Algoritmos de Ordenação

HeapSort

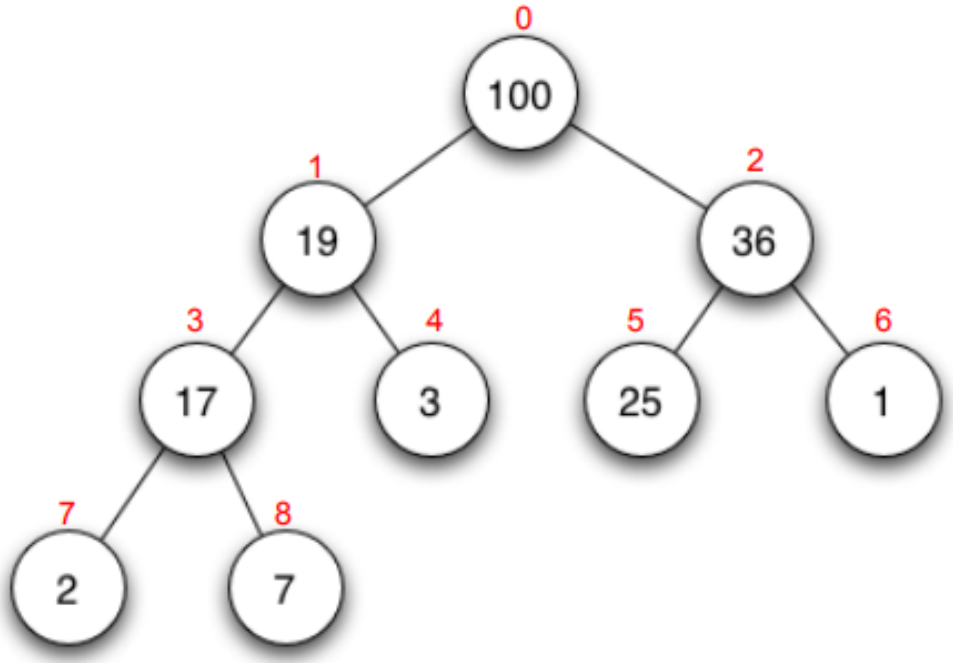
(Fonte: Material adaptado dos Slides do prof. Monael.)

Heap Sort

- É um método de ordenação que o faz através de sucessivas seleções do elemento correto.
- Utiliza um heap binário para manter os elementos.
  - Heap binário é uma árvore binária mantida na forma de vetor.
  - O heap é gerado e mantido no próprio vetor a ser ordenado no segmento não ordenado.
- Prós: Heap Sort é estável. Quick Sort não.
- Contras: Construir o heap consome muita memória.

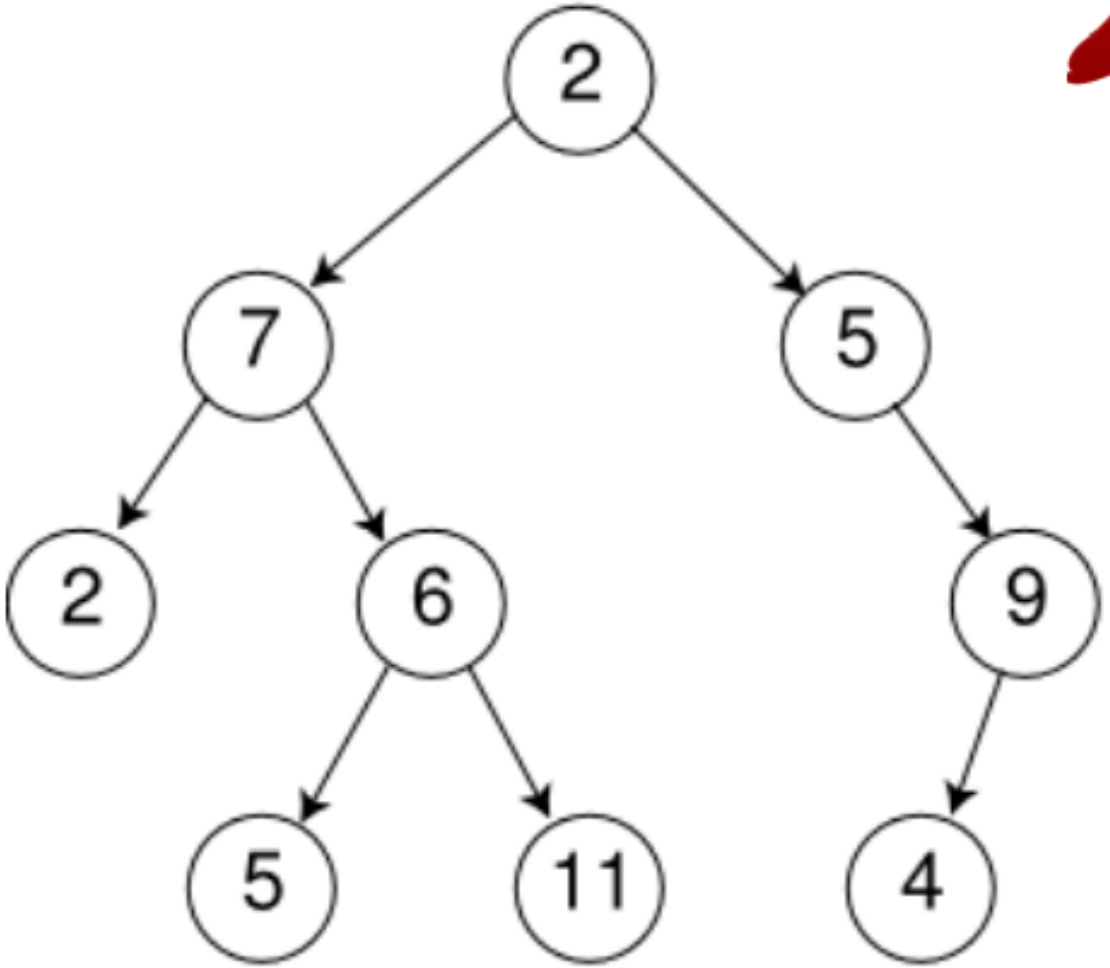
Heap Binário

Um heap-binário é uma árvore binária por níveis, onde o nó pai de uma subárvore sempre é maior ou igual os seus descendentes

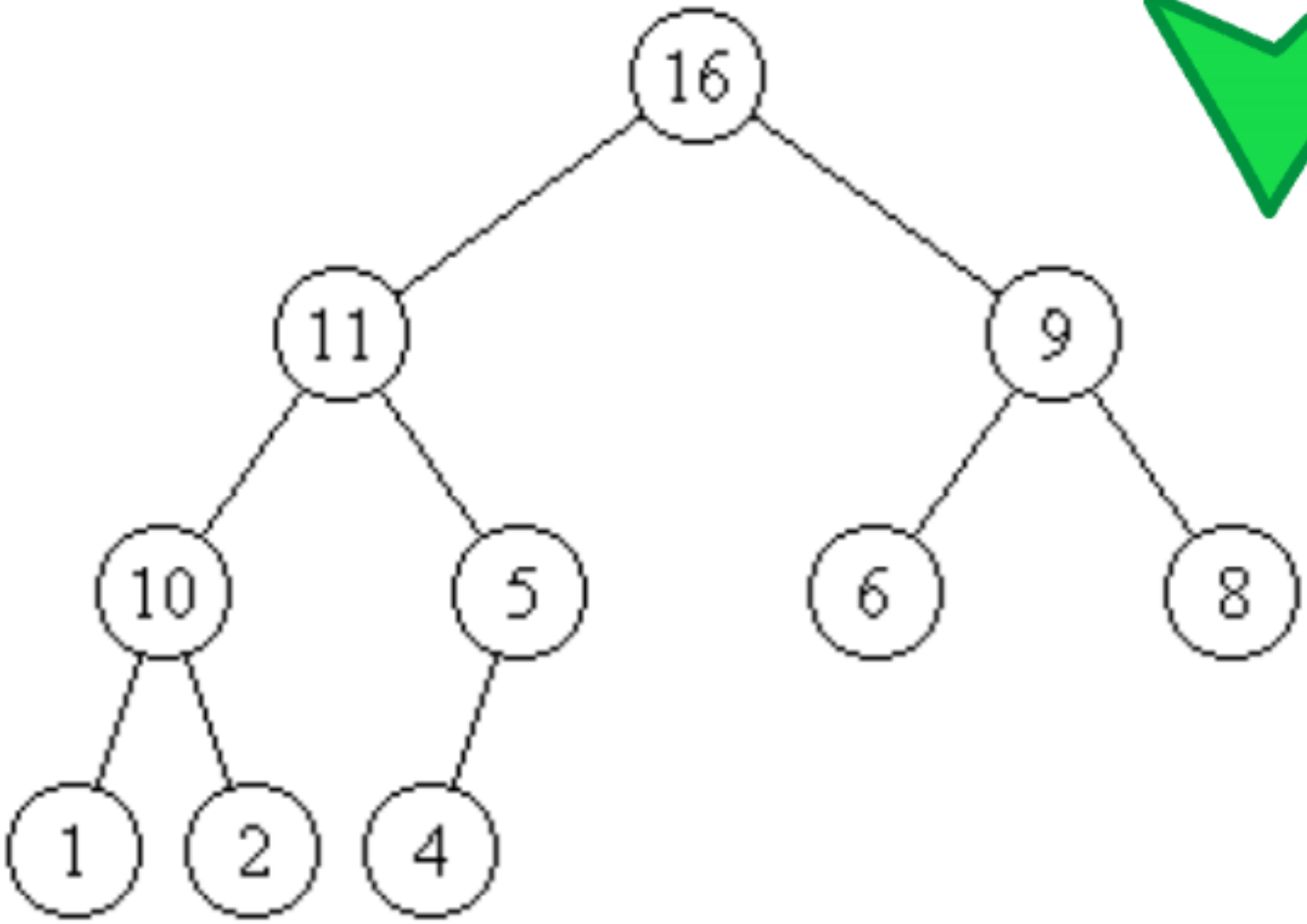


0	1	2	3	4	5	6	7	8
100	19	36	17	3	25	1	2	7

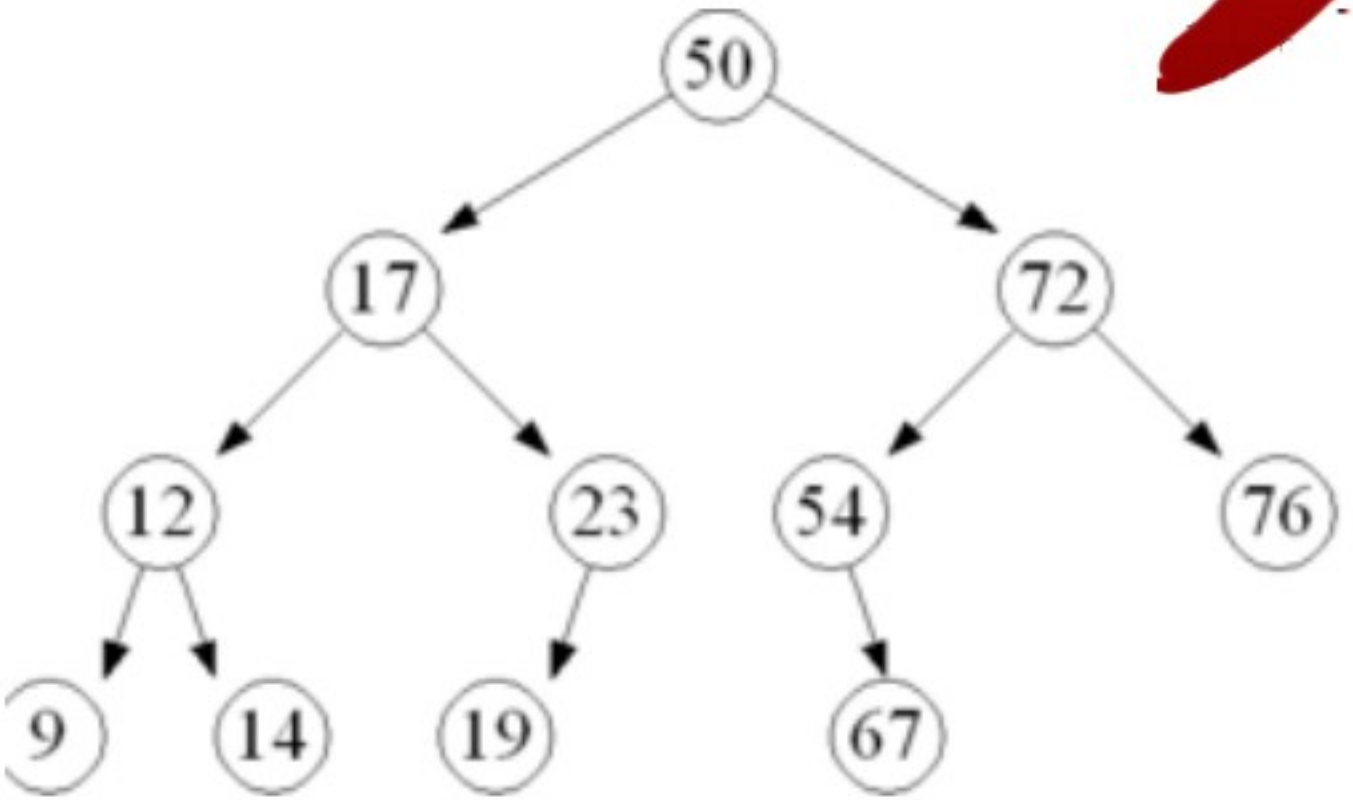
É Heap?



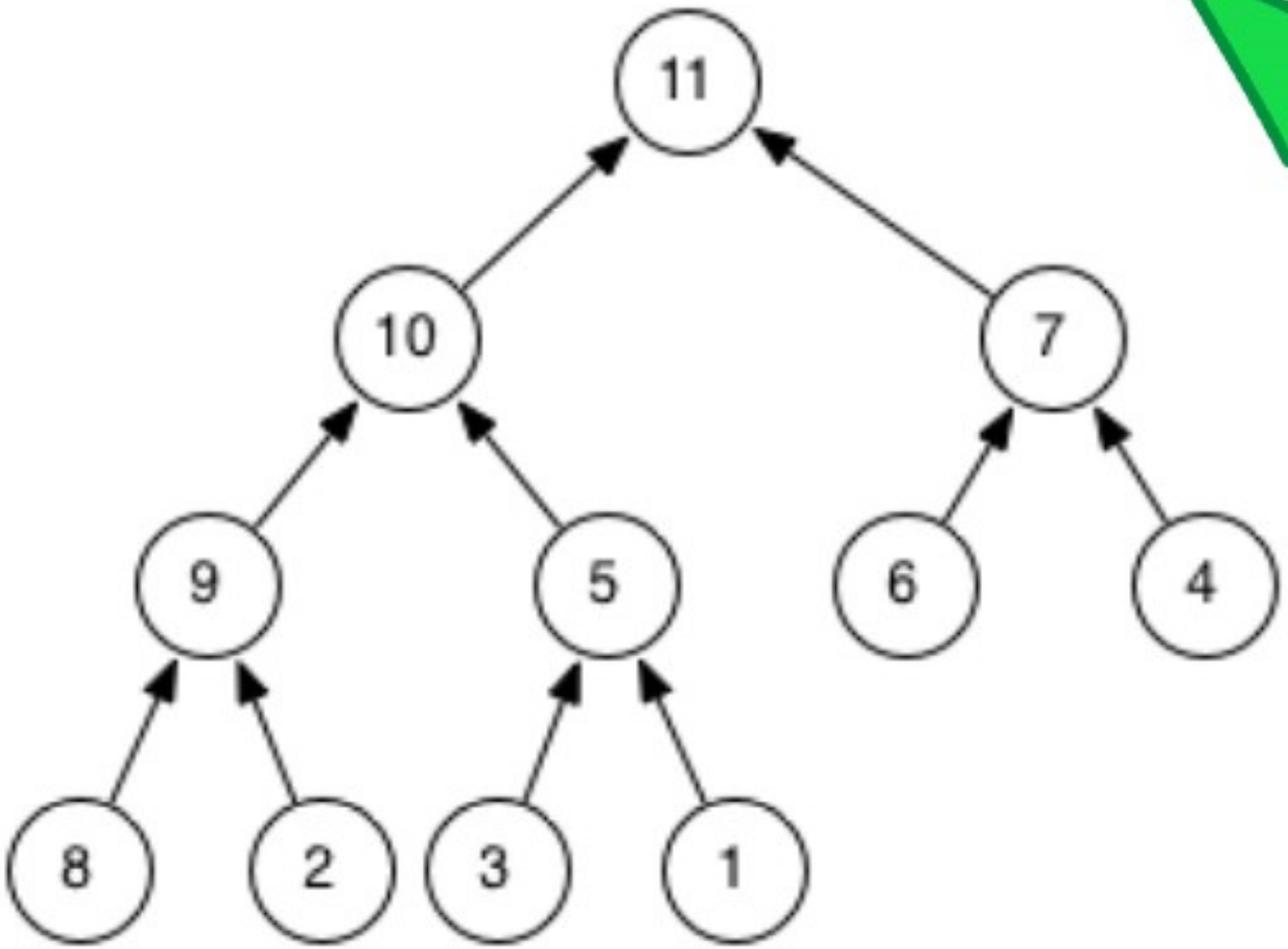
É Heap?



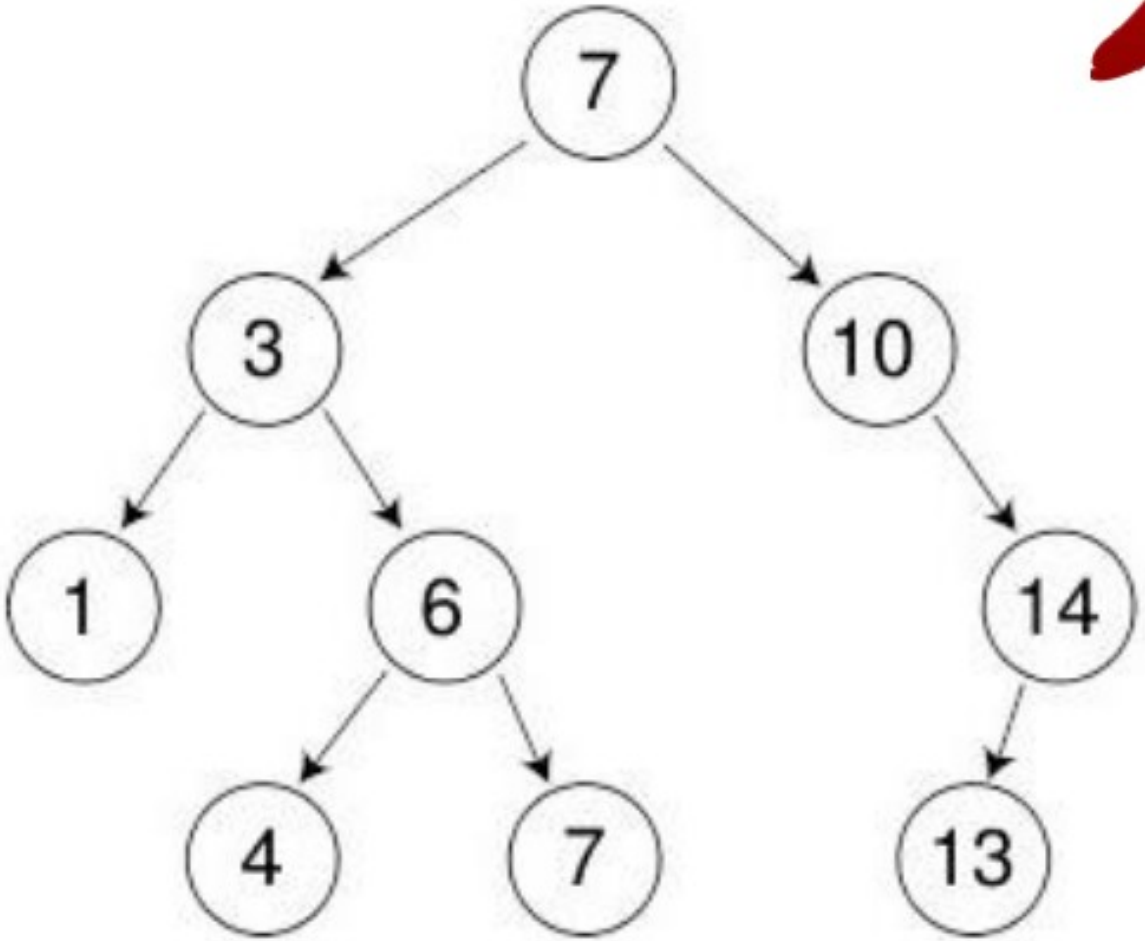
É Heap?



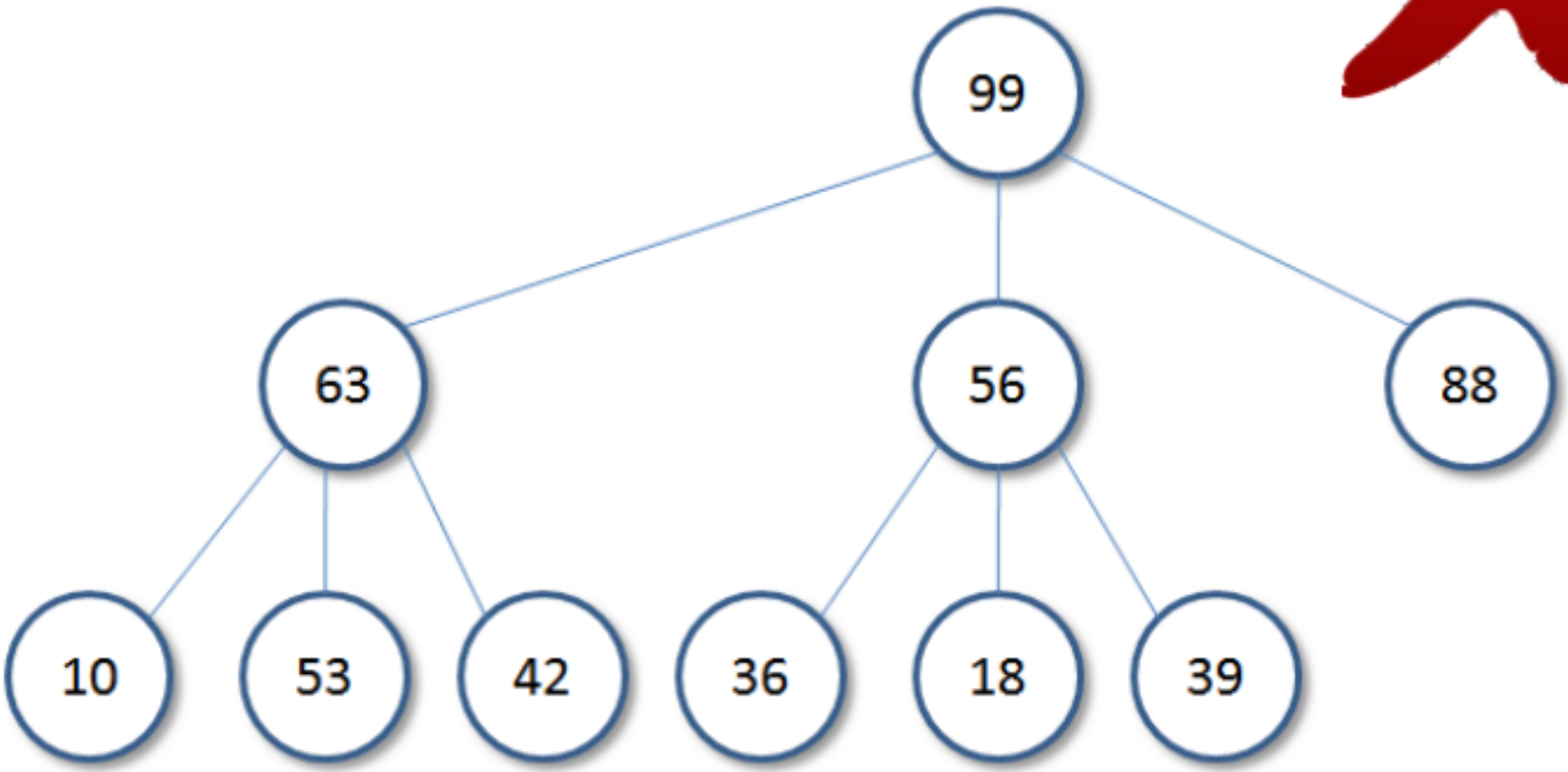
É Heap?



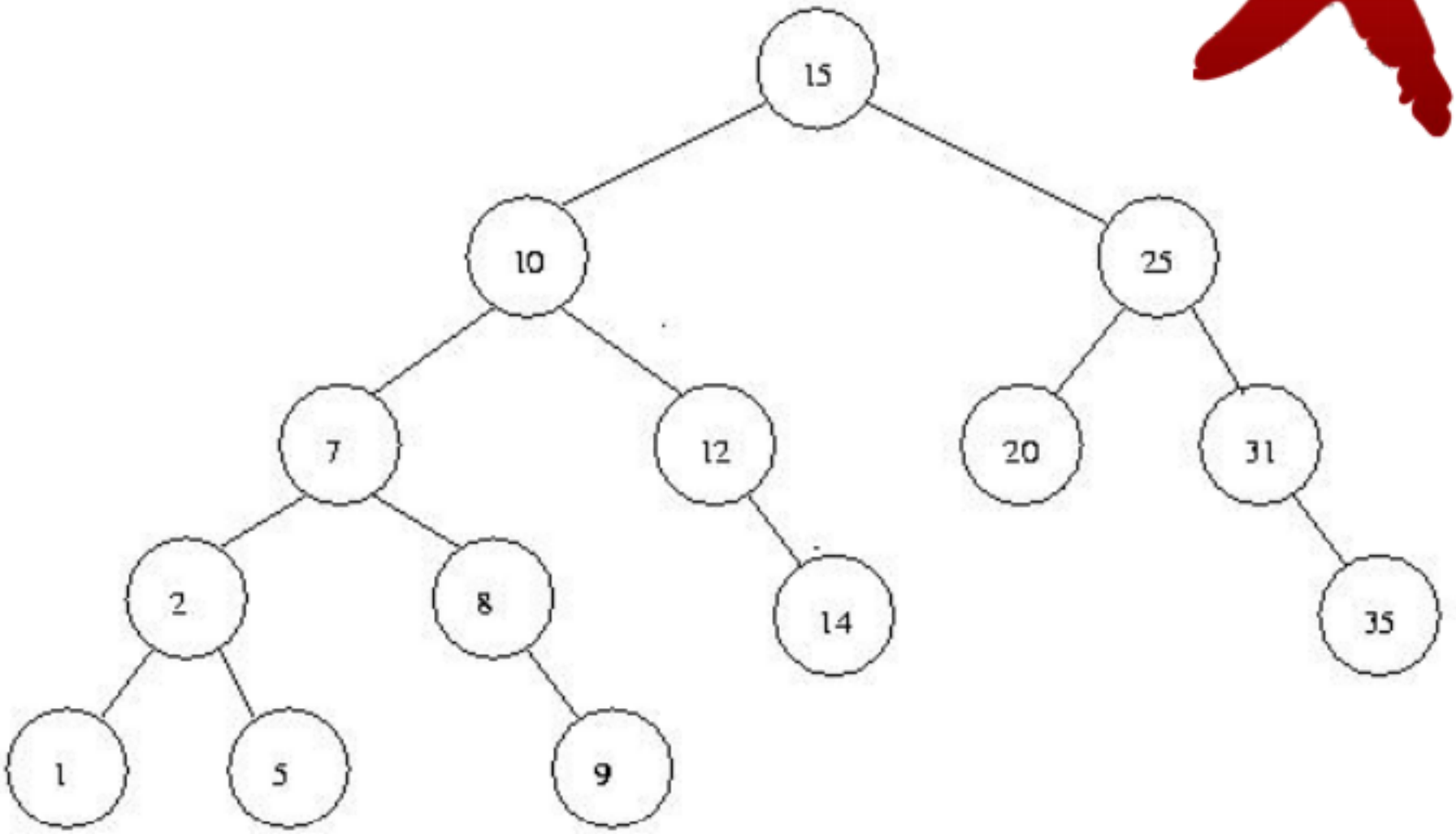
É Heap?



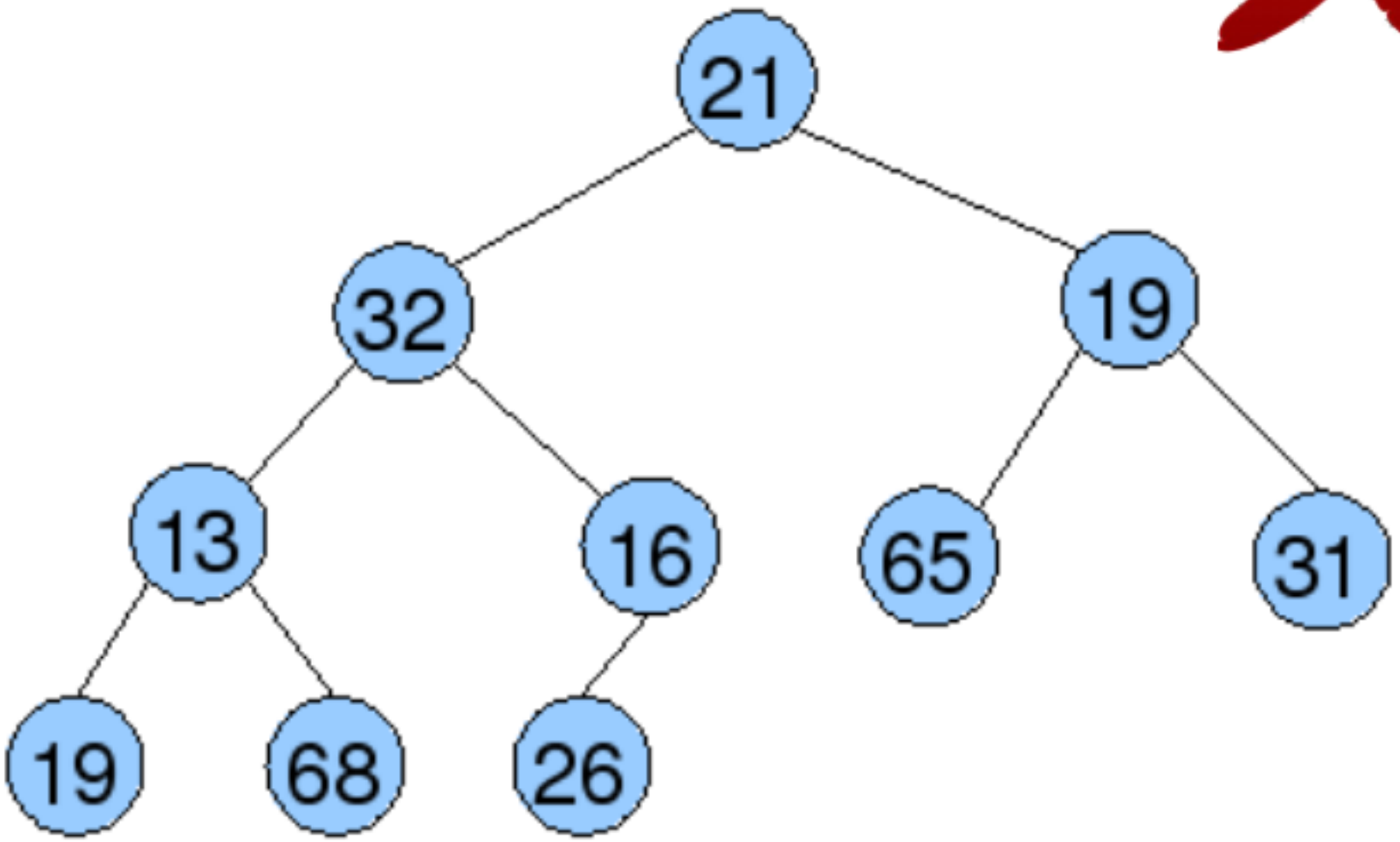
É Heap?



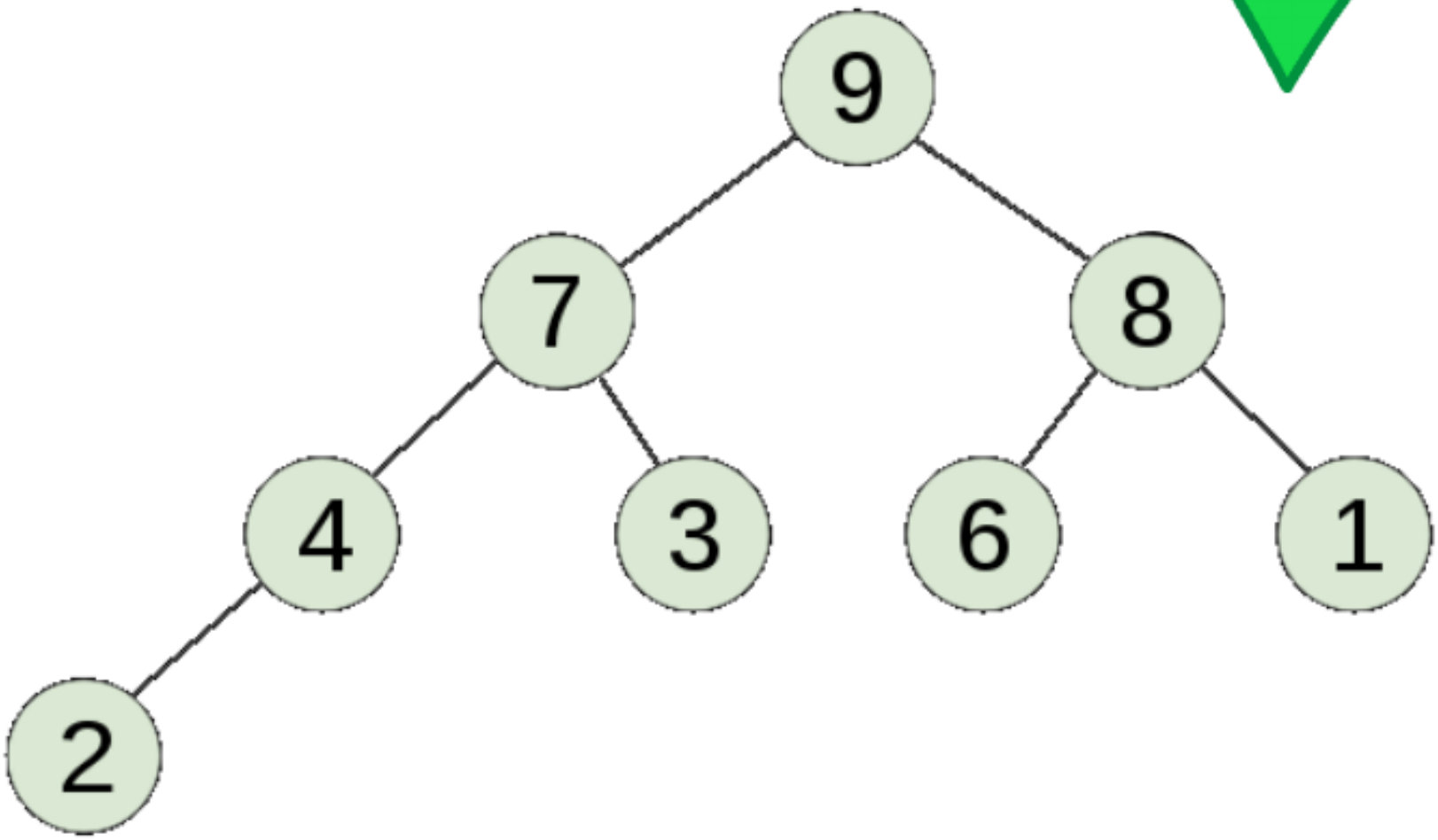
É Heap?



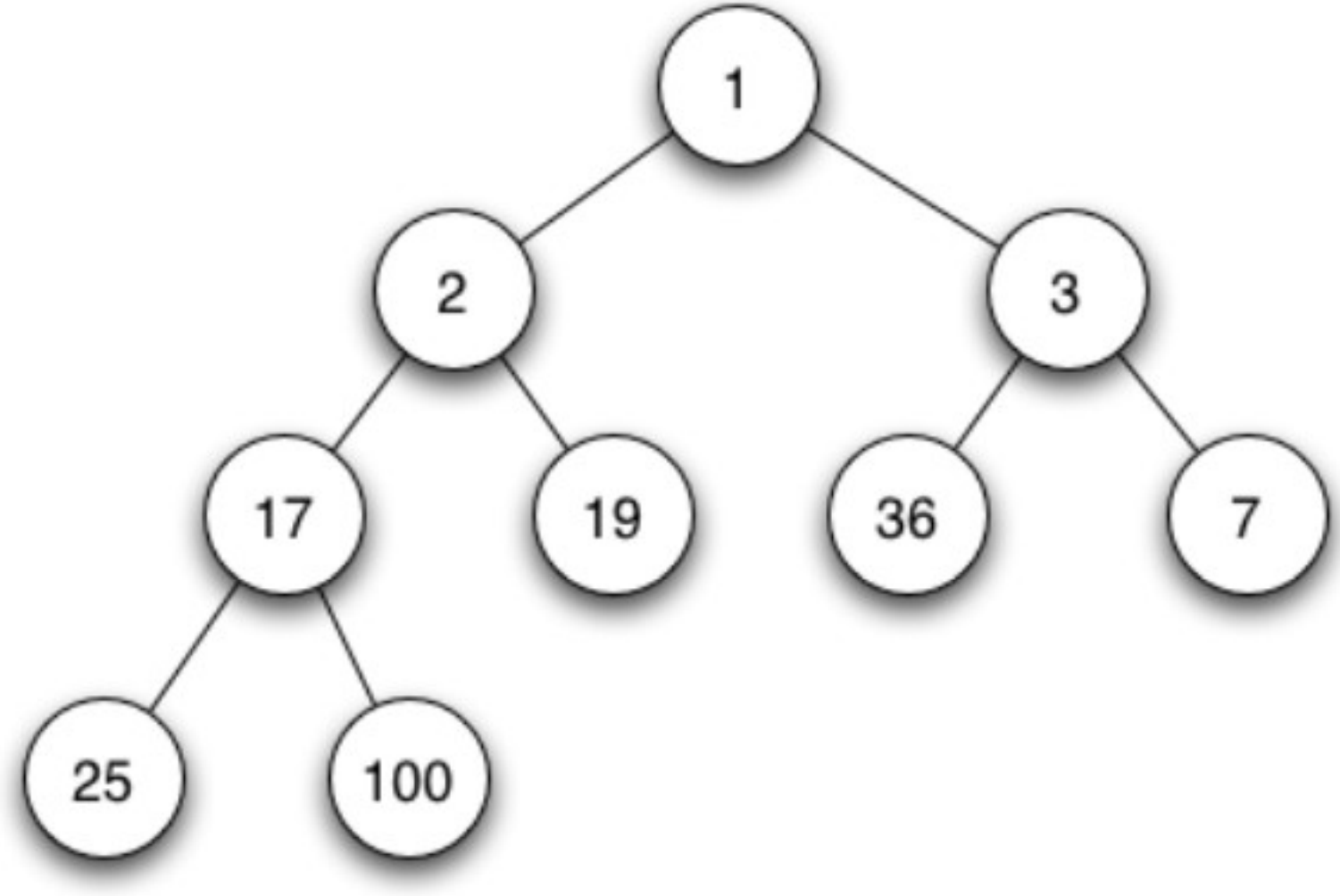
É Heap?



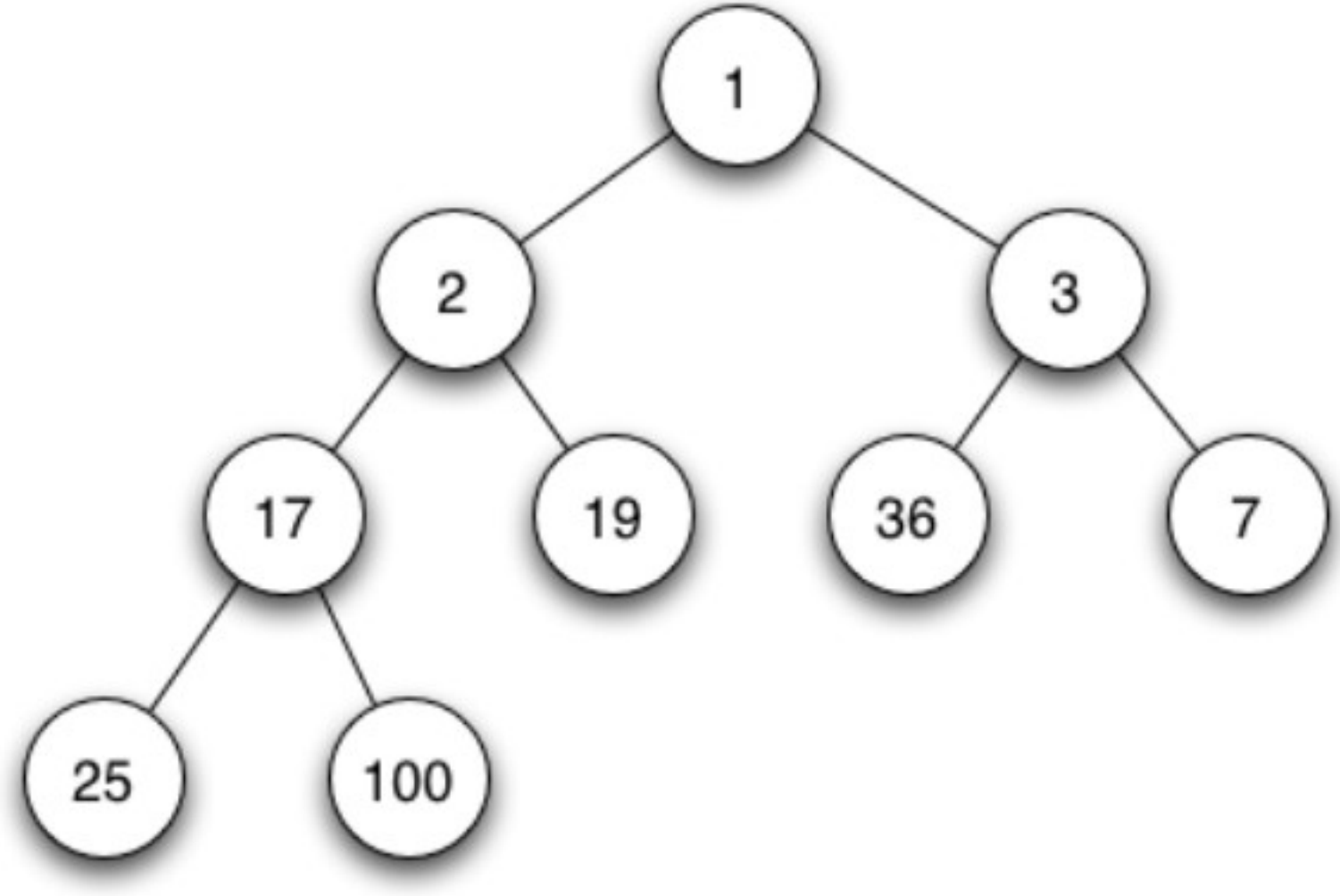
É Heap?



É Heap?

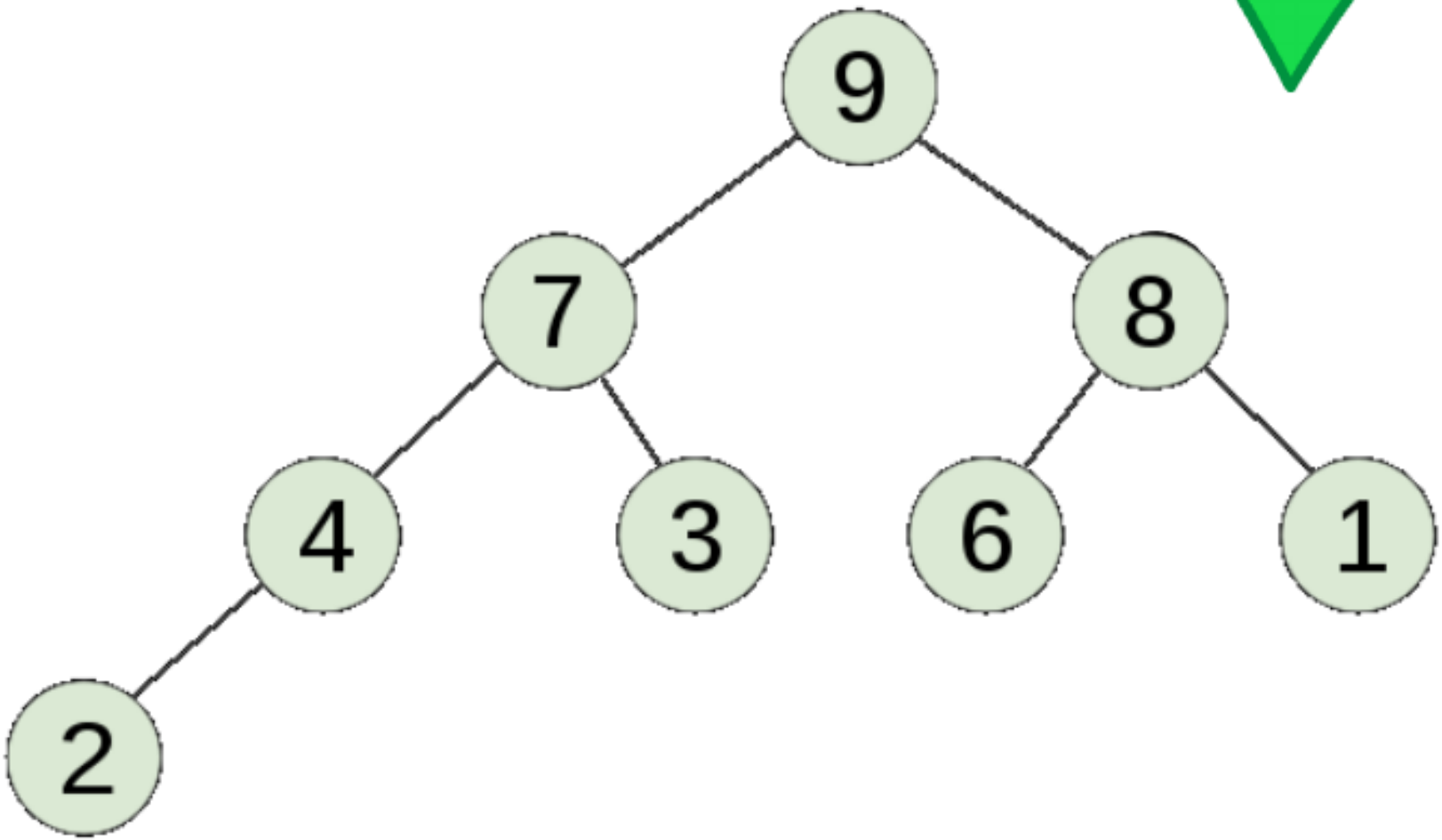


É Heap? Sim! É um min-heap.



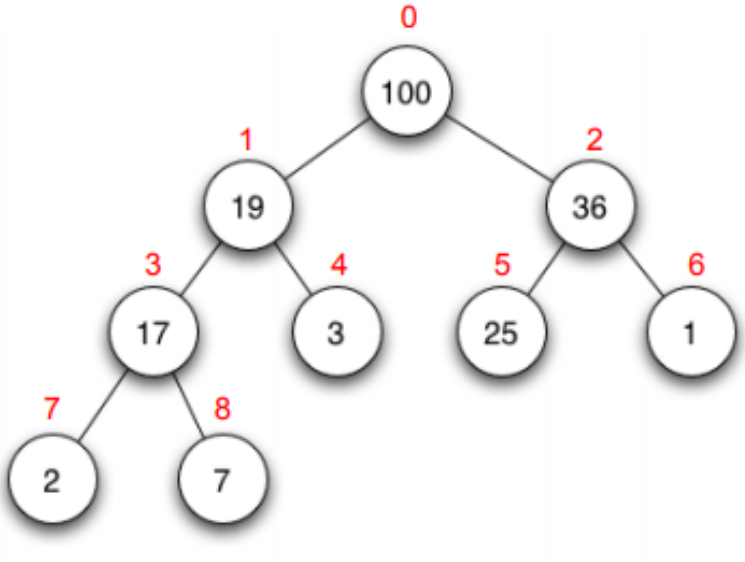
É Heap?

Sim! É um max-heap.



Heap Sort

- Dado o pai, encontrar os filhos:
  - Filho Esquerdo: $2 \cdot i + 1$
  - Filho Direito:
- Dado um filho, encontrar o pai:
- A raiz encontra-se no índice 0 do vetor



0	1	2	3	4	5	6	7	8
100	19	36	17	3	25	1	2	7

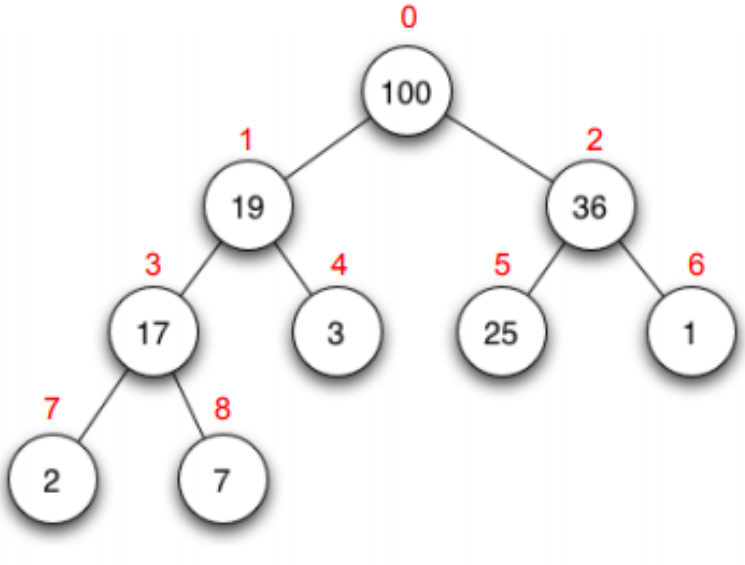
Heap Sort

- Dado o pai, encontrar os filhos:
  - Filho Esquerdo: $2 \cdot i + 1$
  - Filho Direito: 2*i + 2
- Dado um filho, encontrar o pai:

chao((i-1)/2)

- A raiz encontra-se no índice 0 do vetor

0	1	2	3	4	5	6	7	8
100	19	36	17	3	25	1	2	7



Heap Sort

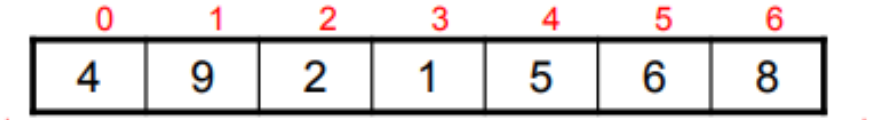
Funcionamento:

1. Transformação de um vetor em um heap binário (controi_Heap)
2. Ordenação
    1. A cada iteração obtem-se o maior elemento do heap (raiz do heap, indice 0 do vetor) e adicione-o em um segmento ordenado.
    2. Após a subtração da raiz, reorganiza-se o heap (ordena_heap)

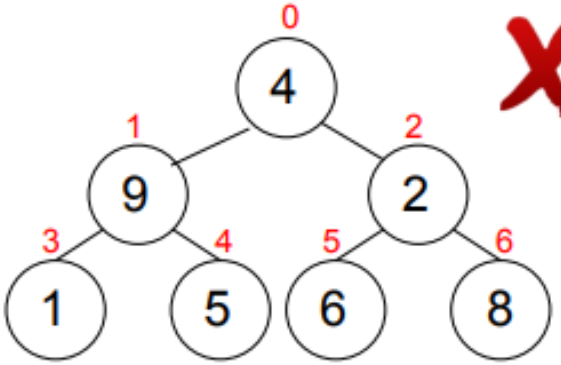
Heap Sort

Idéia:

(0)



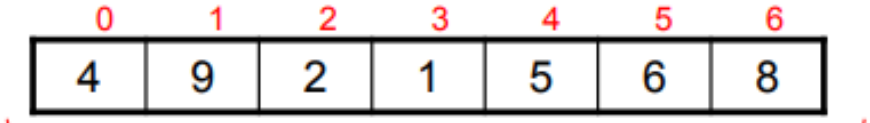
(1) controi_heap(...)



Heap Sort

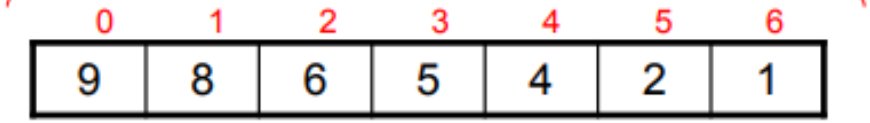
Idéia:

(0)

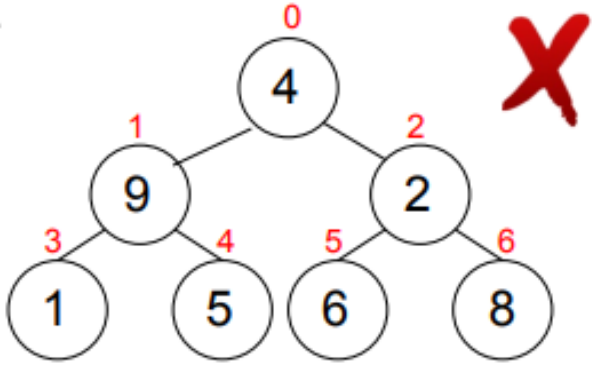


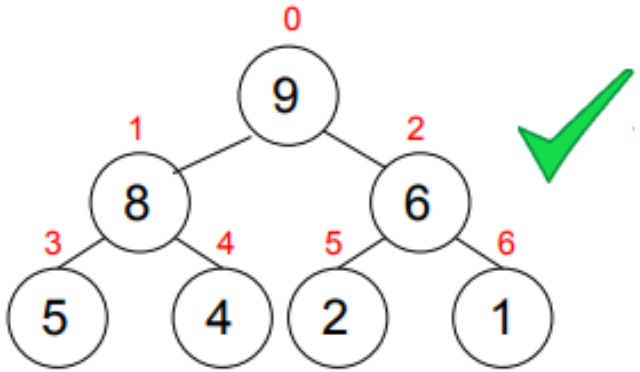
controi_heap(...)

(1)



(2) seleciona_max(...)

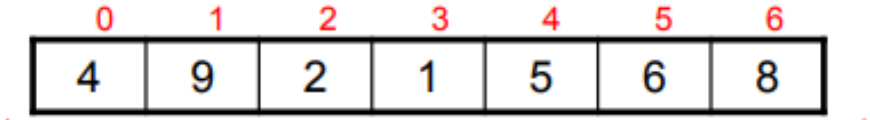




Heap Sort

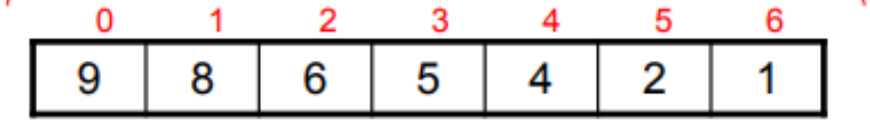
Idéia:

(0)



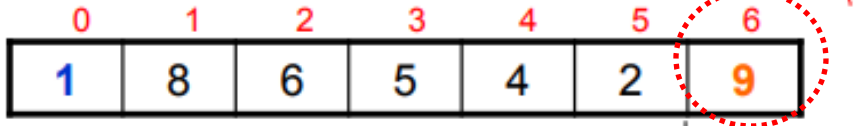
controi_heap(...)

(1)

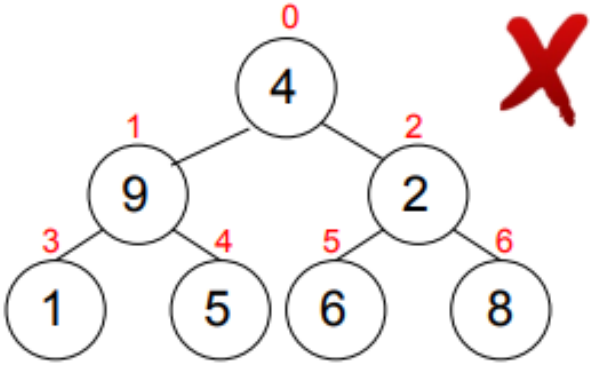


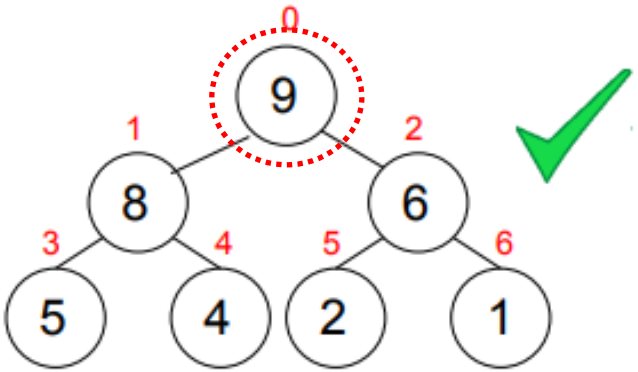
seleciona_max(...)

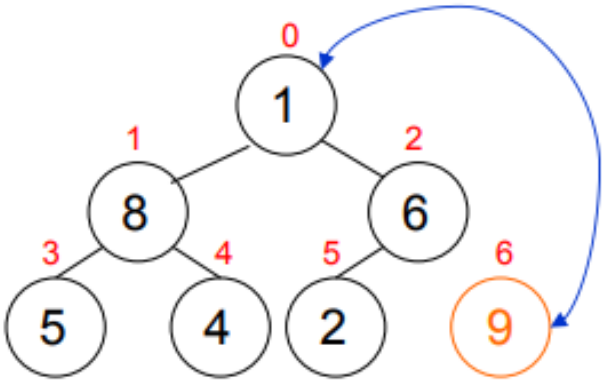
(2)



Seguimento ordenado



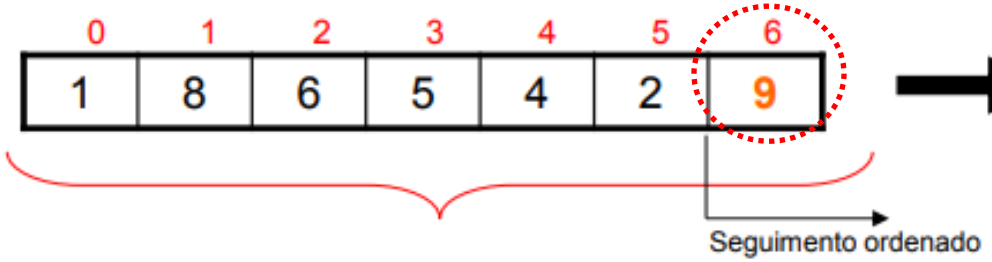


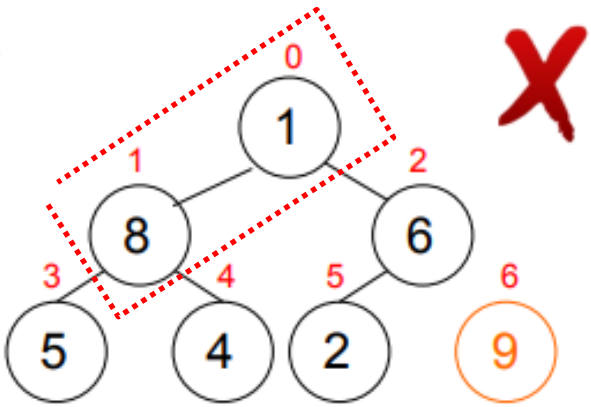


Heap Sort

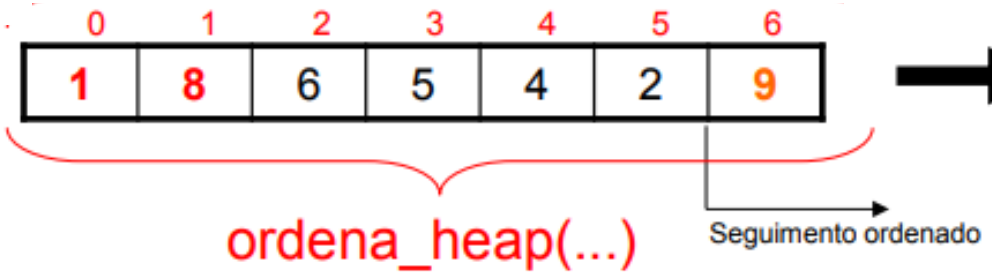
Idéia:

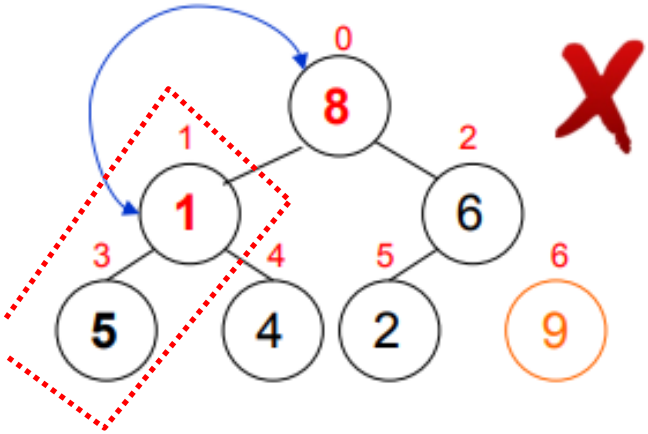
(2.2)





ordena_heap(...)

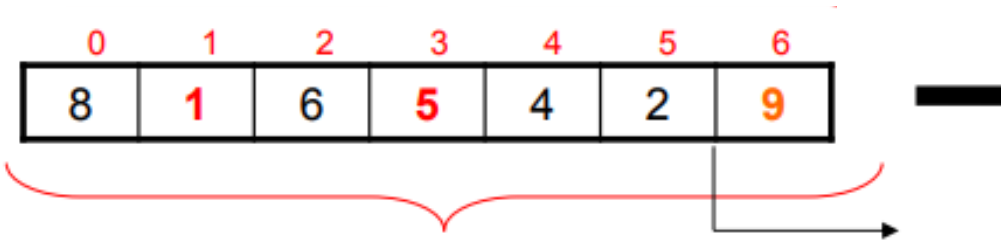




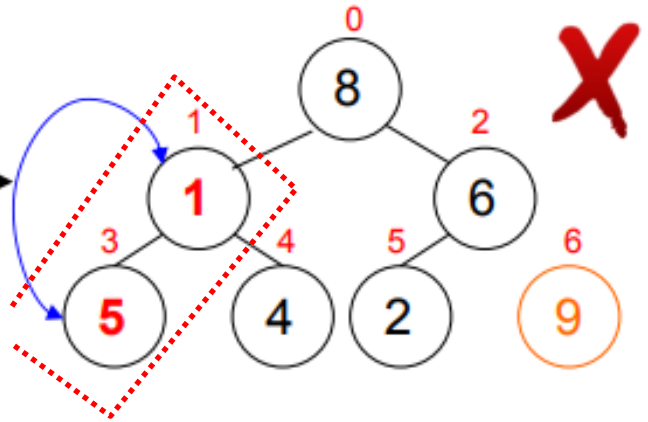
ordena_heap(...)

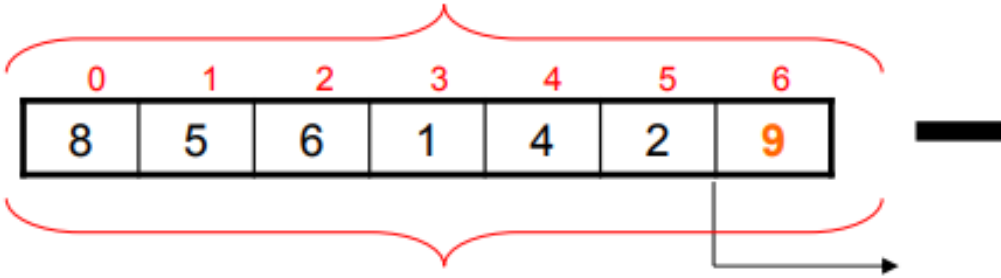
Heap Sort

(2.2)

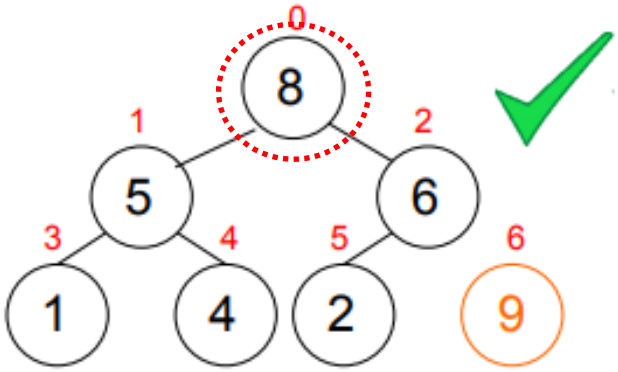


ordena_heap(...)





(2.1) seleciona_max(...)

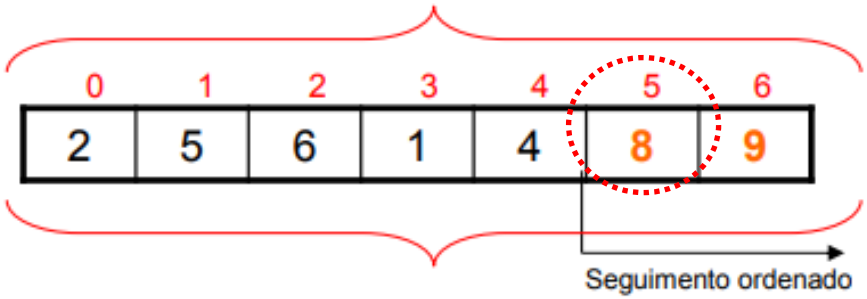


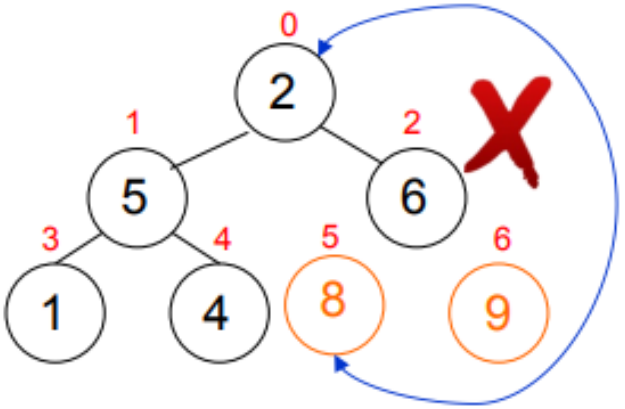
Heap Sort

Idéia:

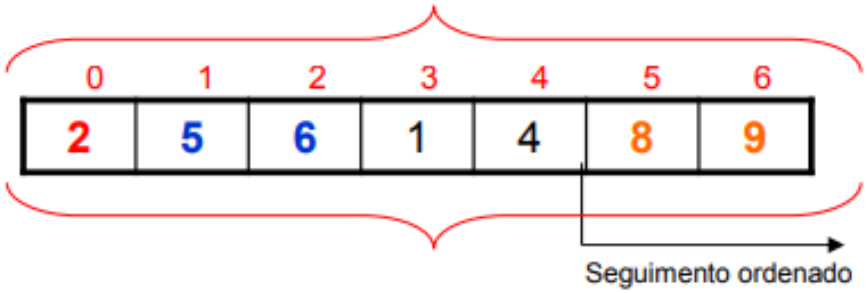
(2.1) seleciona_max(...)

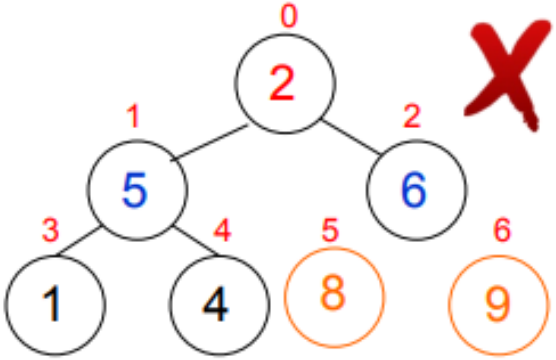
(2.2)



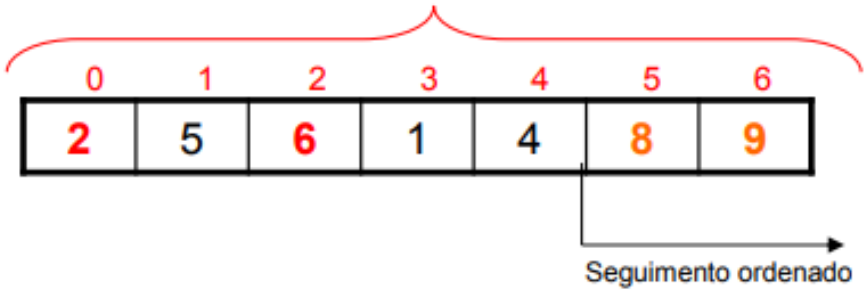


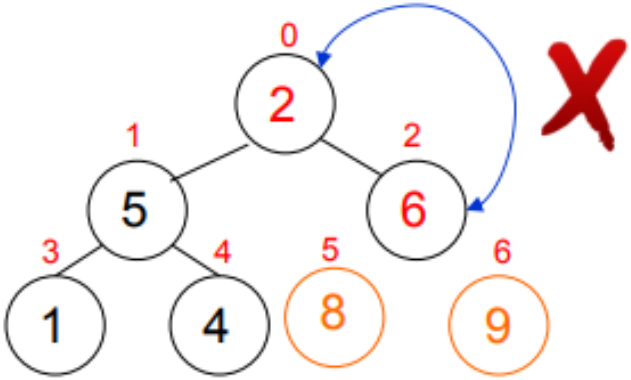
ordena_heap(...)





ordena_heap(...)

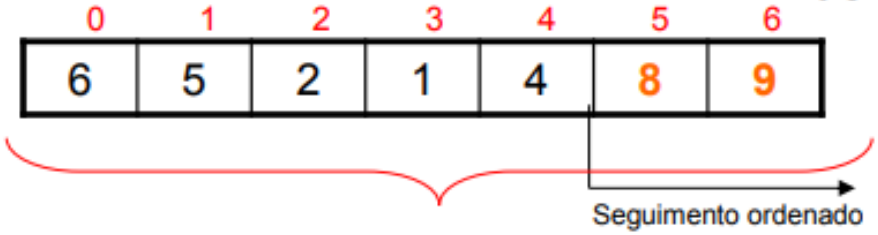




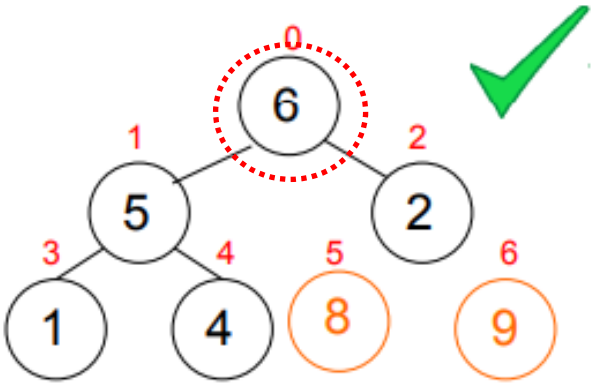
Heap Sort

Idéia:

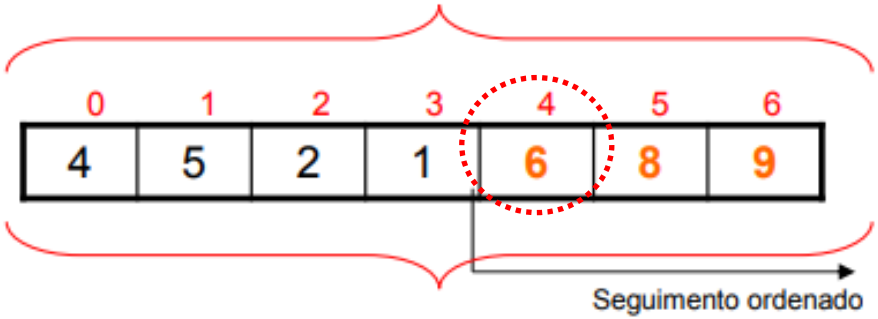
(2.1)



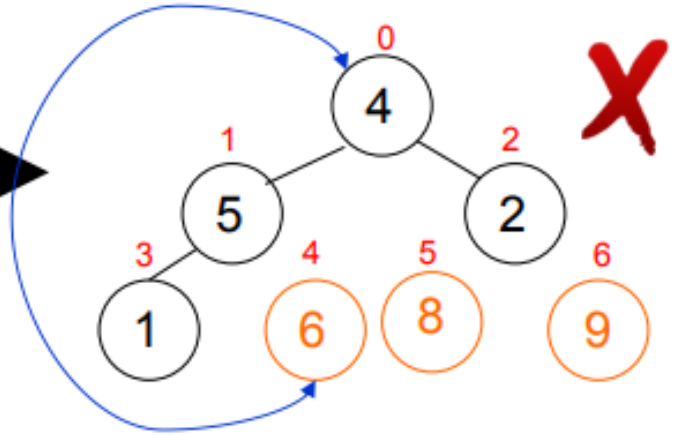
seleciona_max(...)

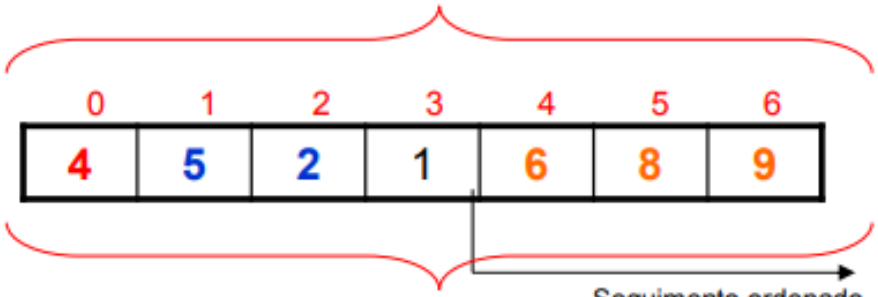


(2.2)

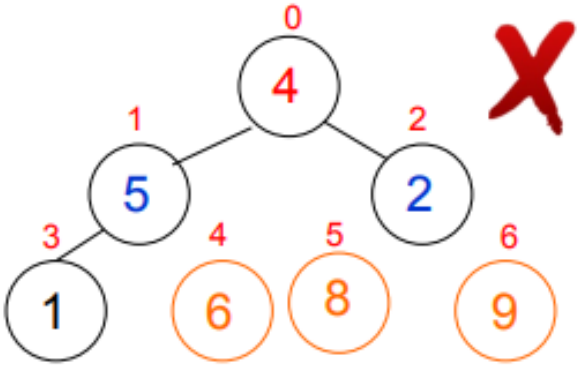


ordena_heap(...)





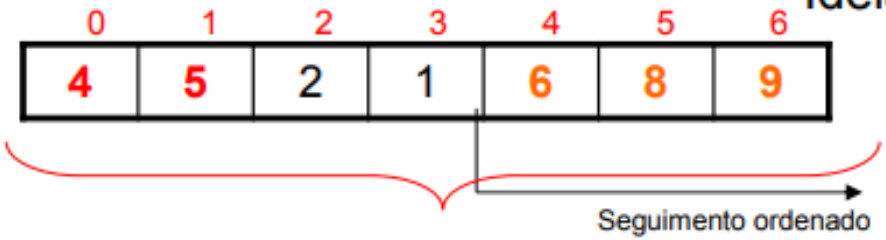
ordena_heap(...)

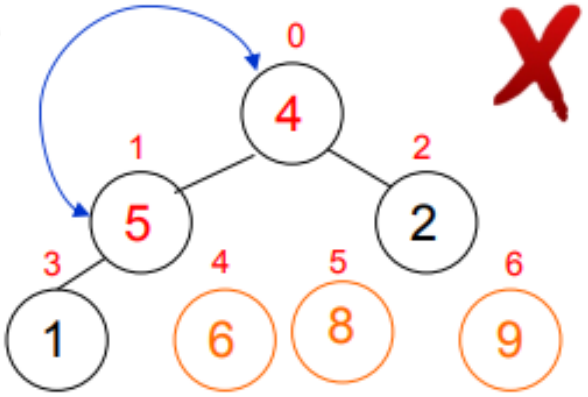


Heap Sort

Idéia:

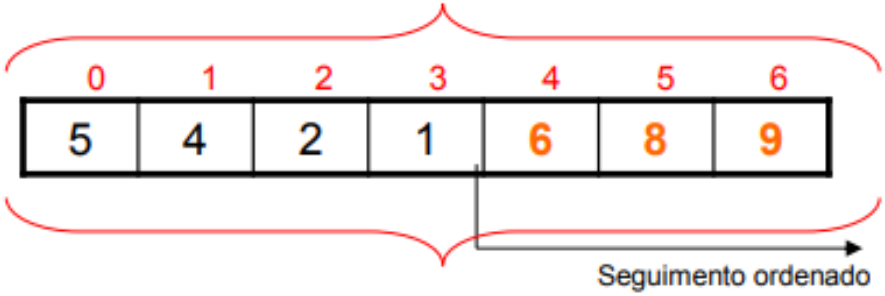
(2.2)

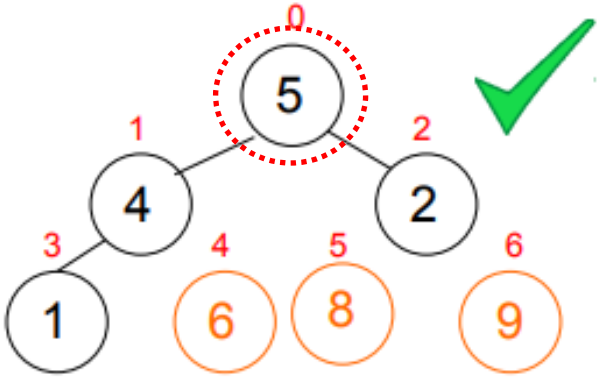




ordena_heap(...)

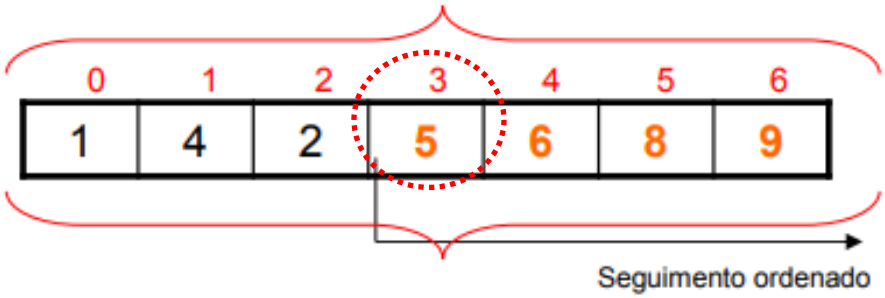
(2.1)

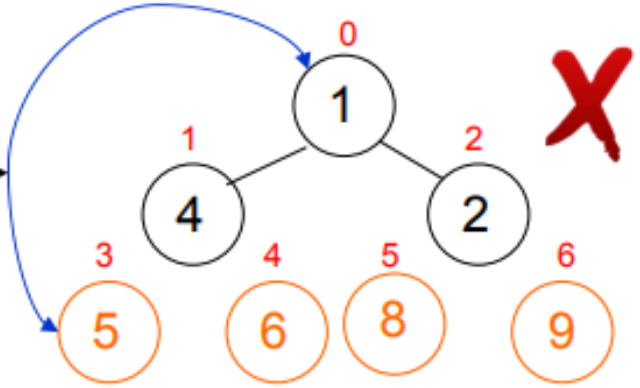




seleciona_max(...)

(2.2)



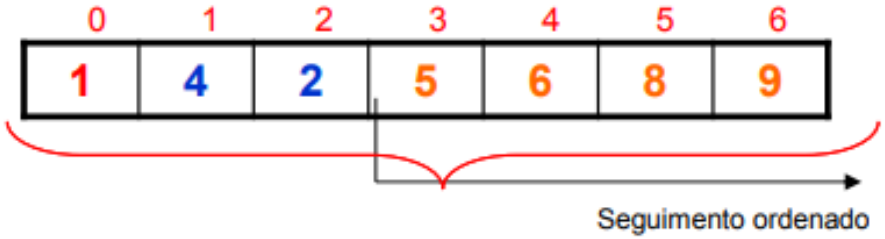


ordena_heap(...)

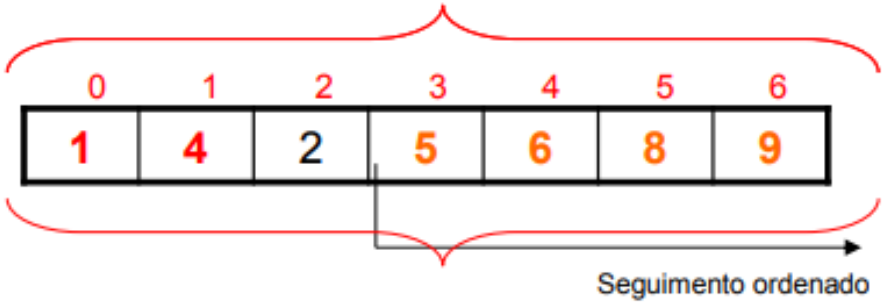
Heap Sort

Idéia:

(2.2)

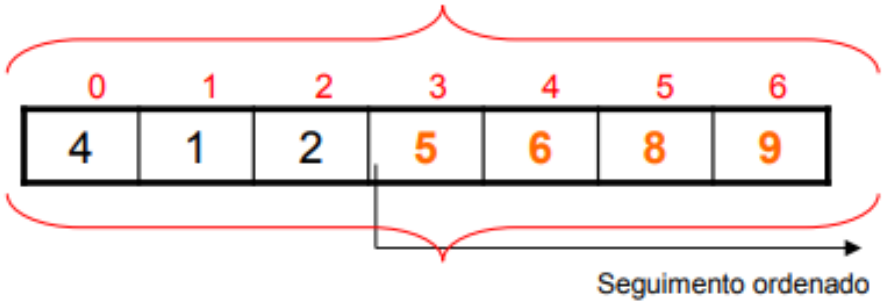


ordena_heap(...)

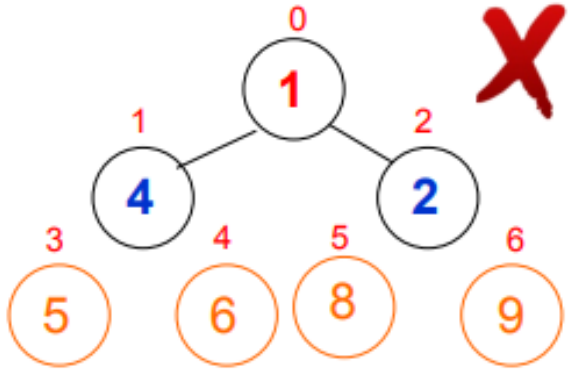


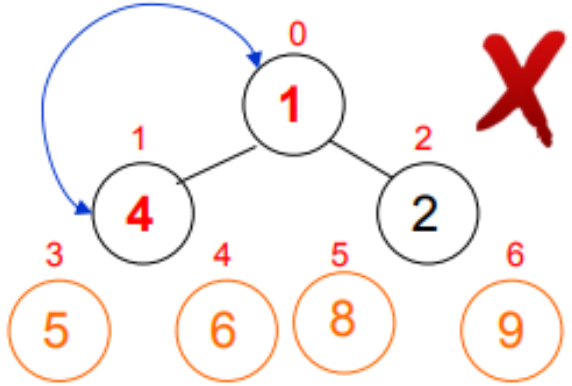
ordena_heap(...)

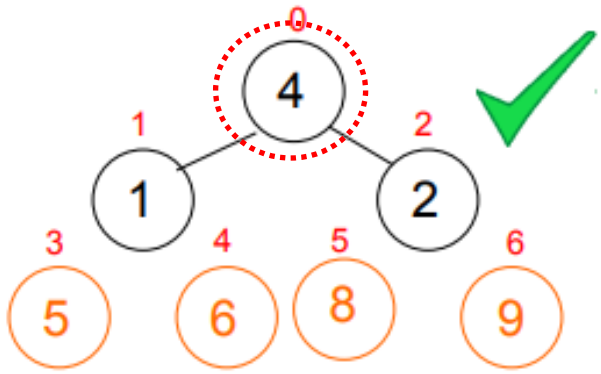
(2.1)



seleciona_max(...)



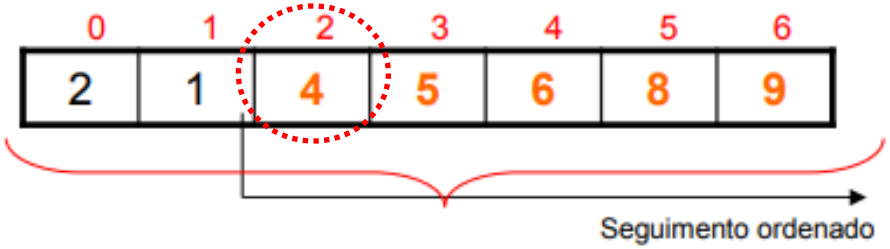




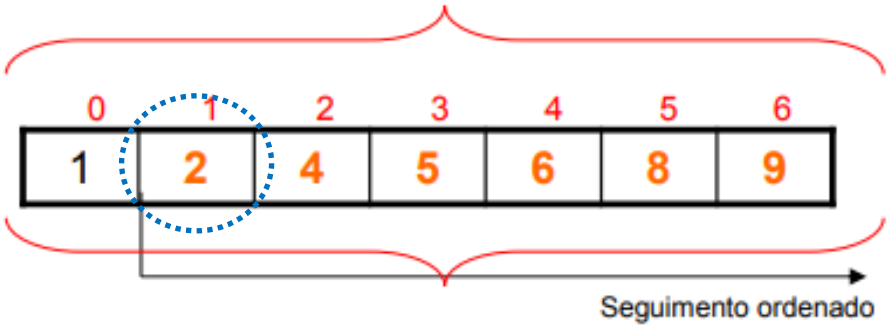
Heap Sort

Idéia:

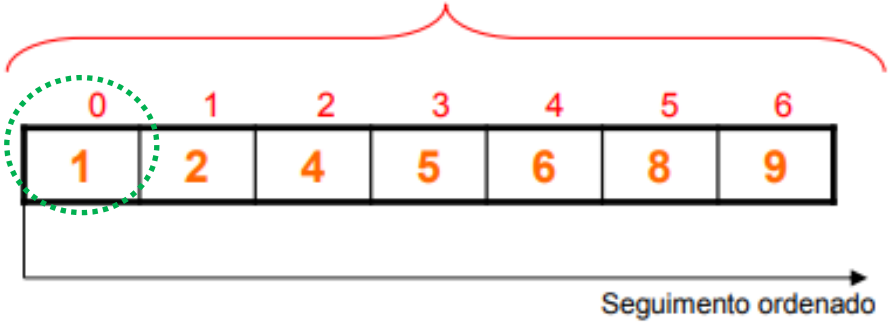
(2.1)

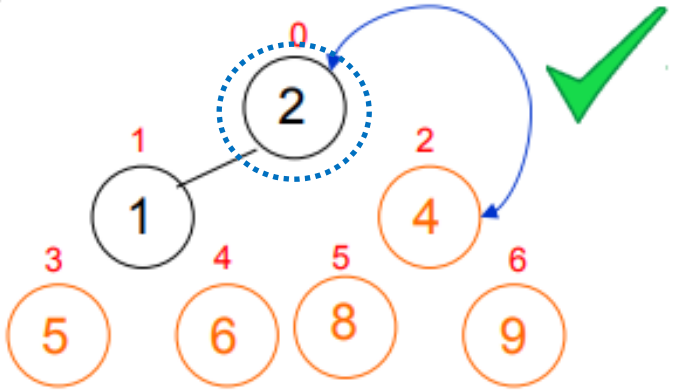


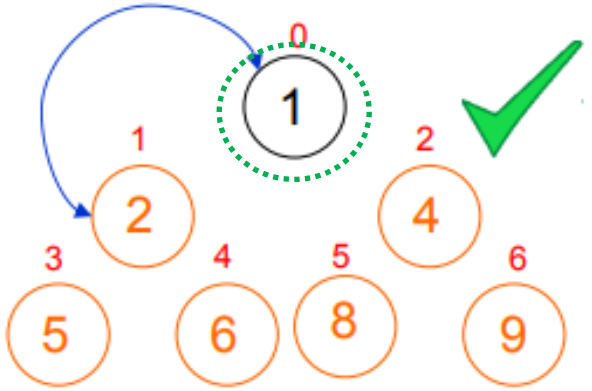
seleciona_max(...)

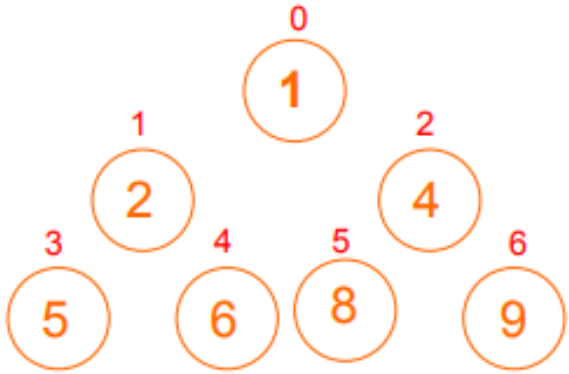


seleciona_max(...)









Heap Sort

Algoritmo:

```
HeapSort(vetor[0, ..., n], tamanho)
1. ConstroiHeap(vetor, tamanho)          → Construa um heap binário com o arranjo inicial.
2. para i de tamanho-1 até 0 passo -1 faça → Para cada item do arranjo.
3.     tamanho ← SelecionaMaximo(vetor, tamanho) → Apanhe o maior e exclua-o do heap.
4.     RestauraHeap(vetor, tamanho, 0)   → Reorganize o arranjo para que volte a ser um heap.
5. fim-para
```

Heap Sort

Algoritmo (Análise de Complexidade):

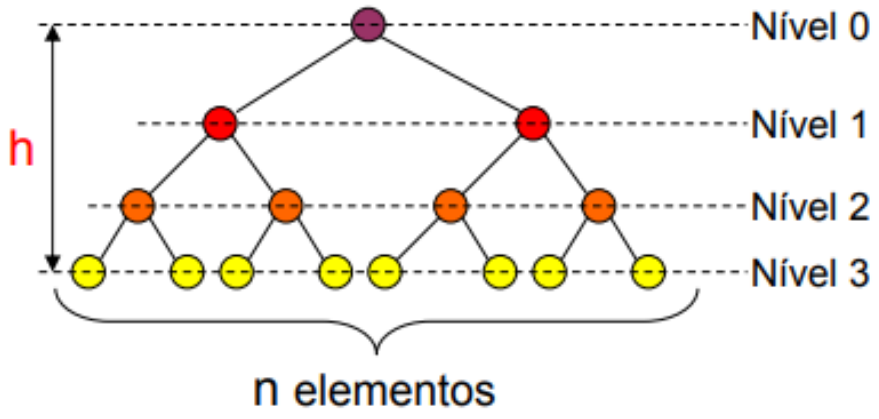
```
HeapSort(vetor[0, ..., n], tamanho)
1. ConstroiHeap(vetor, tamanho)                          → O(n·h)
2. para i de tamanho-1 até 0 passo -1 faça               → O(n)
3.     tamanho ← SelecionaMaximo(vetor, tamanho)         → Θ(1)
4.     RestauraHeap(vetor, tamanho, i)                   → O(h)
5. fim-para
```

O(n·h)

2·O(n·h) = O(n·h)

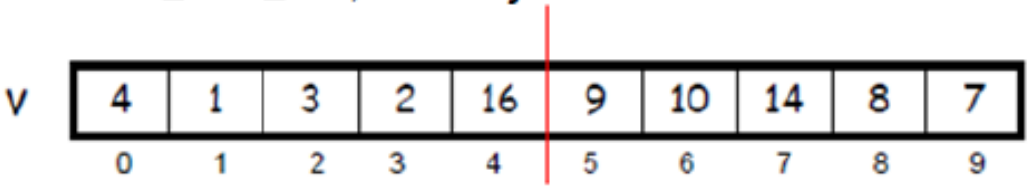
HeapSort é capaz de ordenar os n elementos do arranjo com consumo de tempo proporcional a O(n·h), onde n é o número de itens do arranjo e h a altura do heap.

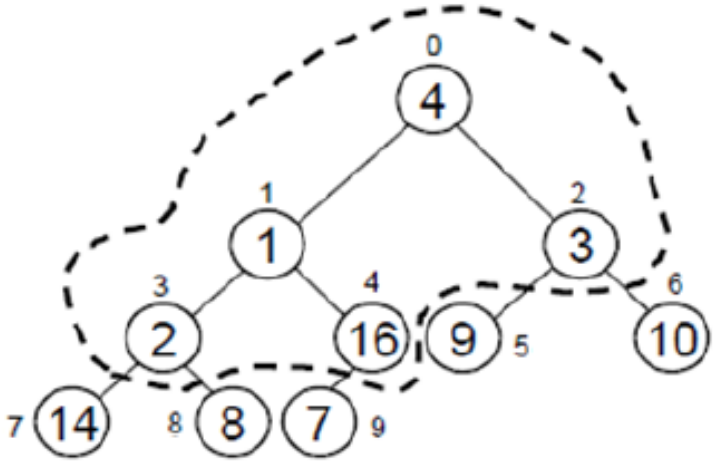
É possível escrevermos h em função de n?



Construção de um Heap

- Construir um Heap a partir de um vetor qualquer:
  - O algoritmo Construir transforma um vetor qualquer em um heap.
  - Como os índices **i**, $\lfloor n/2 \rfloor \leq i < n$, são folhas, basta aplicar **Peneirar** entre as posições 0 e $\lfloor n/2 \rfloor - 1$, ou seja em todos os nós que são pais.





Heap Sort

Algoritmo (Análise de Complexidade):

```
HeapSort(vetor[0, ..., n], tamanho)
1.  ConstroiHeap(vetor, tamanho)                    → O(n·h)
2.  para i de tamanho-1 até 0 passo -1 faça         → O(n)
3.      tamanho ← SelecionaMaximo(vetor, tamanho)   → Θ(1)
4.      RestauraHeap(vetor, tamanho, i)             → O(h)
5.  fim-para
```

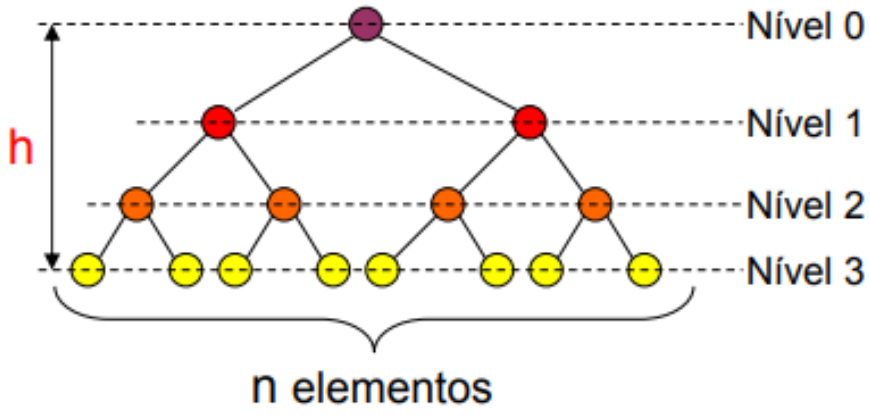
O(n·h)

$2 \cdot O(n \cdot h) = O(n \cdot h)$

HeapSort é capaz de ordenar os n elementos do arranjo com consumo de tempo proporcional a O(n·h), onde n é o número de itens do arranjo e h a altura do heap.

É possível escrevermos h em função de n?

$$h = \lfloor \log_2 n \rfloor$$



n	$\log_2 n$
15	3,91
9	3,17
8	3
7	2,81
4	2
2	1

Heap Sort

Algoritmo (Análise de Complexidade):

```
HeapSort(vetor[0, ..., n], tamanho)
1.  ConstroiHeap(vetor, tamanho)                          → O(n·h)
2.  para i de tamanho-1 até 0 passo -1 faça               → O(n)
3.      tamanho ← SelecionaMaximo(vetor, tamanho)         → Θ(1)
4.      RestauraHeap(vetor, tamanho, i)                   → O(h)
5.  fim-para
```

O(n·h) } 2·O(n·h) = O(n·h)

HeapSort é capaz de ordenar os n elementos do arranjo com consumo de tempo proporcional a O(n·h), onde n é o número de itens do arranjo e h a altura do heap.

É possível escrevermos h em função de n?

Sim!!! $h = \lfloor \log_2 n \rfloor$

Portanto, conclui-se que HeapSort é capaz de ordenar os n elementos do arranjo com consumo de tempo proporcional a $O(n \cdot \log_2 n)$, onde n é o número de itens do arranjo.

Heap Sort

Algoritmo (Análise de Complexidade):

Procedimento	**Consumo no pior caso**
FilhoEsquerdo	$\Theta(1)$
FilhoDireito	$\Theta(1)$
SelecionaMaximo	$\Theta(1)$
RestauraHeap	$O(\log_2 n)$
ConstroiHeap	$O(n \cdot \log_2 n)$
HeapSort	**$O(n \cdot \log_2 n)$**

Algoritmos Clássicos de Ordenação

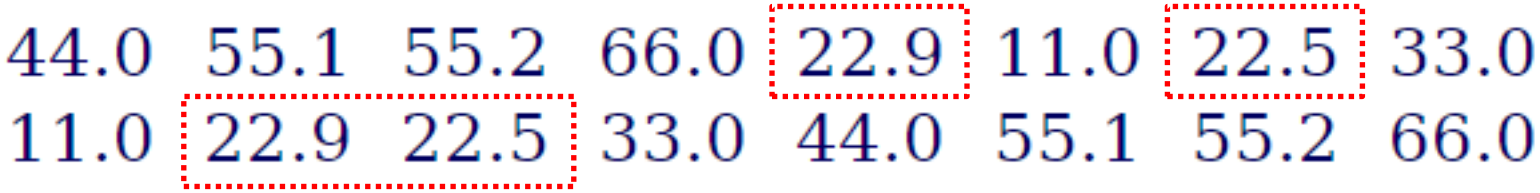
- Quadro Comparativo da complexidade dos algoritmos de ordenação (método de comparação)

Algoritmo	Temporal			Espacial
	Melhor	Médio	Pior	
Bubble Sort	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$
Insertion Sort	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$
Selection Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$
Merge Sort	$\Theta(n \cdot \log_2 n)$	$\Theta(n \cdot \log_2 n)$	$\Theta(n \cdot \log_2 n)$	$\Theta(n)$
Quick Sort	$\Theta(n \cdot \log_2 n)$	$\Theta(n \cdot \log_2 n)$	$\Theta(n^2)$	$O(\log_2 n)$
Heap Sort	$O(n \cdot \log_2 n)$	$\Theta(n \cdot \log_2 n)$	$O(n \cdot \log_2 n)$	$O(1)$

Estável Não Estável

Algoritmos "Estáveis"

- Um algoritmo de ordenação é estável se não altera a posição relativa dos elementos que têm o mesmo valor.
  - Em outras palavras, um algoritmo estável de ordenação mantém a ordem de inserção dos dados no caso de empates.
- Exemplo: ordenação estável da parte inteira



(Fonte: https://www.ime.usp.br/~pf/algoritmos/aulas/ordena.html)