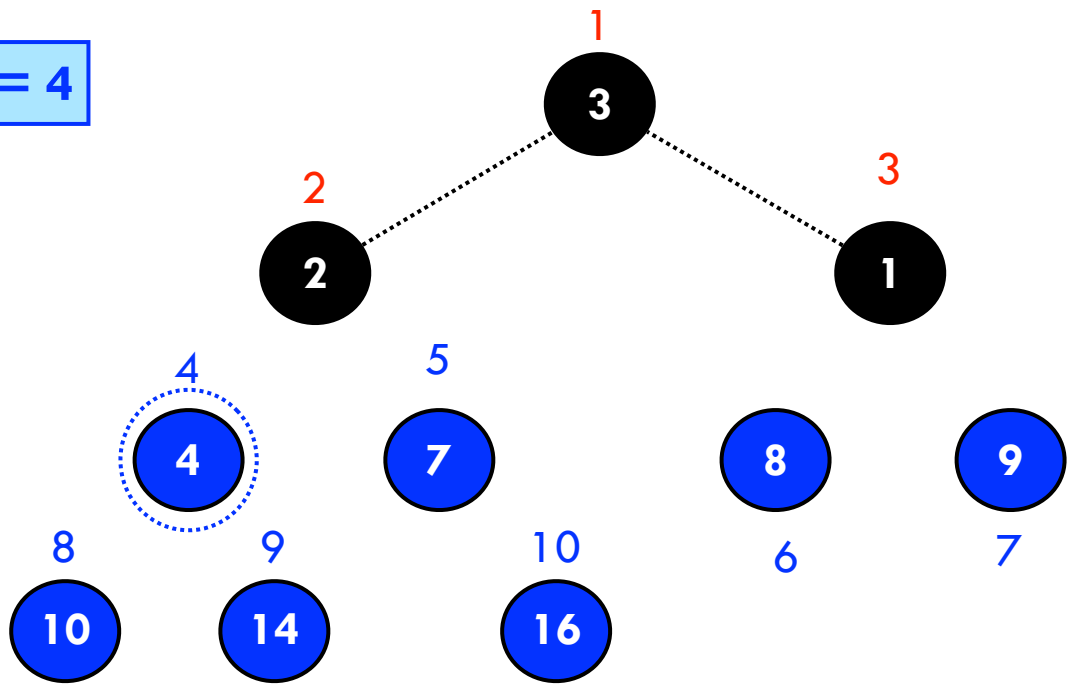
- iteração **i = 5**



HEAP SORT

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

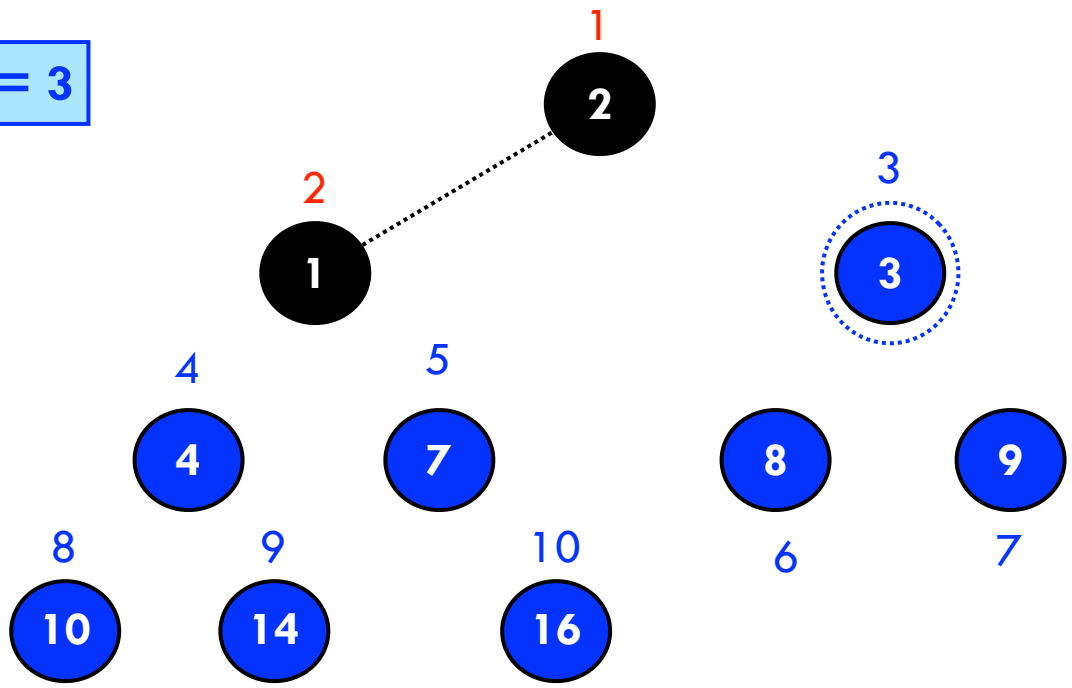
- iteração **i = 4**



HEAP SORT

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

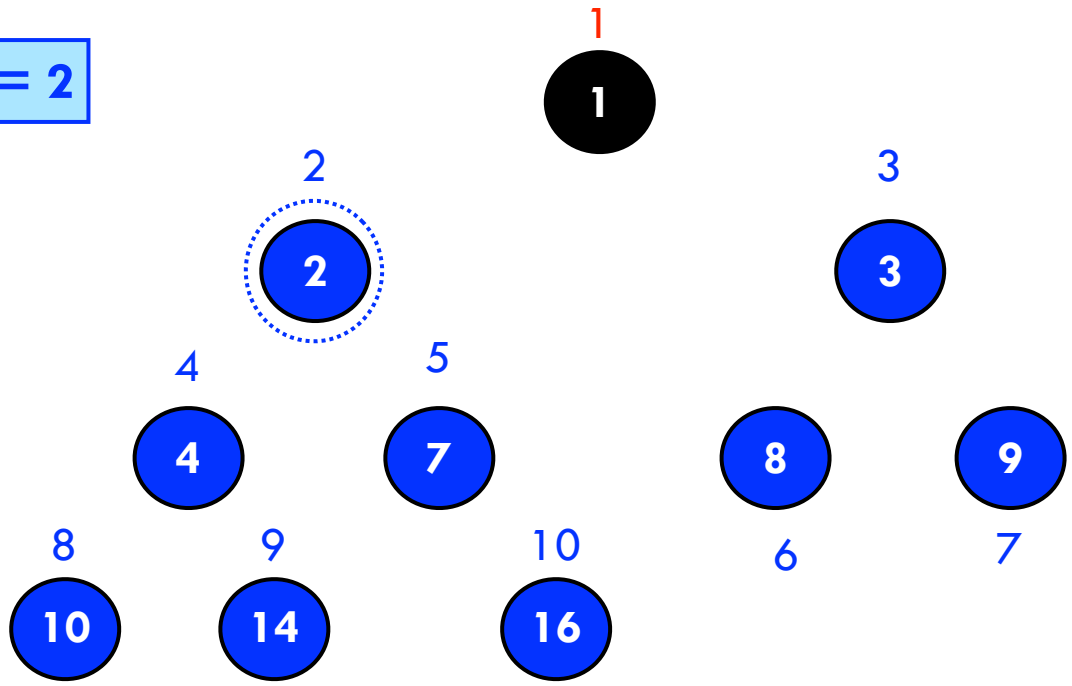
- iteração **i = 3**



HEAP SORT

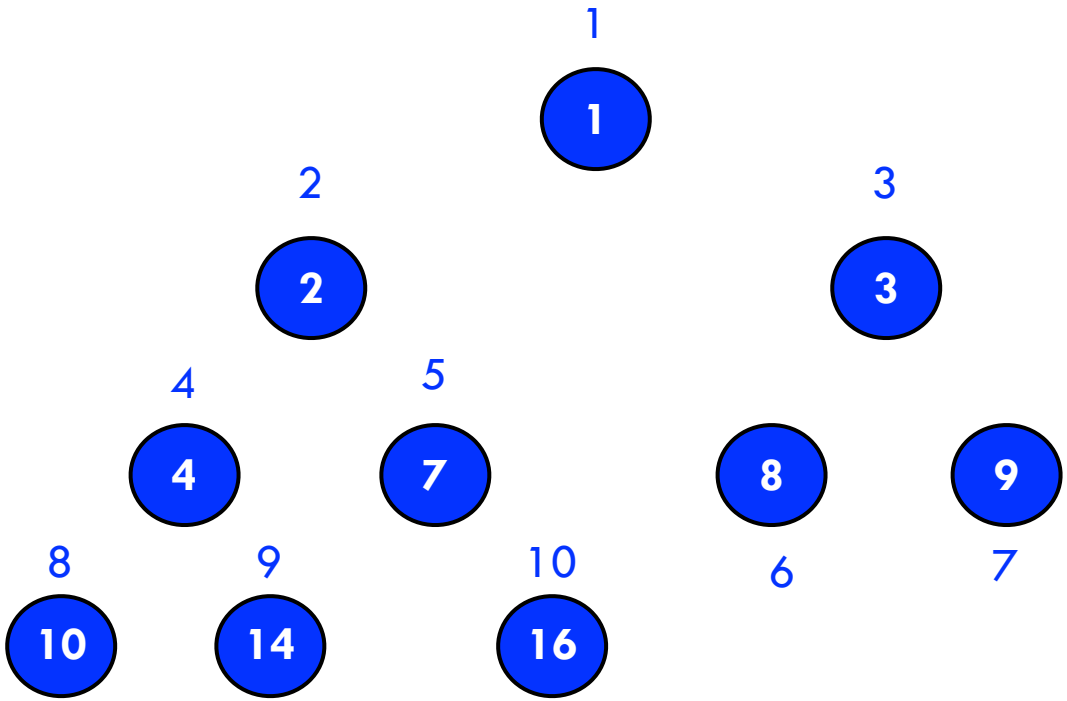
	1	2	3	4	5	6	7	8	9	10
vetor =	1	2	3	4	7	8	9	10	14	16

- iteração **i = 2**



HEAP SORT

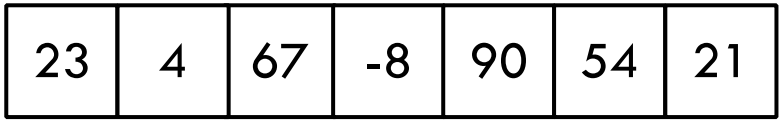
	1	2	3	4	5	6	7	8	9	10
vetor =	1	2	3	4	7	8	9	10	14	16



Roteiro

1. Introdução
2. Heaps
3. Heap Sort
4. Exemplo
5. Exercícios
6. Referências

Exemplo



vetor não ordenado

Roteiro

1. Introdução
2. Heaps
3. Heap Sort
4. Exemplo
5. Exercícios
6. Referências

Exercícios

HANDS ON :)))

Exercícios

1) Execute o teste de mesa (simulação) do algoritmo **Heap Sort** para a sua sequência de números aleatórios, definida na planilha da disciplina.

Exercícios

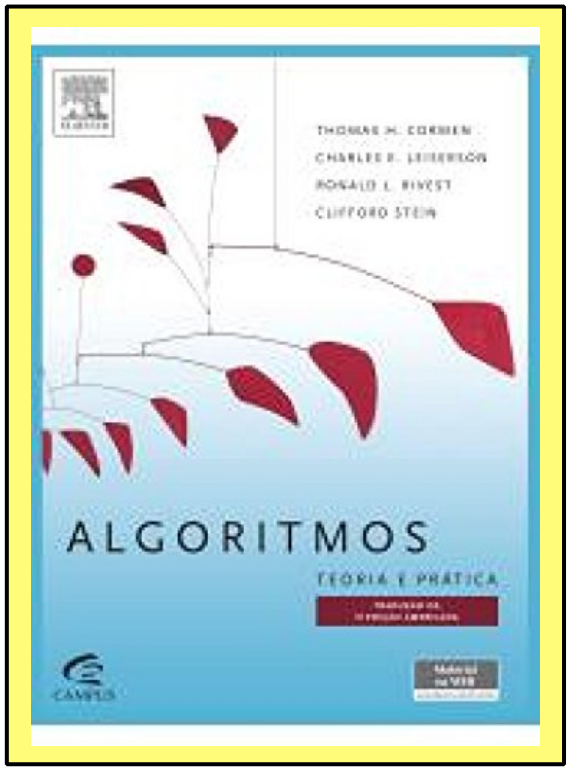
2) Implemente o **heapSort** em Python considerando as seguintes funções

```
/* função principal */
def heapSort(array)

/* constrói heap de máximo a partir do vetor */
def buildMaxHeap(array)

/* reconstrói o heap, desconsiderando o elemento já ordenado */
def maxHeapify(array, i, heapsize)
```

Roteiro

1. Introdução
2. Heaps
3. Heap Sort
4. Exemplo
5. Exercícios
6. Referências

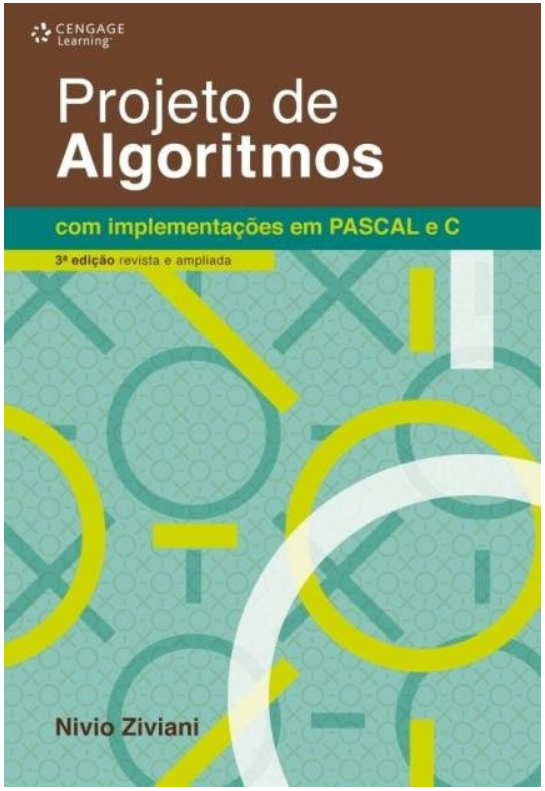
Referências sugeridas



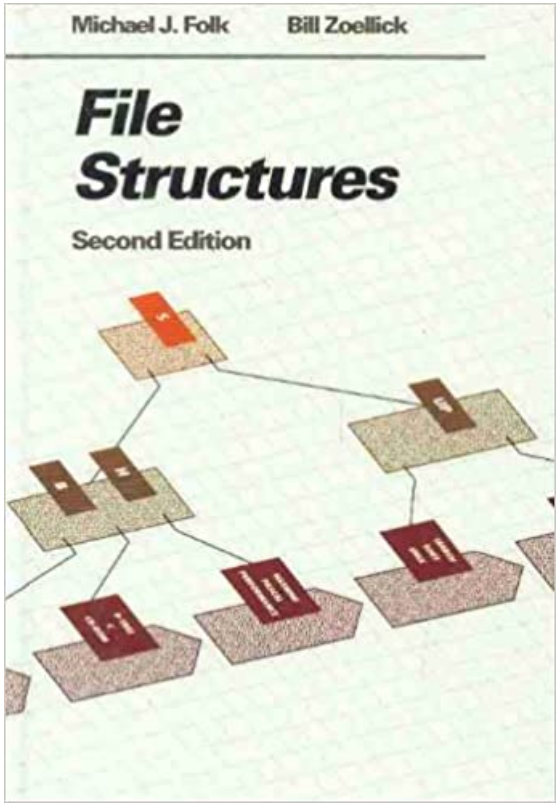
[Cormen et al, 2018]

[Drozdek, 2017]

Referências sugeridas



[Ziviani, 2010]



[Folk & Zoellick, 1992]

Perguntas?

Prof. Rafael G. **Mantovani**

rafaelmantovani@utfpr.edu.br