

# Projeto e Análise de Algoritmos

Fila de prioridade

Lehilton Pedrosa

Primeiro Semestre de 2020

## Selection Sort

# Selecionando o próximo menor

Vamos ordenar usando ideia diferente

- ▶ suponha que o subvetor  $A[1 \dots i - 1]$  está ordenado
- ▶ também, suponha que  $\max A[1 \dots i - 1] \leq \min A[i \dots n]$ .
- ▶ substituimos a posição  $A[i]$  pelo mínimo em  $A[i \dots n]$

Antes de substituir:

1						$i$				$n$
20	25	35	40	44	55	70	80	99	65	85

Após substituir:

1						$i$				$n$
20	25	35	40	44	55	65	80	99	70	85

# Pseudocódigo de SELECTION-SORT

**SELECTION-SORT**( $A, n$ )

```
1  para  $i \leftarrow 1$  até  $n - 1$  faça
2       $min \leftarrow i$ 
3      para  $j \leftarrow i + 1$  até  $n$  faça
4          se  $A[j] < A[min]$  então  $min \leftarrow j$ 
5       $A[i] \leftrightarrow A[min]$ 
```

## Invariante

Ao início de cada iteração:

1.  $A[1 \dots i - 1]$  está ordenado,
2.  $A[1 \dots i - 1] \leq A[i \dots n]$ .

# Complexidade de SELECTION-SORT

<b>SELECTION-SORT</b> ( $A, n$ )	Tempo
1 para $i \leftarrow 1$ até $n - 1$ faça	$\Theta(n)$
2 $min \leftarrow i$	$\Theta(n)$
3     para $j \leftarrow i + 1$ até $n$ faça	$\Theta(n^2)$
4         se $A[j] < A[min]$ então $min \leftarrow j$	$\Theta(n^2)$
5 $A[i] \leftrightarrow A[min]$	$\Theta(n)$

Consumo de tempo no pior caso?  $\Theta(n^2)$

E no melhor caso?

# Uma versão alternativa

Podemos reescrever esse algoritmo

- ▶ ordenamos a partir do final
- ▶ selecionamos o **maior** remanescente
- ▶ refatoramos com uma sub-rotina **MAXIMUM**

**SELECTION-SORT**( $A, n$ )

```
1  para  $i \leftarrow n$  decrescendo até 2 faça
2       $max \leftarrow \text{MAXIMUM}(A, i)$ 
3       $A[i] \leftrightarrow A[max]$ 
```

# Reverendo a complexidade

## SELECTION-SORT( $A, n$ )

```
1  para  $i \leftarrow n$  decrescendo até 2 faça
2       $max \leftarrow \text{MAXIMUM}(A, i)$ 
3       $A[i] \leftrightarrow A[max]$ 
```

- ▶ suponha que  $\text{MAXIMUM}(A, i)$  leva tempo  $O(t(i))$
- ▶ então o tempo total é

$$T(n) = \sum_{i=2}^n O(t(i)) \leq \sum_{i=2}^n O(t(n)) = O(n \cdot t(n))$$

- ▶ vamos tentar otimizar a rotina  $\text{MAXIMUM}(A, i)$

# Heapsort



# O algoritmo HEAP-SORT

Vamos estudar a **fila de prioridade**

- ▶ é uma estrutura de dados também chamada de max-heap
- ▶ implementa **MAXIMUM** com tempo  $O(\log n)$
- ▶ usando um heap, podemos ordenar em  $O(n \log n)$
- ▶ esse algoritmo de ordenação é chamado de **heapsort**

# Representando um heap

Um heap é uma **árvore binária** armazenada em vetor  $A[1 \dots n]$

- ▶ Filhos de um nó  $i$ 
  - ▶ o filho esquerdo é  $2i$
  - ▶ o filho direito é  $2i + 1$
  
- ▶ Pais
  - ▶ o pai de um nó  $i$  é  $\lfloor i/2 \rfloor$
  - ▶ o nó 1 não tem pai.
  
- ▶ Folhas
  - ▶ um nó  $i$  é folha se não tiver filhos, i.e., se  $2i > n$ .
  - ▶ as folhas são  $\lfloor n/2 \rfloor + 1, \dots, n - 1, n$ .



# Árvore completa

Um heap é uma árvore **completa**

- ▶ cada nível  $\ell = 0, 1, 2, \dots$  tem  $2^\ell$  nós (a não ser o último)
- ▶ os nós desse nível  $\ell$  são  $2^\ell, 2^\ell + 1, 2^\ell + 2, \dots, 2^{\ell+1} - 1$

## Nível de um nó

O nível de um nó  $i$  é  $\lfloor \lg i \rfloor$ .

- ▶ se  $i$  está no nível  $\ell$ , então

$$\begin{aligned} 2^\ell &\leq i < 2^{\ell+1} &\Rightarrow \\ \lg 2^\ell &\leq \lg i < \lg 2^{\ell+1} &\Rightarrow \\ \ell &\leq \lg i < \ell + 1 \end{aligned}$$

- ▶ assim,  $\ell = \lfloor \lg i \rfloor$ .

Portanto, o número total de níveis é  $1 + \lfloor \lg n \rfloor$ .

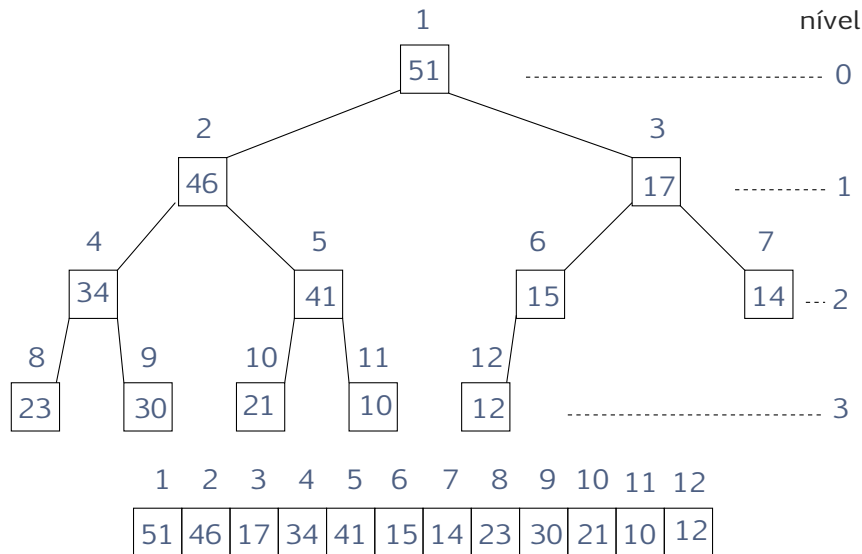
## Definição

Um heap  $A[1 \dots n]$  é chamado de **max-heap** se cada nó tiver valor maior que seus filhos, i.e., para cada  $i \geq n/2$ ,

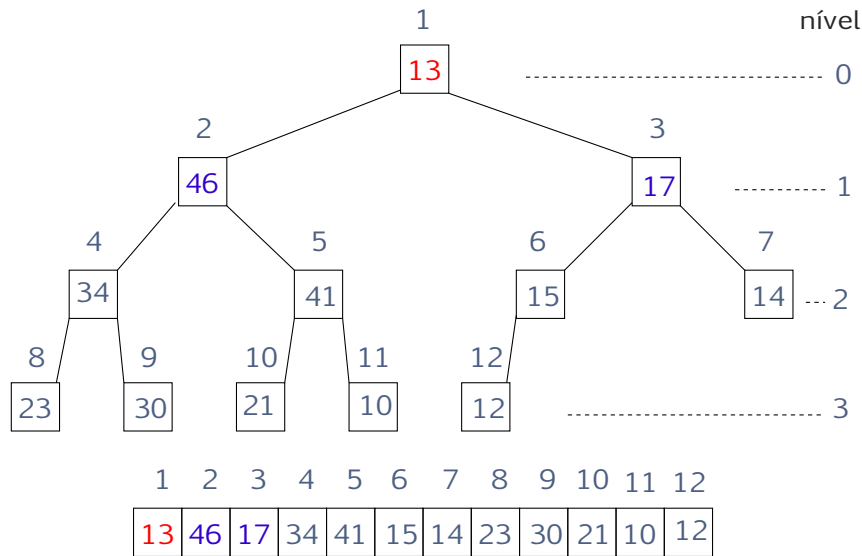
▶  $A[i] \geq A[2i]$  e  $A[i] \geq A[2i + 1]$

- ▶ essa restrição é a **propriedade de max-heap**
- ▶ o valor da raiz é um máximo do heap
- ▶ cada subárvore também é um max-heap

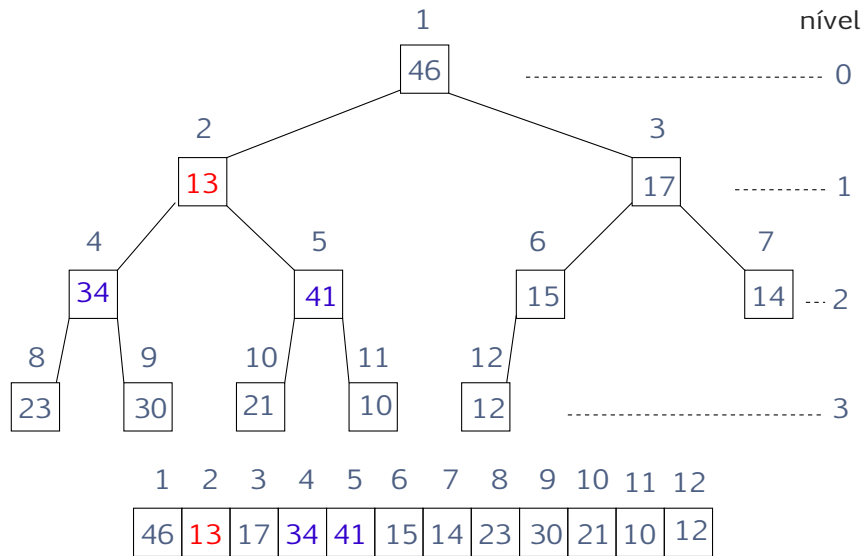
# Exemplo de max-heap



# Manipulação de max-heap

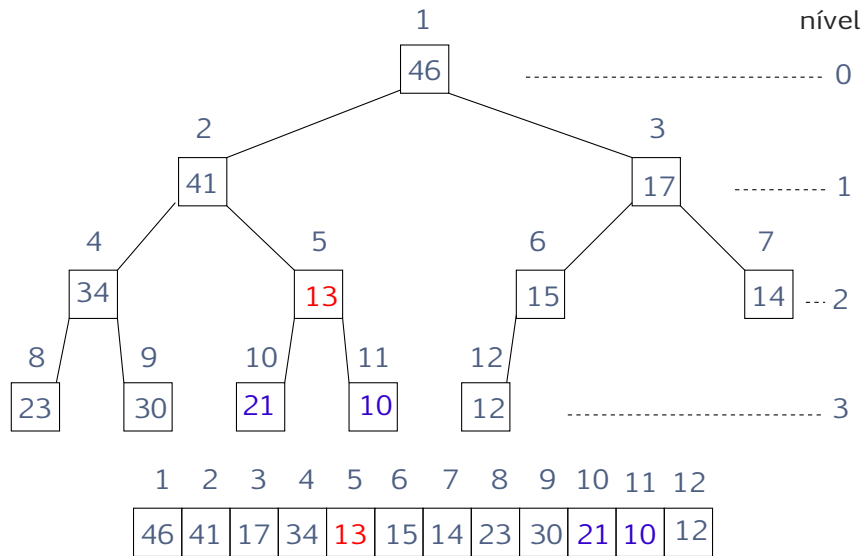


# Manipulação de max-heap

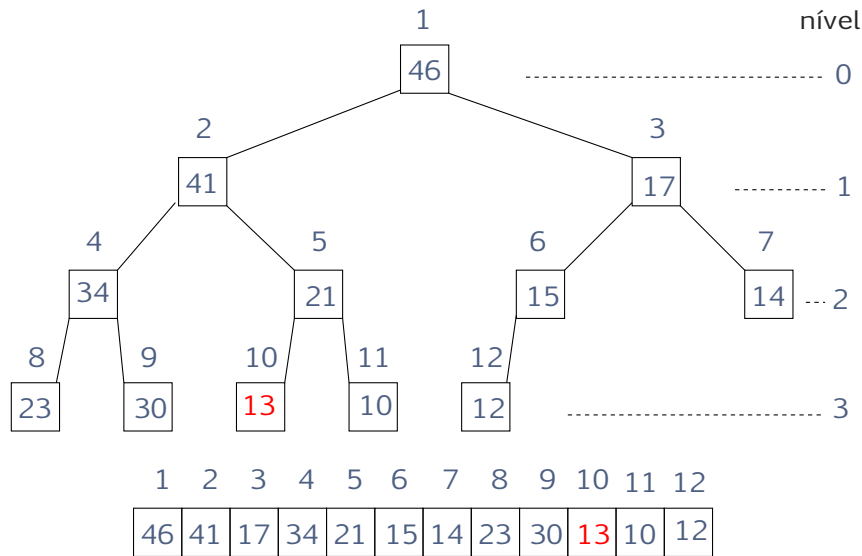




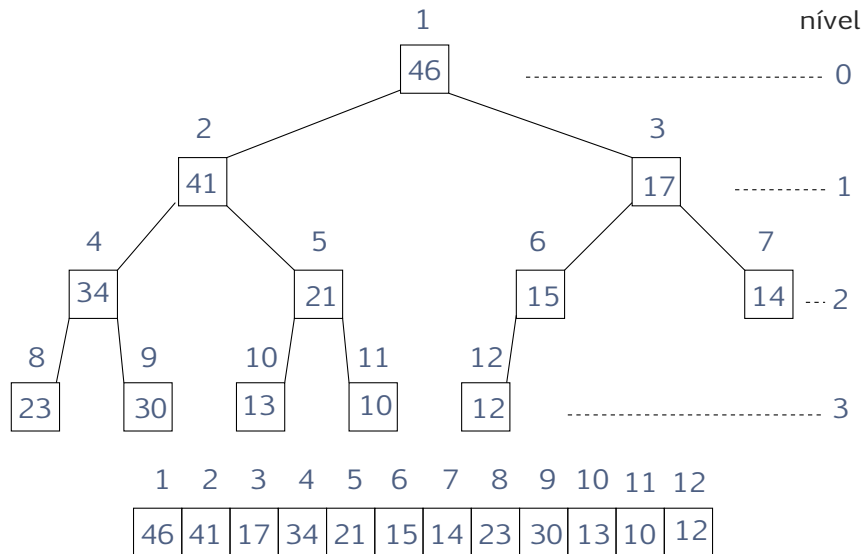
# Manipulação de max-heap



# Manipulação de max-heap



# Manipulação de max-heap



## Consertando um max-heap

- ▶ suponha que as subárvores  $2i$  e  $2i + 1$  são max-heaps
- ▶ como transformar a subárvore  $i$  em um max-heap?

**MAX-HEAPIFY**( $A, n, i$ )

```
1   $e \leftarrow 2i$ 
2   $d \leftarrow 2i + 1$ 
3   $maior \leftarrow i$ 
4  se  $e \leq n$  e  $A[e] > A[i]$ 
5      então  $maior \leftarrow e$ 
6  se  $d \leq n$  e  $A[d] > A[maior]$ 
7      então  $maior \leftarrow d$ 
8  se  $maior \neq i$ 
9      então  $A[i] \leftrightarrow A[maior]$ 
10         MAX-HEAPIFY( $A, n, maior$ )
```

## Lema

MAX-HEAPIFY transforma a subárvore  $i$  em max-heap.

Ideia para demonstração: indução na altura  $h$  do nó  $i$ .

- ▶ se  $h = 0$ , então  $i$  é folha e o algoritmo está correto
- ▶ considere um nó  $i$  com altura  $h > 0$
- ▶ suponha que o algoritmo funciona para árvores menores
  - ▶ antes da linha 8, temos  $A[maior] \geq A[i], A[2i], A[2i + 1]$
  - ▶ após a linha 9, temos  $A[i] \geq A[2i], A[2i + 1]$
  - ▶ segue que  $A[i]$  é máximo no vetor
  - ▶ pela hipótese de indução, MAX-HEAPIFY transforma a subárvore com raiz  $maior$  em max-heap
  - ▶ segue que  $i$  é max-heap

# Complexidade de MAX-HEAPIFY

	MAX-HEAPIFY( $A, n, i$ )	Tempo
1	$e \leftarrow 2i$	$\Theta(1)$
2	$d \leftarrow 2i + 1$	$\Theta(1)$
3	$maior \leftarrow i$	$\Theta(1)$
4	se $e \leq n$ e $A[e] > A[i]$	$\Theta(1)$
4	então $maior \leftarrow e$	$O(1)$
6	se $d \leq n$ e $A[d] > A[maior]$	$\Theta(1)$
7	então $maior \leftarrow d$	$O(1)$
8	se $maior \neq i$	$\Theta(1)$
9	então $A[i] \leftrightarrow A[maior]$	$O(1)$
10	MAX-HEAPIFY( $A, n, maior$ )	$T(h - 1)$

O tempo de execução é  $T(h) = T(h - 1) + \Theta(1) = O(h)$

# Construindo um max-heap

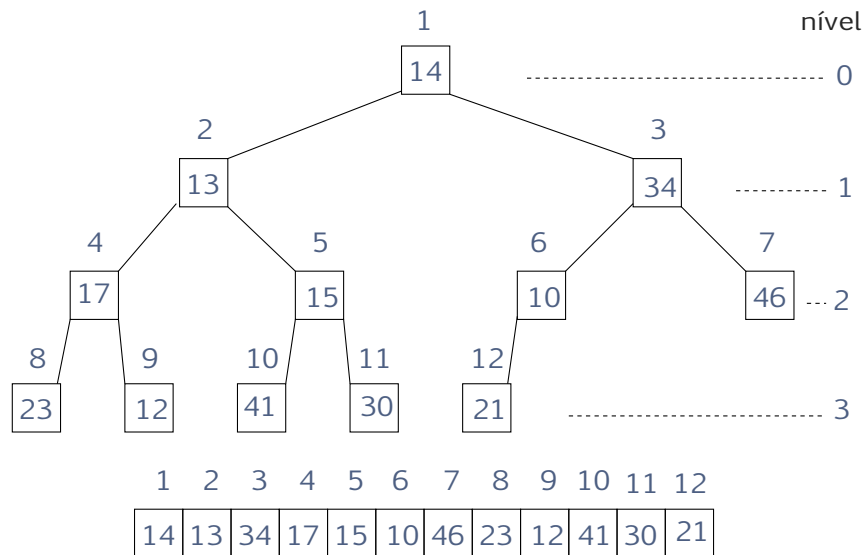
Podemos consertar um vetor inteiro

- ▶ recebemos um vetor  $A[1 \dots n]$  desorganizado
- ▶ mas as folhas já são heap
- ▶ consertamos o penúltimo nível
- ▶ depois o antepenúltimo
- ▶ e assim por diante

**BUILD-MAX-HEAP**( $A, n$ )

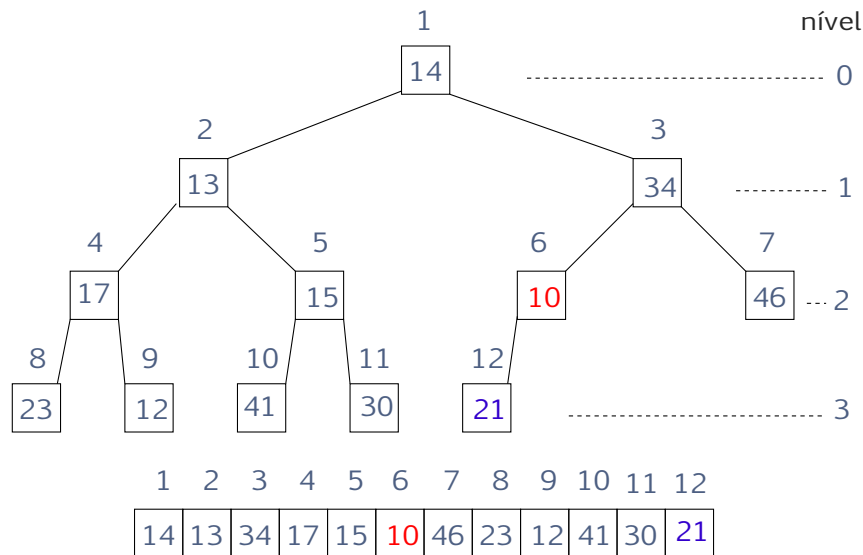
```
1  para  $i \leftarrow \lfloor n/2 \rfloor$  decrescendo até 1 faça
2      MAX-HEAPIFY( $A, n, i$ )
```

# Construção de um max-heap

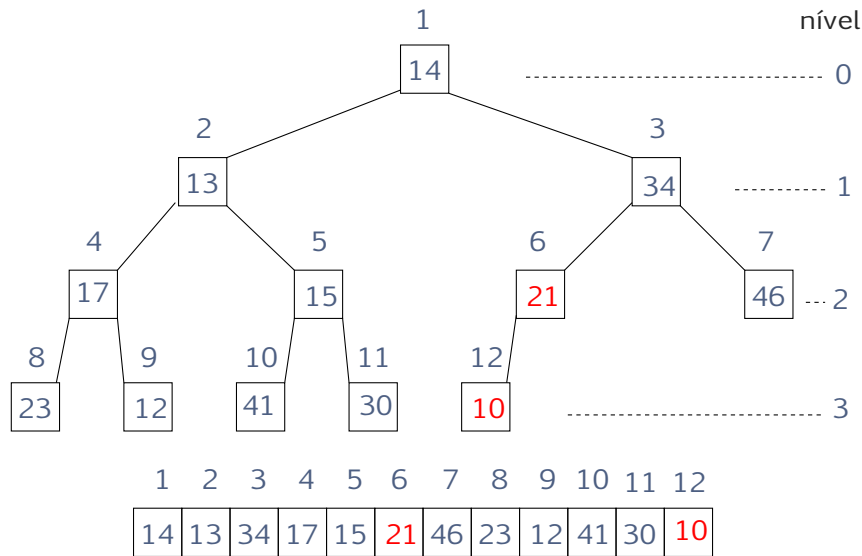




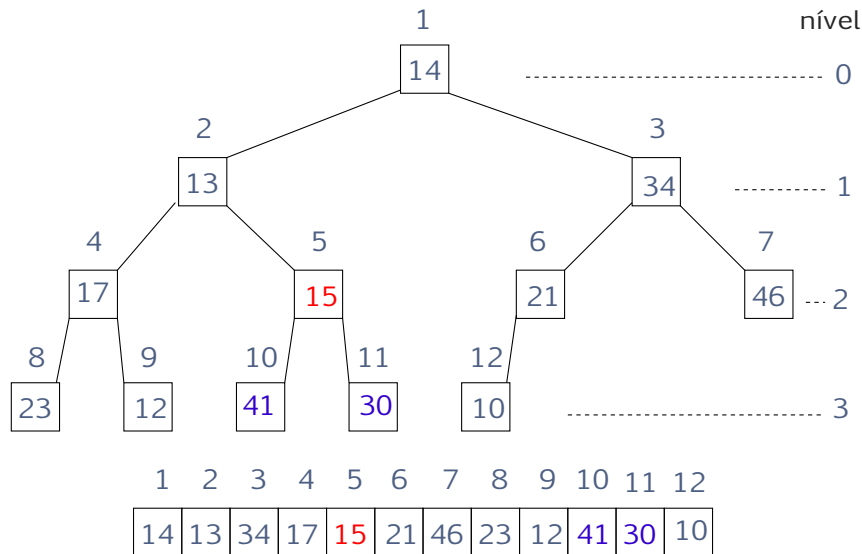
# Construção de um max-heap



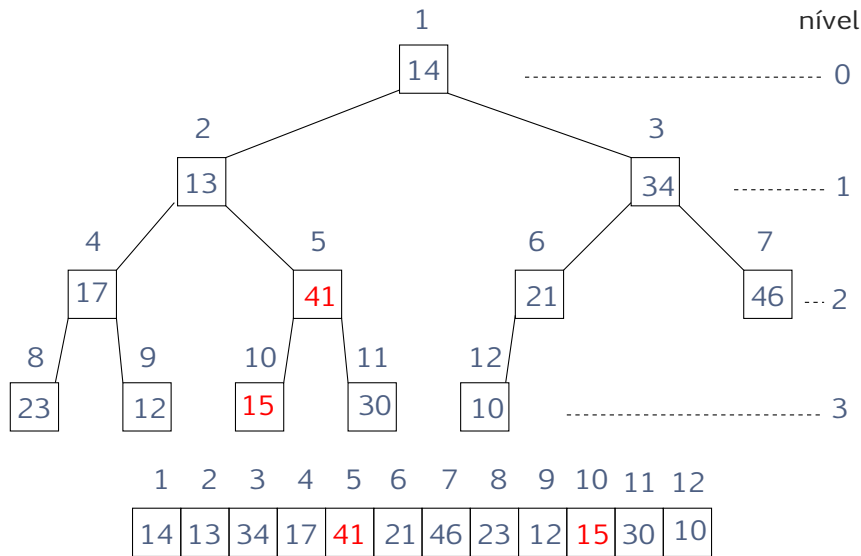
# Construção de um max-heap



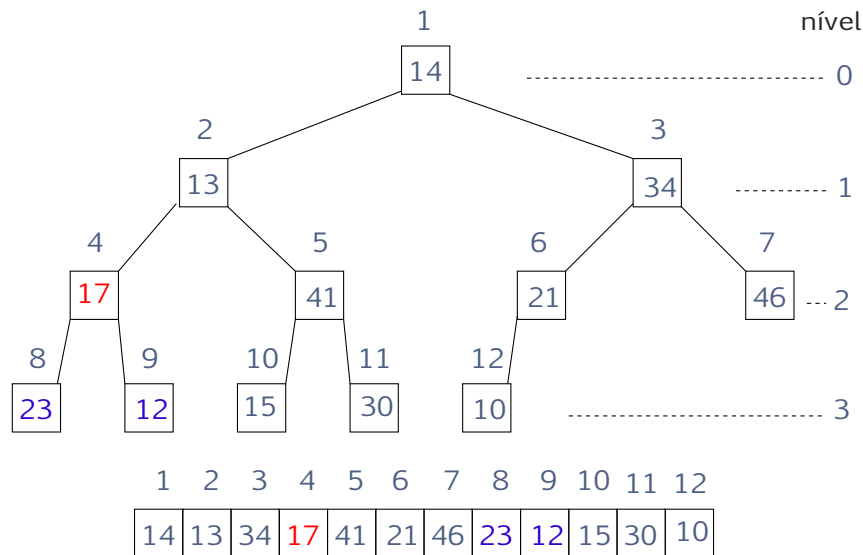
# Construção de um max-heap



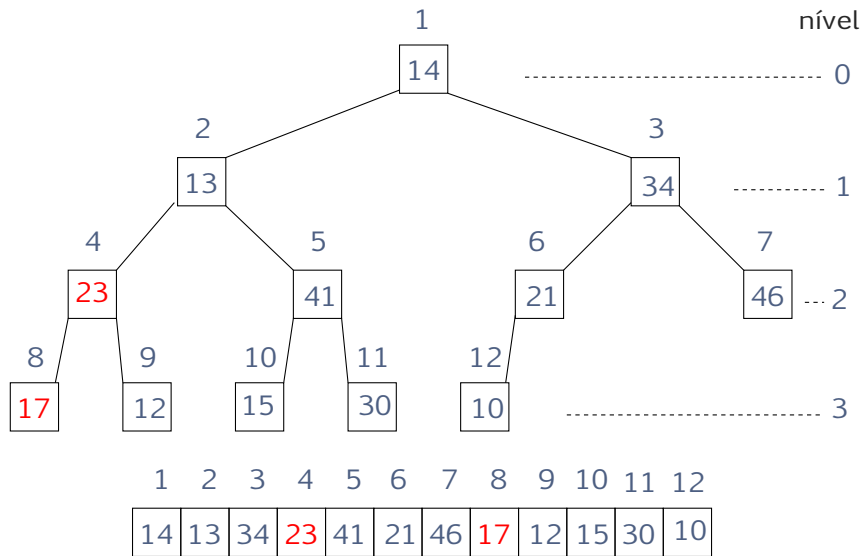
# Construção de um max-heap



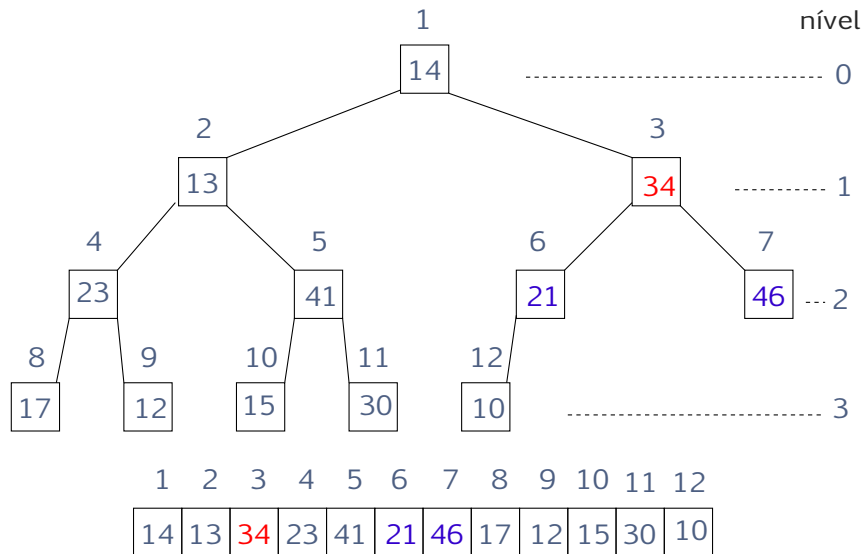
# Construção de um max-heap



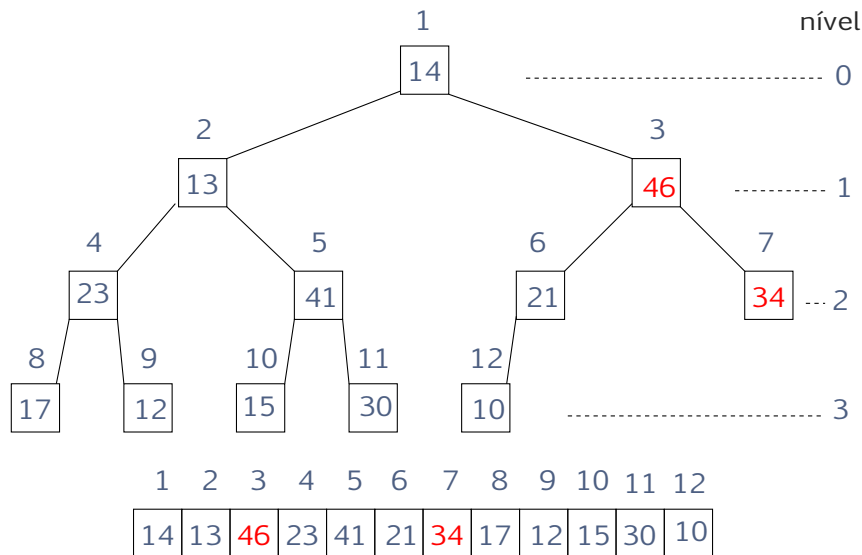
# Construção de um max-heap



# Construção de um max-heap

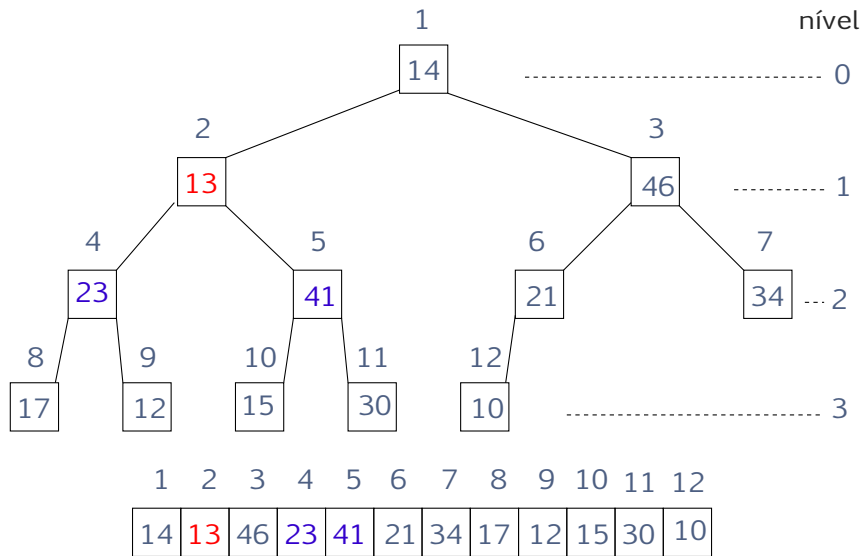


# Construção de um max-heap

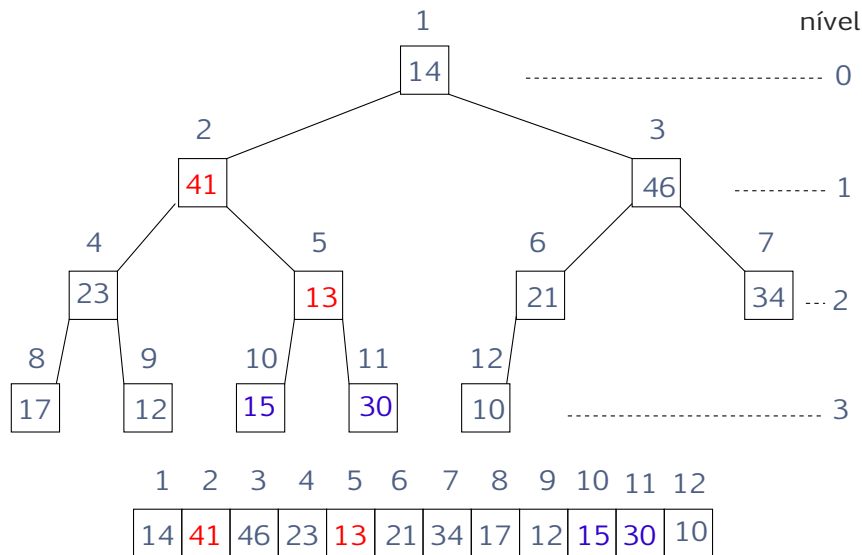




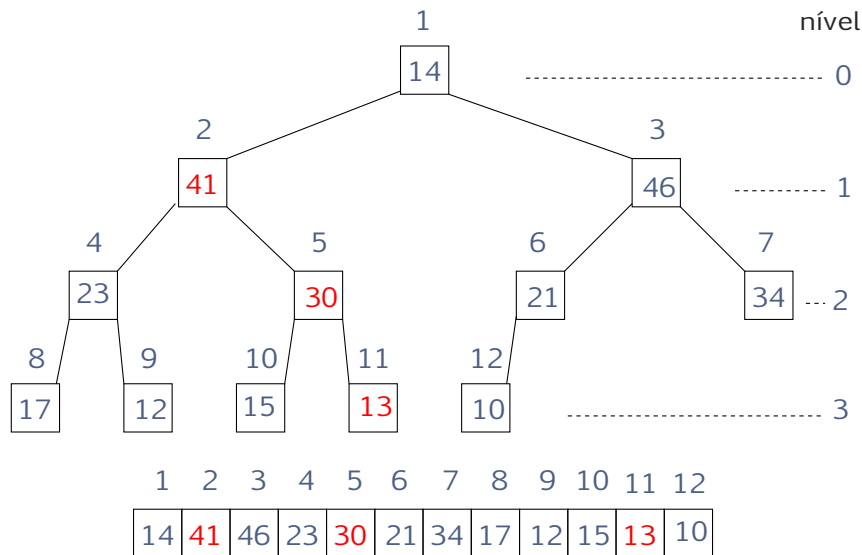
# Construção de um max-heap



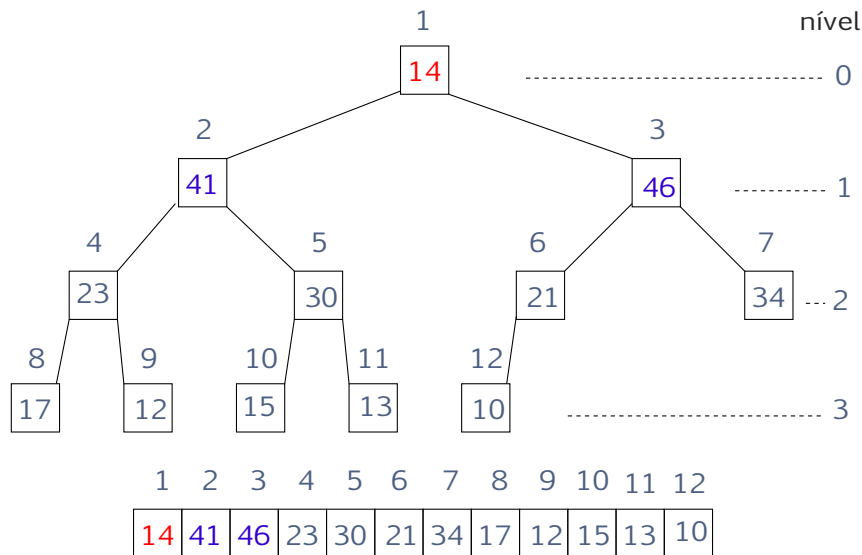
# Construção de um max-heap



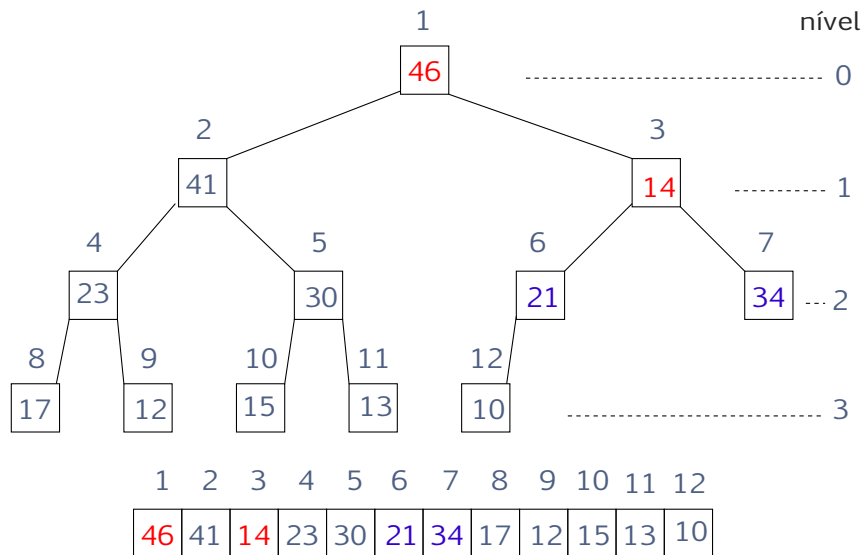
# Construção de um max-heap



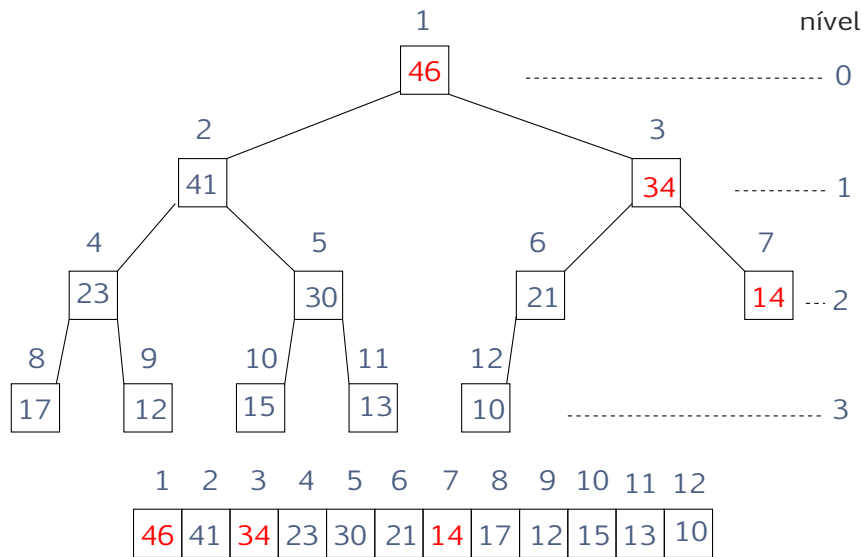
# Construção de um max-heap



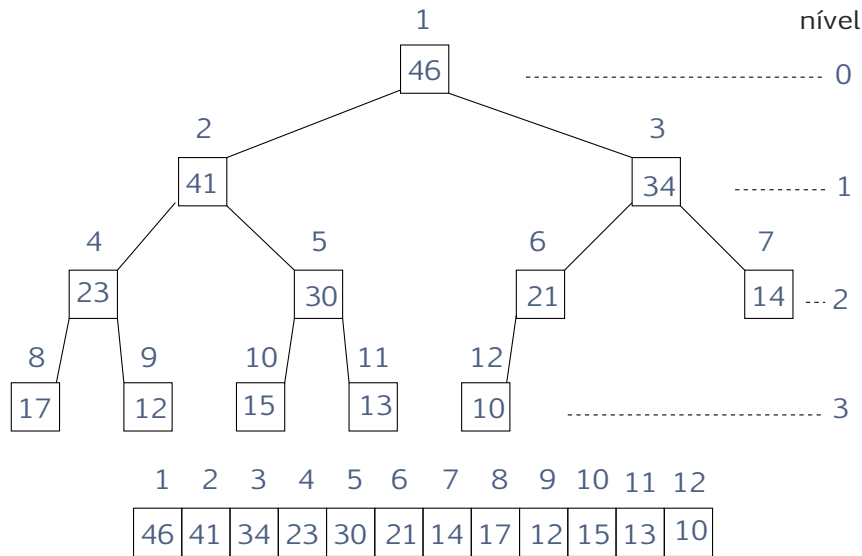
# Construção de um max-heap



# Construção de um max-heap



# Construção de um max-heap



# Análise de BUILD-MAX-HEAP

**BUILD-MAX-HEAP**( $A, n$ )

- 1 para  $i \leftarrow \lfloor n/2 \rfloor$  decrescendo até 1 faça
- 2     **MAX-HEAPIFY**( $A, n, i$ )

## Invariante

No início de cada iteração,  $i + 1, \dots, n$  são raízes de max-heaps.

## Complexidade

- ▶ uma análise rápida leva a  $T(n) = n \cdot O(\lg n) = O(n \lg n)$
- ▶ mas na verdade mostraremos que  $T(n)$  é **linear**!



Análise mais cuidadosa:

- ▶ para um nó de altura  $h$ , MAX-HEAPIFY leva tempo  $O(h)$ 
  - ▶ temos 1 nó de altura  $h$
  - ▶ temos 2 nós de altura  $h/2$
  - ▶ temos 4 nós de altura  $h/4$
  - ▶ e assim por diante
- ▶ vamos somar os tempos de todas as chamadas

## Complexidade de BUILD-MAX-HEAP (cont)

Seja  $k = 1 + \lceil \lg n \rceil$  a altura da árvore inteira, então

$$T(n) = \sum_{h=1}^{k-1} 2^{k-h} \cdot O(h) = O(2^k) \sum_{h=1}^{k-1} \frac{h}{2^h}$$

Note que

$$\begin{aligned} \sum_{h=1}^{k-1} \frac{h}{2^h} &= \left. \frac{1}{2} + \frac{1}{2^2} + \frac{1}{2^3} + \dots + \frac{1}{2^{k-1}} \right\} 1 - \frac{1}{2^{k-1}} < 1 \\ &\quad + \left. \frac{1}{2^2} + \frac{1}{2^3} + \dots + \frac{1}{2^{k-1}} \right\} \frac{1}{2} - \frac{1}{2^{k-1}} < \frac{1}{2} \\ &\quad \quad + \left. \frac{1}{2^3} + \dots + \frac{1}{2^{k-1}} \right\} \frac{1}{4} - \frac{1}{2^{k-1}} < \frac{1}{4} \end{aligned}$$

Ou seja,

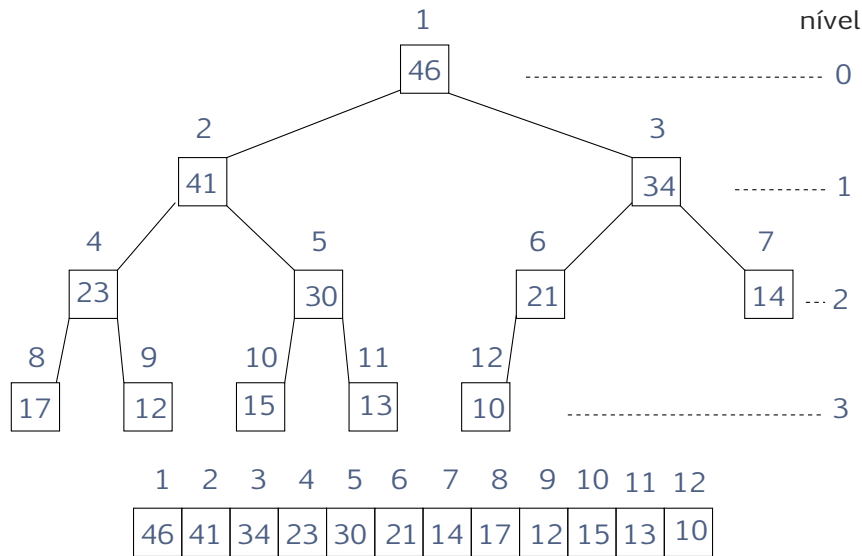
$$T(n) = O(2^k) \sum_{h=0}^k \frac{1}{2^h} \leq O(2^k) \cdot \sum_{h=0}^{\infty} \frac{1}{2^h} = O(2^k) \cdot 2 = O(n)$$

# O algoritmo HEAP-SORT

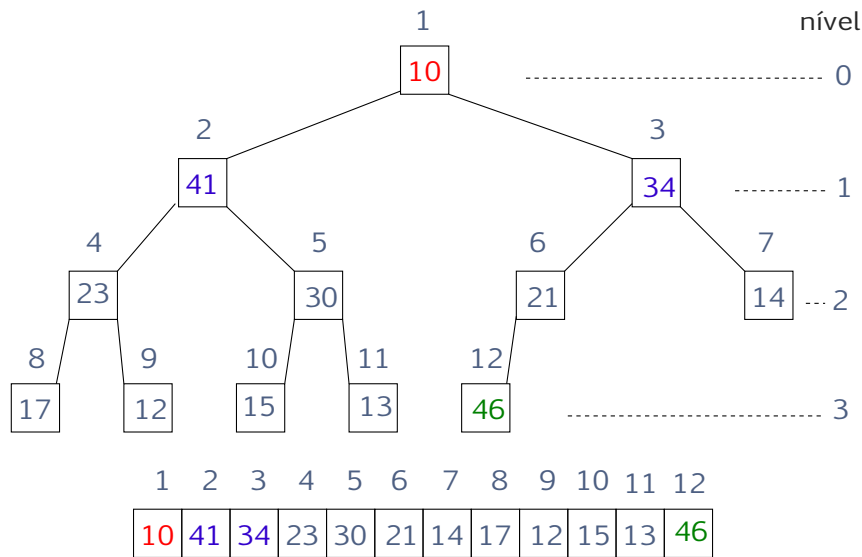
**HEAP-SORT**( $A, n$ )

- 1 BUILD-MAX-HEAP( $A, n$ )
- 2 para  $m \leftarrow n$  decrescendo até 2 faça
- 3      $A[1] \leftrightarrow A[m]$
- 4     MAX-HEAPIFY( $A, m, 1$ )

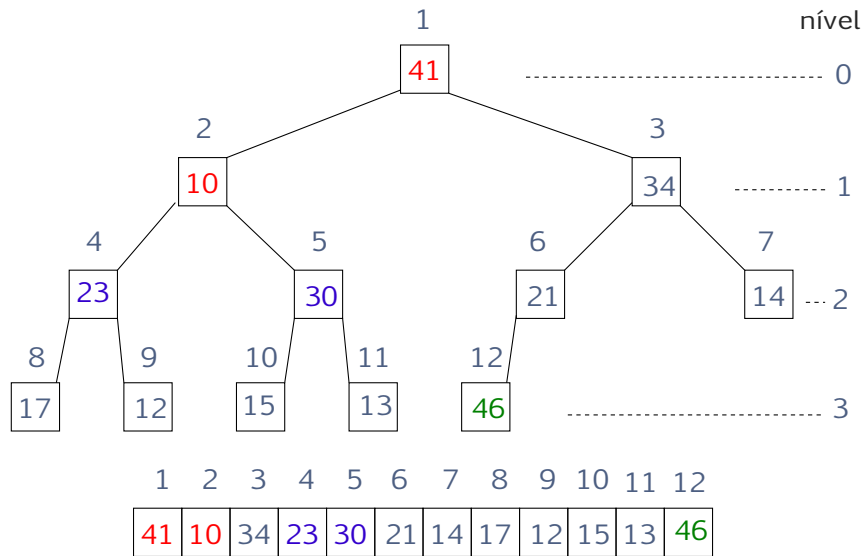
# Execução de HEAP-SORT



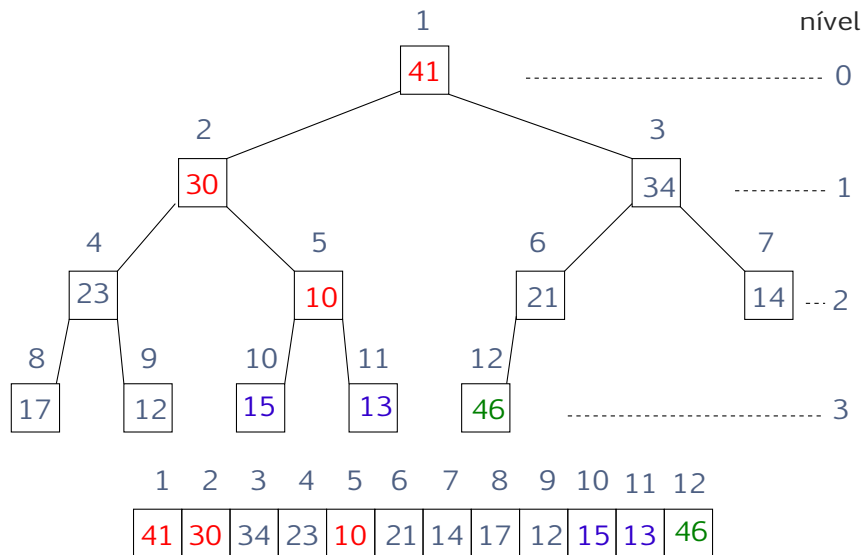
# Execução de HEAP-SORT



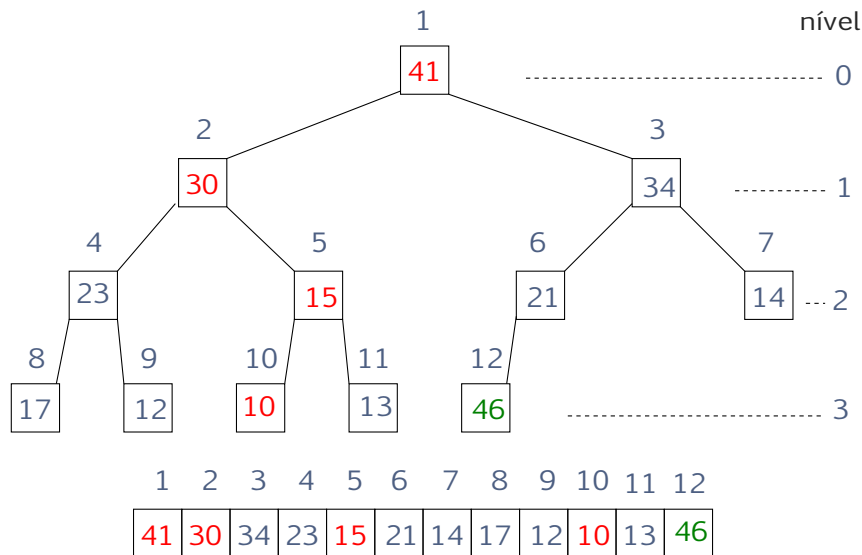
# Execução de HEAP-SORT



# Execução de HEAP-SORT

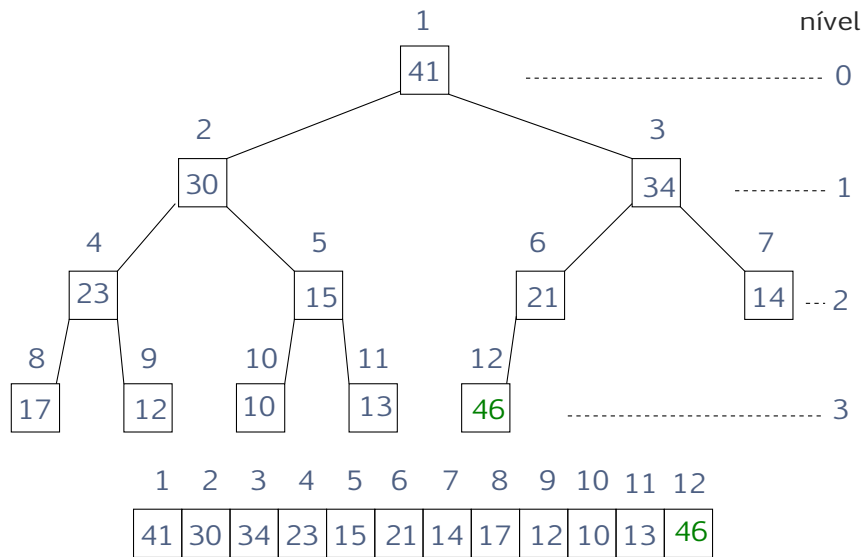


# Execução de HEAP-SORT

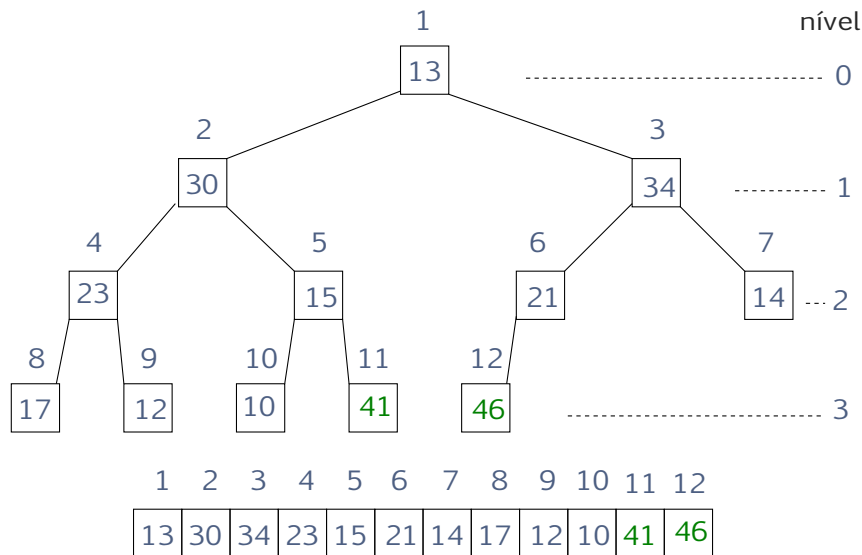




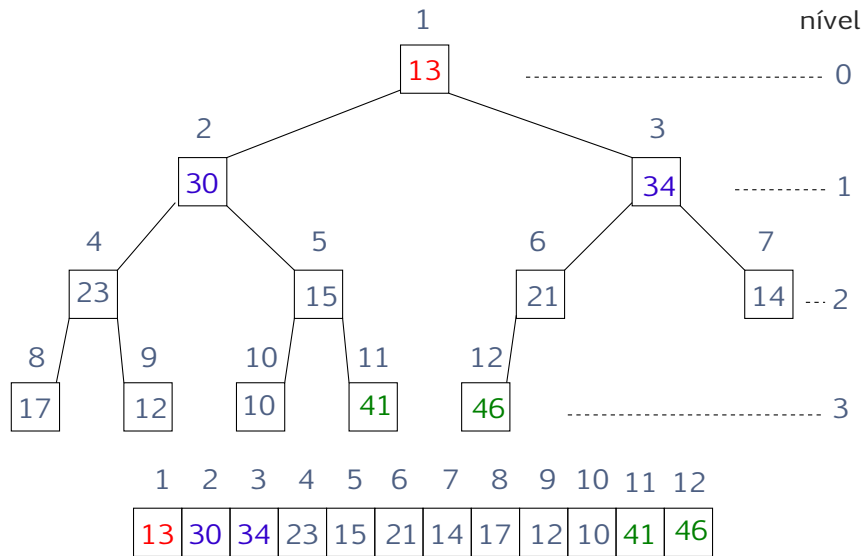
# Execução de HEAP-SORT



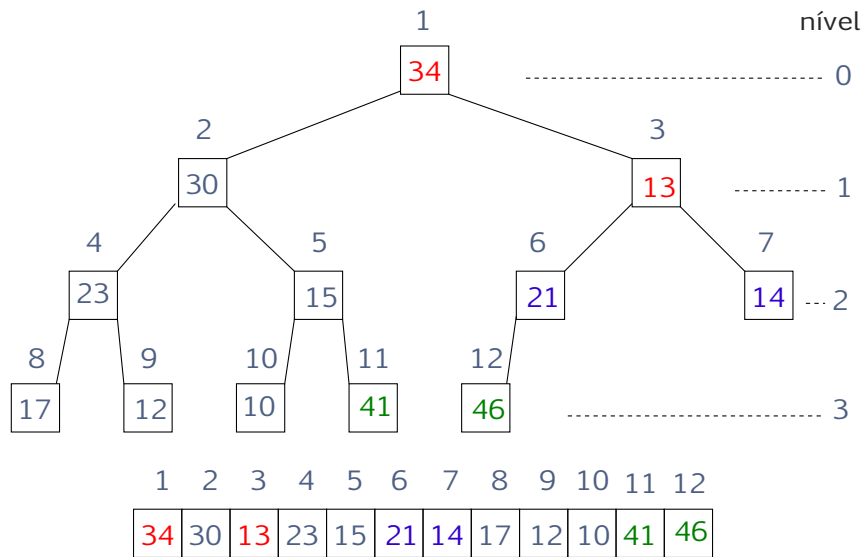
# Execução de HEAP-SORT



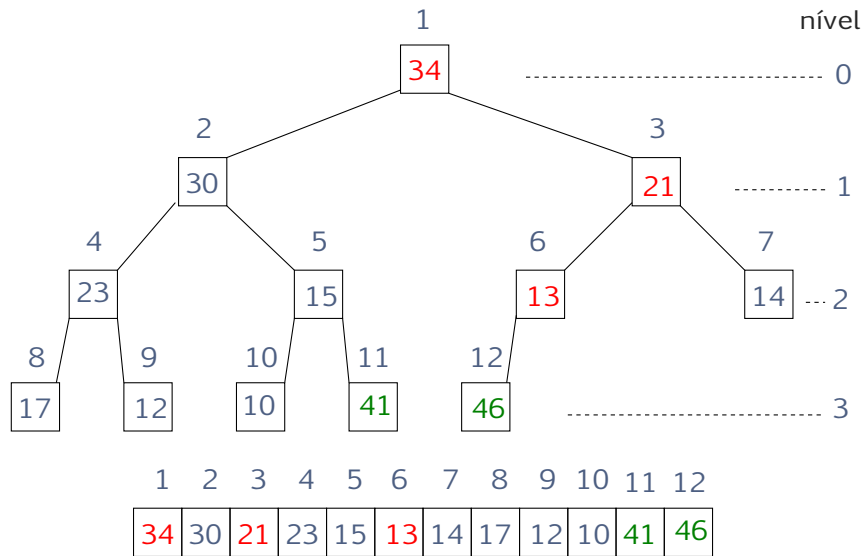
# Execução de HEAP-SORT



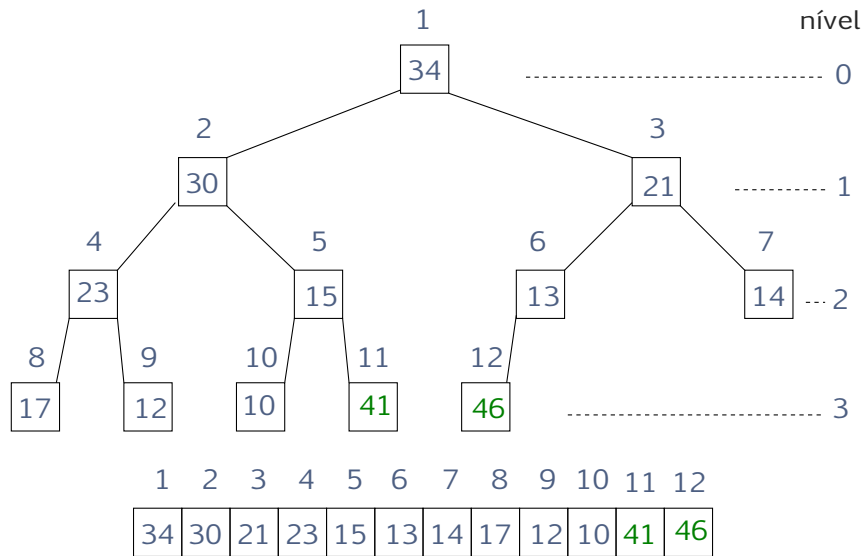
# Execução de HEAP-SORT



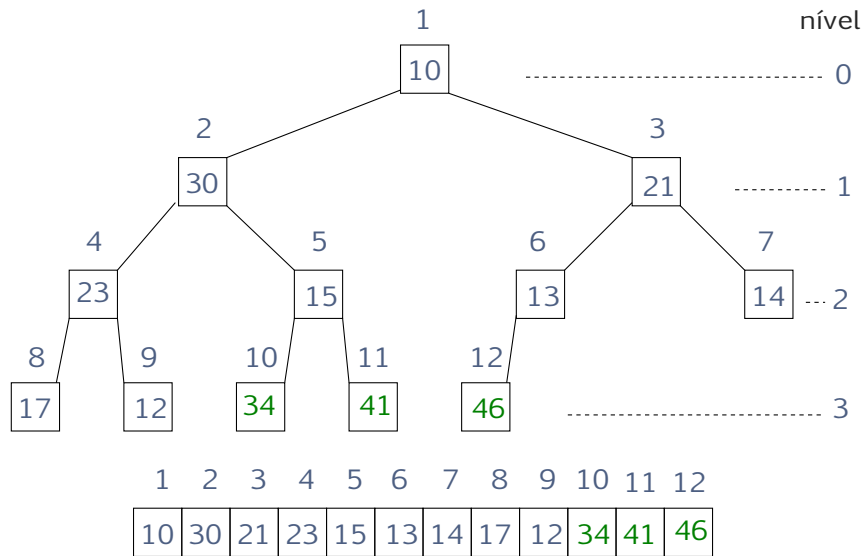
# Execução de HEAP-SORT



# Execução de HEAP-SORT



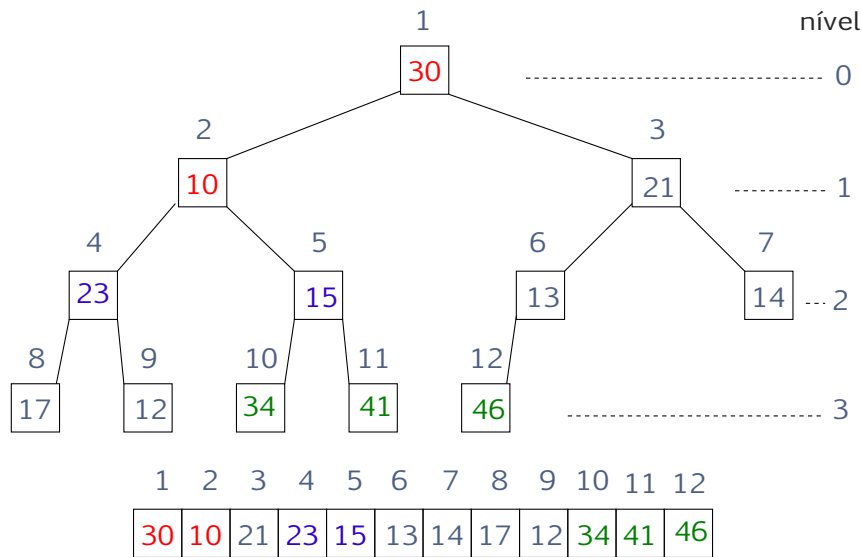
# Execução de HEAP-SORT



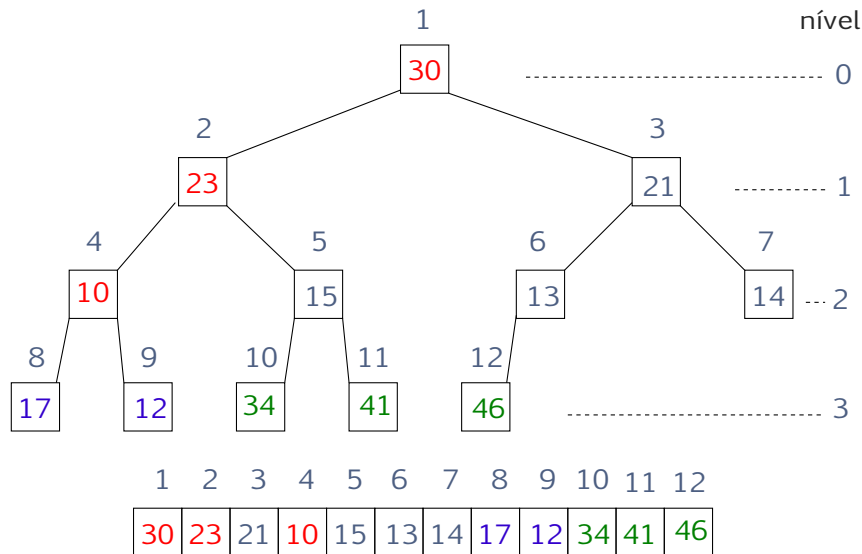




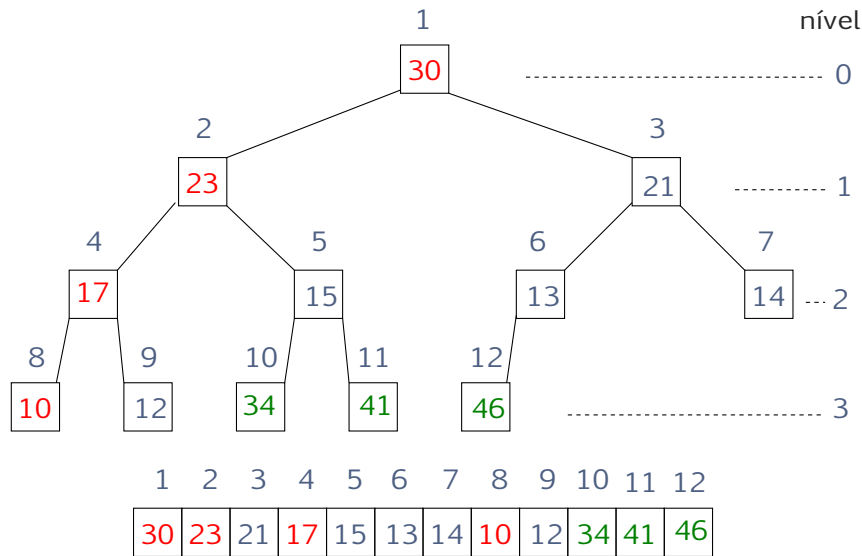
# Execução de HEAP-SORT



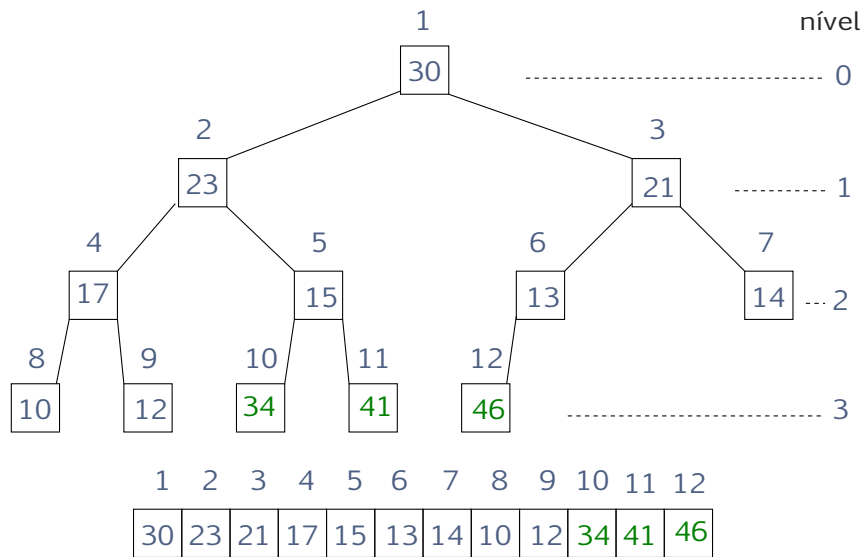
# Execução de HEAP-SORT



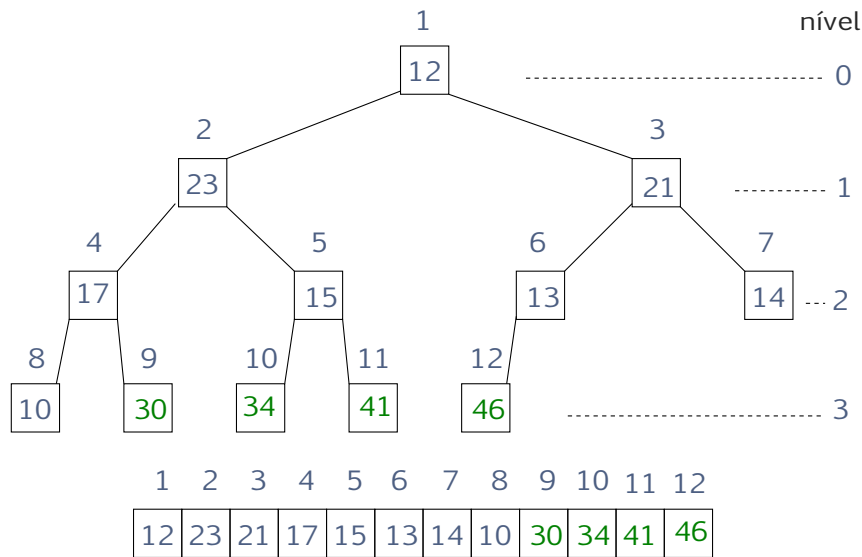
# Execução de HEAP-SORT



# Execução de HEAP-SORT



# Execução de HEAP-SORT



# Análise de HEAP-SORT

**HEAP-SORT**( $A, n$ )

- 1 BUILD-MAX-HEAP( $A, n$ )
- 2 para  $m \leftarrow n$  decrescendo até 2 faça
- 3      $A[1] \leftrightarrow A[m]$
- 4     MAX-HEAPIFY( $A, m, 1$ )

## Invariantes

No início de cada iteração vale

1.  $A[m + 1 \dots n]$  é crescente
2.  $A[1 \dots m] \leq A[m + 1]$
3.  $A[1 \dots m]$  é um max-heap

# Complexidade de HEAP-SORT

<b>HEAP-SORT</b> ( $A, n$ )		Tempo
1	BUILD-MAX-HEAP( $A, n$ )	$\Theta(n)$
2	para $m \leftarrow n$ decrescendo até 2 faça	$\Theta(n)$
3	$A[1] \leftrightarrow A[m]$	$\Theta(n)$
4	MAX-HEAPIFY( $A, m, 1$ )	$n \cdot O(\lg n)$

Consumo de tempo no pior caso?  $O(n \log n)$

E no melhor caso?

Fila de prioridades



## Definição

Uma **fila de prioridades** é um tipo abstrato de dados que consiste de uma coleção  $S$  de itens com prioridades associadas e permite as operações

- ▶  $\text{MAXIMUM}(S)$  devolve um elemento de maior prioridade
- ▶  $\text{EXTRACT-MAX}(S)$  remove um elemento de maior prioridade
- ▶  $\text{INCREASE-KEY}(S, x, p)$  **aumenta** a prioridade de  $x$  para  $p$
- ▶  $\text{INSERT}(S, x, p)$  insere um elemento  $x$  com prioridade  $p$

# Implementação com max-heap

```
HEAP-MAX( $A, n$ )  
1 devolva  $A[1]$ 
```

Complexidade de tempo:  $\Theta(1)$

```
HEAP-EXTRACT-MAX( $A, n$ )  
1  $A[1] \leftarrow A[n]$   
2  $n \leftarrow n - 1$   
3 MAX-HEAPIFY( $A, n, 1$ )
```

Complexidade de tempo:  $O(\lg n)$

# Implementação com max-heap

**HEAP-INCREASE-KEY**( $A, i, chave$ )

```
1   $A[i] \leftarrow chave$   
2  enquanto  $i > 1$  e  $A[\lfloor i/2 \rfloor] < A[i]$  faça  
3       $A[i] \leftrightarrow A[\lfloor i/2 \rfloor]$   
4       $i \leftarrow \lfloor i/2 \rfloor$ 
```

Complexidade de tempo:  $O(\lg n)$

**MAX-HEAP-INSERT**( $A, n, chave$ )

```
1   $n \leftarrow n + 1$   
2   $A[n] \leftarrow -\infty$   
3  HEAP-INCREASE-KEY( $A, n, chave$ )
```

Complexidade de tempo:  $O(\lg n)$