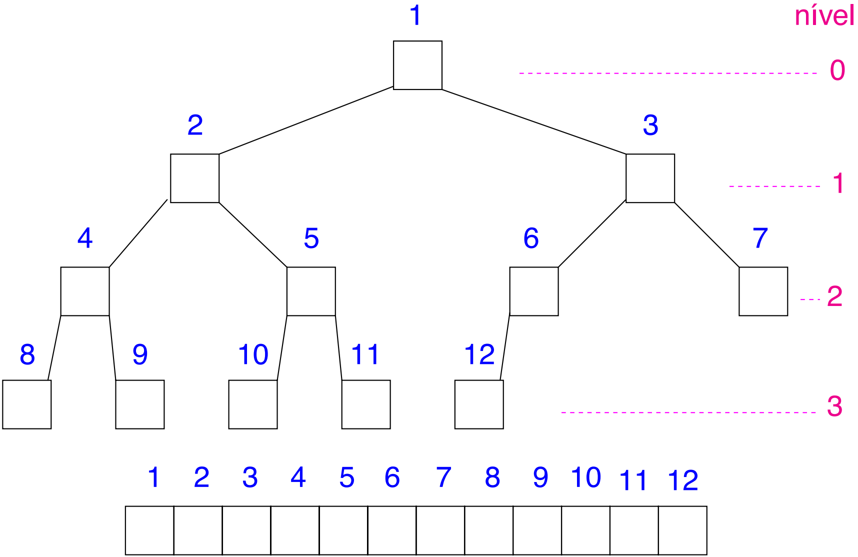
Heapsort

- 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 $A[1 \ldots n]$ representando um heap.

- Cada posição do vetor corresponde a um nó do heap.
- O pai de um nó i é $\lfloor i/2 \rfloor$.
- O nó 1 não tem pai.

- Um nó i tem
  $2i$ como filho esquerdo e
  $2i+1$ como filho direito.
- Naturalmente, o nó i
  tem filho esquerdo apenas se $2i \leq n$ e
  tem filho direito apenas se $2i+1 \leq n$.
- 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 e esses são

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

O nó i pertence ao nível ???.
O nó i pertence ao nível $\lfloor \lg i \rfloor$.

Prova: Se p é o nível do nó i, então

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

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

Portanto o número total de níveis é ???.
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 seqüência

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

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

Assim,

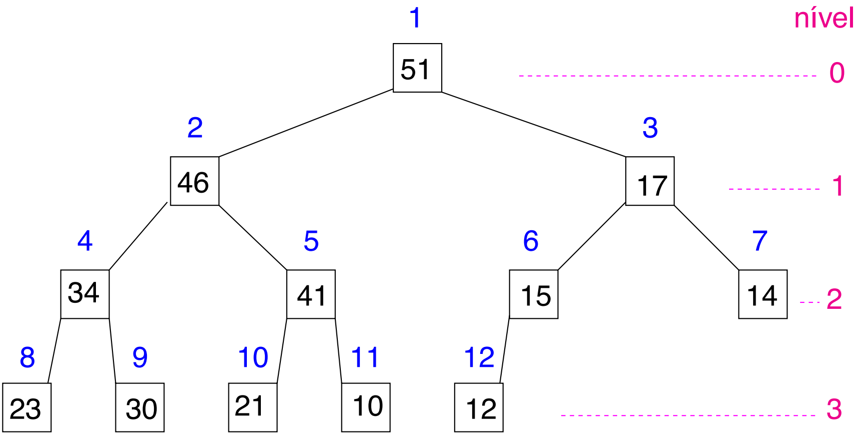
$$\begin{array}{rcccll} 2^h\,i & \leq & n & < & 2^{h+1}\,i & \Rightarrow \\ 2^h & \leq & n/i & < & 2^{h+1} & \Rightarrow \\ h & \leq & \lg(n/i) & < & h+1 & \end{array}$$

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

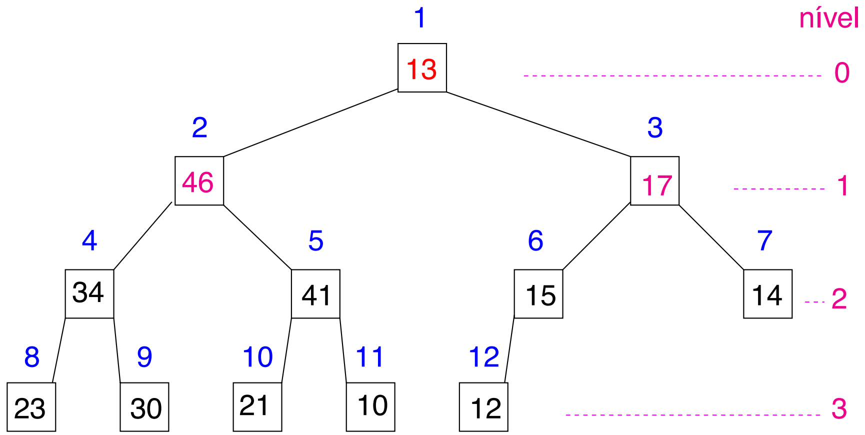


1	2	3	4	5	6	7	8	9	10	11	12
51	46	17	34	41	15	14	23	30	21	10	12

Min-heaps

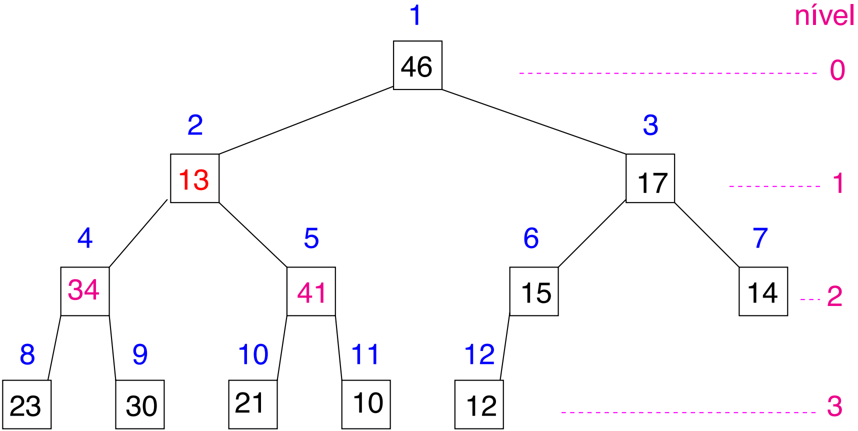
- Um nó i satisfaz a propriedade de (min-)heap se $A[\lfloor i/2 \rfloor] \leq A[i]$ (ou seja, pai $\leq$ filho).
- Uma árvore binária completa é um *min-heap* se todo nó distinto da raiz satisfaz a propriedade de min-heap.
- Vamos nos concentrar apenas em max-heaps.
- Os algoritmos que veremos podem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

Manipulação de max-heap



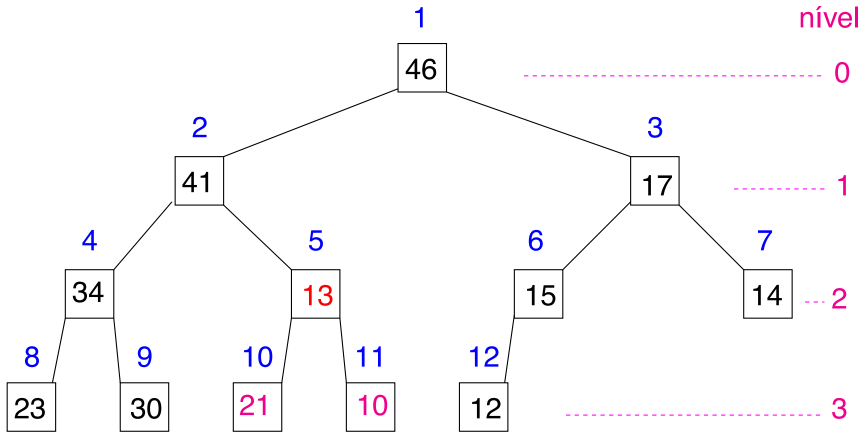
1	2	3	4	5	6	7	8	9	10	11	12
13	46	17	34	41	15	14	23	30	21	10	12

Manipulação de max-heap



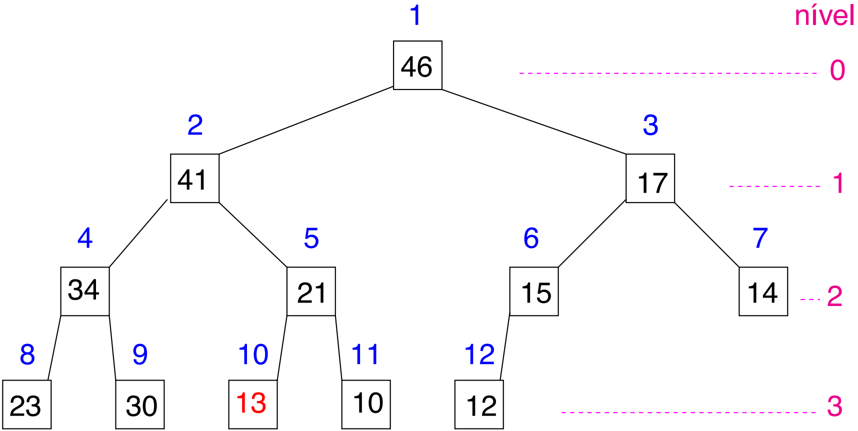
1	2	3	4	5	6	7	8	9	10	11	12
46	13	17	34	41	15	14	23	30	21	10	12

Manipulação de max-heap



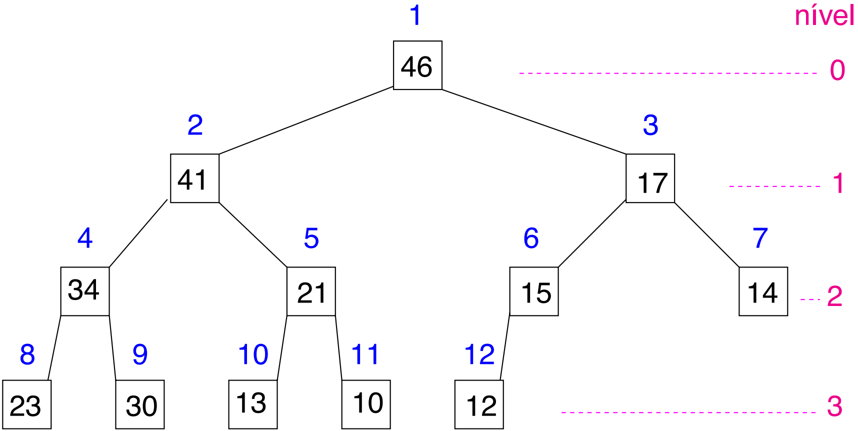
1	2	3	4	5	6	7	8	9	10	11	12
46	41	17	34	13	15	14	23	30	21	10	12

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)

$e \leftarrow 2i$
$d \leftarrow 2i + 1$
se $e \leq n$ **e** $A[e] > A[i]$
  então $\text{maior} \leftarrow e$
  senão $\text{maior} \leftarrow i$
se $d \leq n$ e $A[d] > A[\text{maior}]$
  então $\text{maior} \leftarrow d$
se $\text{maior} \neq i$
  então $A[i] \leftrightarrow A[\text{maior}]$
    MAX-HEAPIFY(A, n, maior)

Corretude de MAXHEAPIFY

A corretude 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).

Corretude de MAXHEAPIFY

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 MAXHEAPIFY

MAX-HEAPIFY(A, n, i)	Tempo
$e \leftarrow 2i$	?
$d \leftarrow 2i + 1$	?
se $e \leq n$ **e** $A[e] > A[i]$	?
então $\text{maior} \leftarrow e$	?
senão $\text{maior} \leftarrow i$	?
se $d \leq n$ e $A[d] > A[\text{maior}]$	?
então $\text{maior} \leftarrow d$	?
se $\text{maior} \neq i$	?
então $A[i] \leftrightarrow A[\text{maior}]$	?
MAX-HEAPIFY(A, n, maior)	?

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

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

Complexidade de MaxHeapify

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 $A[d] > A[\text{maior}]$	$\Theta(1)$
então maior $\leftarrow d$	$O(1)$
se maior $\neq i$	$\Theta(1)$
então $A[i] \leftrightarrow A[\text{maior}]$	$O(1)$
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 MaxHeapify

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

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

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

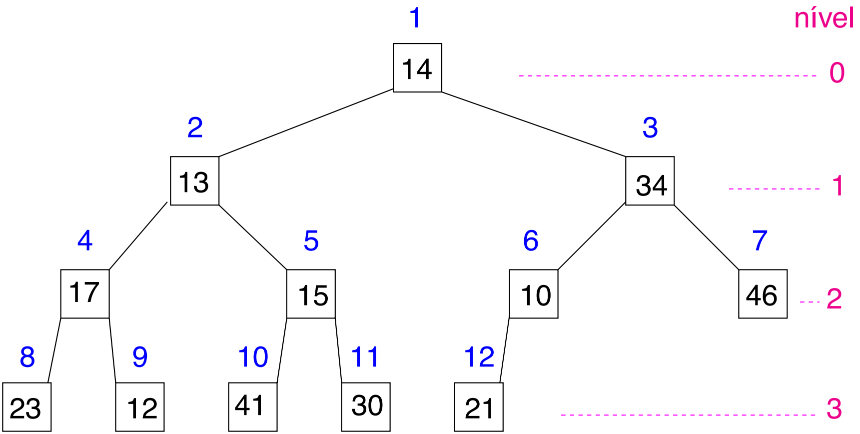
Solução assintótica: $T(n)$ é ???.

Solução assintótica: $T(n)$ é $O(h)$.

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

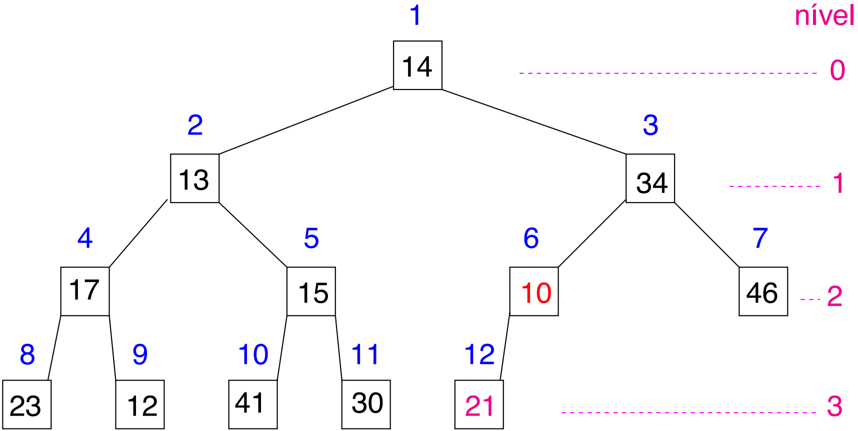
O consumo de tempo do algoritmo Max-Heapify é $O(\lg n)$ (ou melhor ainda, $O(\lg \frac{n}{i})$).

Construção de um max-heap



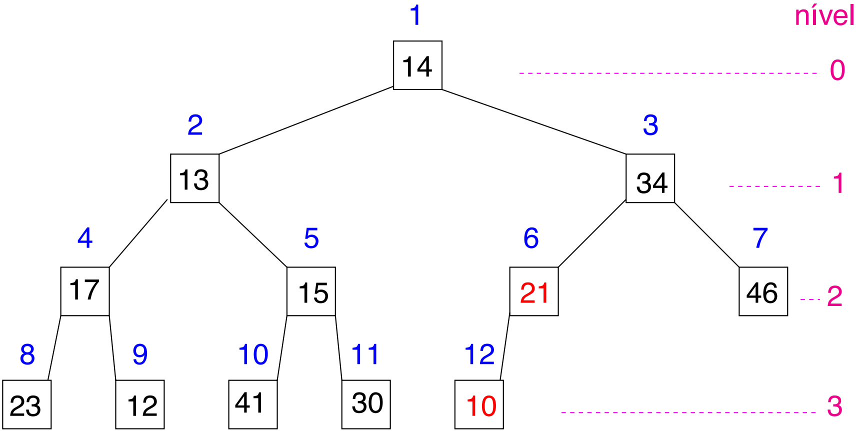
1	2	3	4	5	6	7	8	9	10	11	12
14	13	34	17	15	10	46	23	12	41	30	21

Construção de um max-heap



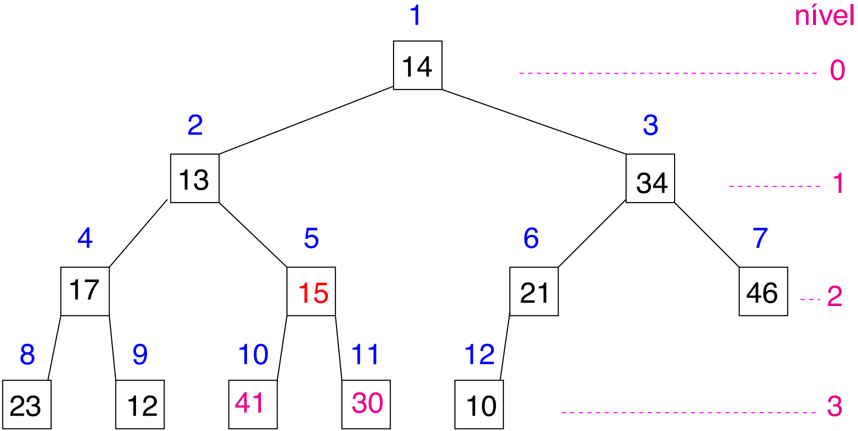
1	2	3	4	5	6	7	8	9	10	11	12
14	13	34	17	15	10	46	23	12	41	30	21

Construção de um max-heap



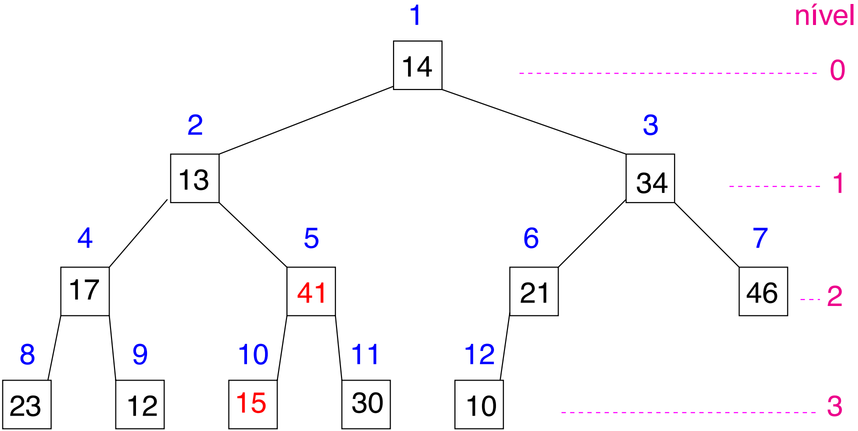
1	2	3	4	5	6	7	8	9	10	11	12
14	13	34	17	15	21	46	23	12	41	30	10

Construção de um max-heap



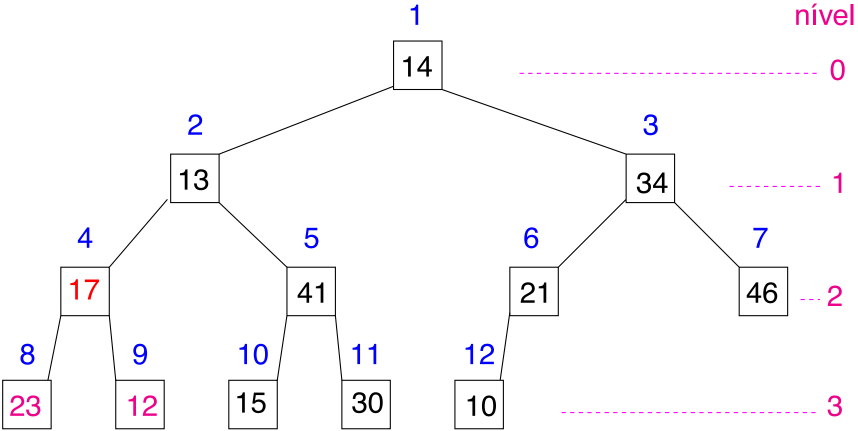
1	2	3	4	5	6	7	8	9	10	11	12
14	13	34	17	15	21	46	23	12	41	30	10

Construção de um max-heap



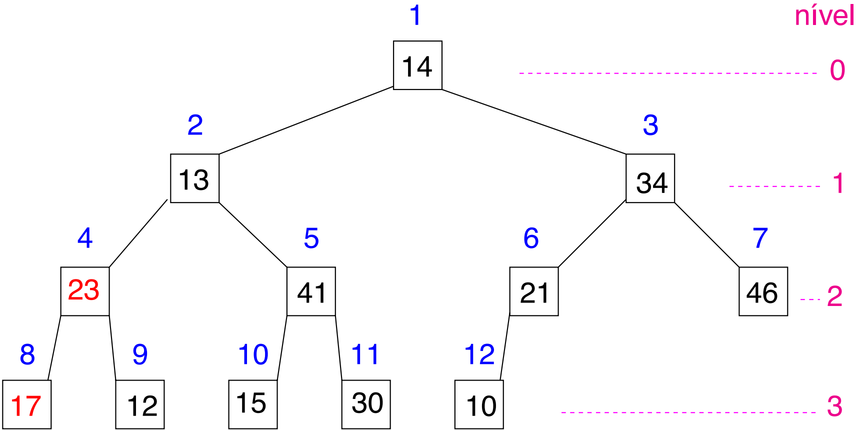
1	2	3	4	5	6	7	8	9	10	11	12
14	13	34	17	41	21	46	23	12	15	30	10

Construção de um max-heap



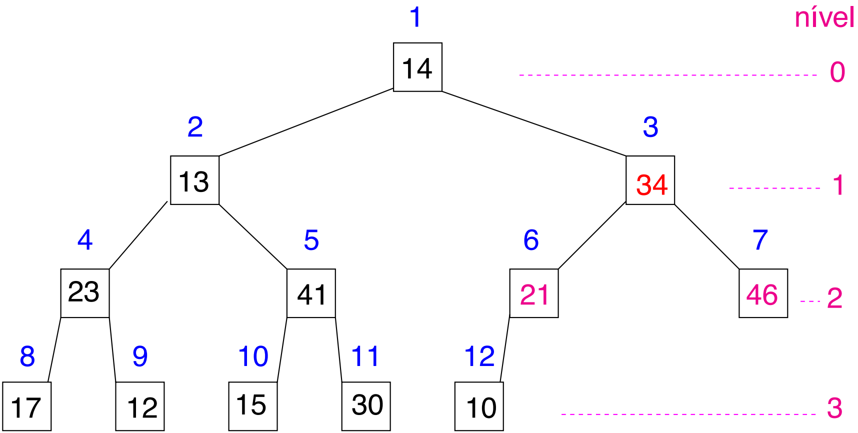
1	2	3	4	5	6	7	8	9	10	11	12
14	13	34	17	41	21	46	23	12	15	30	10

Construção de um max-heap



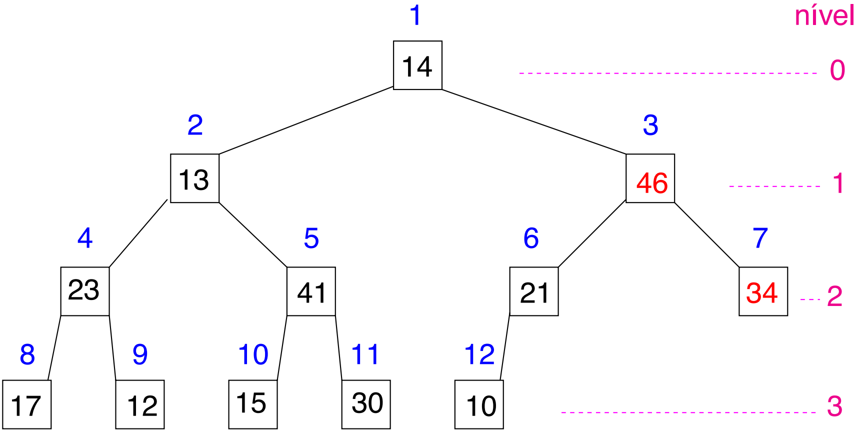
1	2	3	4	5	6	7	8	9	10	11	12
14	13	34	23	41	21	46	17	12	15	30	10

Construção de um max-heap



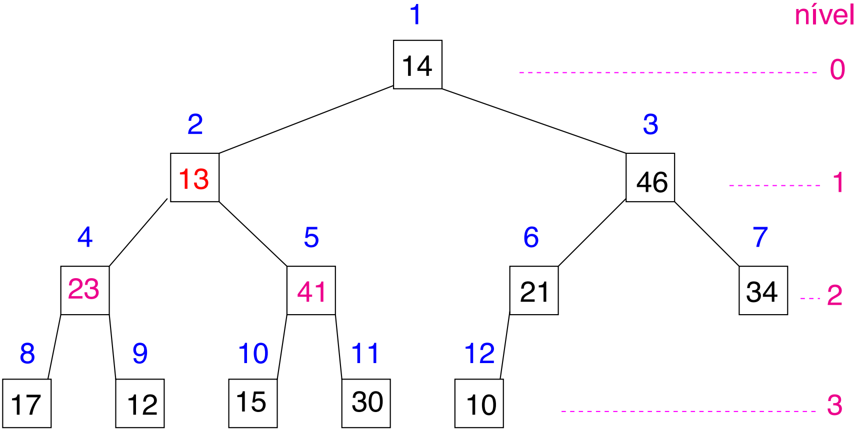
1	2	3	4	5	6	7	8	9	10	11	12
14	13	34	23	41	21	46	17	12	15	30	10

Construção de um max-heap



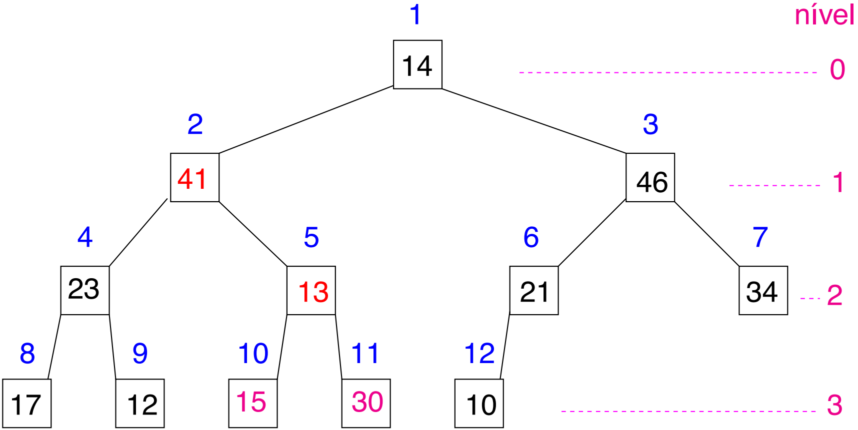
1	2	3	4	5	6	7	8	9	10	11	12
14	13	46	23	41	21	34	17	12	15	30	10

Construção de um max-heap



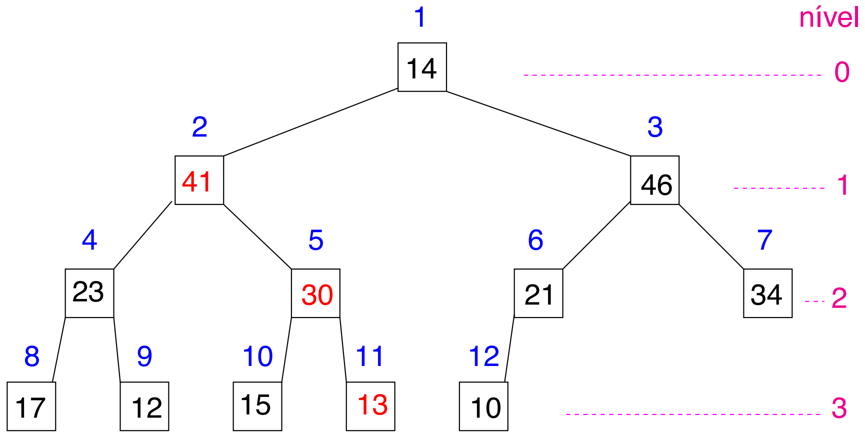
1	2	3	4	5	6	7	8	9	10	11	12
14	13	46	23	41	21	34	17	12	15	30	10

Construção de um max-heap



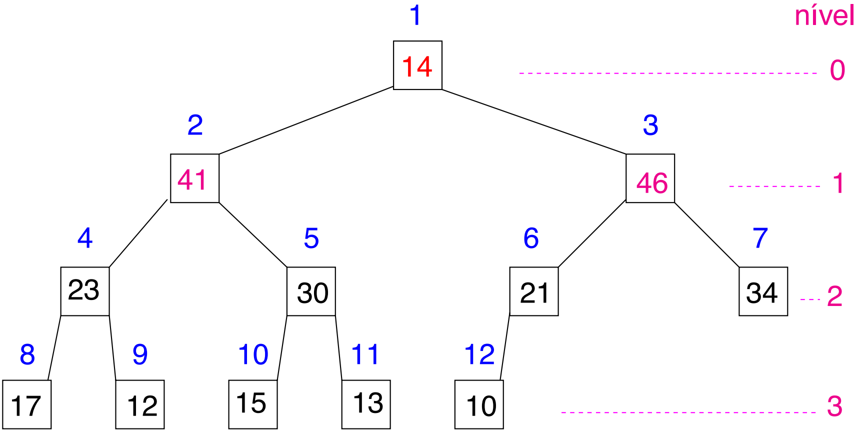
1	2	3	4	5	6	7	8	9	10	11	12
14	41	46	23	13	21	34	17	12	15	30	10

Construção de um max-heap



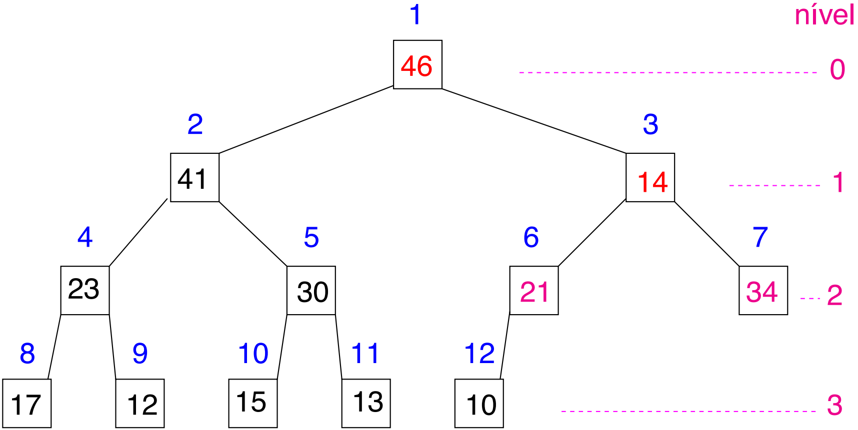
1	2	3	4	5	6	7	8	9	10	11	12
14	41	46	23	30	21	34	17	12	15	13	10

Construção de um max-heap



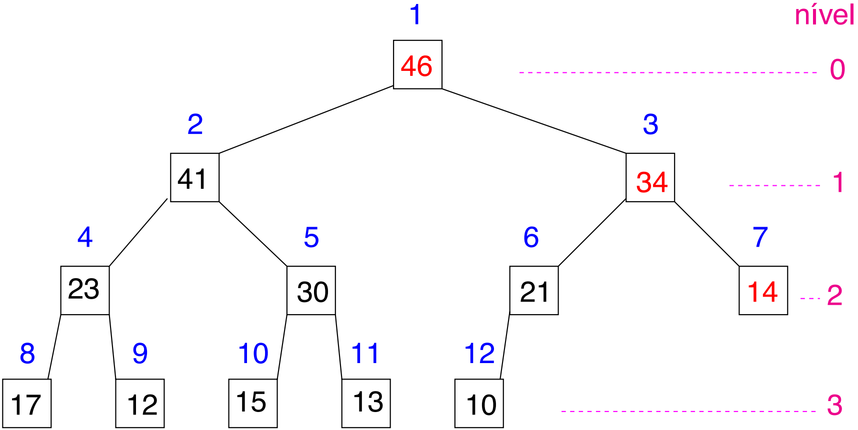
1	2	3	4	5	6	7	8	9	10	11	12
14	41	46	23	30	21	34	17	12	15	13	10

Construção de um max-heap



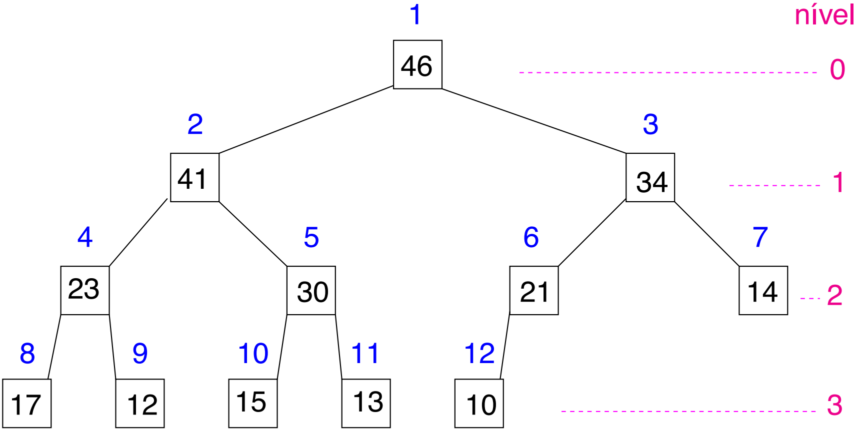
1	2	3	4	5	6	7	8	9	10	11	12
46	41	14	23	30	21	34	17	12	15	13	10

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 $A[1 \ldots n]$ e rearranja A para que seja max-heap.

BUILDMAXHEAP(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, $i + 1, \ldots, n$ 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 $\lfloor \lg n \rfloor + 1$.

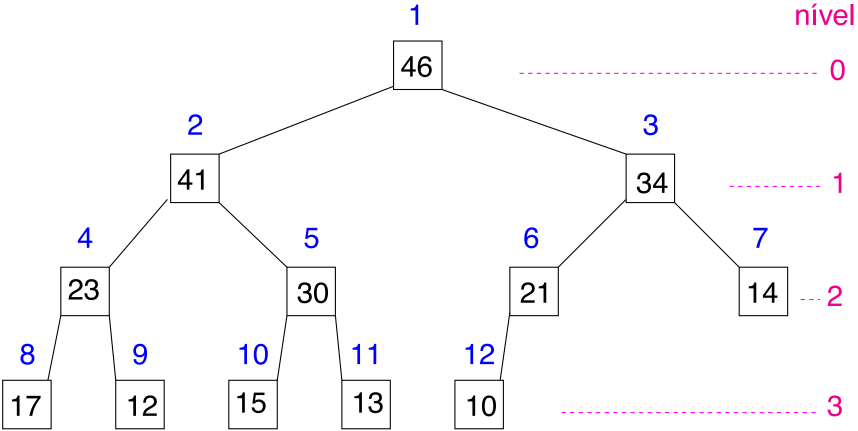
  A complexidade de BUILDMAXHEAP é $T(n) = O(S(\lg n))$.

Construção de um max-heap

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

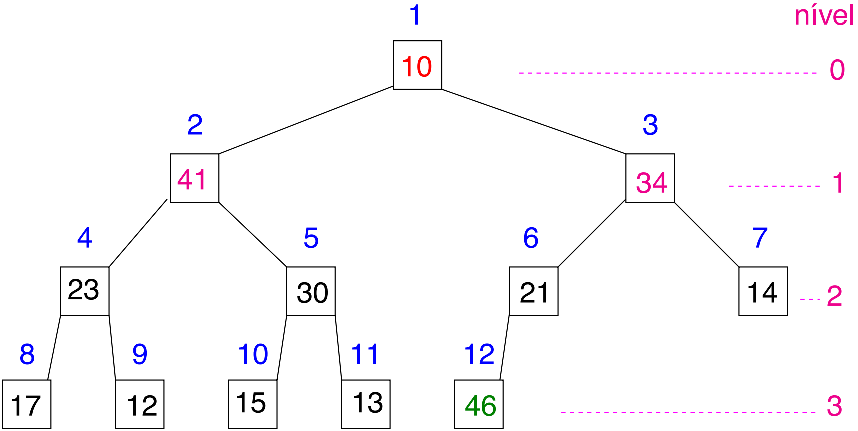
  Mais precisamente, $T(n) = \Theta(n)$. (Por quê?)
- Veja no CLRS uma prova diferente deste fato.

HeapSort

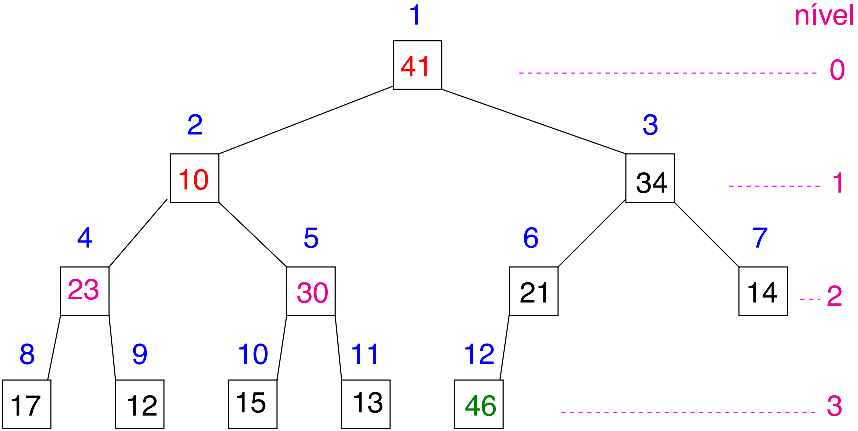


1	2	3	4	5	6	7	8	9	10	11	12
46	41	34	23	30	21	14	17	12	15	13	10

HeapSort

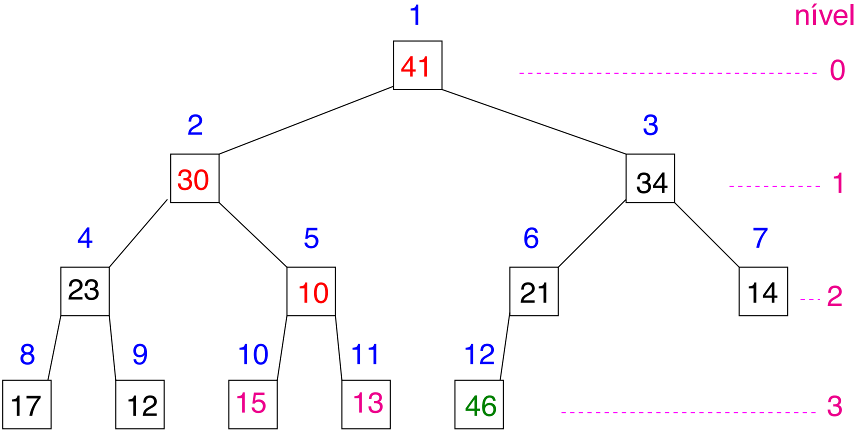


1	2	3	4	5	6	7	8	9	10	11	12
10	41	34	23	30	21	14	17	12	15	13	46



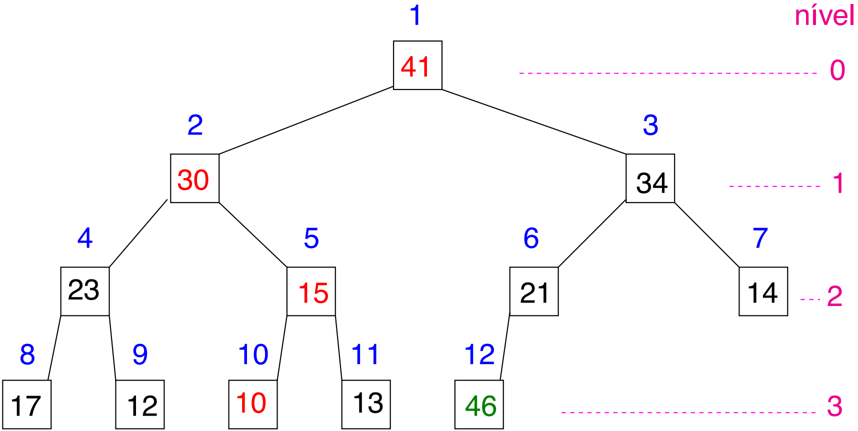
1	2	3	4	5	6	7	8	9	10	11	12
41	10	34	23	30	21	14	17	12	15	13	46

HeapSort

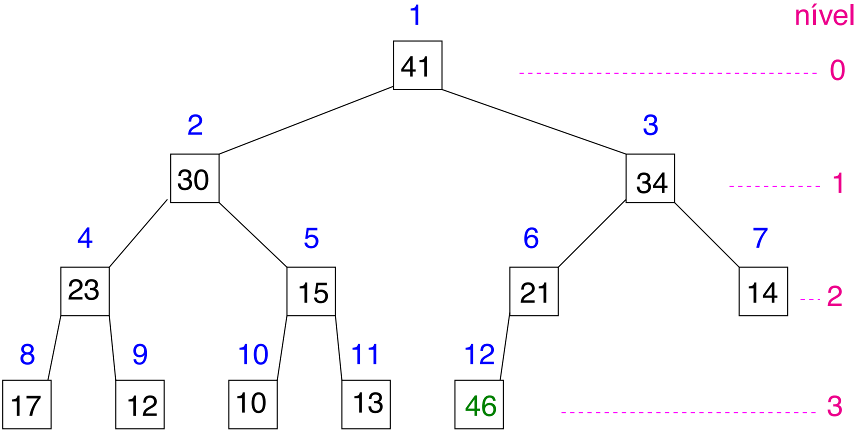


1	2	3	4	5	6	7	8	9	10	11	12
41	30	34	23	10	21	14	17	12	15	13	46

HeapSort

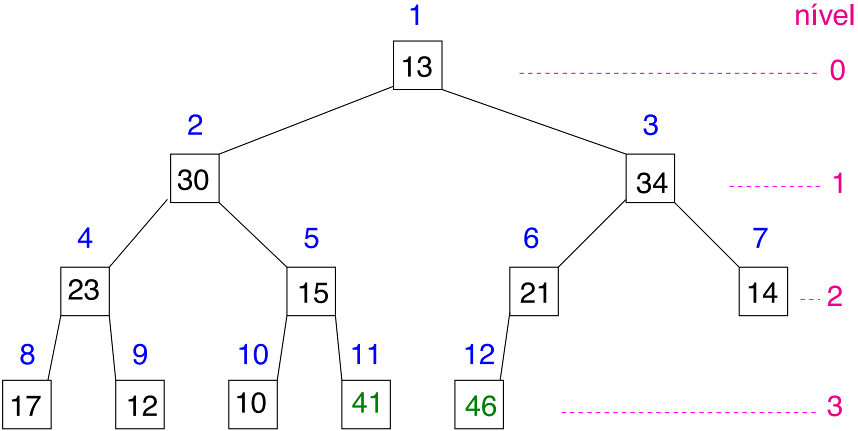


1	2	3	4	5	6	7	8	9	10	11	12
41	30	34	23	15	21	14	17	12	10	13	46



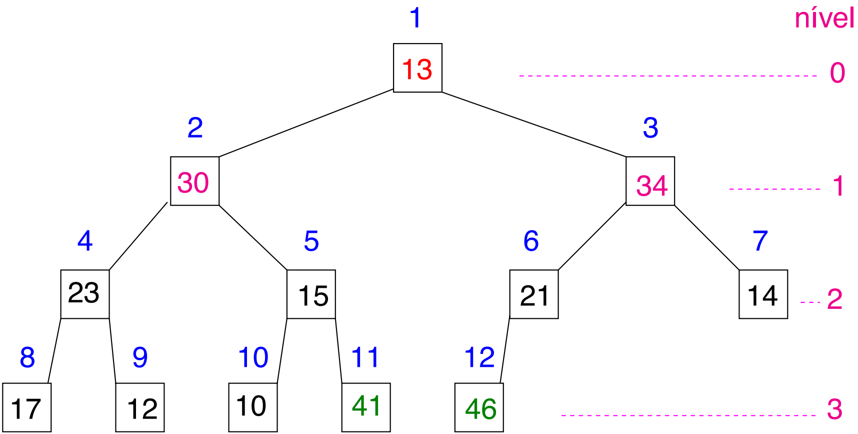
1	2	3	4	5	6	7	8	9	10	11	12
41	30	34	23	15	21	14	17	12	10	13	46

HeapSort

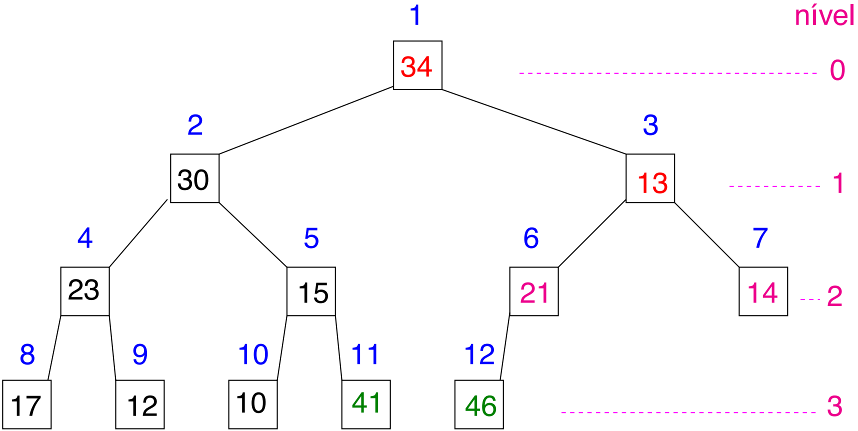


1	2	3	4	5	6	7	8	9	10	11	12
13	30	34	23	15	21	14	17	12	10	41	46

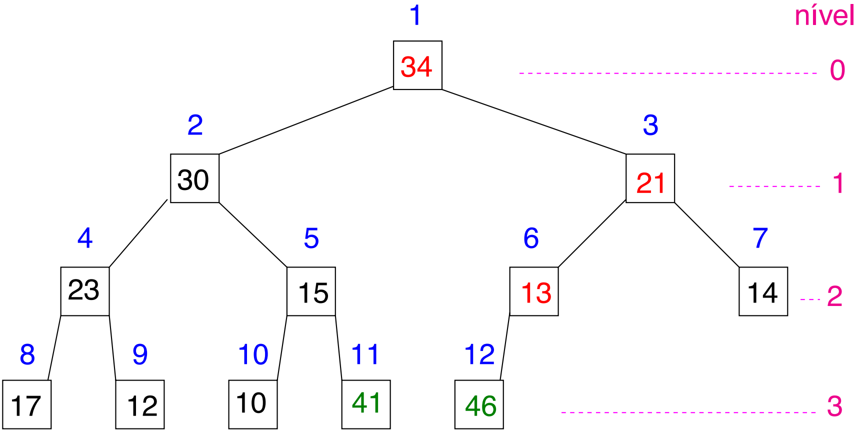
HeapSort



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

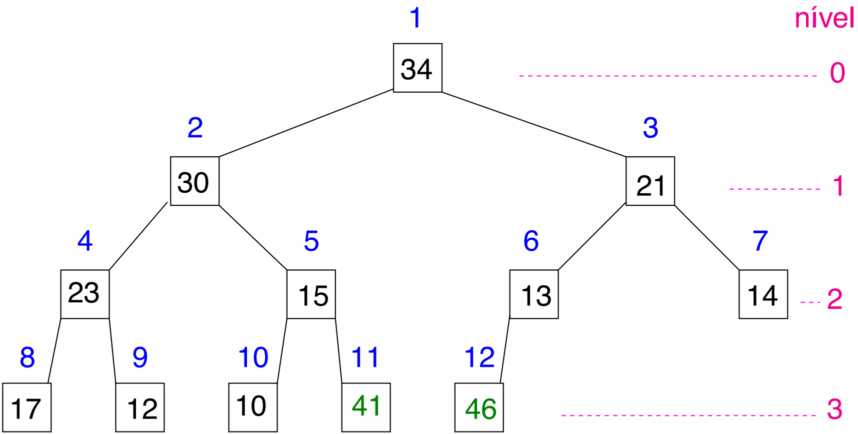


1	2	3	4	5	6	7	8	9	10	11	12
34	30	13	23	15	21	14	17	12	10	41	46



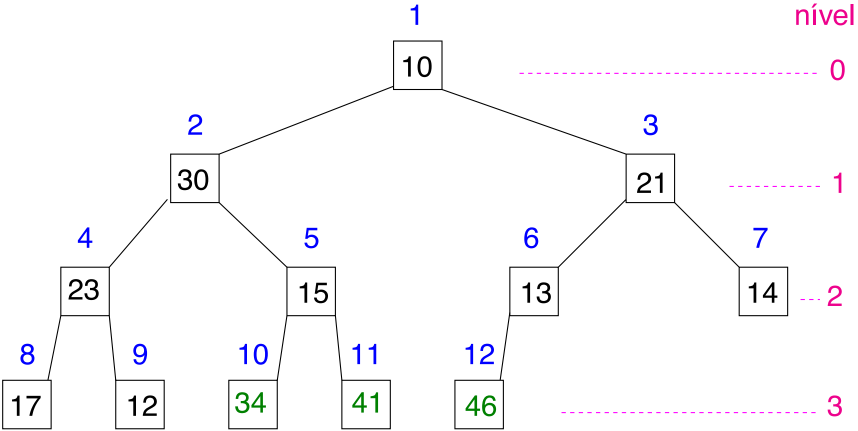
1	2	3	4	5	6	7	8	9	10	11	12
34	30	21	23	15	13	14	17	12	10	41	46

HeapSort



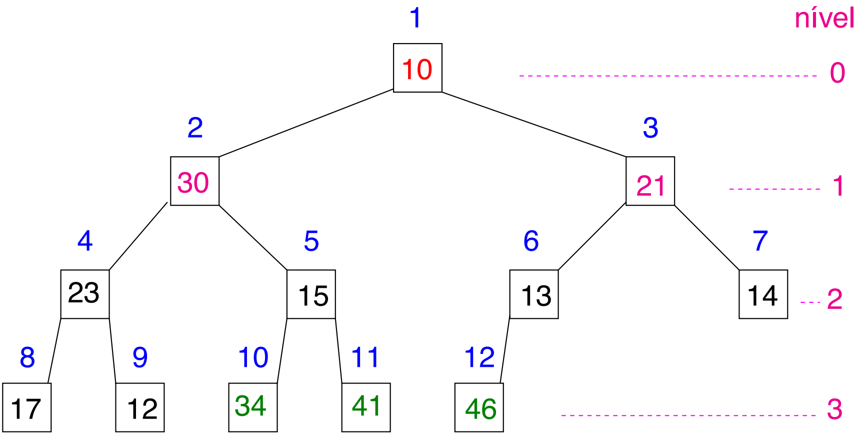
1	2	3	4	5	6	7	8	9	10	11	12
34	30	21	23	15	13	14	17	12	10	41	46

HeapSort



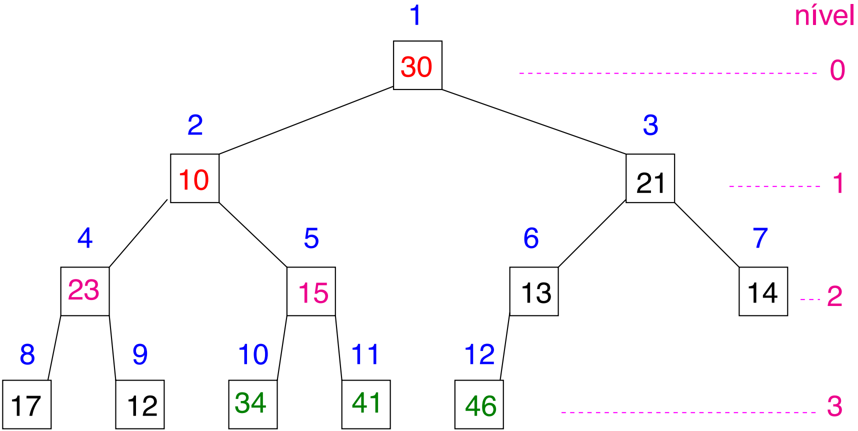
1	2	3	4	5	6	7	8	9	10	11	12
10	30	21	23	15	13	14	17	12	34	41	46

HeapSort



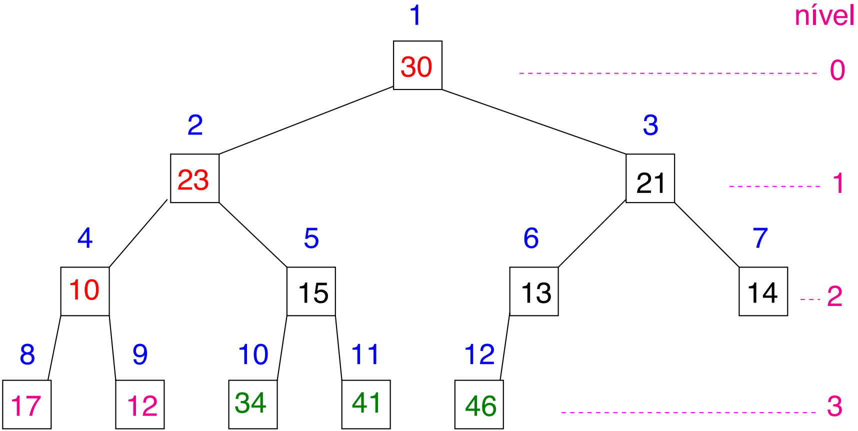
1	2	3	4	5	6	7	8	9	10	11	12
10	30	21	23	15	13	14	17	12	34	41	46

HeapSort



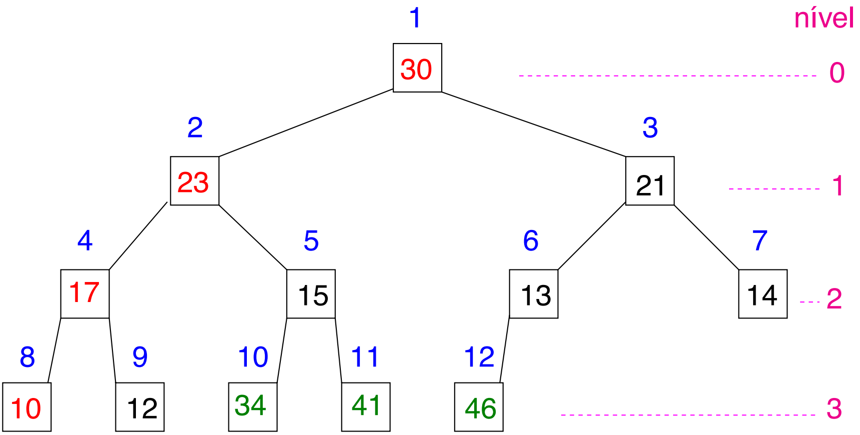
1	2	3	4	5	6	7	8	9	10	11	12
30	10	21	23	15	13	14	17	12	34	41	46

HeapSort



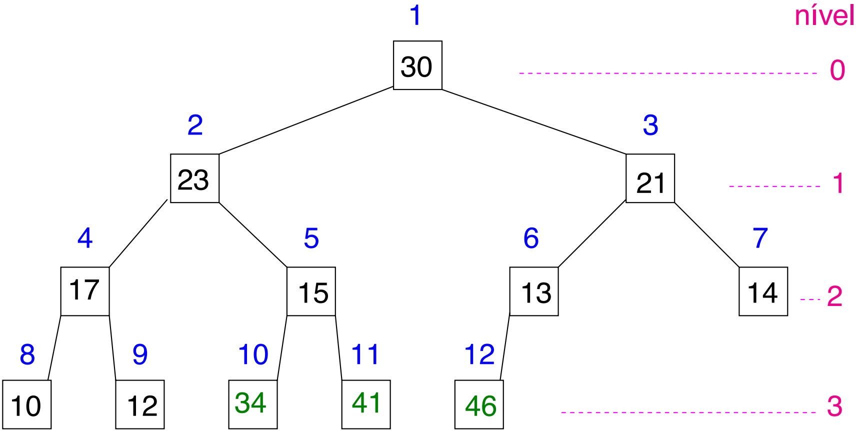
1	2	3	4	5	6	7	8	9	10	11	12
30	23	21	10	15	13	14	17	12	34	41	46

HeapSort



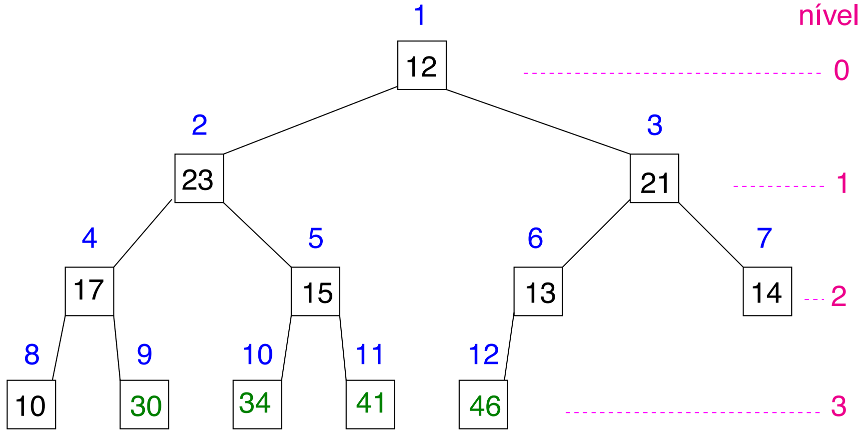
1	2	3	4	5	6	7	8	9	10	11	12
30	23	21	17	15	13	14	10	12	34	41	46

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.

```
HeapSort(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 $A[1 \dots n]$ em ordem crescente.

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

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

HeapSort

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

HEAPSORT(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$(A, m, 1)$	$nO(\lg n)$

$T(n) = ??$ $T(n) = nO(\lg n) + \Theta(4n + 1) = O(n \lg n)$

A complexidade de HEAPSORT 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:

$\textsc{Maximum}(S)$: devolve o elemento de S com a maior prioridade;

$\textsc{Extract-Max}(S)$: remove e devolve o elemento em S com a maior prioridade;

$\textsc{Increase-Key}(S, x, p)$: aumenta o valor da prioridade do elemento x para p; e

$\textsc{Insert}(S, x, p)$: 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: $O(\lg n)$.