Aula 06 - Heapsort

Algoritmo de Ordenação

Prof. Marco Aurélio Stefanes
marco em dct.ufms.br

www.dct.ufms.br/ ~ marco

Ordenação

$A[1 \ldots n]$ é crescente se $A[1] \leq \cdots \leq A[n]$.

Problema: Rearranjar um vetor $A[1 \ldots n]$ de modo que ele fique crescente.

Entra:

1										n
33	55	33	44	33	22	11	99	22	55	77

Sai:

1										n
11	22	22	33	33	33	44	55	55	77	99

Ordenação

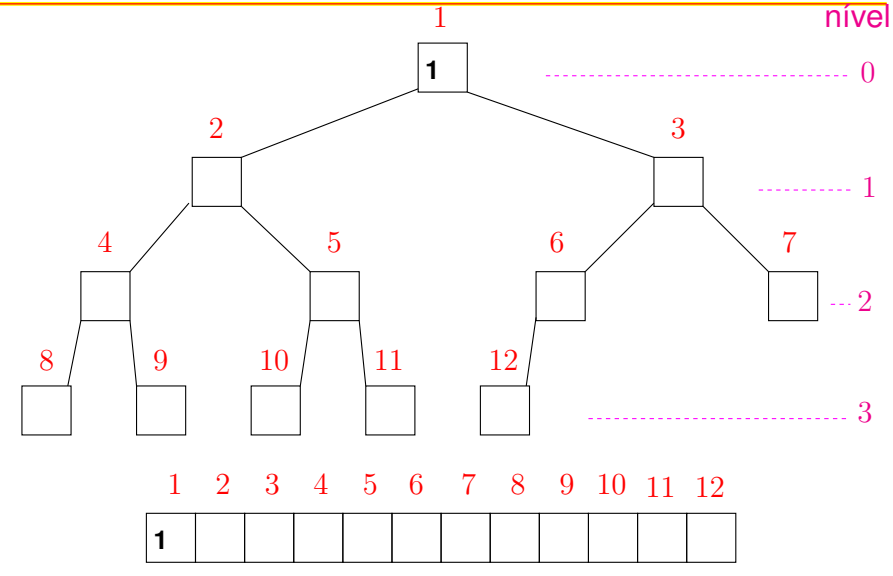
$A[1 \ldots n]$ é crescente se $A[1] \leq \cdots \leq A[n]$.

Problema: Rearranjar um vetor $A[1 \ldots n]$ de modo que ele fique crescente.

Entra:

1										n
33	55	33	44	33	22	11	99	22	55	77

Representar árvores em vetores



Pais e filhos

$A[1 \ldots m]$ é um vetor representando uma árvore.
Diremos que para qualquer índice ou nó i,

- $\lfloor i/2 \rfloor$ é o *pai* de i;
- $2i$ é o *filho esquerdo* de i;
- $2i + 1$ é o *filho direito*.

O nó 1 não tem pai e é chamado de *raiz*.

Um nó i só tem filho esquerdo se $2i \leq m$.

Um nó i só tem filho direito se $2i + 1 \leq m$.

Um nó i é um *folha* se não tem filhos, ou seja $2i > m$.

Todo nó i é raiz da subárvore formada por

$$A[i, 2i, 2i + 1, 4i, 4i + 1, 4i + 2, 4i + 3, 8i, \ldots, 8i + 7, \ldots]$$

Níveis

Cada nível p, exceto talvez o último, tem exatamente 2^p nós e esses são

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

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 ???.

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 $\lfloor \lg i \rfloor$.

Prova Se p é o nível do nó i, então

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

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

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 $\lfloor \lg i \rfloor$.

Prova Se p é o nível do nó i, então

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

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

Portanto o número total de níveis é ???.

Altura

A *altura* de um nó i é o maior comprimento de um caminho de i a uma folha.

Em outras palavras, a altura de um nó i é o maior comprimento de uma seqüência da forma

$$\{\mathit{filho}(i), \mathit{filho}(\mathit{filho}(i)), \mathit{filho}(\mathit{filho}(\mathit{filho}(i))), \ldots\}$$

onde $\mathit{filho}(i)$ vale $2i$ ou $2i + 1$.

Os nós que têm *altura zero* são as folhas.

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 $\lfloor \lg i \rfloor$.

Prova Se p é o nível do nó i, então

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

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

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

Altura

A *altura* de um nó i é o maior comprimento de um caminho de i a uma folha.

Em outras palavras, a altura de um nó i é o maior comprimento de uma seqüência da forma

$$\{\mathit{filho}(i), \mathit{filho}(\mathit{filho}(i)), \mathit{filho}(\mathit{filho}(\mathit{filho}(i))), \ldots\}$$

onde $\mathit{filho}(i)$ vale $2i$ ou $2i + 1$.

Os nós que têm *altura zero* são as folhas.

A altura de um nó i é ????.

Exercício 12.A

A altura de um nó i é o comprimento da seqüência

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

onde $2^h\,i \le m < 2^{(h+1)}\,i$. Assim,

$$\begin{array}{rcccl} 2^h\,i & \le & m & < & 2^{h+1}i \quad \Rightarrow \\ 2^h & \le & m/i & < & \lg 2^{h+1} \quad \Rightarrow \\ h & \le & \lg(m/i) & < & h+1 \end{array}$$

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

Exercício 12.B

Mostre que $A[1 \ldots m]$ tem no máximo $\lceil m/2^{h+1} \rceil$ nós com altura h.

Exemplo: N_h = número de nós à altura h

m	$\lfloor m/2^{0+1} \rfloor$	N_0	$\lceil m/2^{0+1} \rceil$	$\lfloor m/2^{1+1} \rfloor$	N_1	$\lceil m/2^{1+1} \rceil$
16	8	8	8	4	4	4
17	8	9	9	4	4	5
18	9	9	9	4	5	5
19	9	10	10	4	5	5
20	10	10	10	5	5	5
21	10	11	11	5	5	6
22	11	11	11	5	6	6
23	11	12	12	5	6	6
24	12	12	12	6	6	6

Resumão

Considere uma árvore representada em um vetor $A[1 \ldots m]$.

filho esquerdo de i:	$2\,i$
filho direito de i:	$2\,i+1$
pai de i:	$\lfloor i/2 \rfloor$
nível da raiz:	0
nível de i:	$\lfloor \lg i \rfloor$
altura da raiz:	$\lfloor \lg m \rfloor$
altura da árvore:	$\lfloor \lg m \rfloor$
altura de i:	$\lfloor \lg(m/i) \rfloor$
altura de uma folha:	0
total de nós de altura h	$\le \lceil m/2^{h+1} \rceil$ (Exercício 12.B)

Heap

Um vetor $A[1 \ldots m]$ é um *heap* se

$$A[\lfloor i/2 \rfloor] \ge A[i]$$

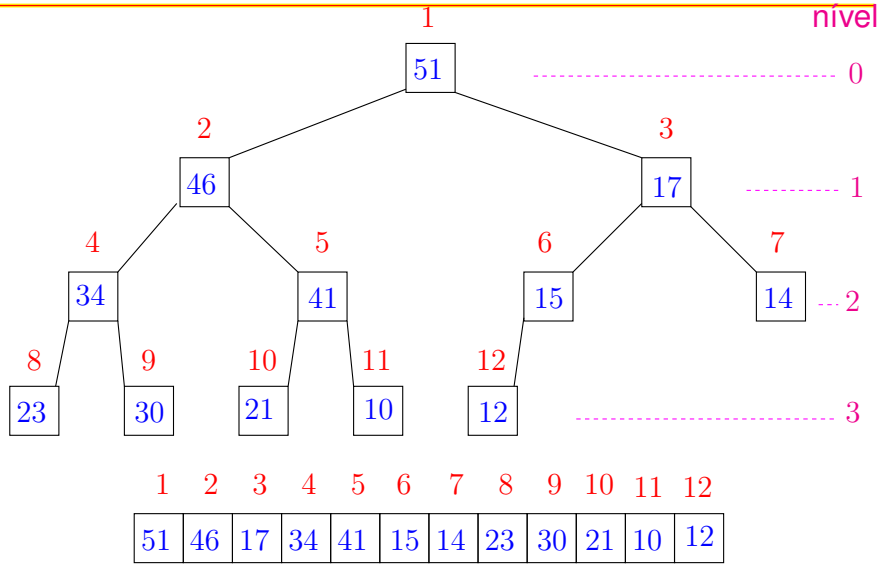
para todo $i = 2, 3, \ldots, m$.

De uma forma mais geral, $A[j \ldots m]$ é um *heap* se

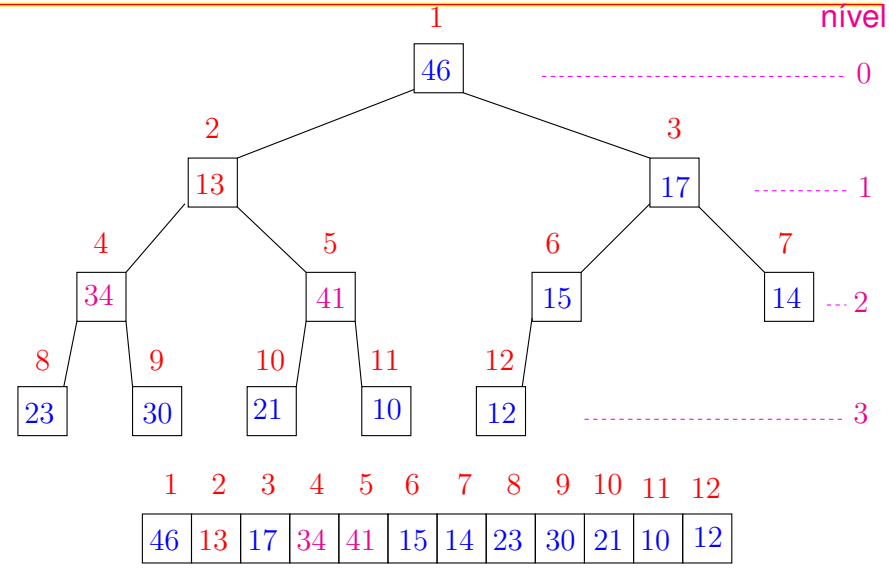
$$A[\lfloor i/2 \rfloor] \ge A[i]$$

para todo $i = 2j, 2j+1, 4j, \ldots, 4j+3, 8j, \ldots, 8j+7, \ldots$.
Neste caso também diremos que a subárvore com raiz j é um *heap*.

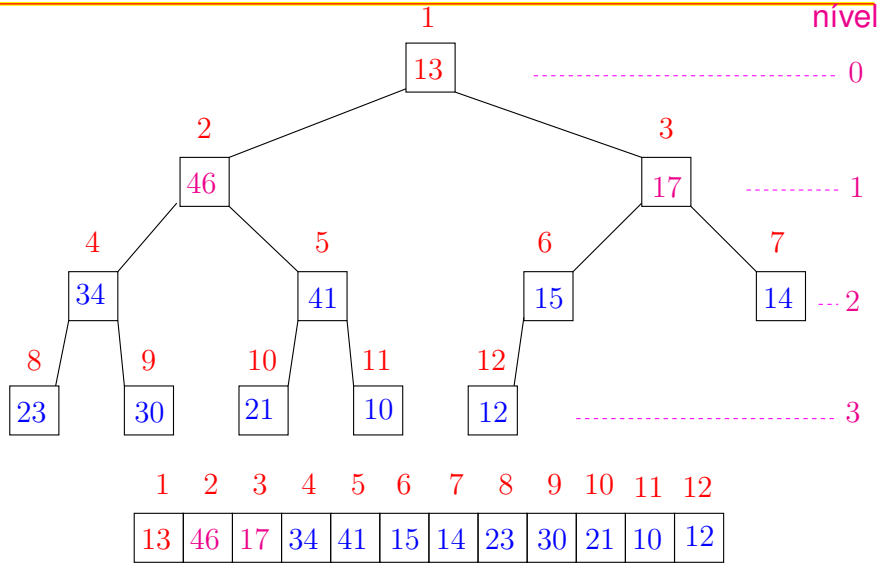
Heap



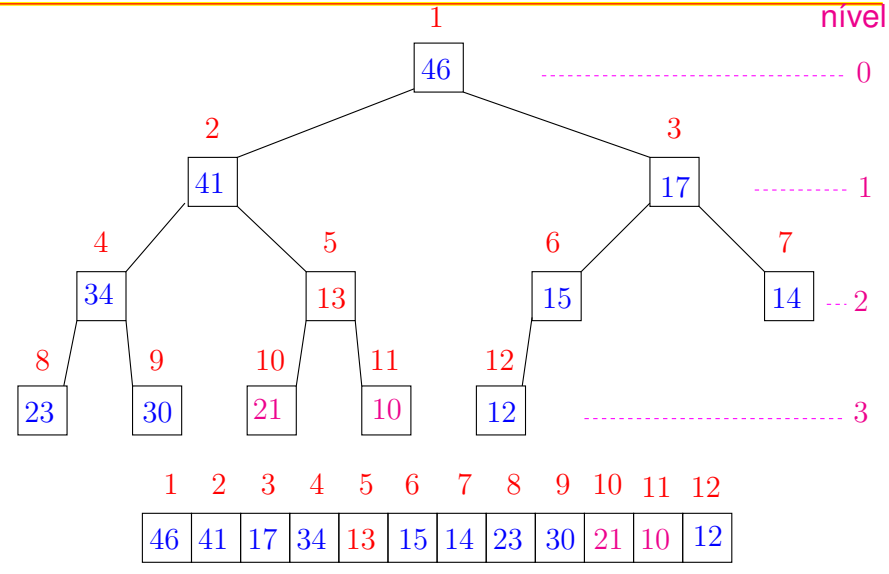
Manipulação de heap



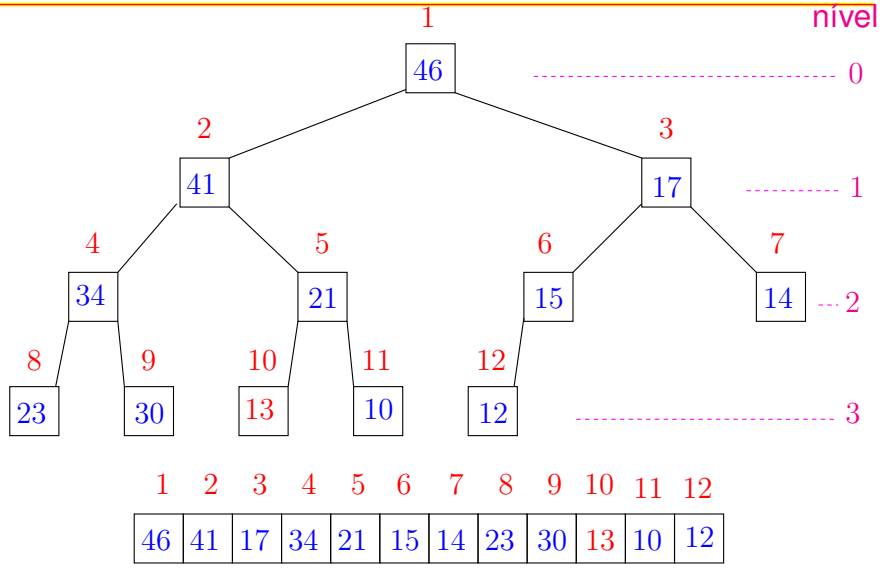
Manipulação de heap



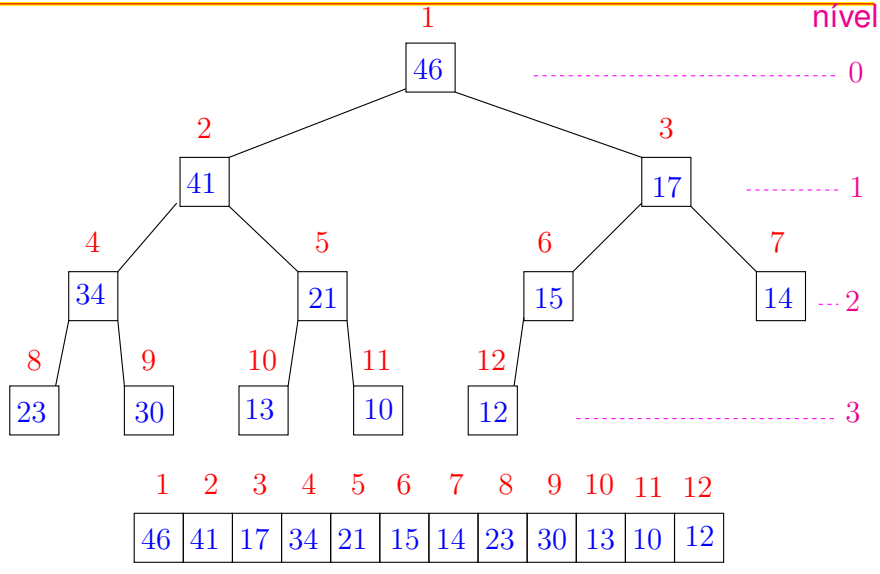
Manipulação de heap



Manipulação de heap



Manipulação de heap



Manipulação de heap

Recebe $A[1 \ldots m]$ e $i \geq 1$ tais que subárvores com raiz $2i$ e $2i+1$ são heaps e rearranja A de modo que subárvore com raiz i seja heap.

Heapify (A, m, i)

```
1:  e = 2i
2:  d = 2i + 1
3:  if e ≤ m e A[e] > A[i] then
4:      maior = e
5:  else
6:      maior = i
7:  if d ≤ m e A[d] > A[maior] then
8:      maior = d
9:  if maior ≠ i then
10:     A[i] ↔ A[maior]
11:     Heapify(A, m, maior)
```


Correção do Heapify

- Caso 1 - $A[i] \geq \max(A[2i], A[2i+1])$.
  A subárvore i já possui propriedade heap
- Caso 2 - $A[i] \leq \max(A[2i], A[2i+1])$
  Suponha SPG $A[2i] \geq A[2i+1]$.
  Trocando os valores de $A[i]$ com $A[2i]$, o nó i satisfaz a propriedade $A[i] \leq A[2i]$ e $A[i] \leq A[2i+1]$. A subárvore com raiz $2i+1$ não foi alterada e, por hipótese, satisfaz a propriedade heap. Porém o procedimento precisa ser chamado para o nó $2i$

Consumo de tempo

h := altura de $i = \lfloor \lg \frac{m}{i} \rfloor$
$T(h)$:= consumo de tempo no pior caso

linha		todas as execuções da linha
1-3	$=$	$3\,\Theta(1)$
4-5	$=$	$2\,O(1)$
6	$=$	$\Theta(1)$
7	$=$	$O(1)$
8	$=$	$\Theta(1)$
9	$=$	$O(1)$
10	$\leq$	$T(h-1)$
total	$\leq$	$T(h-1) + \Theta(5) + O(2)$

Consumo de tempo

h := altura de $i = \lfloor \lg \frac{m}{i} \rfloor$
$T(h)$:= consumo de tempo no pior caso

Recorrência associada:

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

pois altura de $maior$ é $h-1$.

Solução assintótica: $T(n)$ é ???.

Consumo de tempo

h := altura de $i = \lfloor \lg \frac{m}{i} \rfloor$
$T(h)$:= consumo de tempo no pior caso

Recorrência associada:

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

pois altura de $maior$ é $h-1$.

Consumo de tempo

h := altura de $i = \lfloor \lg \frac{m}{i} \rfloor$
$T(h)$:= consumo de tempo no pior caso

Recorrência associada:

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

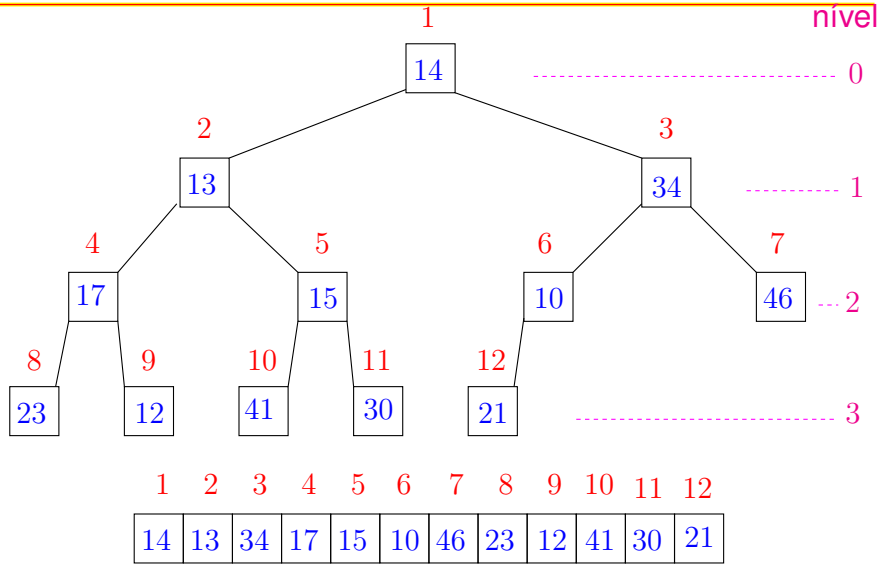
pois altura de $maior$ é $h-1$.

Solução assintótica: $T(n)$ é $O(h)$.

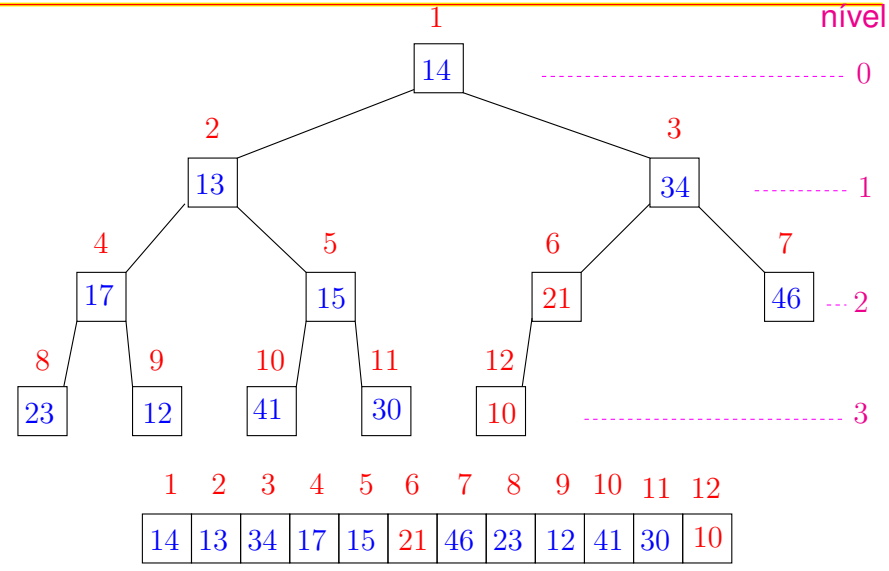
Como $h \leq \lg m$, podemos dizer que:

O consumo de tempo do algoritmo Heapify é $O(\lg m)$ (ou melhor

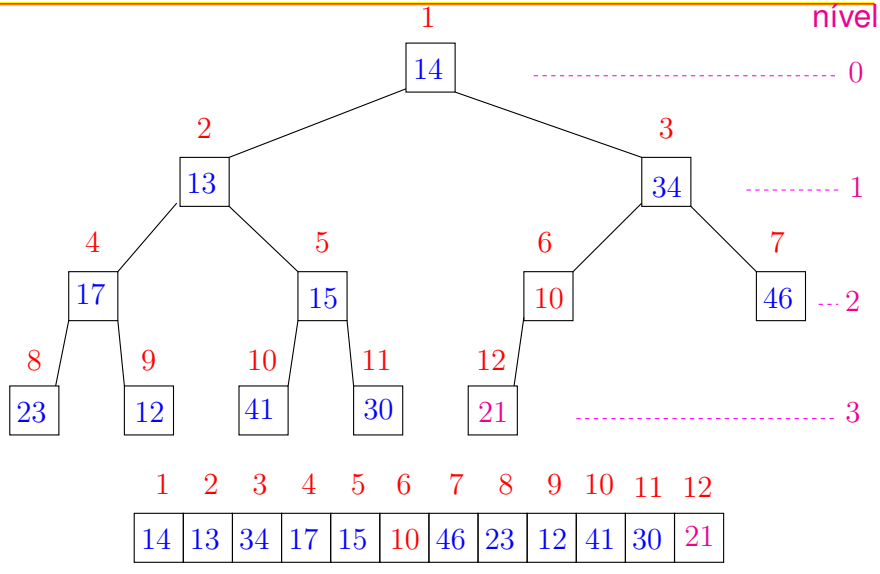
Construção de um Heap



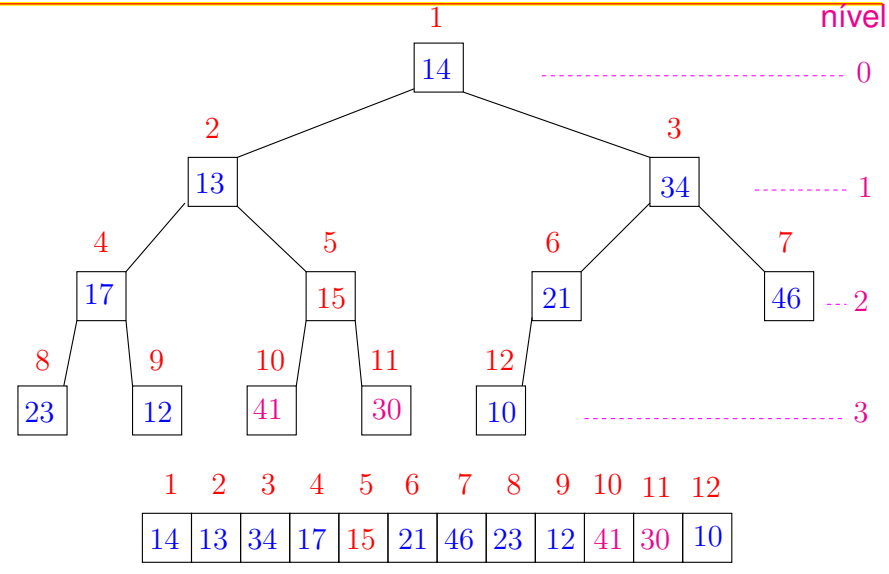
Construção de um Heap



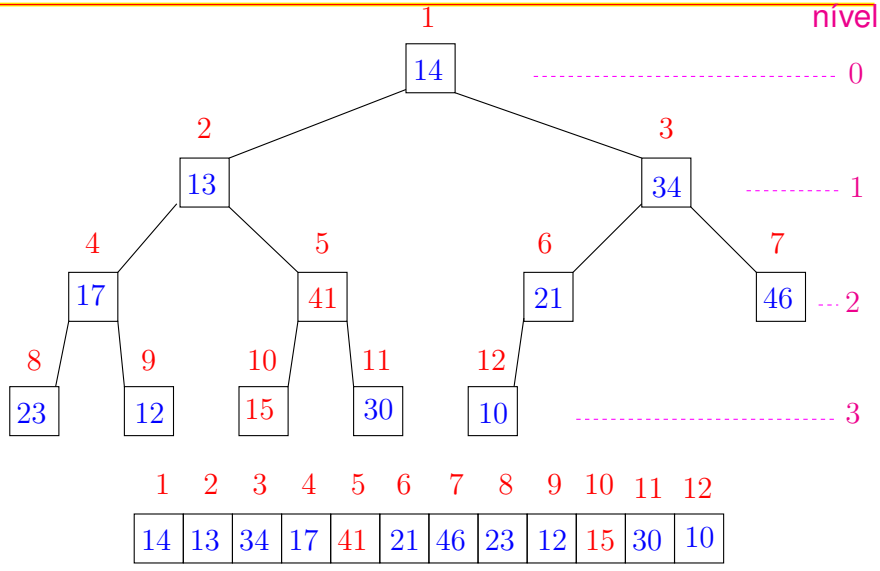
Construção de um Heap



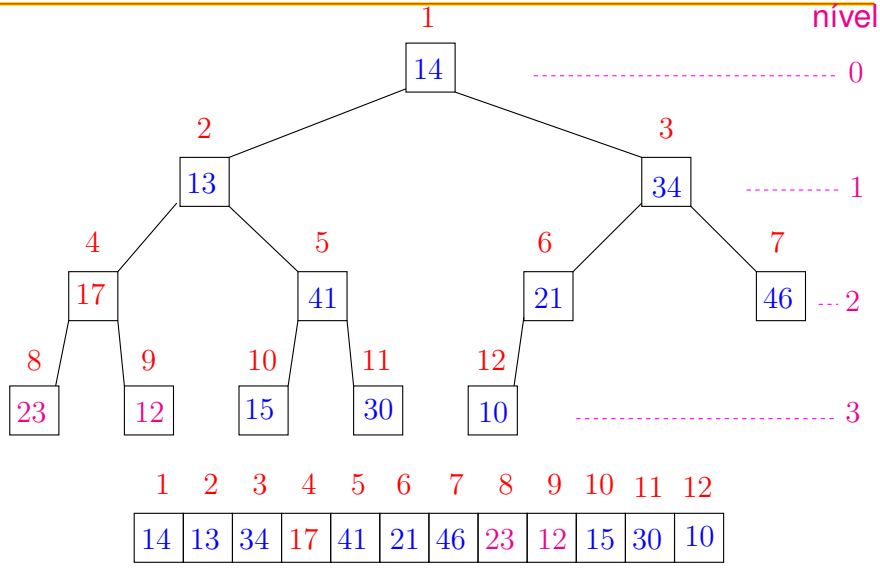
Construção de um Heap



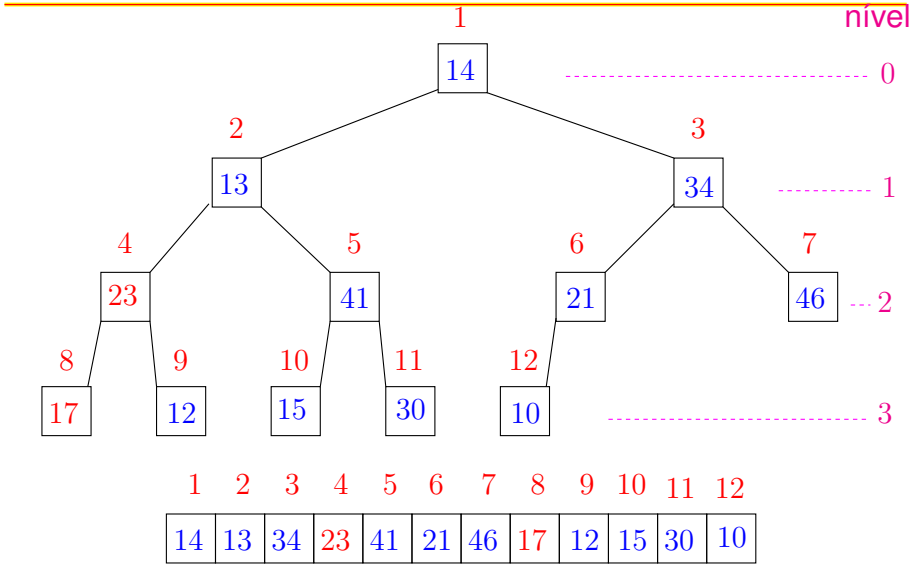
Construção de um Heap



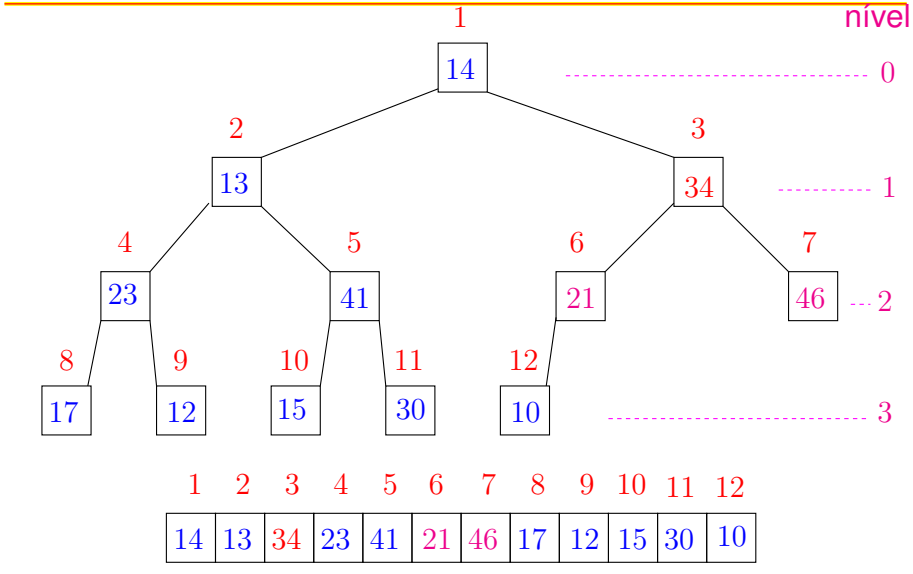
Construção de um Heap



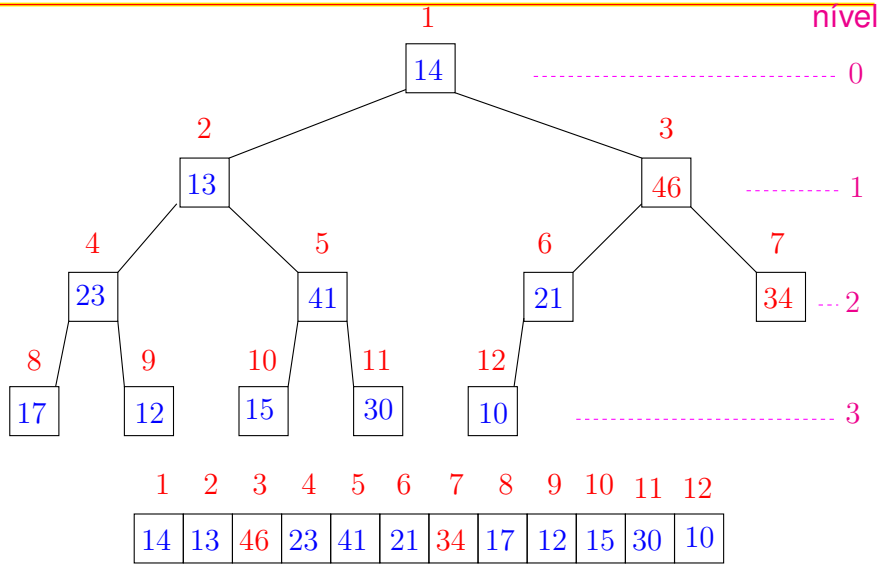
Construção de um Heap



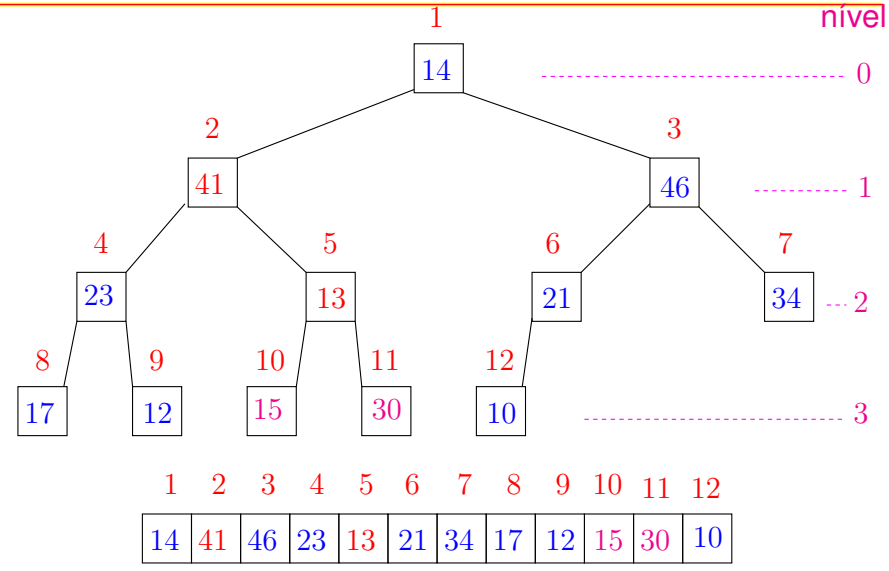
Construção de um Heap



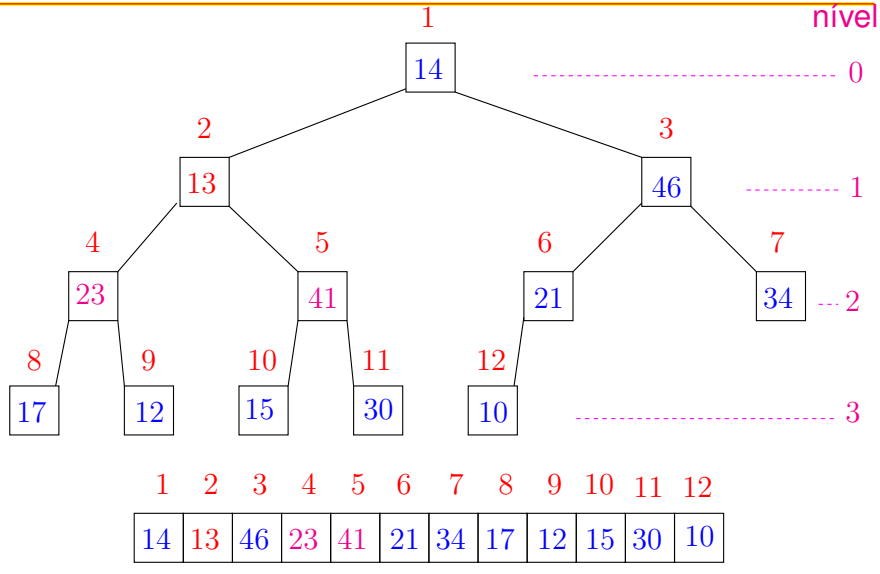
Construção de um Heap



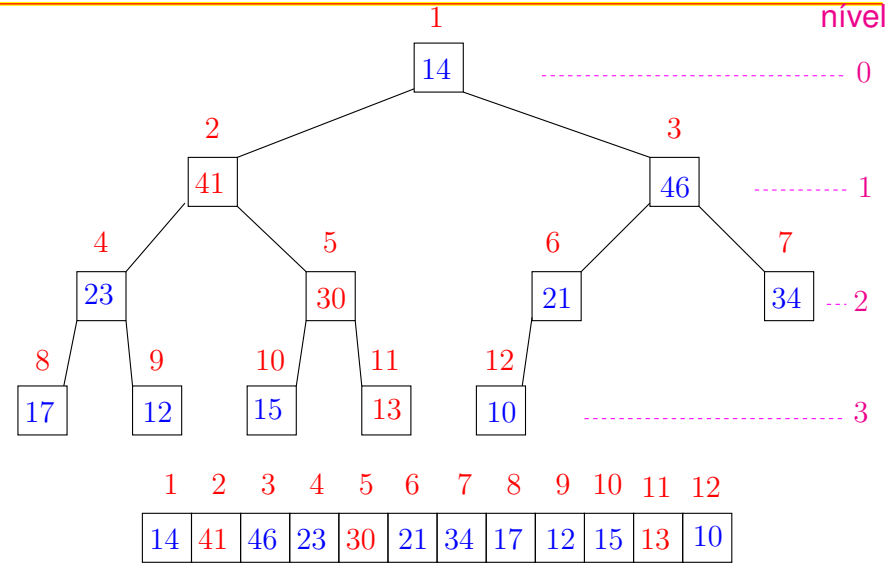
Construção de um Heap



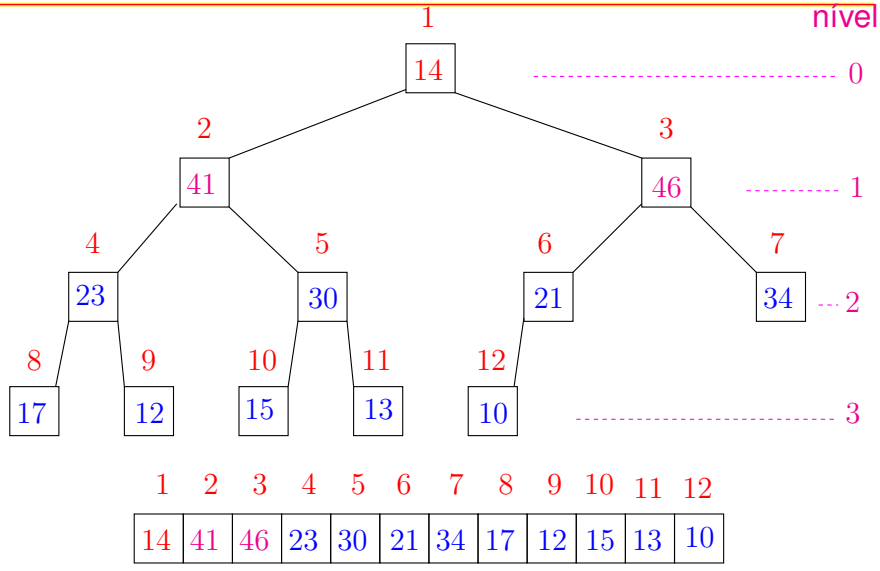
Construção de um Heap



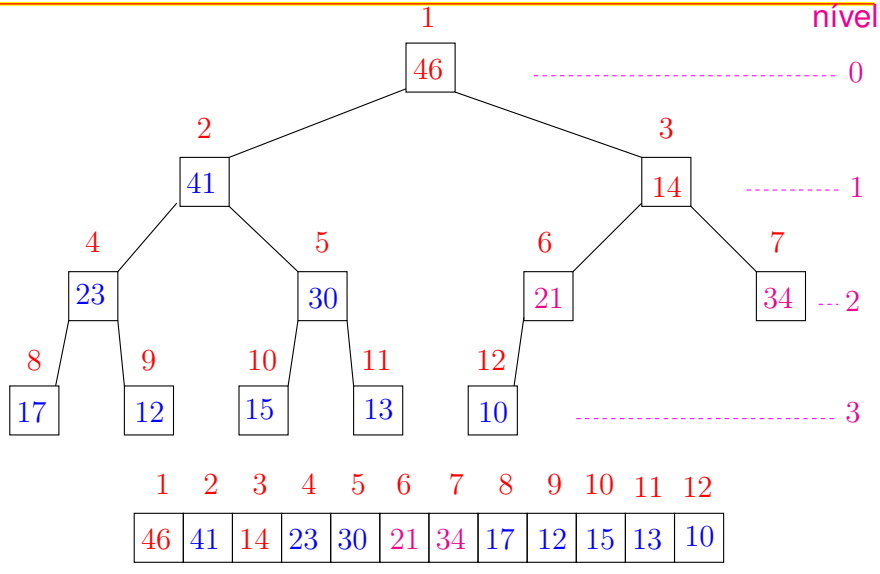
Construção de um Heap



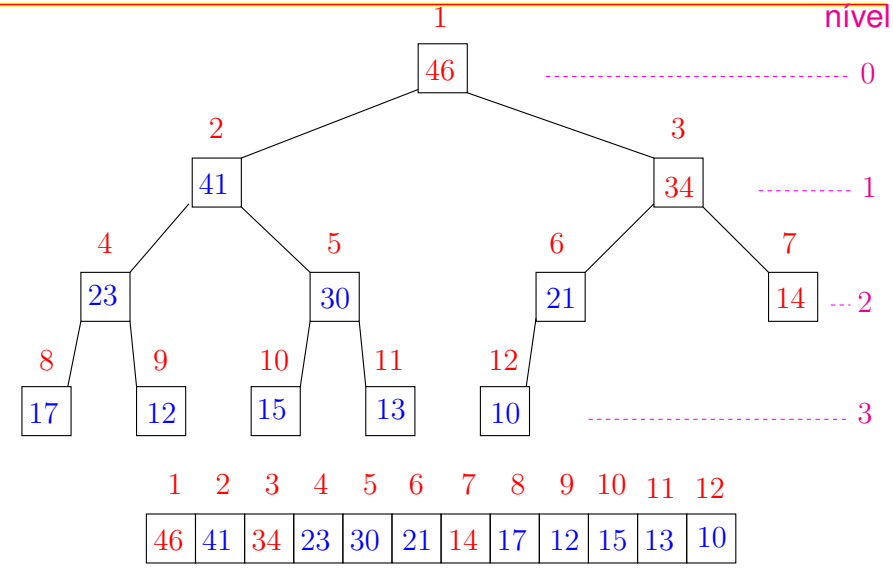
Construção de um Heap



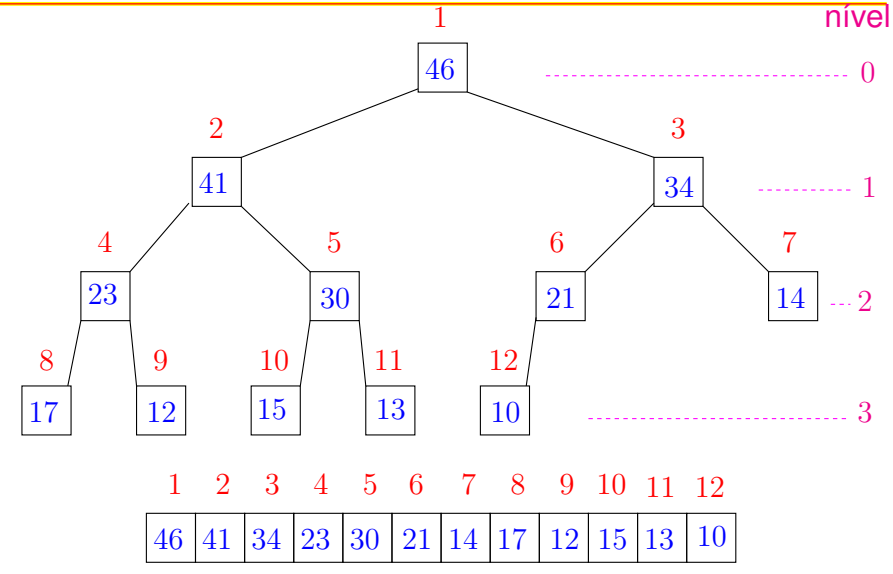
Construção de um Heap



Construção de um Heap



Construção de um Heap



Construção de um Heap

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

BuildHeap (A, n)

1: **for** $i = \lfloor n/2 \rfloor$ até 1 **do**
2: Heapify(A, n, i)

Relação invariante:

(i0) no início de cada iteração, $i+1, \ldots, n$ são raízes de heaps.

$T(n)$:= consumo de tempo no pior caso

$T(n)$ é $O(n)$

Prova O consumo de Heapify(A, n, i) é proporcional a $h = \lfloor \lg \frac{n}{i} \rfloor$. Logo,

$$\begin{aligned} T(n) &= \sum_{h=1}^{\lfloor \lg n \rfloor} \left(\lceil \frac{n}{2^{h+1}} \rceil \right) h \\ &\leq \sum_{h=1}^{\lfloor \lg n \rfloor} \frac{n}{2^h} h \quad \text{(Exercício 12.C)} \\ &\leq n \left(\frac{1}{2^1} + \frac{2}{2^2} + \frac{3}{2^3} + \cdots + \frac{\lfloor \lg n \rfloor}{2^{\lfloor \lg n \rfloor}} \right) \\ &< n \frac{1/2}{(1-1/2)^2} \\ &= 2n. \end{aligned}$$

Construção de um Heap

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

BuildHeap (A, n)

1: **for** $i = \lfloor n/2 \rfloor$ até 1 **do**
2: Heapify(A, n, i)

Relação invariante:

(i0) no início de cada iteração, $i+1, \ldots, n$ são raízes de heaps.

$T(n)$:= consumo de tempo no pior caso

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

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

$T(n)$ é $O(n)$

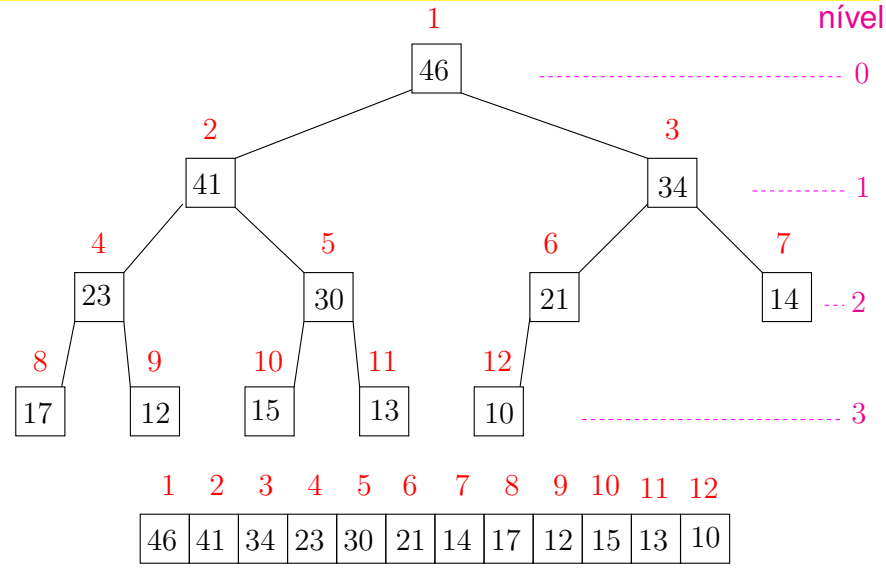
Prova O consumo de tempo de Heapify(A, n, i) é a $O(h) = O(\lfloor \lg \frac{n}{i} \rfloor)$. Logo,

$$\begin{aligned} T(n) &= \sum_{h=0}^{\lfloor \lg n \rfloor} \left(\lceil \frac{n}{2^{h+1}} \rceil \right) O(h) \\ &= O\left(n \sum_{h=0}^{\lfloor \lg n \rfloor} \frac{h}{2^h} \right) \quad \text{(Exercício 12.C)} \\ &= O\left(n \sum_{h=0}^{\infty} \frac{h}{2^h} \right) \\ &= O\left(n \frac{1/2}{(1-1/2)^2} \right) \\ &= O(2n) = O(n) \end{aligned}$$

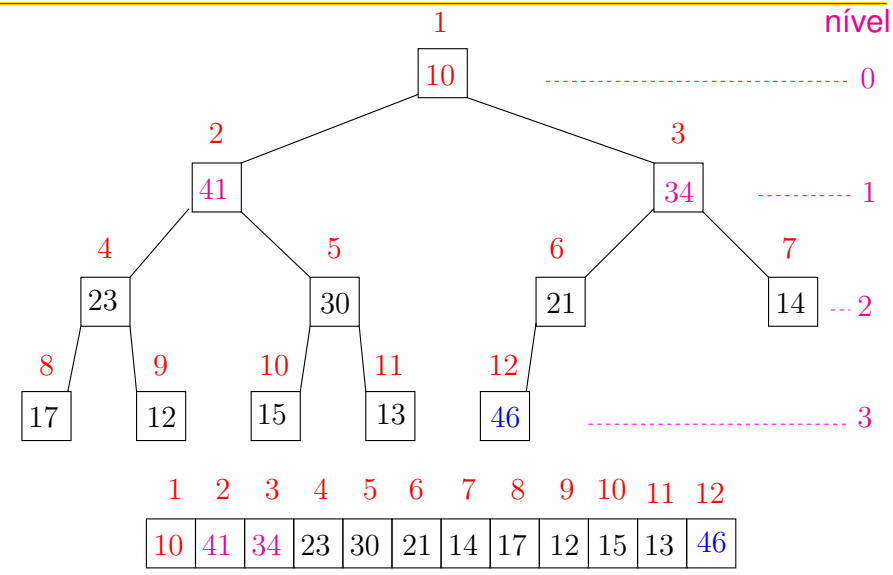
Conclusão

O consumo de tempo do algoritmo Heapify é $\Theta(n)$.

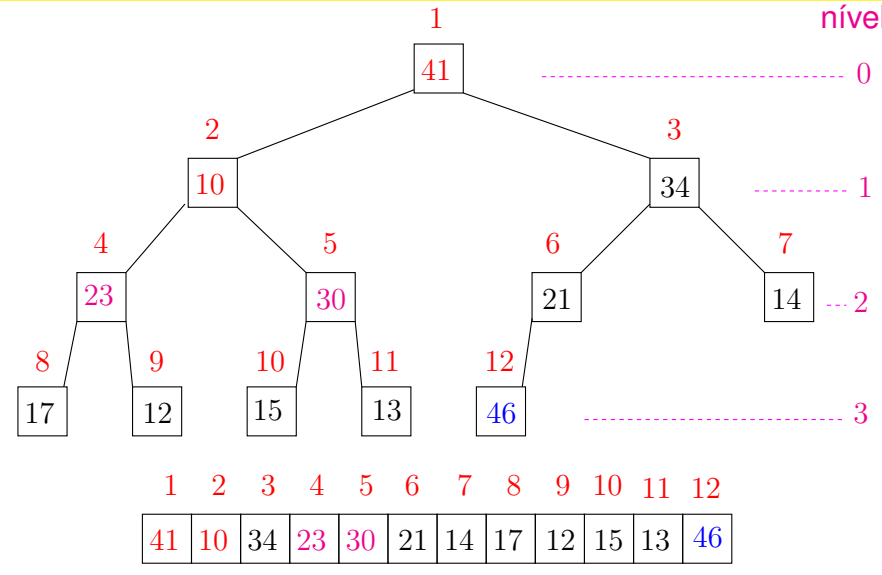
Heap sort



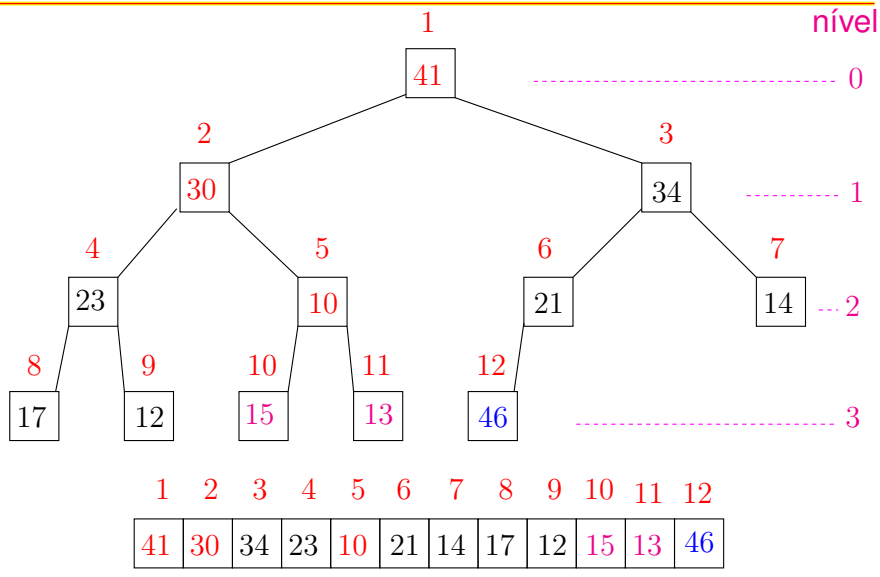
Heap sort



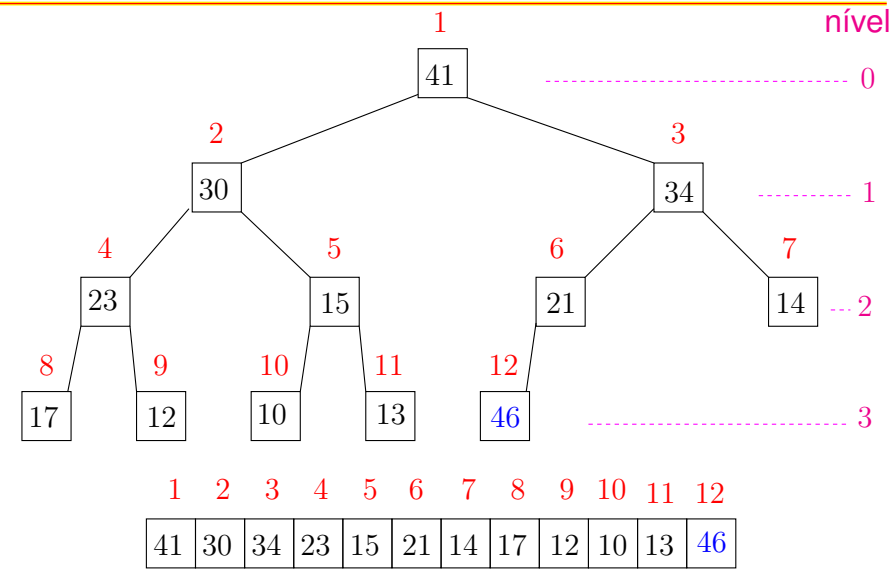
Heap sort



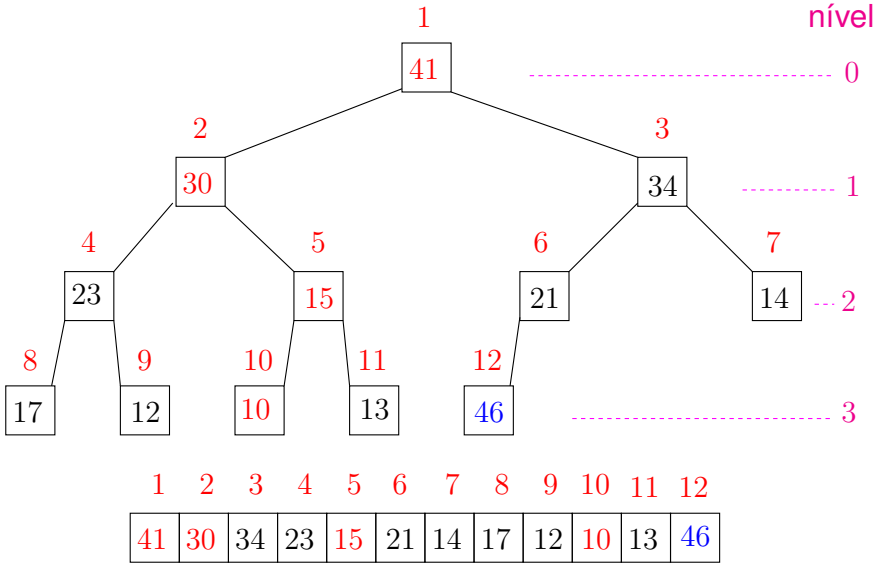
Heap sort



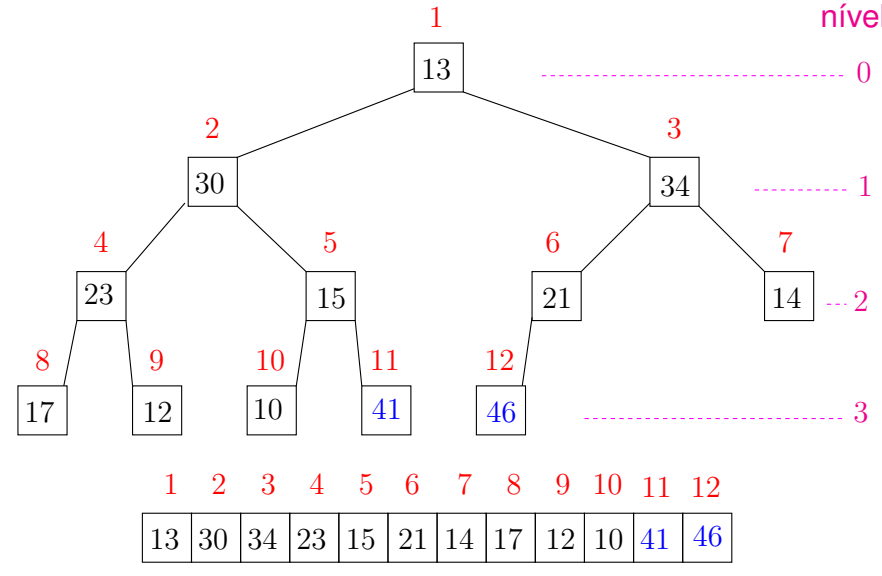
Heap sort



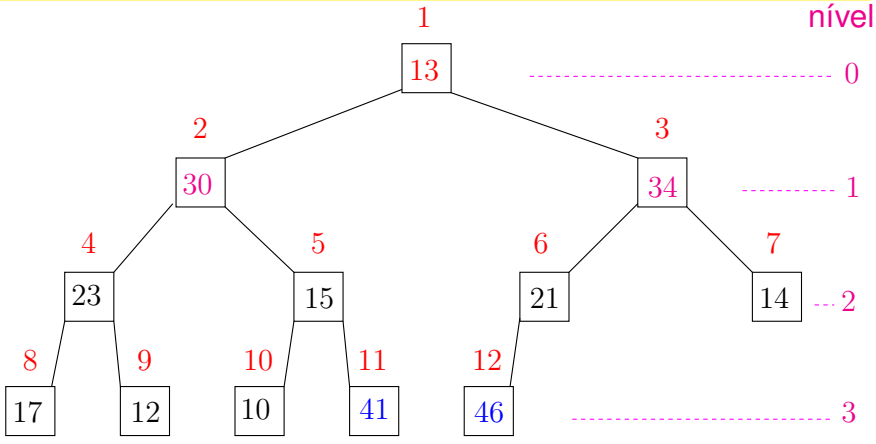
Heap sort



Heap sort

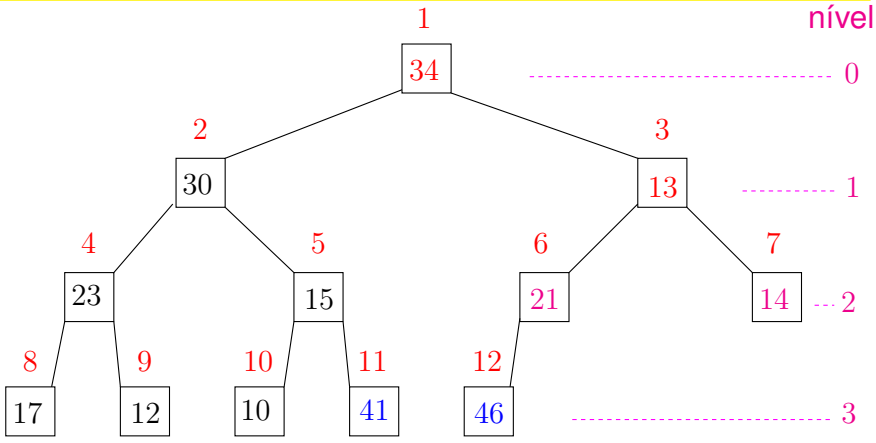


Heap sort



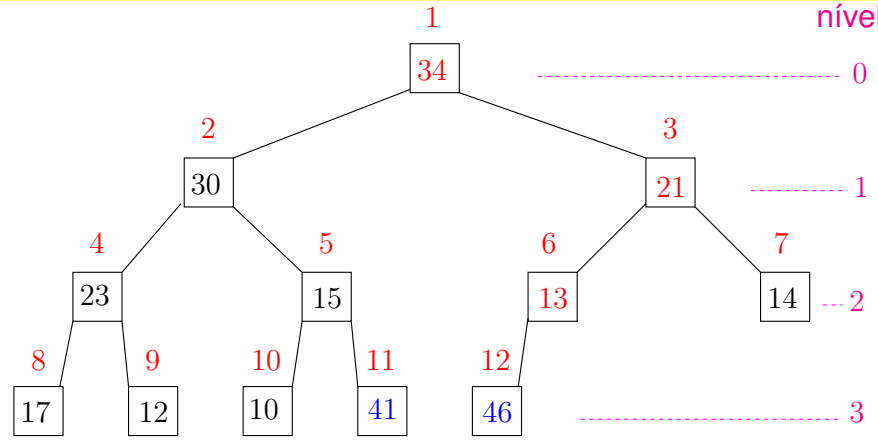
1	2	3	4	5	6	7	8	9	10	11	12
13	30	34	23	15	21	14	17	12	10	41	46

Heap sort



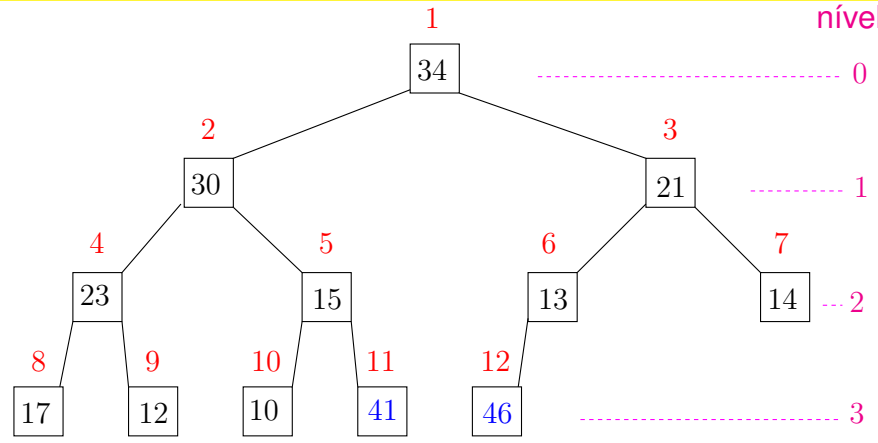
1	2	3	4	5	6	7	8	9	10	11	12
34	30	13	23	15	21	14	17	12	10	41	46

Heap sort



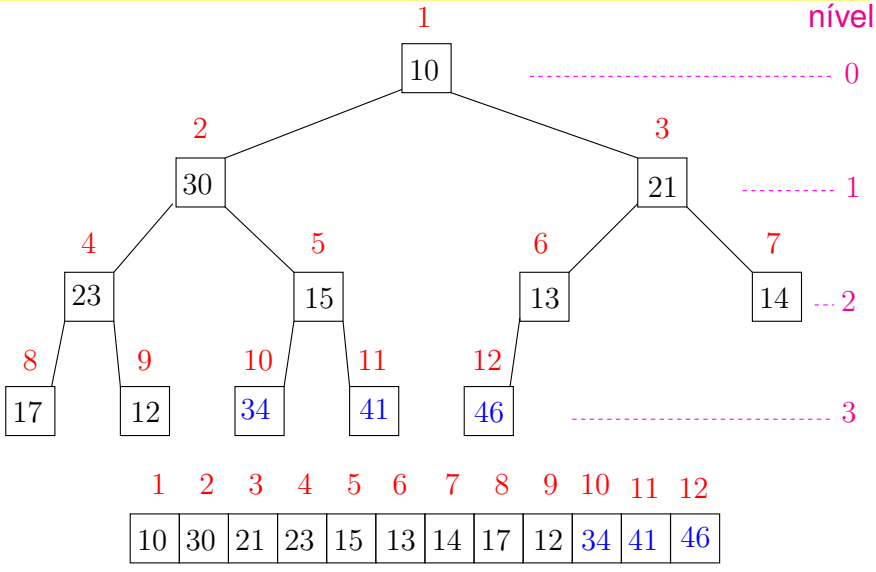
1	2	3	4	5	6	7	8	9	10	11	12
34	30	21	23	15	13	14	17	12	10	41	46

Heap sort

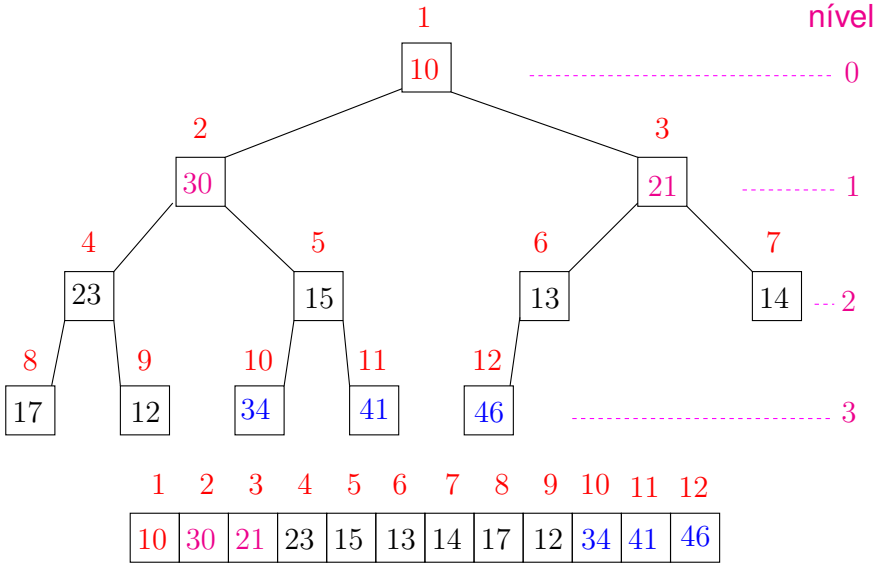


1	2	3	4	5	6	7	8	9	10	11	12
34	30	21	23	15	13	14	17	12	10	41	46

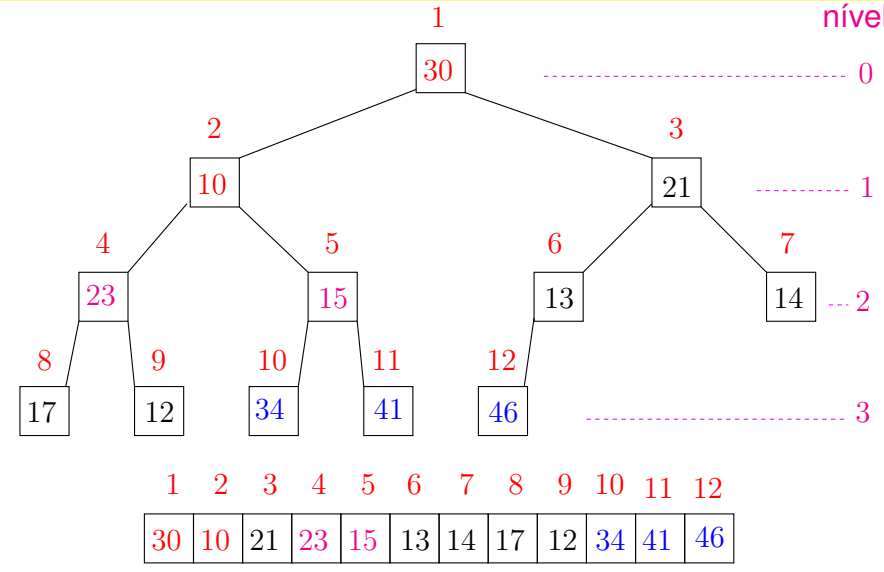
Heap sort



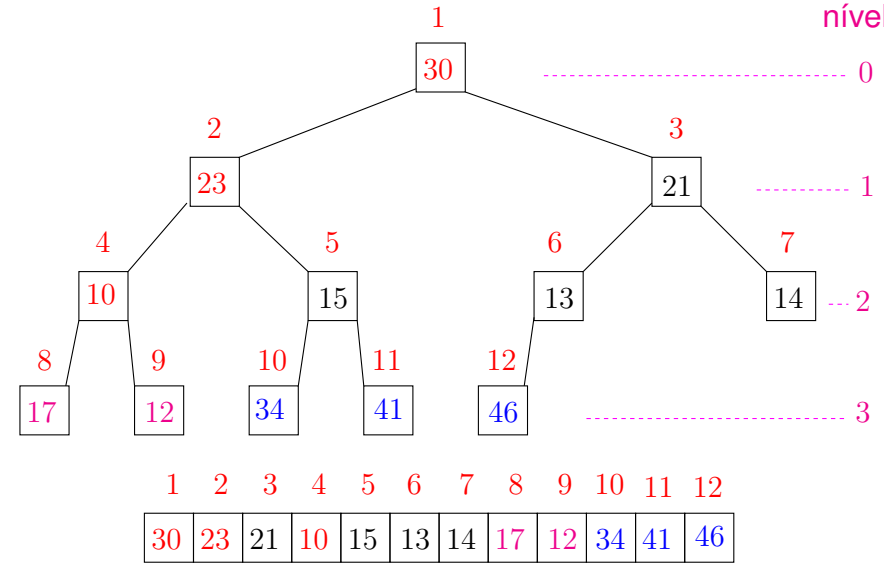
Heap sort



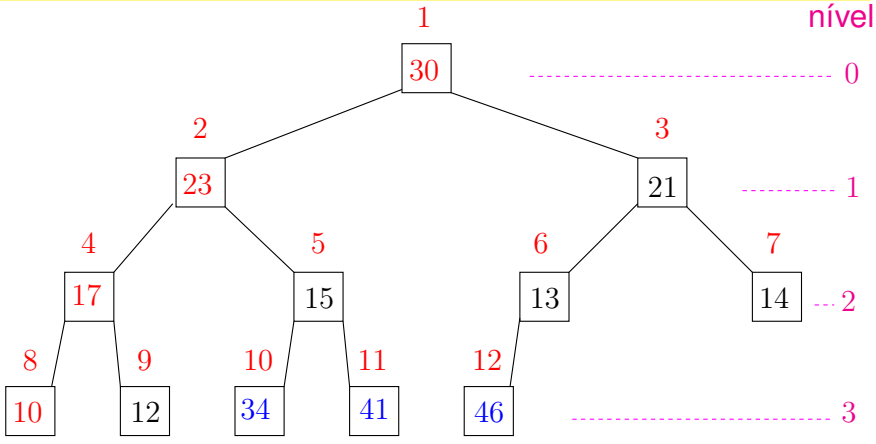
Heap sort



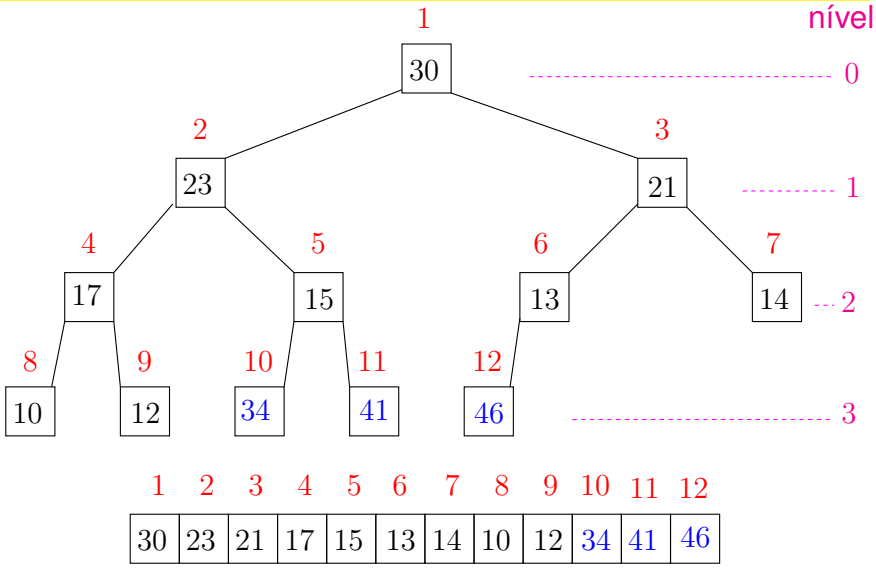
Heap sort



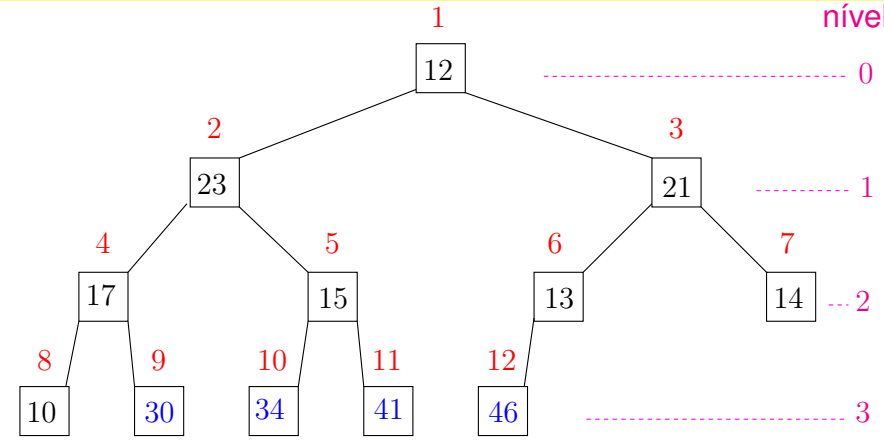
Heap sort



Heap sort



Heap sort



Heap sort

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

Heapsort(A, n)

1: BuildHeap(A, n) ▷ pré-processamento
2: $m = n$
3: **for** $i = n$ até 2 **do**
4: $\quad A[1] \leftrightarrow A[i]$
5: $\quad m = m - 1$
6: $\quad$ Heapify$(A, m, 1)$

Relações invariantes: Na linha 3 vale que:

(i0) $A[m \ldots n]$ é crescente;

(i1) $A[1 \ldots m] \leq A[m + 1]$;

(i2) $A[1 \ldots m]$ é um heap.

Consumo de tempo

linha		todas as execuções da linha
1	$=$	$\Theta(n)$
2	$=$	$\Theta(1)$
3	$=$	$\Theta(n)$
4	$=$	$\Theta(n)$
5	$=$	$\Theta(n)$
6	$=$	$nO(\lg n)$
total	$=$	$nO(\lg n) + \Theta(4n+1) = O(n \lg n)$

O consumo de tempo do algoritmo Heapsort é $O(n \lg n)$.

Fila de prioridades

- Estrutura de dados relacionada com um conjunto S de elementos, cada um deles associado a uma chave
- Uma fila de prioridades suporta as seguintes operações
  - Insere(S, x, n): insere o elemento x em S
  - Máximo(S, n): devolve o elemento de S com a maior chave
  - Extrai_max(S, n): remove e devolve o elemento de S com a maior chave

Implementação com heap

Fila de prioridades

Extrai_max(A, n)
if $n < 1$ **then**
  "Erro"
else
  $x = A[1]$
  $A[1] = A[n]$
  $n = n - 1$
  Heapipy$(A, 1, n)$
  devolva x

Complexidade: $O(\lg n)$