Projeto e Análise de Algoritmos*

Ordenação

Lehilton Pedrosa

Segundo Semestre de 2017

*Criado por C. de Souza, C. da Silva, O. Lee, F. Miyazawa et al.

Ordenação

O problema da ordenação

Problema:

Rearranjar um vetor $A[1 \dots n]$ de inteiros de modo que fique em ordem crescente.

Ou simplesmente:

Problema:

Ordenar um vetor $A[1 \dots n]$ de inteiros.

Algoritmos de ordenação

Veremos vários algoritmos de ordenação:

- *Insertion sort*
- *Selection sort*
- *Mergesort*
- *Heapsort*
- *Quicksort*

Insertion sort

- **Idéia básica:** a cada passo mantemos o subvetor $A[1 \dots j-1]$ ordenado e inserimos o elemento $A[j]$ neste subvetor.
- Repetimos o processo para $j = 2, \dots, n$ e ordenamos o vetor.

Insertion Sort

Insertion sort – pseudocódigo

```
INSERTION-SORT(A, n)
    para j ← 2 até n faça
        chave ← A[j]
        ▷ Insere A[j] no subvetor ordenado A[1..j − 1]
        i ← j − 1
        enquanto i ≥ 1 e A[i] > chave faça
            A[i + 1] ← A[i]
            i ← i − 1
        A[i + 1] ← chave
```

Já analisamos antes a correção e complexidade.

Vamos analisar novamente a complexidade usando a notação assintótica.

Complexidade de tempo de Insertion sort

INSERTION-SORT(A, n)	Tempo
1 **para** $j \leftarrow 2$ **até** n **faça**	$\Theta(n)$
2 $\quad chave \leftarrow A[j]$	$\Theta(n)$
3 $\quad \triangleright$ Insere $A[j]$ em $A[1 \ldots j-1]$	
4 $\quad i \leftarrow j-1$	$\Theta(n)$
5 $\quad$ **enquanto** $i \geq 1$ **e** $A[i] > chave$ **faça**	$nO(n) = O(n^2)$
6 $\qquad A[i+1] \leftarrow A[i]$	$nO(n) = O(n^2)$
7 $\qquad i \leftarrow i-1$	$nO(n) = O(n^2)$
8 $\quad A[i+1] \leftarrow chave$	$O(n)$

Consumo de tempo no pior caso: $O(n^2)$

Insertion sort

- Complexidade de tempo no pior caso: $\Theta(n^2)$
  Vetor em ordem decrescente
  $\Theta(n^2)$ comparações
  $\Theta(n^2)$ movimentações
- Complexidade de tempo no melhor caso: $\Theta(n)$
  (vetor em ordem crescente)
  $O(n)$ comparações
  zero movimentações
- Complexidade de espaço/consumo espaço: $\Theta(n)$

Um pouco de terminologia

- Um algoritmo A tem complexidade de tempo (no **pior caso**) $O(f(n))$ se para **qualquer** entrada de tamanho n ele gasta tempo no **máximo** $O(f(n))$.
- Um algoritmo A tem complexidade de tempo no pior caso $\Theta(f(n))$ se para qualquer entrada de tamanho n ele gasta tempo no **máximo** $O(f(n))$ e para alguma entrada de tamanho n ele gasta tempo pelo menos $\Omega(f(n))$.
- Por exemplo, INSERTION-SORT tem complexidade de tempo no **pior caso** $\Theta(n^2)$.

Um pouco de terminologia

- Faz sentido dizer que um algoritmo tem complexidade de tempo no **pior caso** pelo menos $O(f(n))$?
- Faz sentido dizer que um algoritmo tem complexidade de tempo no **pior caso** $\Omega(f(n))$?
- Faz sentido dizer que um algoritmo tem complexidade de tempo no **melhor caso** $\Omega(f(n))$?

Selection Sort

Selection sort

- Mantemos um subvetor $A[1 \dots i-1]$ tal que:
  1. $A[1 \dots i-1]$ está **ordenado** e
  2. $A[1 \dots i-1] \leq A[i \dots n]$.

  A cada passo selecionamos o menor elemento em $A[i \dots n]$ e o colocamos em $A[i]$.
- Repetimos o processo para $i = 1, \dots, n-1$ e ordenamos vetor.

Selection sort – pseudocódigo

```
SELECTION-SORT(A, n)
    para i ← 1 até n − 1 faça
        min ← i
        para j ← i + 1 até n faça
            se A[j] < A[min] então min ← j
        A[i] ↔ A[min]
```

Invariantes:

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

Complexidade de Selection sort

SELECTION-SORT(A, n)	Tempo
para $i \leftarrow 1$ **até** $n - 1$ **faça**	$\Theta(n)$
$\quad min \leftarrow i$	$\Theta(n)$
$\quad$ **para** $j \leftarrow i + 1$ **até** n **faça**	$\Theta(n^2)$
$\quad\quad$ **se** $A[j] < A[min]$ **então** $min \leftarrow j$	$\Theta(n^2)$
$\quad A[i] \leftrightarrow A[min]$	$\Theta(n)$

Consumo de tempo no pior caso: $O(n^2)$

Selection sort

- Complexidade de tempo no pior caso: $\Theta(n^2)$
  $\Theta(n^2)$ comparações
  $\Theta(n)$ movimentações
- Complexidade de tempo no melhor caso: $\Theta(n^2)$
  Mesmo que o pior caso.
- Complexidade de espaço/consumo espaço: $\Theta(n)$

Conhecimento geral

- Para vetores com no máximo 10 elementos, o melhor algoritmo de ordenação costuma ser *Insertion sort*.
- Para um vetor que está quase ordenado, *Insertion sort* também é a melhor escolha.
- Algoritmos super-eficientes assintoticamente tendem a fazer muitas movimentações, enquanto *Insertion sort* faz poucas movimentações quando o vetor está quase ordenado.

Merge Sort

Mergesort

Vimos que o algoritmo *Mergesort* é um exemplo clássico de paradigma de divisão-e-conquista.

- Divisão: divida o vetor de n elementos em subvetores de tamanhos $\lceil n/2 \rceil$ e $\lfloor n/2 \rfloor$.
- Conquista: recursivamente ordene cada subvetor.
- Combinação: intercale os subvetores ordenados para obter o vetor ordenado.

Mergesort – pseudocódigo

```
MERGE-SORT(A, p, r)
    se p < r
        então q ← ⌊(p + r)/2⌋
              MERGE-SORT(A, p, q)
              MERGE-SORT(A, q + 1, r)
              INTERCALA(A, p, q, r)
```

A complexidade de MERGE-SORT é dada pela recorrência:

$$T(n) = T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + O(f(n)),$$

onde $f(n)$ é a complexidade de INTERCALA.

Correção do Mergesort

MERGE-SORT(A, p, r)
  se $p < r$
    então $q \leftarrow \lfloor (p + r)/2 \rfloor$
      MERGE-SORT(A, p, q)
      MERGE-SORT$(A, q + 1, r)$
      INTERCALA(A, p, q, r)

O algoritmo está correto?

A correção do algoritmo Mergesort apoia-se na correção do algoritmo Intercala e segue facilmente **por indução** em $n := r - p + 1$.

Você consegue ver por quê?

Correção do Mergesort

Base: Mergesort ordena vetores de tamanho 0 ou 1.

Hipótese de indução: Mergesort ordena vetores com $< n$ elementos.

Passo de indução: por hipótese de indução, Mergesort ordena os dois subvetores (de tamanho $\lceil n/2 \rceil$ e $\lfloor n/2 \rfloor$).

Pela correção de Intercala, segue que o vetor resultante da intercalação é um vetor ordenado de n elementos.

Complexidade de Mergesort

Merge-Sort(A, p, r)

se $p < r$

  então $q \leftarrow \lfloor (p + r)/2 \rfloor$

    Merge-Sort(A, p, q)

    Merge-Sort$(A, q + 1, r)$

    Intercala(A, p, q, r)

$T(n)$: complexidade de pior caso de Merge-Sort.
Então

$$T(n) = T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + \Theta(n).$$

A solução da recorrência é $T(n) = \Theta(n \lg n)$.

Mergesort

- Complexidade de tempo: $\Theta(n \lg n)$
  $\Theta(n \lg n)$ comparações
  $\Theta(n \lg n)$ movimentações

  O pior caso e o melhor caso têm a mesma complexidade.
- Complexidade de espaço/consumo espaço: $\Theta(n)$

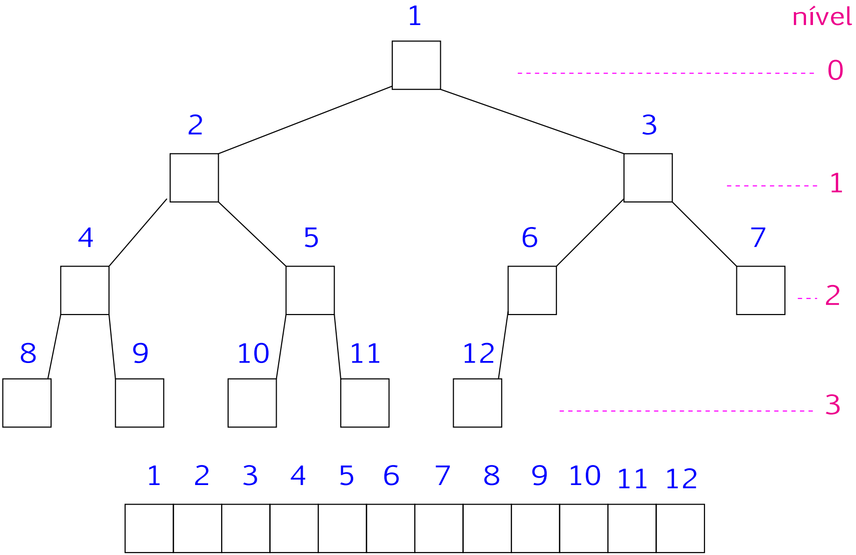
  O *Mergesort* usa um vetor auxiliar de tamanho n para fazer a intercalação, mas o espaço ainda é $\Theta(n)$.
- O *Mergesort* é util para ordenação externa, quando não é possível armazenar todos os elementos na memória primária.

Heap Sort

Heapsort

- O *Heapsort* é um algoritmo de ordenação que usa uma estrutura de dados sofisticada chamada *heap*.
- A complexidade de pior caso é $\Theta(n \lg n)$.
- *Heaps* podem ser utilizados para implementar filas de prioridade que são extremamente úteis em outros algoritmos.
- Um *heap* é um vetor A que simula uma árvore binária completa, com exceção possivelmente do último nível.

Heaps



Heaps

- Considere um vetor $A[1 \dots n]$ representando um heap.
- Cada posição do vetor corresponde a um nó do heap.

Pais

- O pai de um nó i é $\lfloor i/2 \rfloor$.
- O nó 1 não tem pai.

Heaps

Filhos

Um nó i tem

- $2i$ como filho esquerdo e
- $2i+1$ como filho direito.

- O nó i tem filho esquerdo apenas se $2i \leq n$ e
- O nó i tem filho direito apenas se $2i+1 \leq n$.

Folhas

- Um nó i é uma folha se não tem filhos, ou seja, se $2i > n$.
- As folhas são $\lfloor n/2 \rfloor + 1, \ldots, n-1, n$.

Níveis

- Cada nível p, exceto talvez o último, tem exatamente 2^p nós
- Esses nós são

$$2^p, 2^p + 1, 2^p + 2, \dots, 2^{p+1} - 1.$$

Nível do item i

- O nó i pertence ao nível $???\lfloor \lg i \rfloor$.
- Prova: Se p é o nível do nó i, então

$$\begin{array}{rcccll} 2^p & \leq & i & < & 2^{p+1} & \Rightarrow \\ \lg 2^p & \leq & \lg i & < & \lg 2^{p+1} & \Rightarrow \\ p & \leq & \lg i & < & p+1 & \end{array}$$

Logo, $p = \lfloor \lg i \rfloor$.

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

Altura

- A altura de um nó i é o maior comprimento de um caminho de i a uma folha.
- Os nós que têm altura zero são as folhas.

Qual é a altura de um nó i?

Altura

A altura de um nó i é o comprimento da sequência

$$2\,i, 2^2\,i, 2^3\,i, \ldots, 2^h\,i$$

onde $2^h\,i \leq n < 2^{(h+1)}\,i$.

Assim,

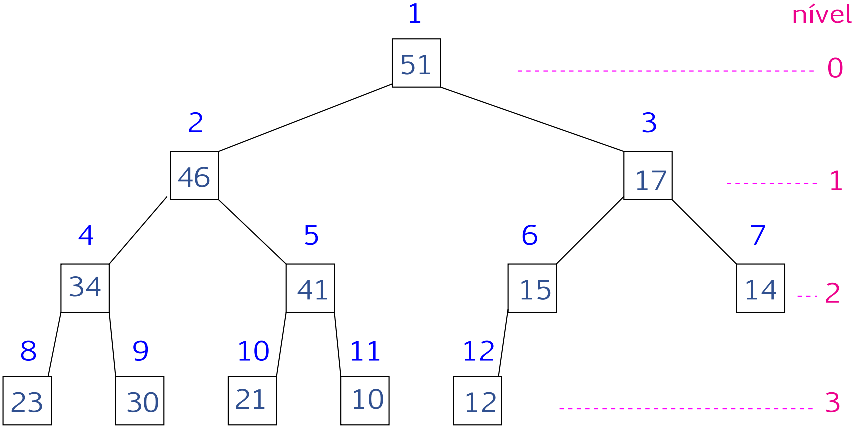
$$\begin{array}{rcccll} 2^h\,i & \leq & n & < & 2^{h+1}i & \Rightarrow \\ 2^h & \leq & n/i & < & 2^{h+1} & \Rightarrow \\ h & \leq & \lg(n/i) & < & h+1 & \end{array}$$

Portanto, a altura de i é $\lfloor \lg(n/i) \rfloor$.

Max-heaps

- Um nó i satisfaz a propriedade de (max-)heap se $A[\lfloor i/2 \rfloor] \geq A[i]$ (ou seja, pai $\geq$ filho).
- Uma árvore binária completa é um *max-heap* se todo nó distinto da raiz satisfaz a propriedade de heap.
- O máximo ou maior elemento de um max-heap está na raiz.

Max-heap

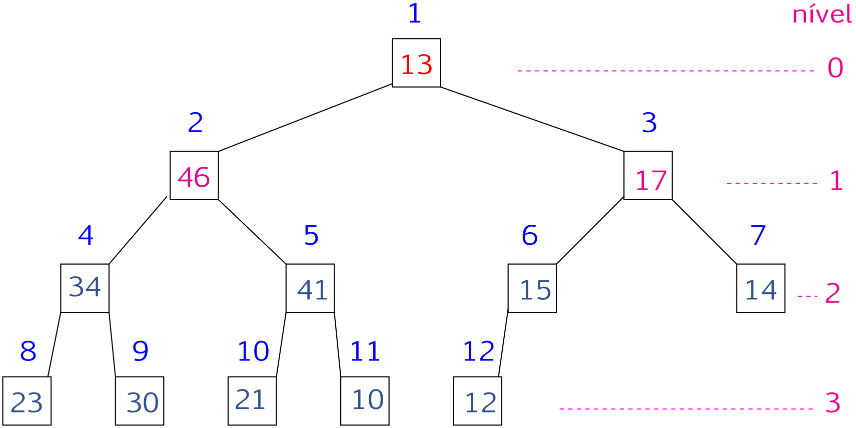


1	2	3	4	5	6	7	8	9	10	11	12
51	46	17	34	41	15	14	23	30	21	10	12

Min-heaps

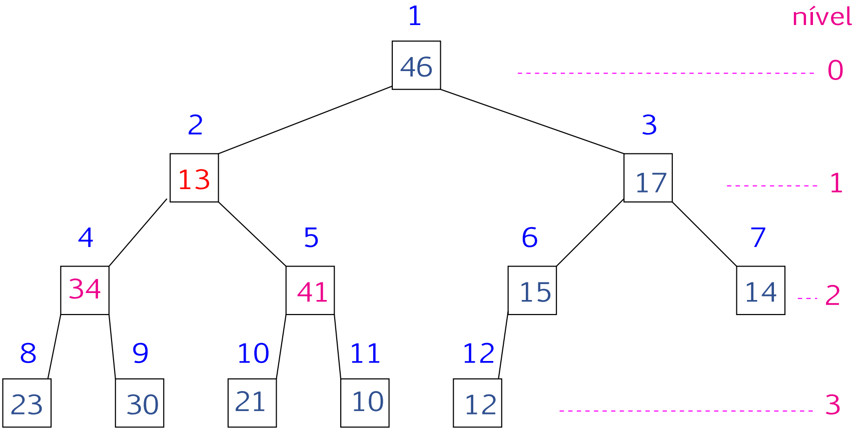
- Um nó i satisfaz a propriedade de (min-)heap se $A[\lfloor i/2 \rfloor] \leq A[i]$ (ou seja, pai $\leq$ filho).
- Uma árvore binária completa é um *min-heap* se todo nó distinto da raiz satisfaz a propriedade de min-heap.
- Vamos nos concentrar apenas em max-heaps.
- Os algoritmos que veremos podem ser facilmente modificados para trabalhar com min-heaps.

Manipulação de max-heap



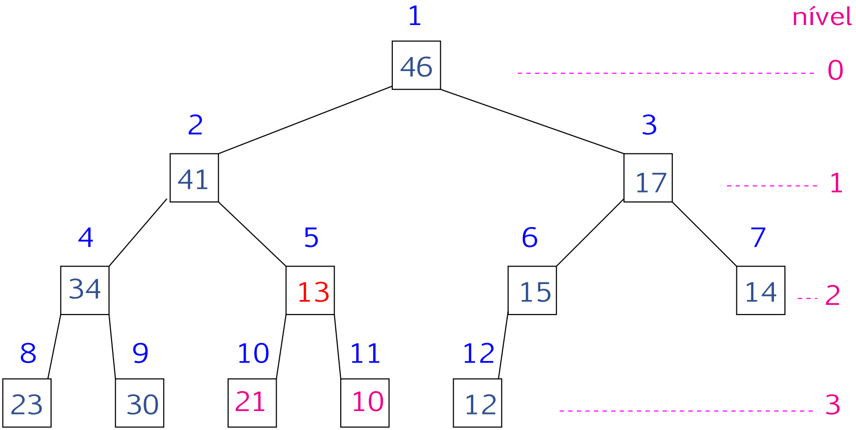
1	2	3	4	5	6	7	8	9	10	11	12
13	46	17	34	41	15	14	23	30	21	10	12

Manipulação de max-heap



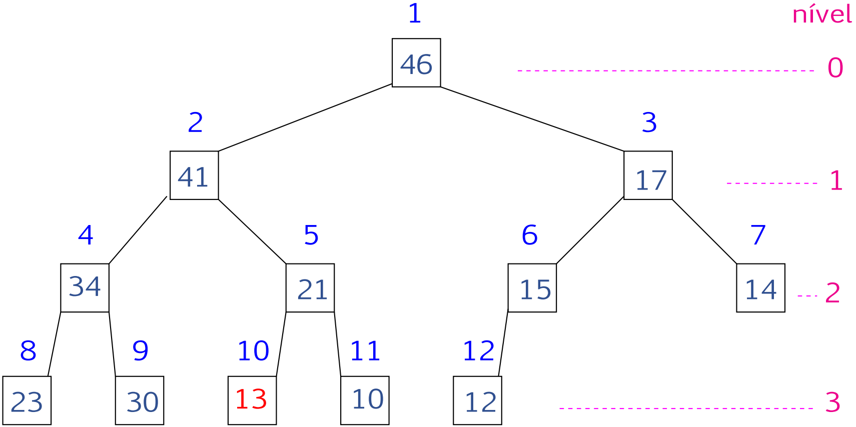
1	2	3	4	5	6	7	8	9	10	11	12
46	13	17	34	41	15	14	23	30	21	10	12

Manipulação de max-heap



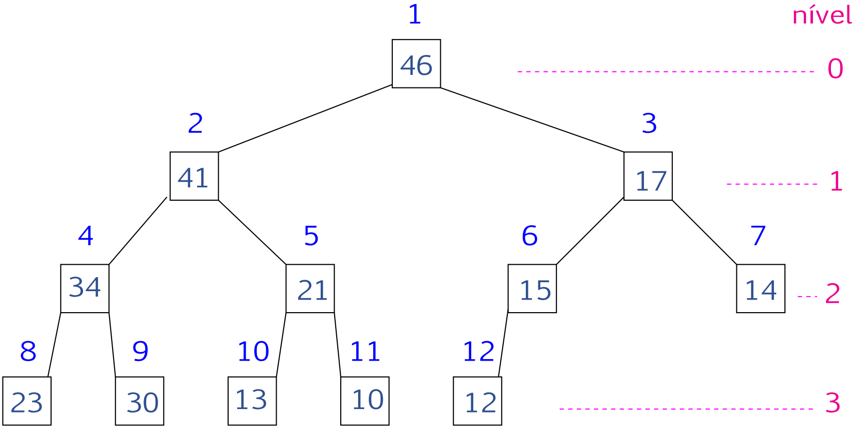
1	2	3	4	5	6	7	8	9	10	11	12
46	41	17	34	13	15	14	23	30	21	10	12

Manipulação de max-heap



1	2	3	4	5	6	7	8	9	10	11	12
46	41	17	34	21	15	14	23	30	13	10	12

Manipulação de max-heap



1	2	3	4	5	6	7	8	9	10	11	12
46	41	17	34	21	15	14	23	30	13	10	12

Manipulação de max-heap

Recebe $A[1 \dots n]$ e $i \geq 1$ tais que subárvores com raízes $2i$ e $2i+1$ são max-heaps e rearranja A de modo que subárvore com raiz i seja um max-heap.

```
MAX-HEAPIFY(A, n, i)
    e ← 2i
    d ← 2i + 1
    se e ≤ n e A[e] > A[i]
        então maior ← e
        senão maior ← i
    se d ≤ n e A[d] > A[maior]
        então maior ← d
    se maior ≠ i
        então A[i] ↔ A[maior]
                MAX-HEAPIFY(A, n, maior)
```

Correção de Max-Heapify

A correção de Max-Heapify segue por indução na altura h do nó i.

Base: para $h = 0$, o algoritmo funciona.

Hipótese de indução: Max-Heapify funciona para heaps de altura $< h$.

Passo de indução:

A variável *maior* na linha 8 guarda o índice do maior elemento entre $A[i]$, $A[2i]$ e $A[2i+1]$.
Após a troca na linha 9, temos $A[2i], A[2i+1] \leq A[i]$.

O algoritmo Max-Heapify transforma a subárvore com raiz *maior* em um max-heap (hipótese de indução).

Correção de Max-Heapify

Passo de indução:

A variável *maior* na linha 8 guarda o índice do maior elemento entre $A[i]$, $A[2i]$ e $A[2i+1]$.
Após a troca na linha 9, temos $A[2i], A[2i+1] \leq A[i]$.

O algoritmo Max-Heapify transforma a subárvore com raiz *maior* em um max-heap (hipótese de indução).

A subárvore cuja raiz é o irmão de *maior* continua sendo um max-heap.

Logo, a subárvore com raiz i torna-se um max-heap e portanto, o algoritmo Max-Heapify está correto.

Complexidade de Max-Heapify

Max-Heapify(A, n, i)	Tempo
$e \leftarrow 2i$	$\Theta(1)$
$d \leftarrow 2i+1$	$\Theta(1)$
se $e \leq n$ **e** $A[e] > A[i]$	$\Theta(1)$
então $maior \leftarrow e$	$O(1)$
senão $maior \leftarrow i$	$O(1)$
se $d \leq n$ e $A[d] > A[maior]$	$\Theta(1)$
então $maior \leftarrow d$	$O(1)$
se $maior \neq i$	$\Theta(1)$
então $A[i] \leftrightarrow A[maior]$	$O(1)$
Max-Heapify$(A, n, maior)$	$T(h-1)$

h := altura de i = $\lfloor \lg \frac{n}{i} \rfloor$

$T(h) \leq T(h-1) + \Theta(5) + O(2)$.

Complexidade de MAX-HEAPIFY

h := altura de $i = \lfloor \lg \frac{n}{i} \rfloor$

$T(h)$:= complexidade de tempo no pior caso

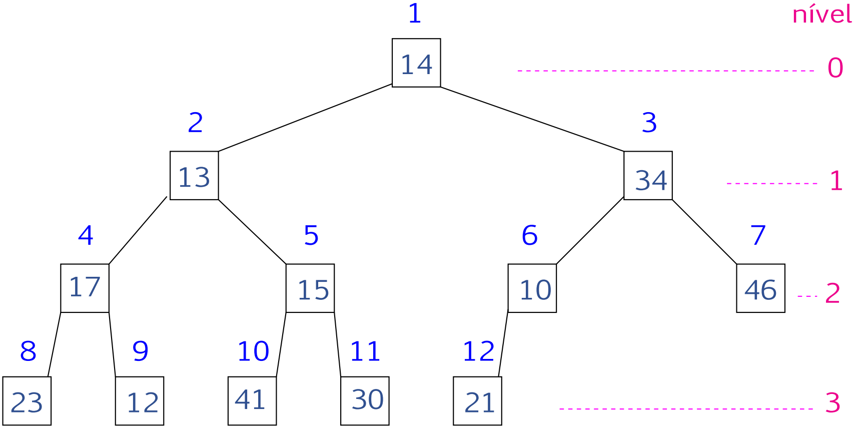
$T(h) \leq T(h-1) + \Theta(1)$

Solução assintótica: $T(n)$ é $O(h)$.

Como $h \leq \lg n$, podemos dizer que:

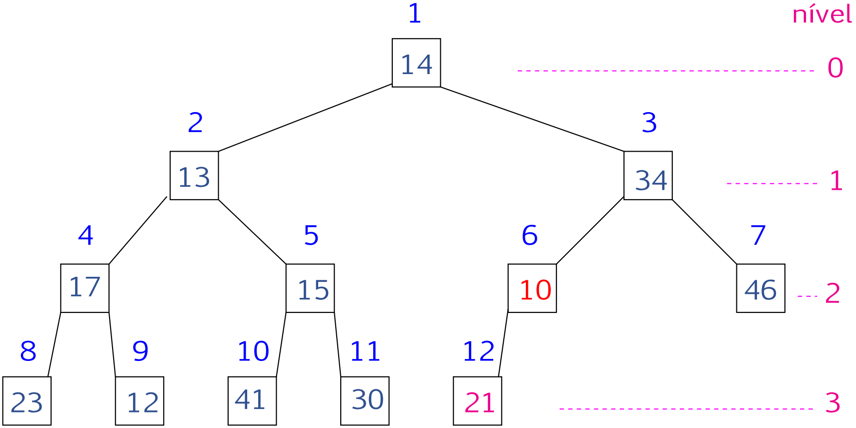
O consumo de tempo do algoritmo MAX-HEAPIFY é $O(\lg n)$
(ou melhor ainda, $O(\lg \frac{n}{i})$).

Construção de um max-heap



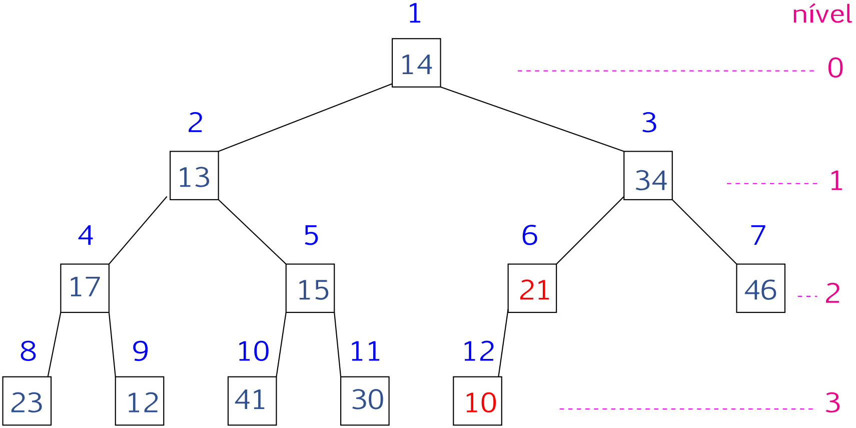
1	2	3	4	5	6	7	8	9	10	11	12
14	13	34	17	15	10	46	23	12	41	30	21

Construção de um max-heap



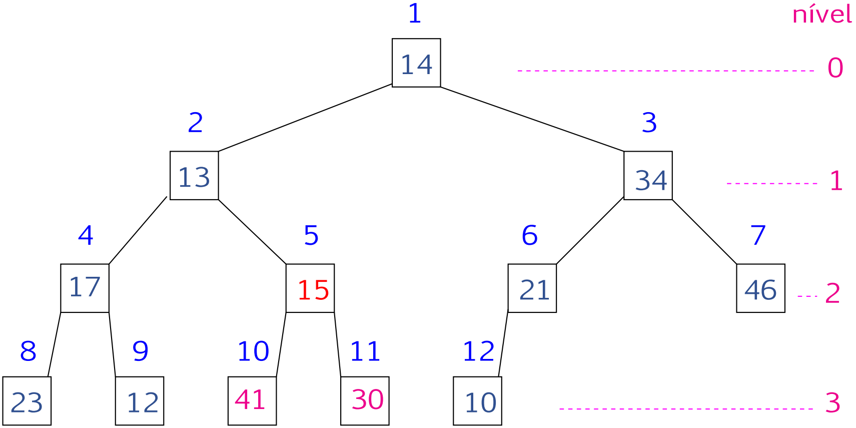
1	2	3	4	5	6	7	8	9	10	11	12
14	13	34	17	15	10	46	23	12	41	30	21

Construção de um max-heap



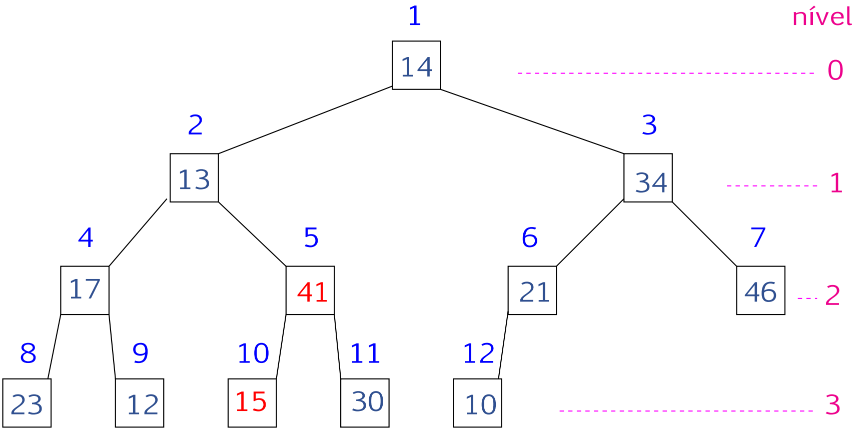
1	2	3	4	5	6	7	8	9	10	11	12
14	13	34	17	15	21	46	23	12	41	30	10

Construção de um max-heap



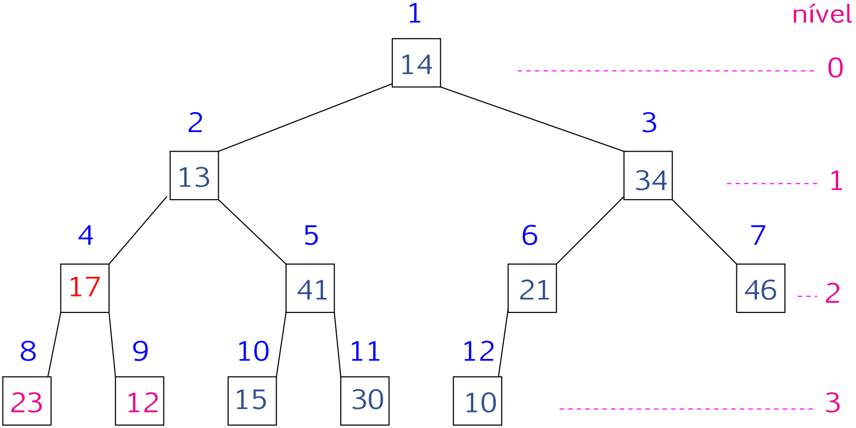
1	2	3	4	5	6	7	8	9	10	11	12
14	13	34	17	15	21	46	23	12	41	30	10

Construção de um max-heap



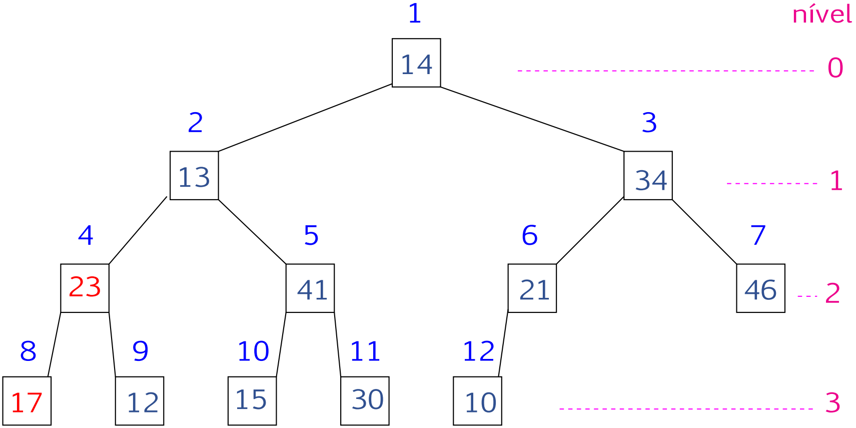
1	2	3	4	5	6	7	8	9	10	11	12
14	13	34	17	41	21	46	23	12	15	30	10

Construção de um max-heap



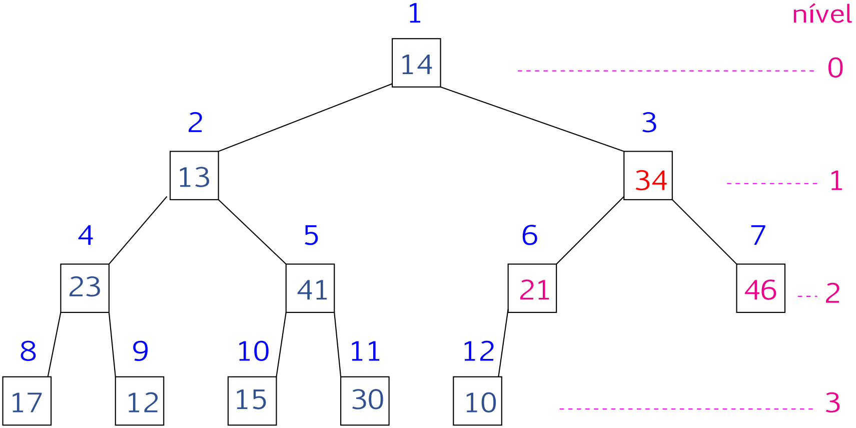
1	2	3	4	5	6	7	8	9	10	11	12
14	13	34	17	41	21	46	23	12	15	30	10

Construção de um max-heap



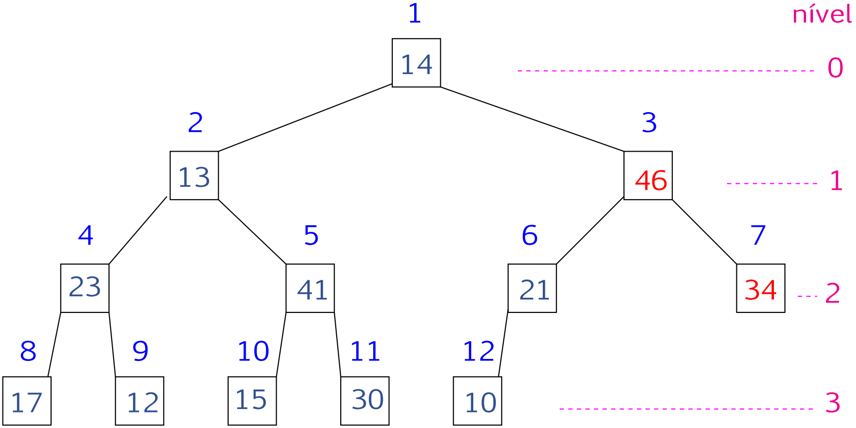
1	2	3	4	5	6	7	8	9	10	11	12
14	13	34	23	41	21	46	17	12	15	30	10

Construção de um max-heap



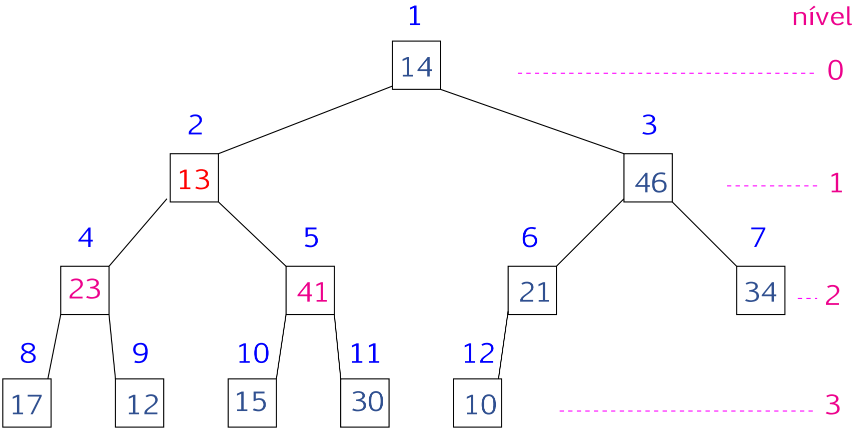
1	2	3	4	5	6	7	8	9	10	11	12
14	13	34	23	41	21	46	17	12	15	30	10

Construção de um max-heap



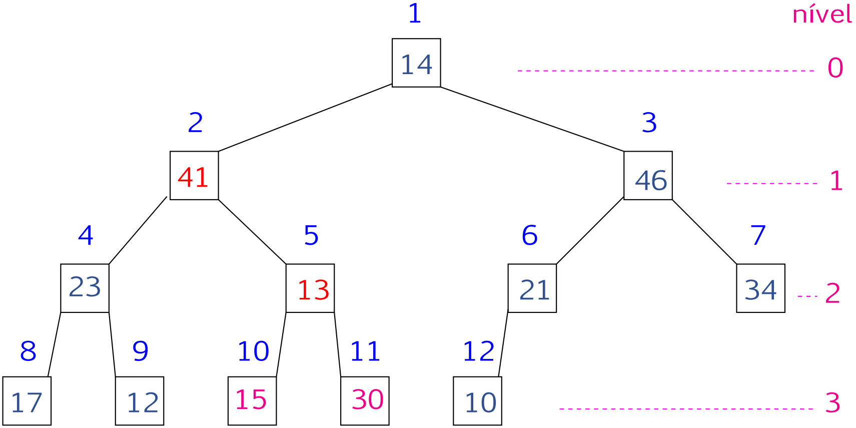
1	2	3	4	5	6	7	8	9	10	11	12
14	13	46	23	41	21	34	17	12	15	30	10

Construção de um max-heap



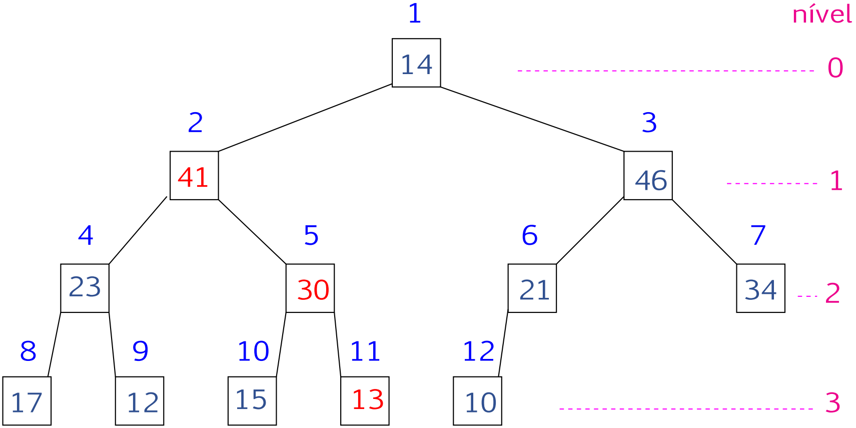
1	2	3	4	5	6	7	8	9	10	11	12
14	13	46	23	41	21	34	17	12	15	30	10

Construção de um max-heap



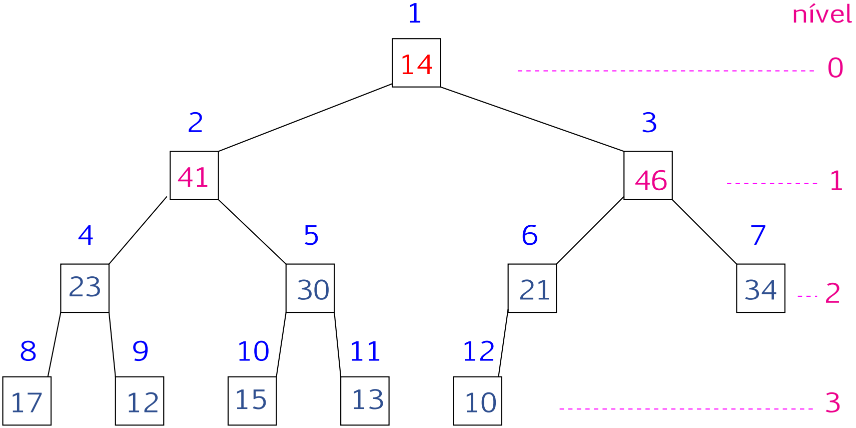
1	2	3	4	5	6	7	8	9	10	11	12
14	41	46	23	13	21	34	17	12	15	30	10

Construção de um max-heap



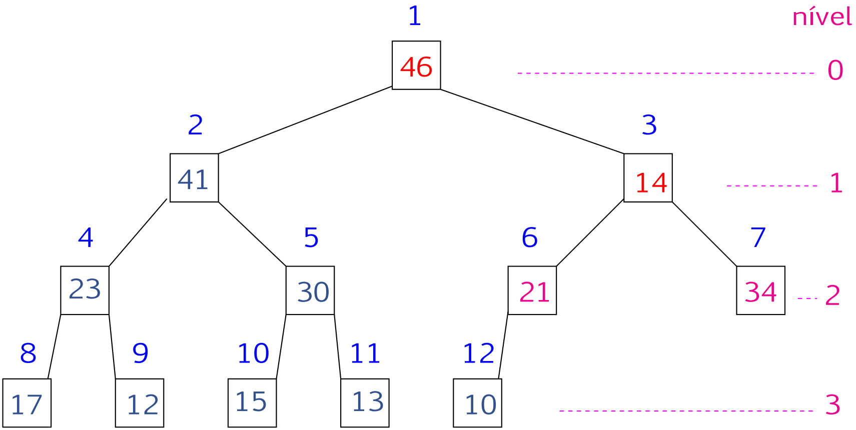
1	2	3	4	5	6	7	8	9	10	11	12
14	41	46	23	30	21	34	17	12	15	13	10

Construção de um max-heap



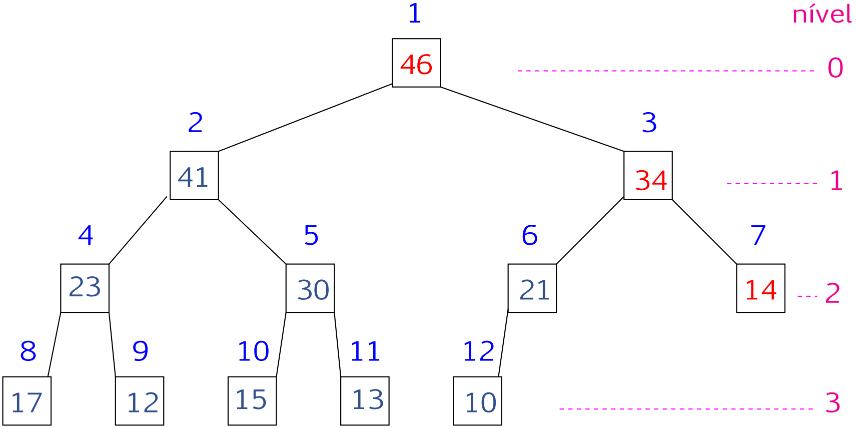
1	2	3	4	5	6	7	8	9	10	11	12
14	41	46	23	30	21	34	17	12	15	13	10

Construção de um max-heap



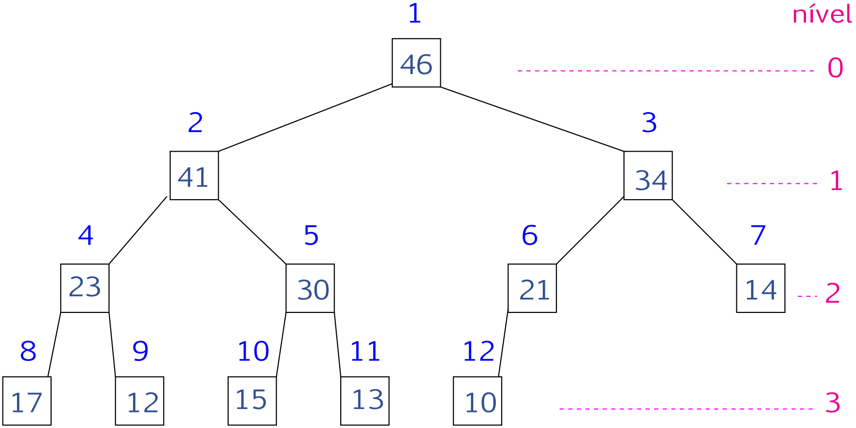
1	2	3	4	5	6	7	8	9	10	11	12
46	41	14	23	30	21	34	17	12	15	13	10

Construção de um max-heap



1	2	3	4	5	6	7	8	9	10	11	12
46	41	34	23	30	21	14	17	12	15	13	10

Construção de um max-heap



1	2	3	4	5	6	7	8	9	10	11	12
46	41	34	23	30	21	14	17	12	15	13	10

Construção de um max-heap

Recebe um vetor $A[1\dots n]$ e rearranja A para que seja max-heap.

```
BUILD-MAX-HEAP(A, n)
    para i ← ⌊n/2⌋ decrescendo até 1 faça
        MAX-HEAPIFY(A, n, i)
```

Invariante:

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

$T(n) =$ complexidade de tempo no pior caso

Análise grosseira: $T(n)$ é $\frac{n}{2}\,O(\lg n) = O(n \lg n)$.

Construção de um max-heap

Análise mais cuidadosa: $T(n)$ é $O(n)$.

- Na iteração i são feitas $O(h_i)$ comparações e trocas no pior caso, onde h_i é a altura da subárvore de raiz i.
- Seja $S(h)$ a soma das alturas de todos os nós de uma árvore binária completa de altura h.
- A altura de um heap é $\lfloor \lg n \rfloor + 1$.

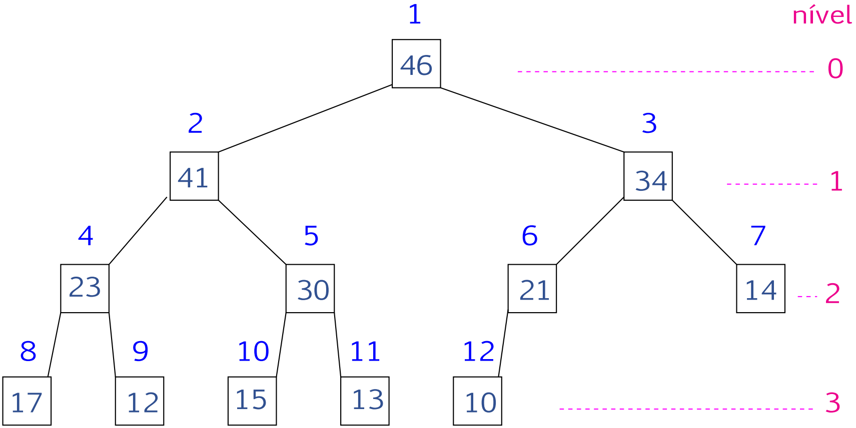
  A complexidade de Build-Max-Heap é $T(n) = O(S(\lg n))$.

Construção de um max-heap

- Pode-se provar por indução que $S(h) = 2^{h+1} - h - 2$.
- Logo, a complexidade de BUILD-MAX-HEAP é $T(n) = O(S(\lg n)) = O(n)$.

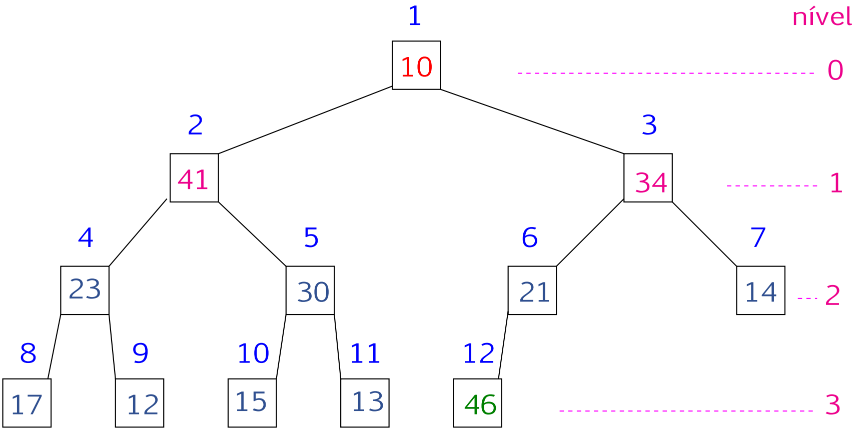
  Mais precisamente, $T(n) = \Theta(n)$. (Por quê?)
- Veja no CLRS uma prova diferente deste fato.

HeapSort



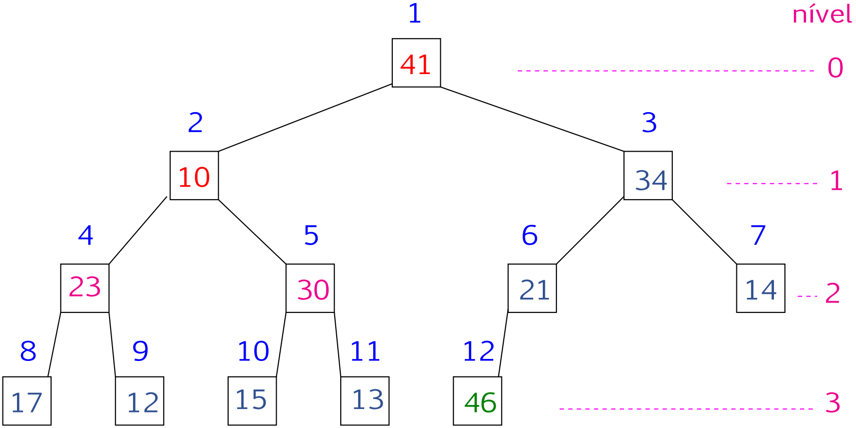
1	2	3	4	5	6	7	8	9	10	11	12
46	41	34	23	30	21	14	17	12	15	13	10

HeapSort



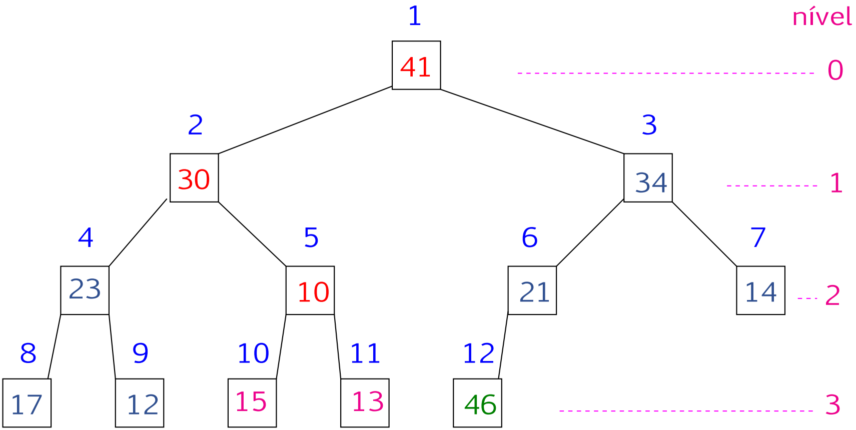
1	2	3	4	5	6	7	8	9	10	11	12
10	41	34	23	30	21	14	17	12	15	13	46

HeapSort



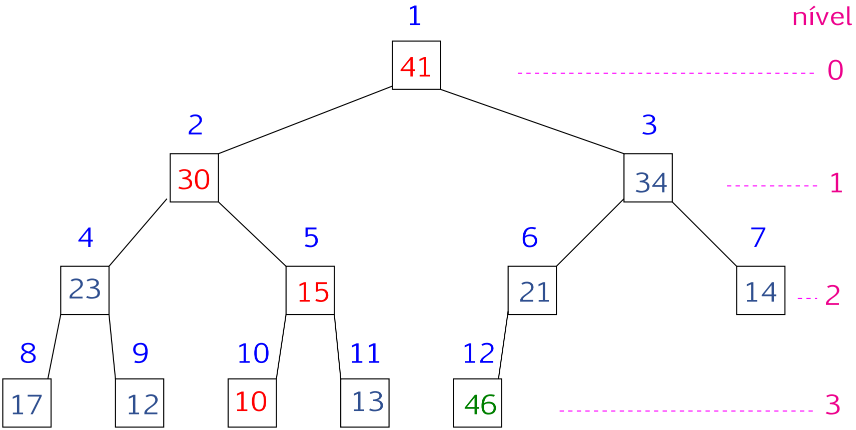
1	2	3	4	5	6	7	8	9	10	11	12
41	10	34	23	30	21	14	17	12	15	13	46

HeapSort



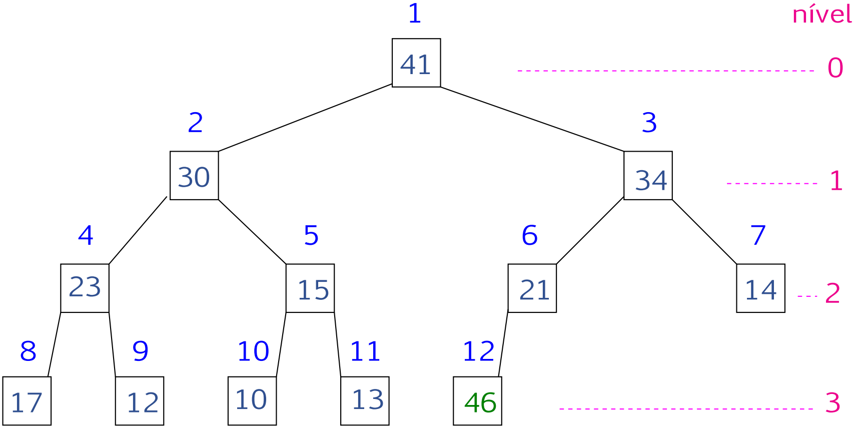
1	2	3	4	5	6	7	8	9	10	11	12
41	30	34	23	10	21	14	17	12	15	13	46

HeapSort



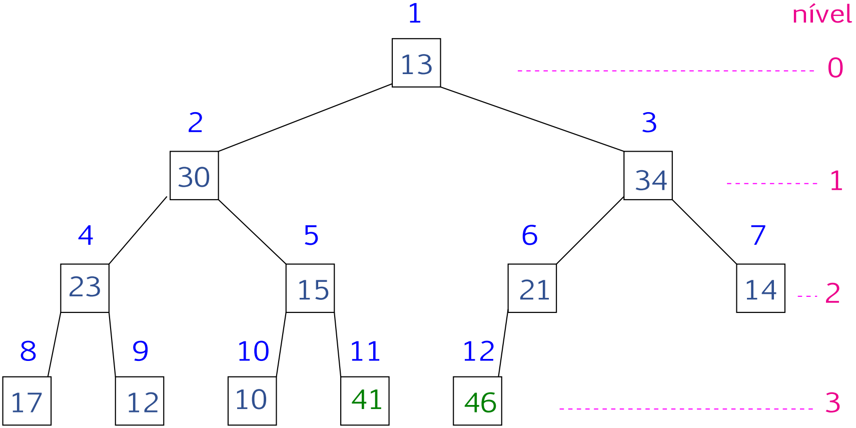
1	2	3	4	5	6	7	8	9	10	11	12
41	30	34	23	15	21	14	17	12	10	13	46

HeapSort



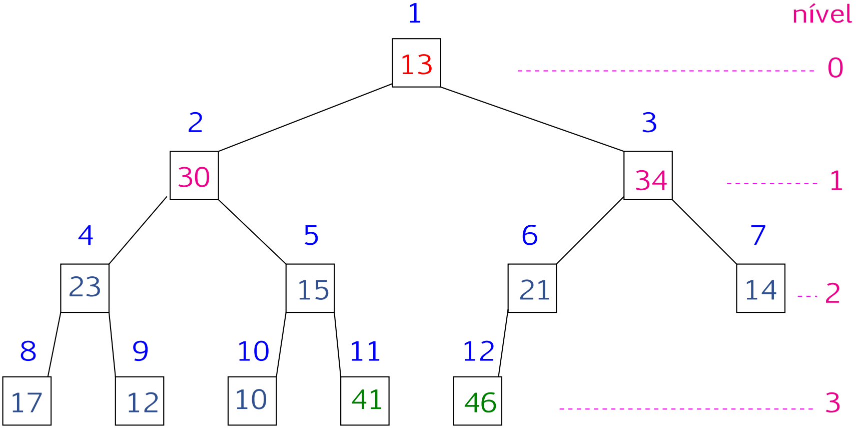
1	2	3	4	5	6	7	8	9	10	11	12
41	30	34	23	15	21	14	17	12	10	13	46

HeapSort



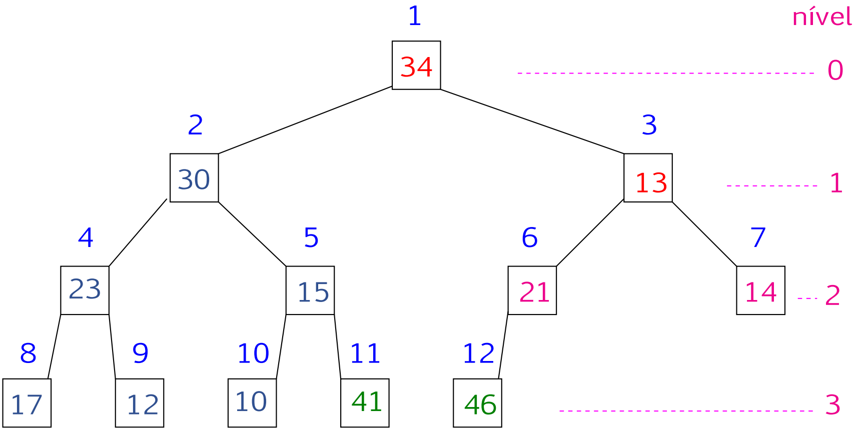
1	2	3	4	5	6	7	8	9	10	11	12
13	30	34	23	15	21	14	17	12	10	41	46

HeapSort



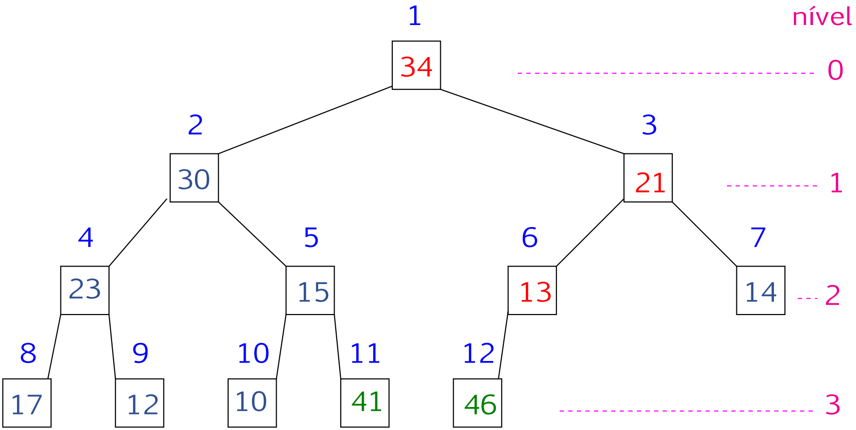
1	2	3	4	5	6	7	8	9	10	11	12
13	30	34	23	15	21	14	17	12	10	41	46

HeapSort



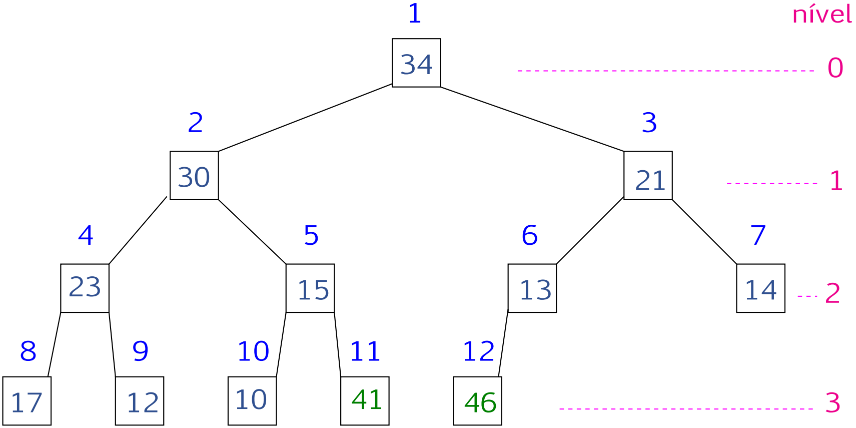
1	2	3	4	5	6	7	8	9	10	11	12
34	30	13	23	15	21	14	17	12	10	41	46

HeapSort



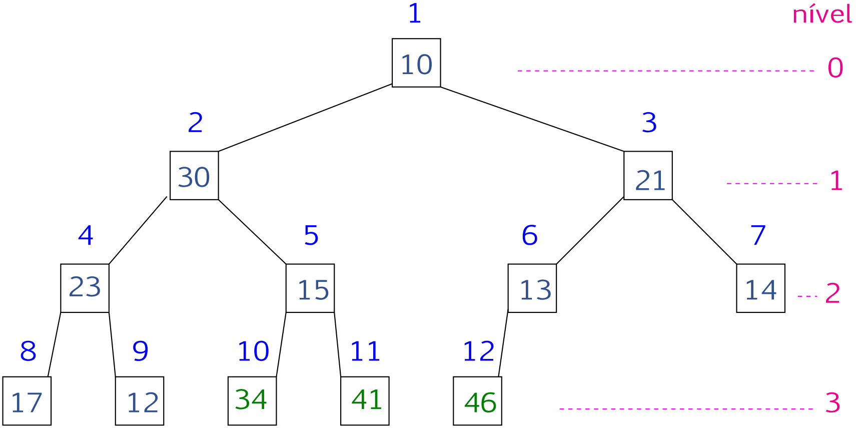
1	2	3	4	5	6	7	8	9	10	11	12
34	30	21	23	15	13	14	17	12	10	41	46

HeapSort



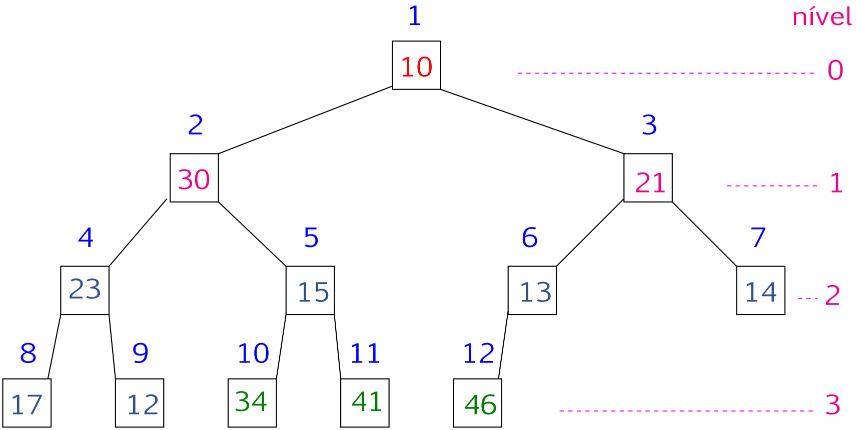
1	2	3	4	5	6	7	8	9	10	11	12
34	30	21	23	15	13	14	17	12	10	41	46

HeapSort



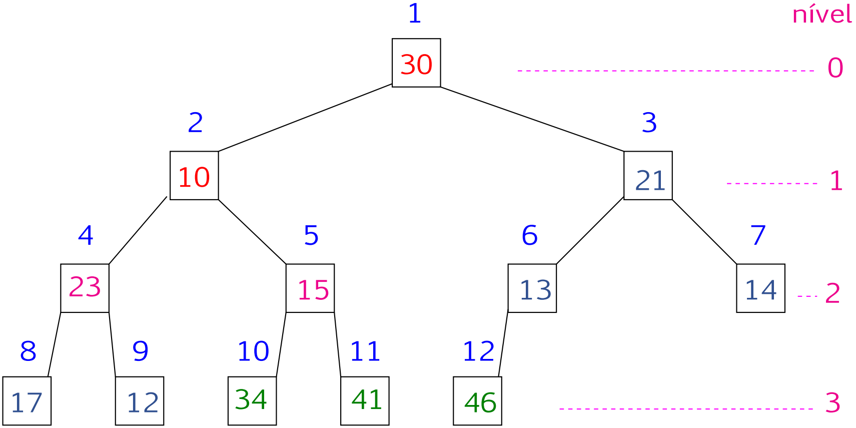
1	2	3	4	5	6	7	8	9	10	11	12
10	30	21	23	15	13	14	17	12	34	41	46

HeapSort



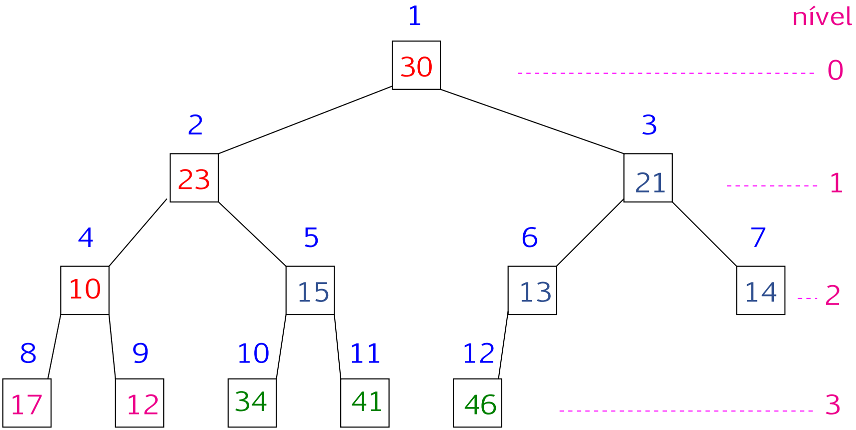
1	2	3	4	5	6	7	8	9	10	11	12
10	30	21	23	15	13	14	17	12	34	41	46

HeapSort



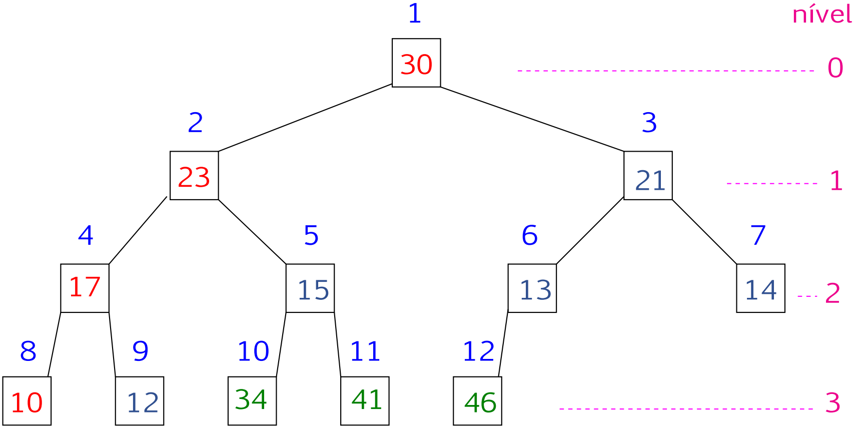
1	2	3	4	5	6	7	8	9	10	11	12
30	10	21	23	15	13	14	17	12	34	41	46

HeapSort



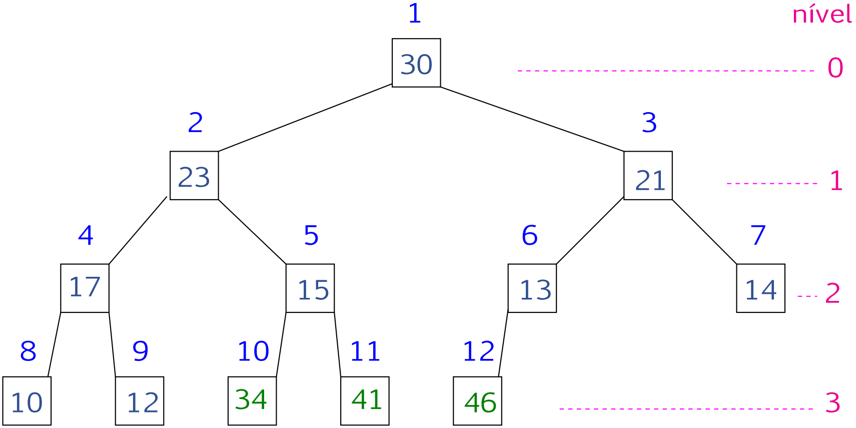
1	2	3	4	5	6	7	8	9	10	11	12
30	23	21	10	15	13	14	17	12	34	41	46

HeapSort



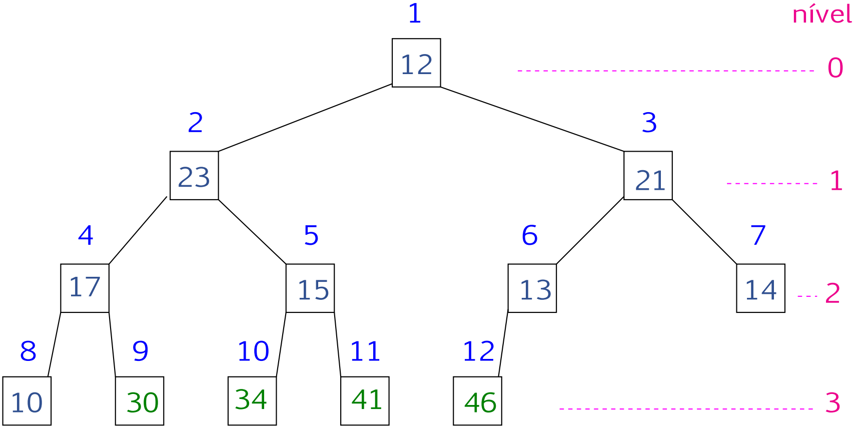
1	2	3	4	5	6	7	8	9	10	11	12
30	23	21	17	15	13	14	10	12	34	41	46

HeapSort



1	2	3	4	5	6	7	8	9	10	11	12
30	23	21	17	15	13	14	10	12	34	41	46

HeapSort



1	2	3	4	5	6	7	8	9	10	11	12
12	23	21	17	15	13	14	10	30	34	41	46

HeapSort

Algoritmo rearranja $A[1 \dots n]$ em ordem crescente.

```
HEAP-SORT(A, n)
1    BUILD-MAX-HEAP(A, n)
2    m ← n
3    para i ← n decrescendo até 2 faça
4        A[1] ↔ A[i]
5        m ← m − 1
6        MAX-HEAPIFY(A, m, 1)
```

Invariantes:

No início de cada iteração na linha 3 vale que:

1. $A[m+1 \dots n]$ é crescente e contém os $n - m$ maiores elementos de $A[1 \dots n]$;
2. $A[1 \dots m] \leq A[m+1]$;
3. $A[1 \dots m]$ é um max-heap.

HeapSort

Algoritmo rearranja $A[1 \ldots n]$ em ordem crescente.

HEAP-SORT(A, n)		Tempo
1	BUILD-MAX-HEAP(A, n)	$\Theta(n)$
2	$m \leftarrow n$	$\Theta(1)$
3	**para** $i \leftarrow n$ **decrescendo até** 2 **faça**	$\Theta(n)$
4	$A[1] \leftrightarrow A[i]$	$\Theta(n)$
5	$m \leftarrow m - 1$	$\Theta(n)$
6	MAX-HEAPIFY$(A, m, 1)$	$nO(\lg n)$

A complexidade de HEAP-SORT no pior caso é $O(n \lg n)$.

Como seria a complexidade de tempo no melhor caso?

Filas com prioridades

Uma fila com prioridades é um tipo abstrato de dados que consiste de uma coleção S de itens, cada um com um valor ou prioridade associada.

Algumas operações típicas em uma fila com prioridades são:

MAXIMUM(S): devolve o elemento de S com a maior prioridade;

EXTRACT-MAX(S): remove e devolve o elemento em S com a maior prioridade;

INCREASE-KEY(S, x, p): aumenta o valor da prioridade do elemento x para p; e

INSERT(S, x, p): insere o elemento x em S com prioridade p.

Implementação com max-heap

Heap-Max(A, n)
devolva $A[1]$

Complexidade de tempo: $\Theta(1)$.

Heap-Extract-Max(A, n)
$\triangleright\ n \geq 1$
$\max \leftarrow A[1]$
$A[1] \leftarrow A[n]$
$n \leftarrow n - 1$
Max-Heapify$(A, n, 1)$
devolva $\max$

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

Implementação com max-heap

Heap-Increase-Key$(A, i, chave)$

▷ Supõe que $chave \geq A[i]$
$A[i] \leftarrow chave$
enquanto $i > 1$ e $A[\lfloor i/2 \rfloor] < A[i]$ **faça**
    $A[i] \leftrightarrow A[\lfloor i/2 \rfloor]$
    $i \leftarrow \lfloor i/2 \rfloor$

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

Max-Heap-Insert$(A, n, chave)$

$n \leftarrow n + 1$
$A[n] \leftarrow -\infty$
Heap-Increase-Key$(A, n, chave)$

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