

Ordenação: algoritmo Heapsort

PF 10

<http://www.ime.usp.br/~pf/algoritmos/aulas/hpsrt.html>

Ordenação

$v[1 : n]$ é **crescente** se $v[1] \leq \dots \leq v[n-1]$.

Problema: Rearranjar uma lista $v[1 : n]$ 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

Heapsort

O **Heapsort** ilustra o uso de **estruturas de dados** no projeto de algoritmos eficientes.

Rearranjar uma lista $v[1 : n]$ de modo que ele fique **crecente**.

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 por seleção

$i = 5$

	1				max						n
38	50	20	44	10	50	55	60	75	85	99	

Ordenação por seleção

$i = 5$

1

j max

n

38	50	20	44	10	50	55	60	75	85	99
----	----	----	----	----	----	----	----	----	----	----

Ordenação por seleção

$i = 5$

1			j	max						n
38	50	20	44	10	50	55	60	75	85	99

1		j	max							n
38	50	20	44	10	50	55	60	75	85	99

Ordenação por seleção

$i = 5$

1			j	max							n
38	50	20	44	10	50	55	60	75	85	99	

1		j	max								n
38	50	20	44	10	50	55	60	75	85	99	

1	j		max								n
38	50	20	44	10	50	55	60	75	85	99	

Ordenação por seleção

$i = 5$

1			j	max							n
38	50	20	44	10	50	55	60	75	85	99	

1		j	max								n
38	50	20	44	10	50	55	60	75	85	99	

1	j		max								n
38	50	20	44	10	50	55	60	75	85	99	

	j	max									n
38	50	20	44	10	50	55	60	75	85	99	

Ordenação por seleção

$i = 5$

1			j	max							n
38	50	20	44	10	50	55	60	75	85	99	

1		j	max								n
38	50	20	44	10	50	55	60	75	85	99	

1	j		max								n
38	50	20	44	10	50	55	60	75	85	99	

	j	max									n
38	50	20	44	10	50	55	60	75	85	99	

1	max										n
38	50	20	44	10	50	55	60	75	85	99	

Ordenação por seleção

1			<i>i</i>							<i>n</i>
38	10	20	44	50	50	55	60	75	85	99

Ordenação por seleção

1			i							n
38	10	20	44	50	50	55	60	75	85	99

1			i							n
38	10	20	44	50	50	55	60	75	85	99

Ordenação por seleção

1			<i>i</i>							<i>n</i>
38	10	20	44	50	50	55	60	75	85	99

1		<i>i</i>								<i>n</i>
20	10	38	44	50	50	55	60	75	85	99

1	<i>i</i>									<i>n</i>
20	10	38	44	50	50	55	60	75	85	99

Ordenação por seleção

1			<i>i</i>							<i>n</i>
38	10	20	44	50	50	55	60	75	85	99

1		<i>i</i>								<i>n</i>
20	10	38	44	50	50	55	60	75	85	99

1	<i>i</i>									<i>n</i>
10	20	38	44	50	50	55	60	75	85	99

1										<i>n</i>
10	20	38	44	50	50	55	60	75	85	99

Função selecao

Algoritmo rearranja $v[0 : n]$ em ordem crescente

```
def selecao(v):
0     n = len(v)
1     for i in range(n-1, 0, -1):    #B#
2         max = i
3         for j in range(i-1, -1, -1):
4             if v[j] > v[max]:    max = j
5         v[i], v[max] = v[max], v[i]
```

Função selecao

Algoritmo rearranja $v[\underline{1} : n]$ em ordem crescente

```
def selecao(v):
0     n = len(v)
1     for i in range(n-1, 1, -1):    #B#
2         max = i
3         for j in range(i-1, 0, -1):
4             if v[j] > v[max]:    max = j
5             v[i], v[max] = v[max], v[i]
```

Função selecao

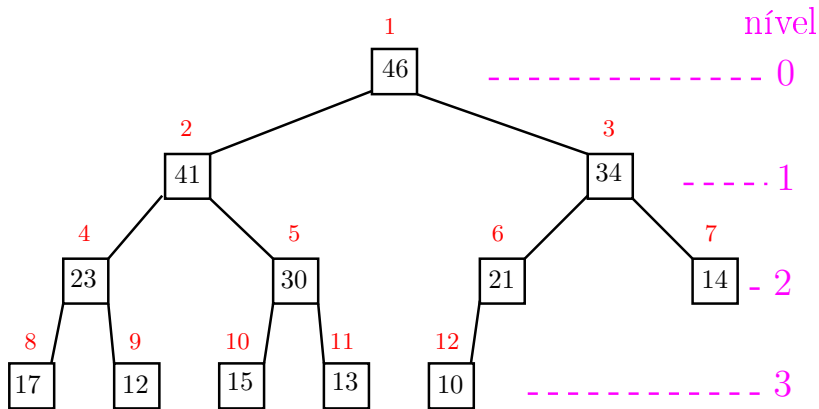
Relações invariantes: Em #B# vale que:

(i0) $v[i+1 : n]$ é crescente;

(i1) $v[1 : i+1] \leq v[i+1]$;

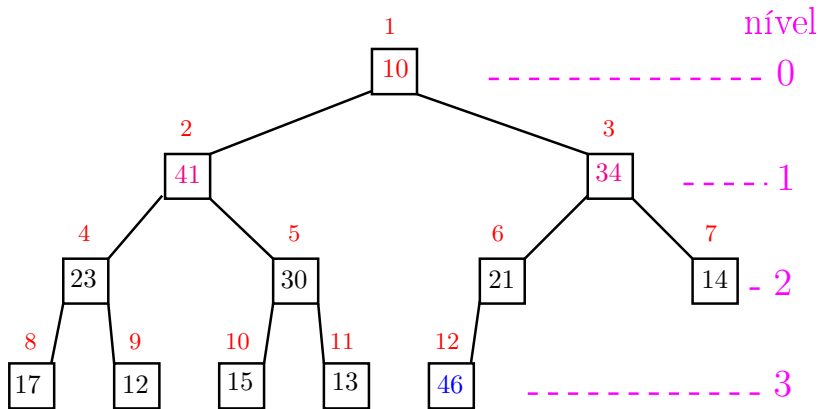
1			<i>i</i>							<i>n</i>
38	10	20	44	50	50	55	60	75	85	99

Heapsort



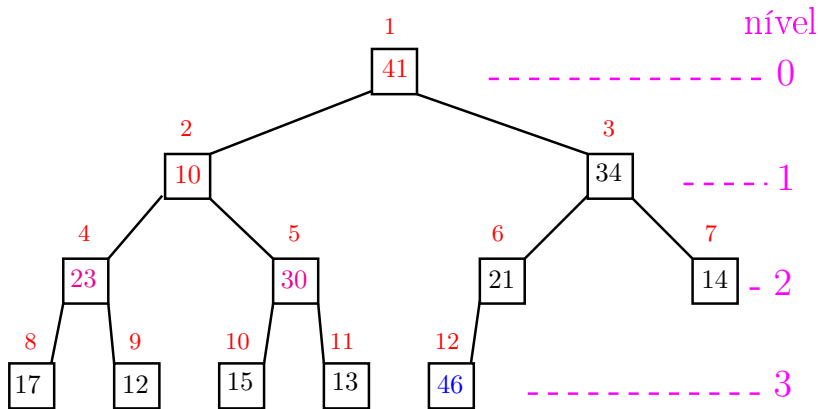
1	2	3	4	5	6	7	8	9	10	11	12
46	41	34	23	30	21	14	17	12	15	13	10

Heapsort



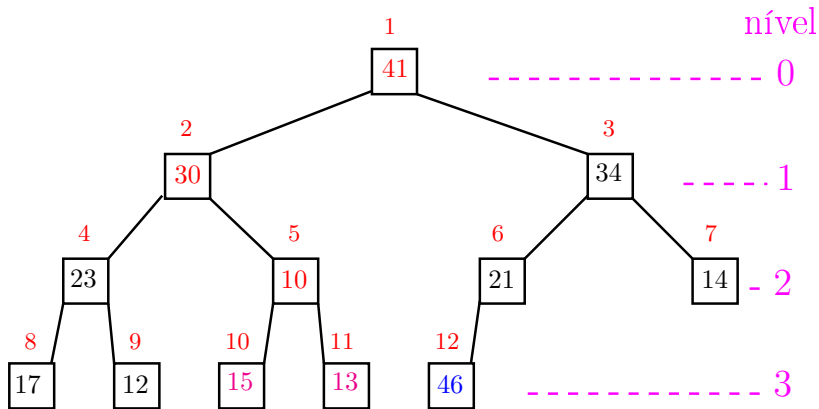
1	2	3	4	5	6	7	8	9	10	11	12
10	41	34	23	30	21	14	17	12	15	13	46

Heapsort



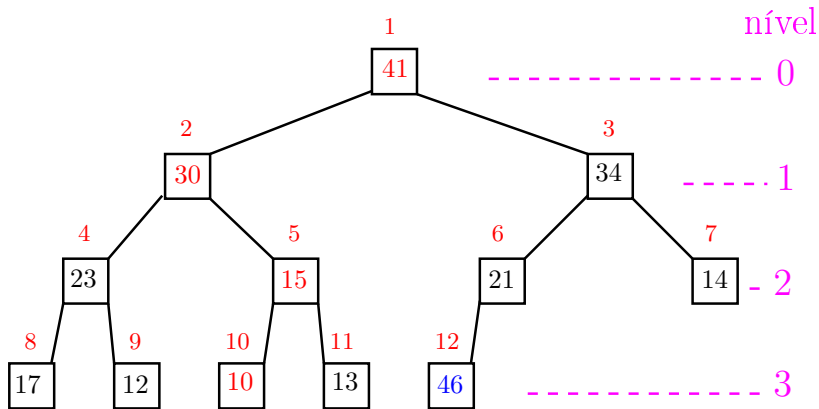
1	2	3	4	5	6	7	8	9	10	11	12
41	10	34	23	30	21	14	17	12	15	13	46

Heapsort



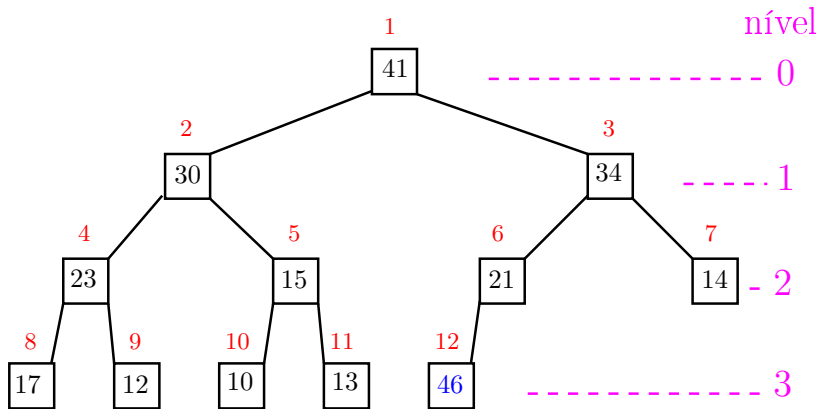
1	2	3	4	5	6	7	8	9	10	11	12
41	30	34	23	10	21	14	17	12	15	13	46

Heapsort



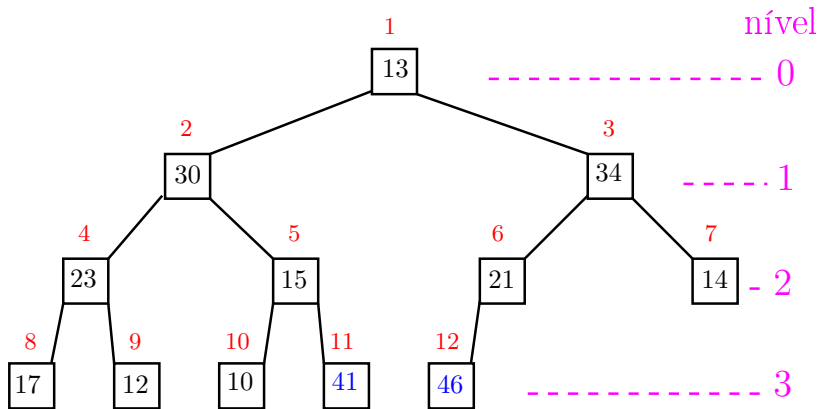
1	2	3	4	5	6	7	8	9	10	11	12
41	30	34	23	15	21	14	17	12	10	13	46

Heapsort



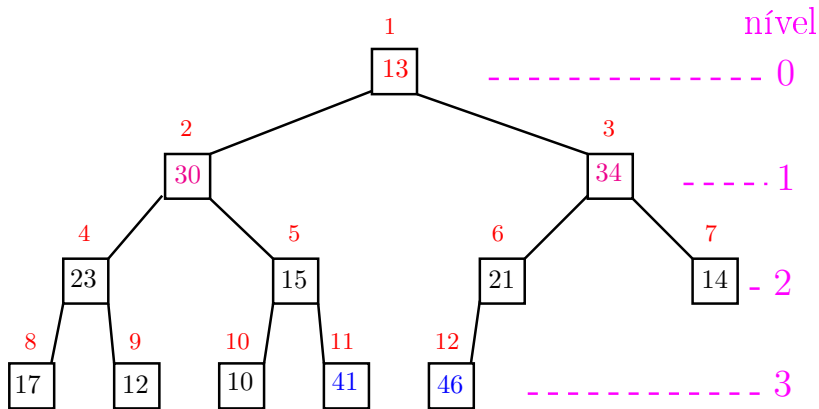
1	2	3	4	5	6	7	8	9	10	11	12
41	30	34	23	15	21	14	17	12	10	13	46

Heapsort



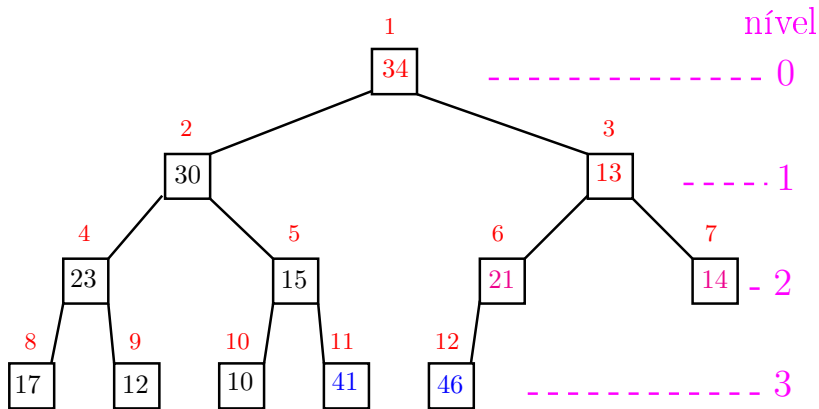
1	2	3	4	5	6	7	8	9	10	11	12
13	30	34	23	15	21	14	17	12	10	41	46

Heapsort



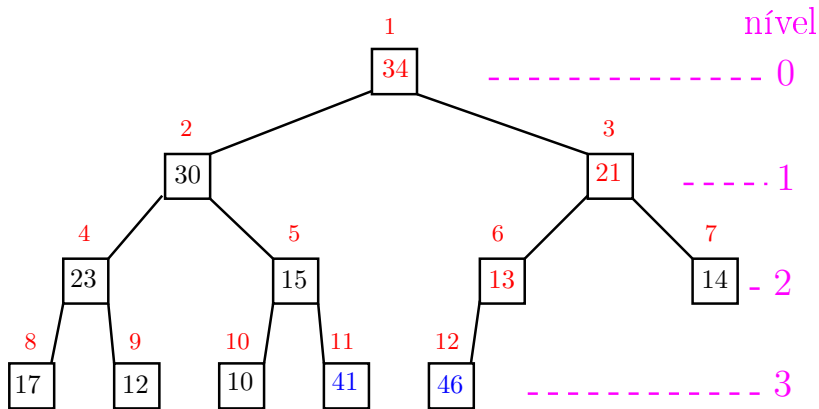
1	2	3	4	5	6	7	8	9	10	11	12
13	30	34	23	15	21	14	17	12	10	41	46

Heapsort



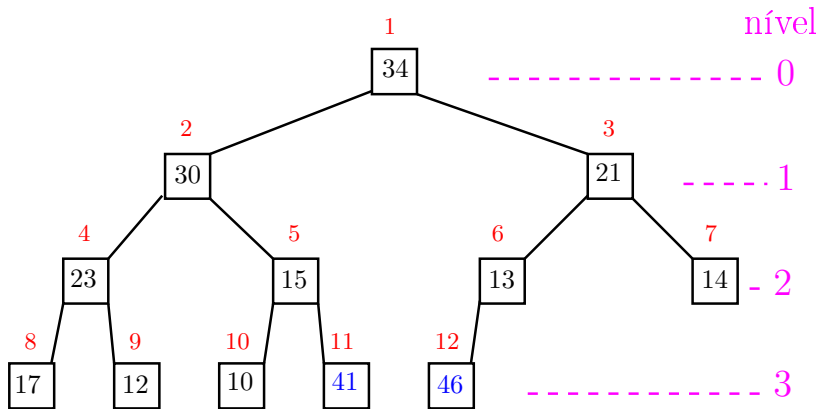
1	2	3	4	5	6	7	8	9	10	11	12
34	30	13	23	15	21	14	17	12	10	41	46

Heapsort



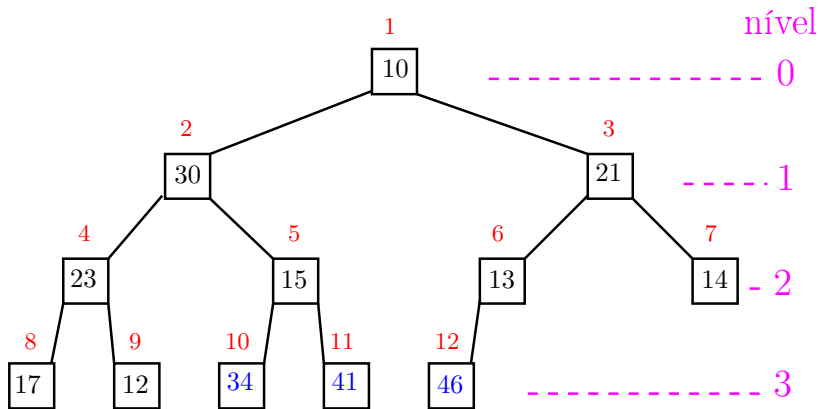
1	2	3	4	5	6	7	8	9	10	11	12
34	30	21	23	15	13	14	17	12	10	41	46

Heapsort



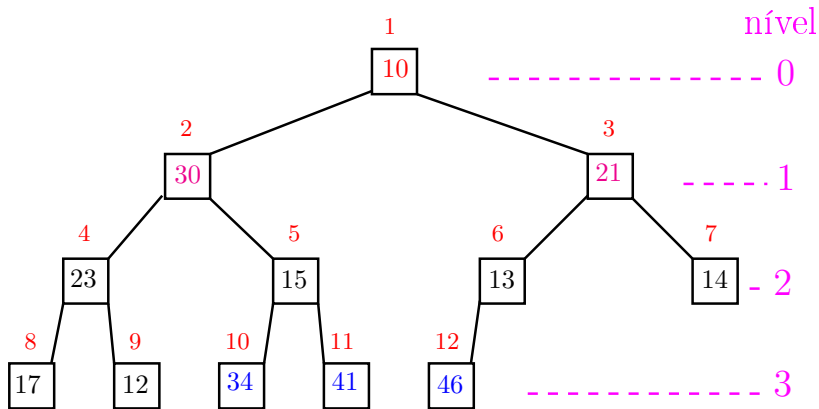
1	2	3	4	5	6	7	8	9	10	11	12
34	30	21	23	15	13	14	17	12	10	41	46

Heapsort



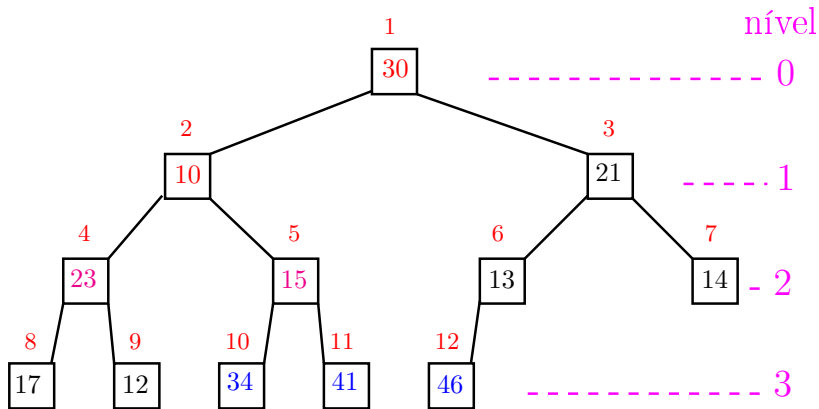
1	2	3	4	5	6	7	8	9	10	11	12
10	30	21	23	15	13	14	17	12	34	41	46

Heapsort



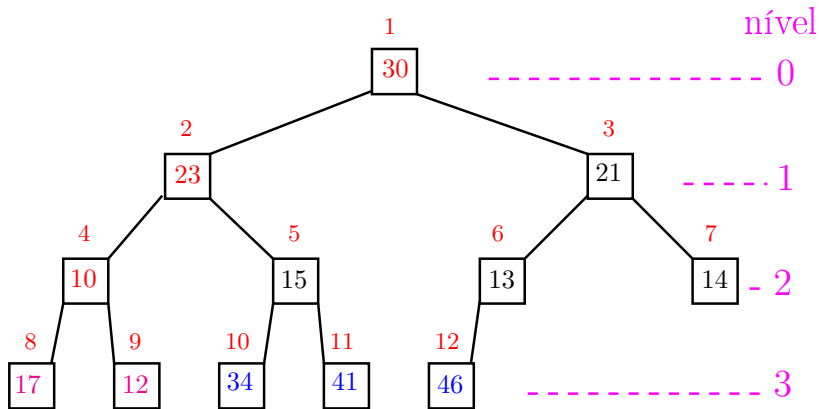
1	2	3	4	5	6	7	8	9	10	11	12
10	30	21	23	15	13	14	17	12	34	41	46

Heapsort



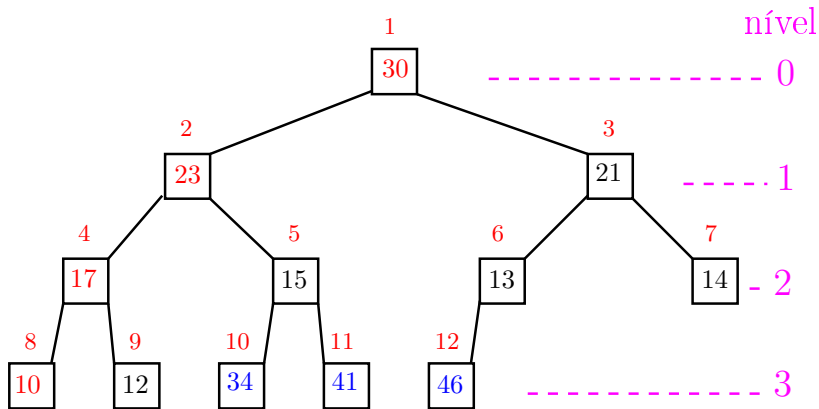
1	2	3	4	5	6	7	8	9	10	11	12
30	10	21	23	15	13	14	17	12	34	41	46

Heapsort



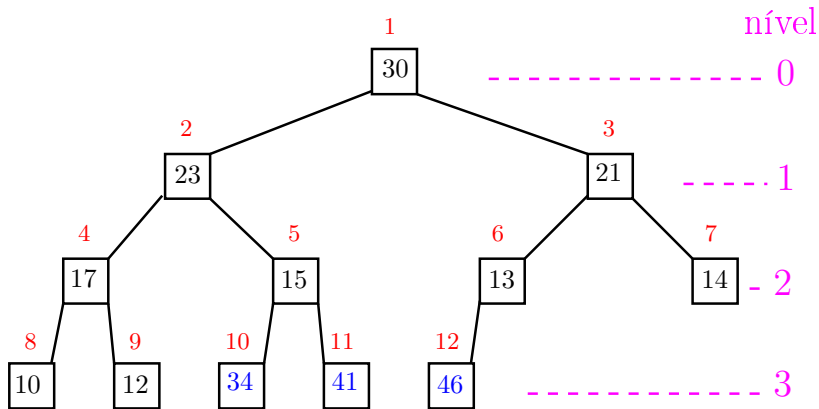
1	2	3	4	5	6	7	8	9	10	11	12
30	23	21	10	15	13	14	17	12	34	41	46

Heapsort



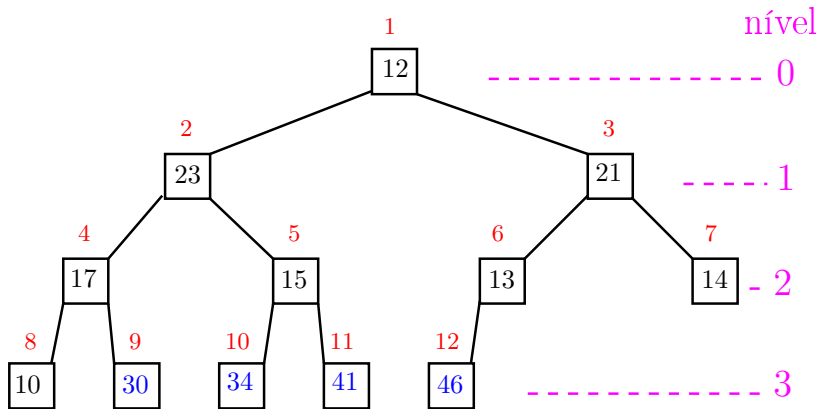
1	2	3	4	5	6	7	8	9	10	11	12
30	23	21	17	15	13	14	10	12	34	41	46

Heapsort



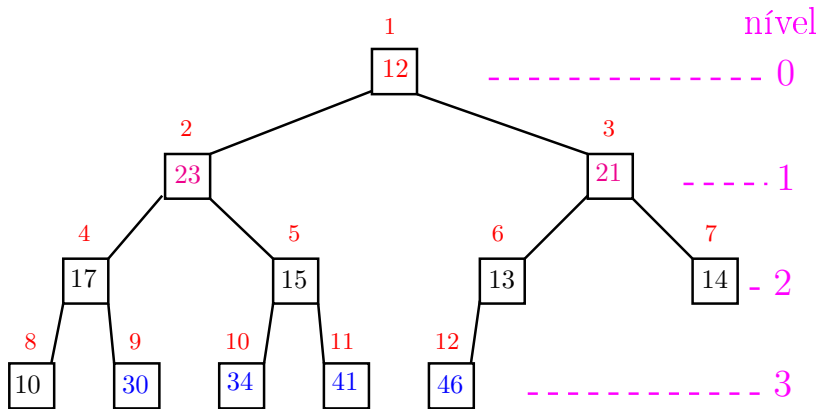
1	2	3	4	5	6	7	8	9	10	11	12
30	23	21	17	15	13	14	10	12	34	41	46

Heapsort



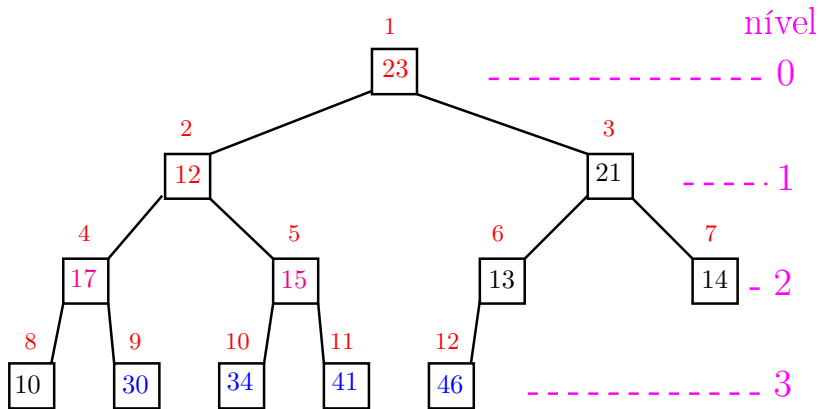
1	2	3	4	5	6	7	8	9	10	11	12
12	23	21	17	15	13	14	10	30	34	41	46

Heapsort



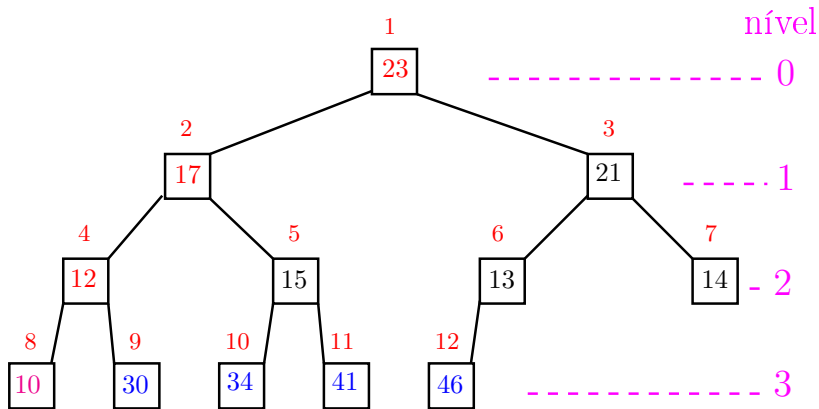
1	2	3	4	5	6	7	8	9	10	11	12
12	23	21	17	15	13	14	10	30	34	41	46

Heapsort



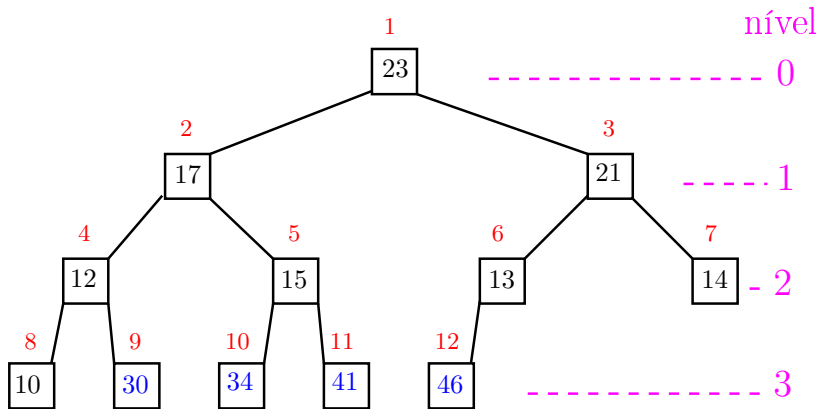
1	2	3	4	5	6	7	8	9	10	11	12
23	12	21	17	15	13	14	10	30	34	41	46

Heapsort



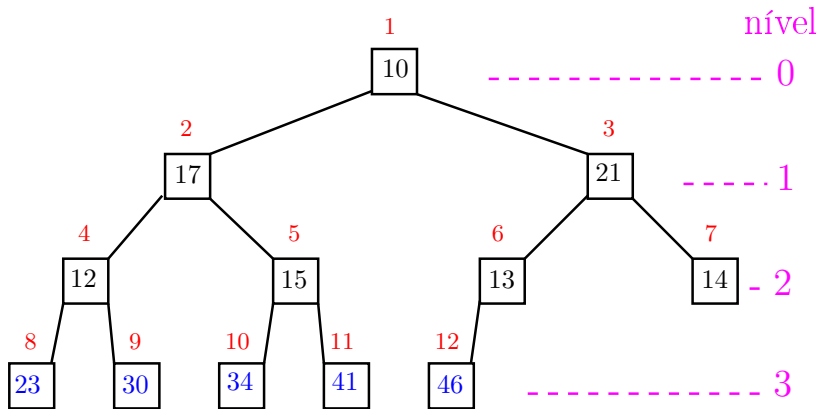
1	2	3	4	5	6	7	8	9	10	11	12
23	17	21	12	15	13	14	10	30	34	41	46

Heapsort



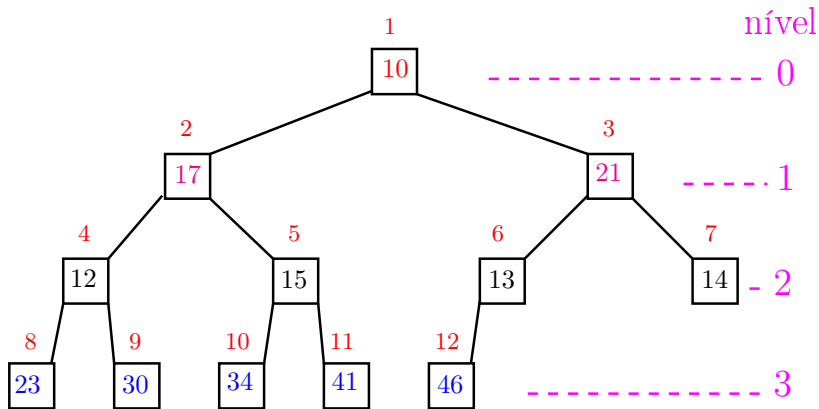
1	2	3	4	5	6	7	8	9	10	11	12
23	17	21	12	15	13	14	10	30	34	41	46

Heapsort



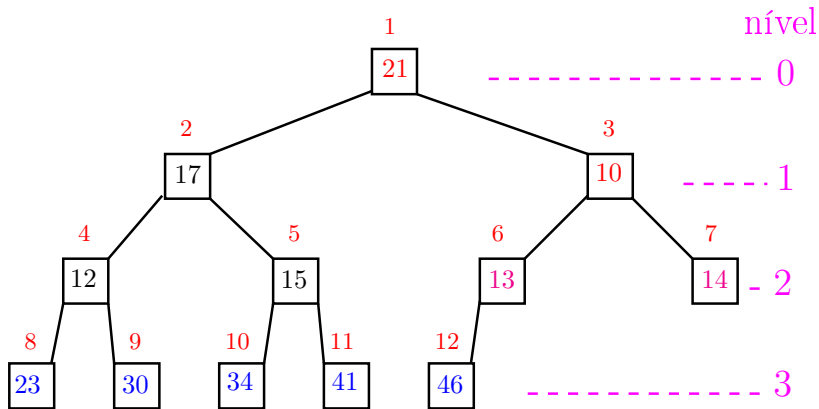
1	2	3	4	5	6	7	8	9	10	11	12
10	17	21	12	15	13	14	23	30	34	41	46

Heapsort



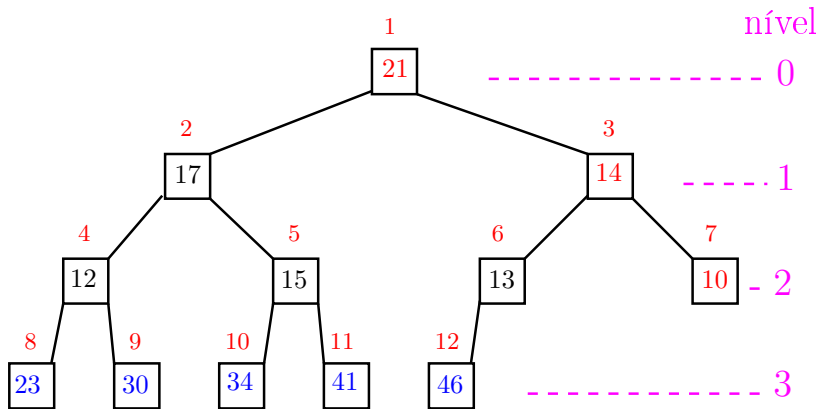
1	2	3	4	5	6	7	8	9	10	11	12
10	17	21	12	15	13	14	23	30	34	41	46

Heapsort



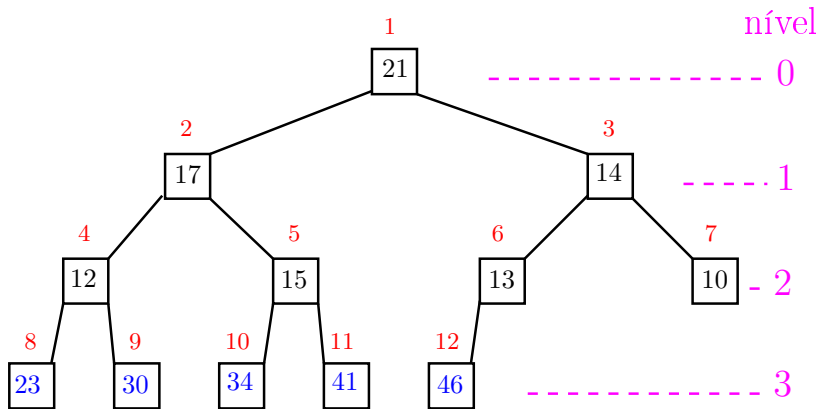
1	2	3	4	5	6	7	8	9	10	11	12
21	17	10	12	15	13	14	23	30	34	41	46

Heapsort



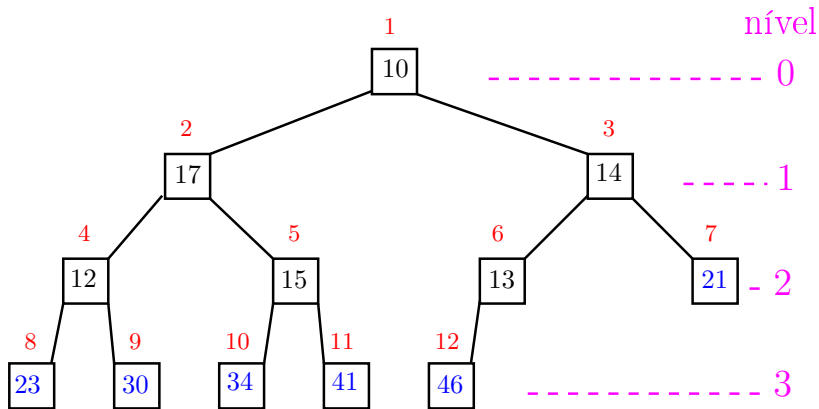
1	2	3	4	5	6	7	8	9	10	11	12
21	17	14	12	15	13	10	23	30	34	41	46

Heapsort



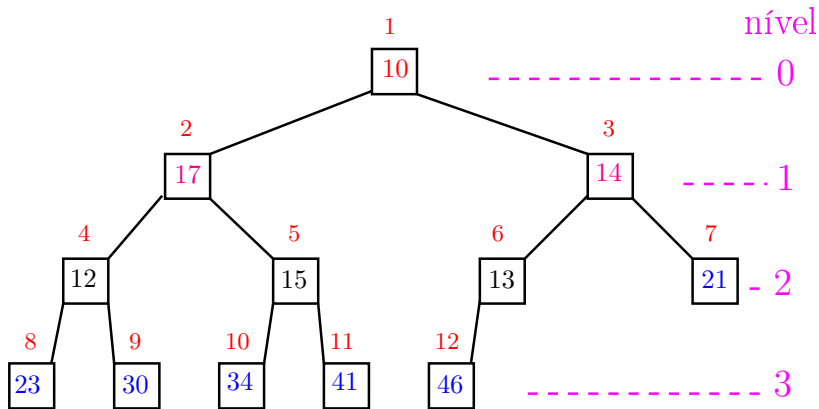
1	2	3	4	5	6	7	8	9	10	11	12
21	17	14	12	15	13	10	23	30	34	41	46

Heapsort



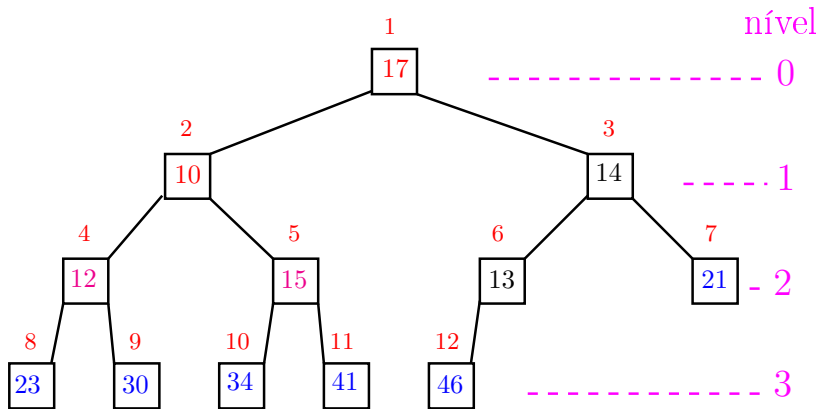
1	2	3	4	5	6	7	8	9	10	11	12
10	17	14	12	15	13	21	23	30	34	41	46

Heapsort



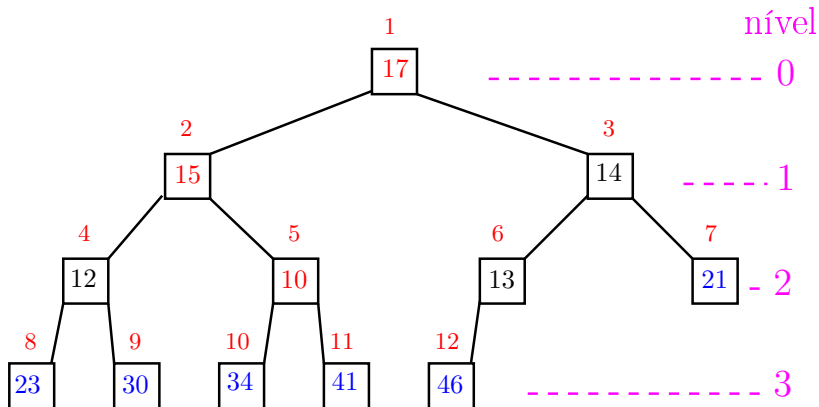
1	2	3	4	5	6	7	8	9	10	11	12
10	17	14	12	15	13	21	23	30	34	41	46

Heapsort



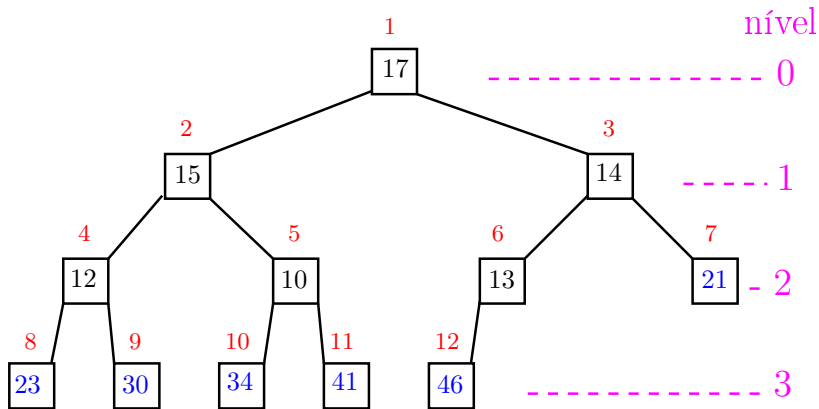
1	2	3	4	5	6	7	8	9	10	11	12
17	10	14	12	15	13	21	23	30	34	41	46

Heapsort



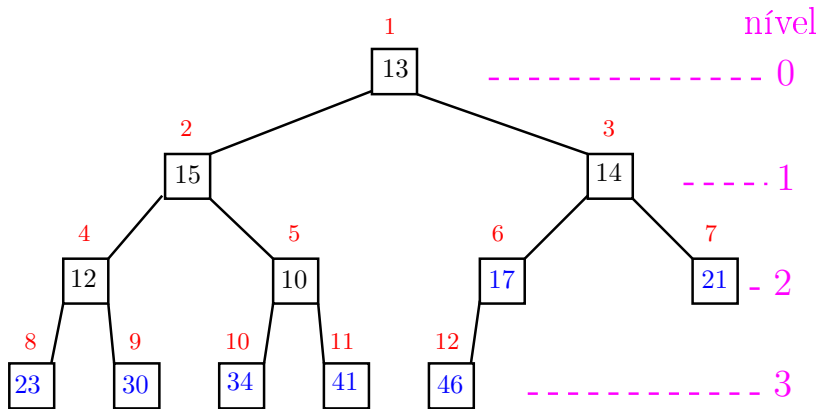
1	2	3	4	5	6	7	8	9	10	11	12
17	15	14	12	10	13	21	23	30	34	41	46

Heapsort



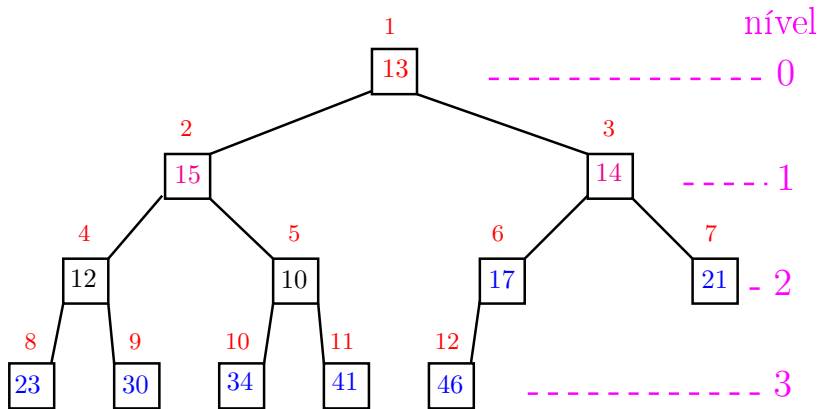
1	2	3	4	5	6	7	8	9	10	11	12
17	15	14	12	10	13	21	23	30	34	41	46

Heapsort



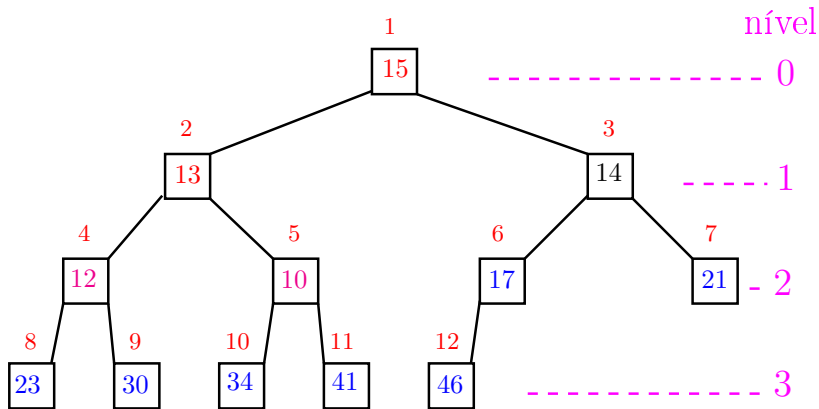
1	2	3	4	5	6	7	8	9	10	11	12
13	15	14	12	10	17	21	23	30	34	41	46

Heapsort



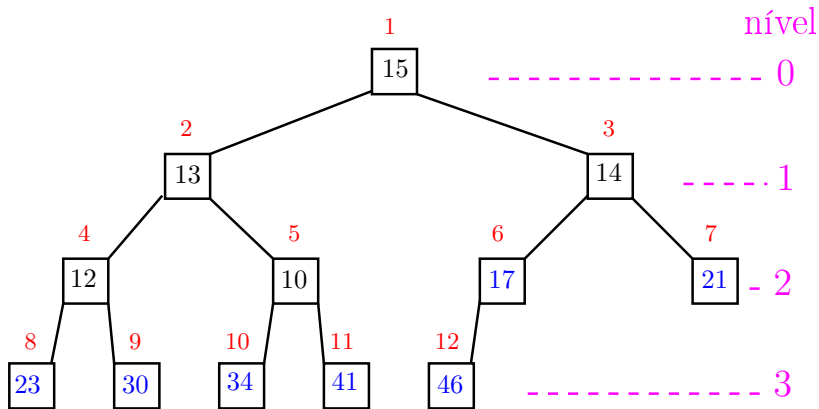
1	2	3	4	5	6	7	8	9	10	11	12
13	15	14	12	10	17	21	23	30	34	41	46

Heapsort



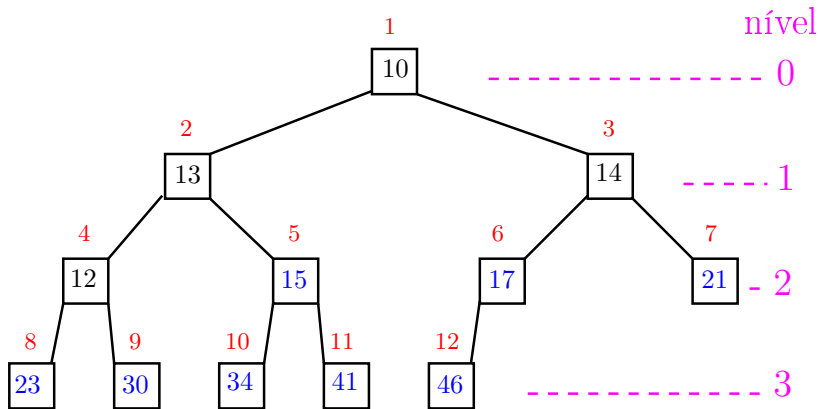
1	2	3	4	5	6	7	8	9	10	11	12
15	13	14	12	10	17	21	23	30	34	41	46

Heapsort



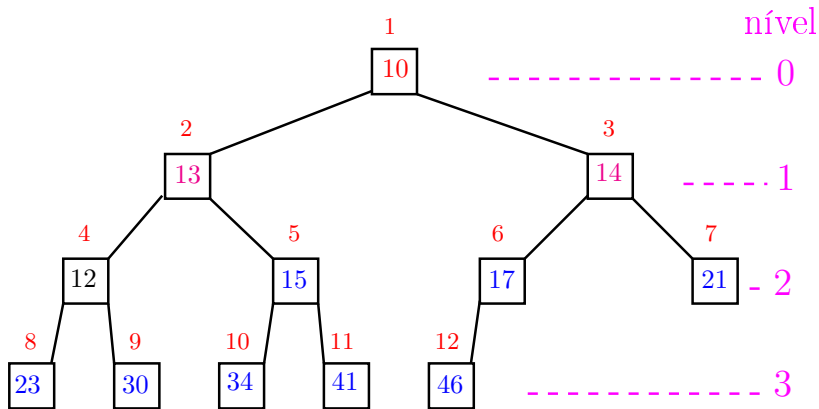
1	2	3	4	5	6	7	8	9	10	11	12
15	13	14	12	10	17	21	23	30	34	41	46

Heapsort



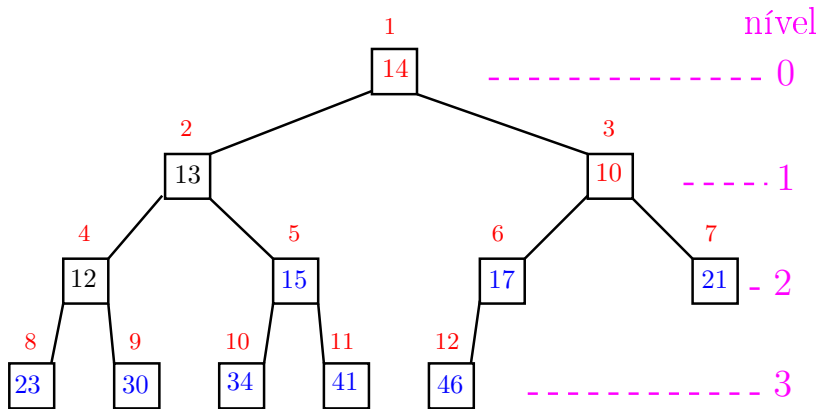
1	2	3	4	5	6	7	8	9	10	11	12
10	13	14	12	15	17	21	23	30	34	41	46

Heapsort



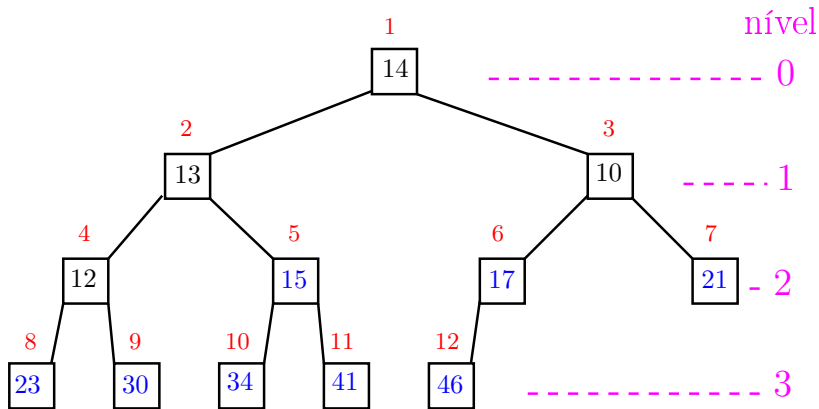
1	2	3	4	5	6	7	8	9	10	11	12
10	13	14	12	15	17	21	23	30	34	41	46

Heapsort



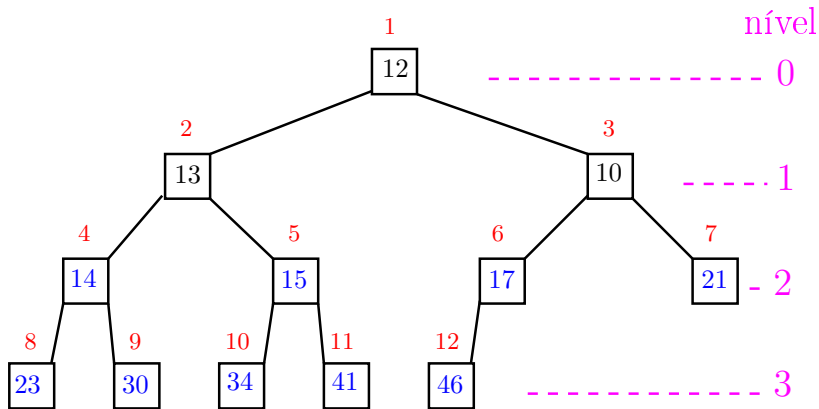
1	2	3	4	5	6	7	8	9	10	11	12
14	13	10	12	15	17	21	23	30	34	41	46

Heapsort



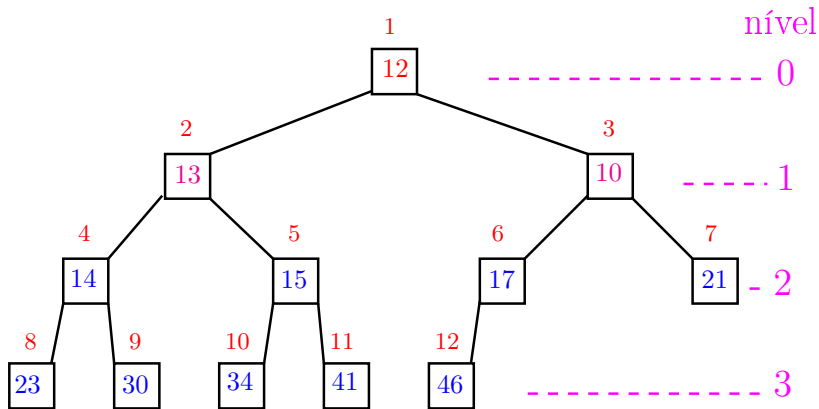
1	2	3	4	5	6	7	8	9	10	11	12
14	13	10	12	15	17	21	23	30	34	41	46

Heapsort



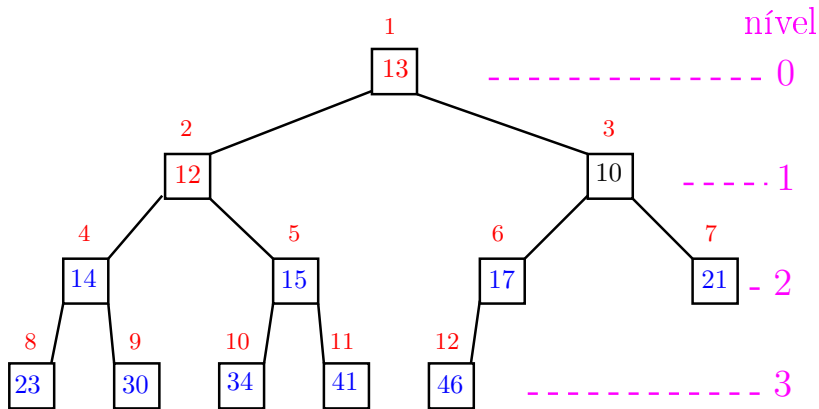
1	2	3	4	5	6	7	8	9	10	11	12
12	13	10	14	15	17	21	23	30	34	41	46

Heapsort



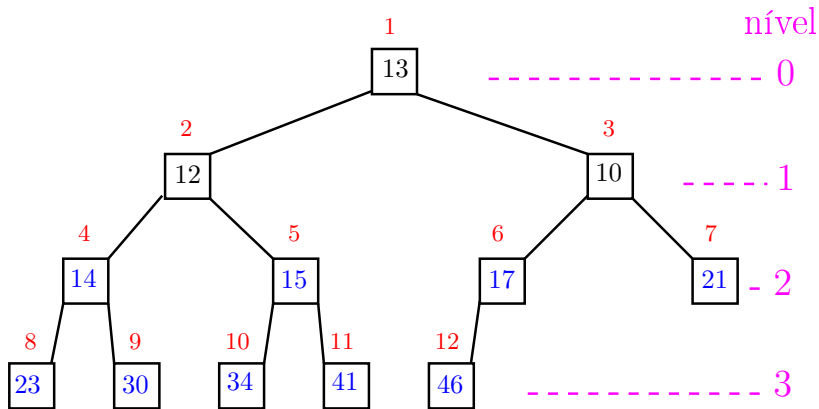
1	2	3	4	5	6	7	8	9	10	11	12
12	13	10	14	15	17	21	23	30	34	41	46

Heapsort



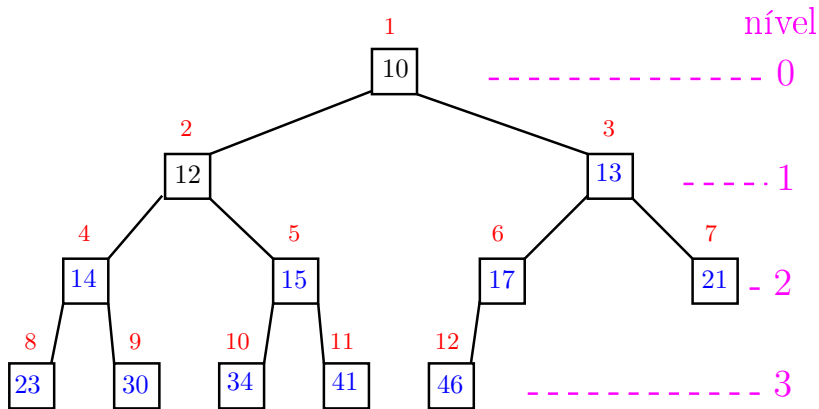
1	2	3	4	5	6	7	8	9	10	11	12
13	12	10	14	15	17	21	23	30	34	41	46

Heapsort



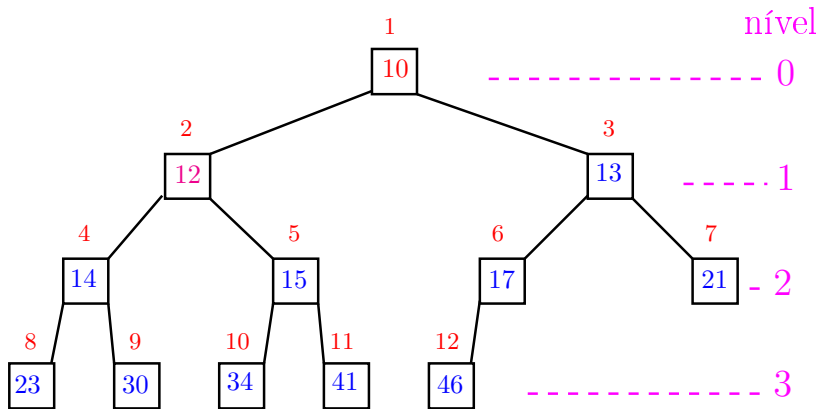
1	2	3	4	5	6	7	8	9	10	11	12
13	12	10	14	15	17	21	23	30	34	41	46

Heapsort



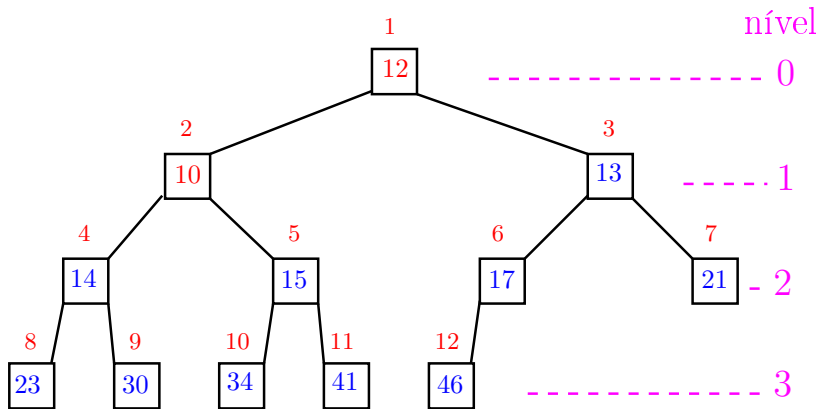
1	2	3	4	5	6	7	8	9	10	11	12
10	12	13	14	15	17	21	23	30	34	41	46

Heapsort



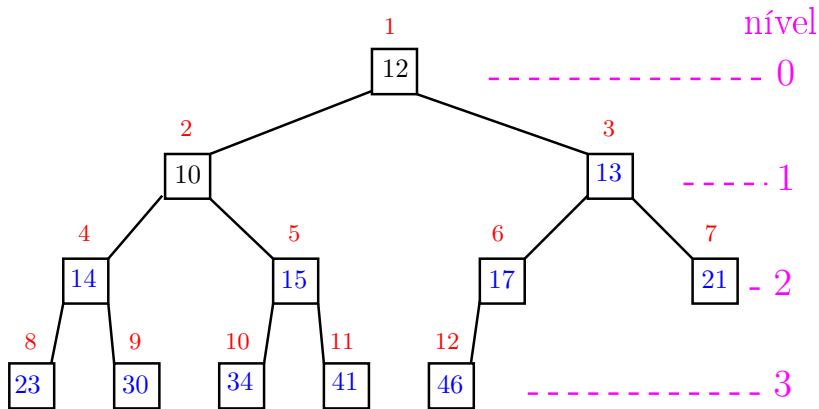
1	2	3	4	5	6	7	8	9	10	11	12
10	12	13	14	15	17	21	23	30	34	41	46

Heapsort



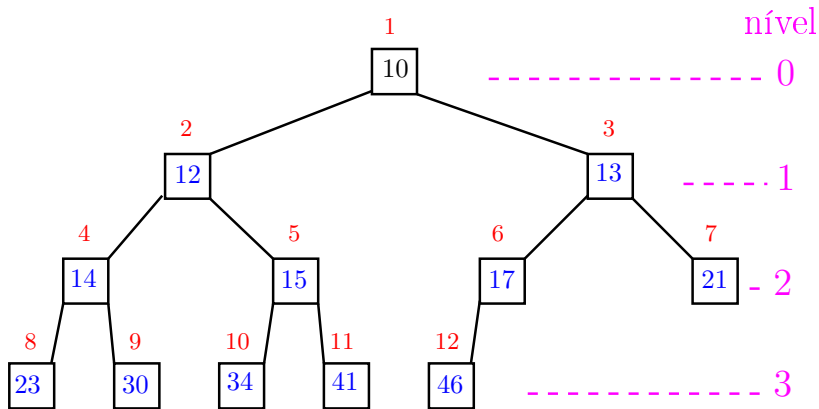
1	2	3	4	5	6	7	8	9	10	11	12
12	10	13	14	15	17	21	23	30	34	41	46

Heapsort



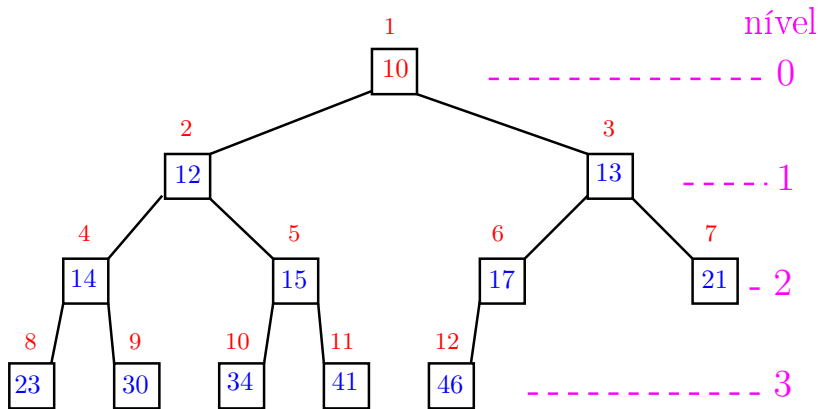
1	2	3	4	5	6	7	8	9	10	11	12
12	10	13	14	15	17	21	23	30	34	41	46

Heapsort



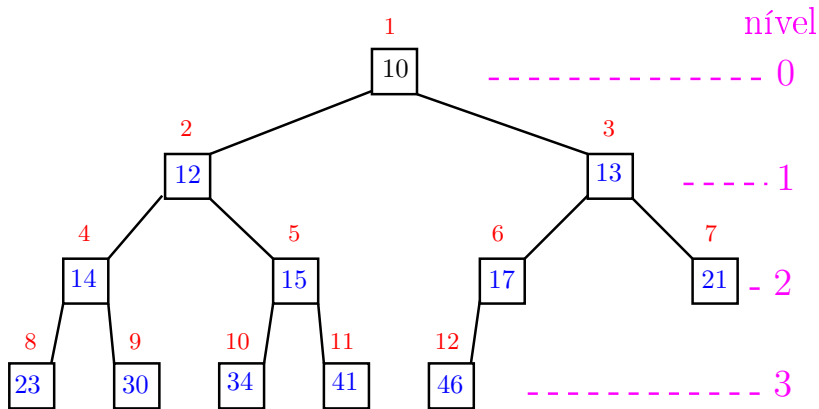
1	2	3	4	5	6	7	8	9	10	11	12
10	12	13	14	15	17	21	23	30	34	41	46

Heapsort



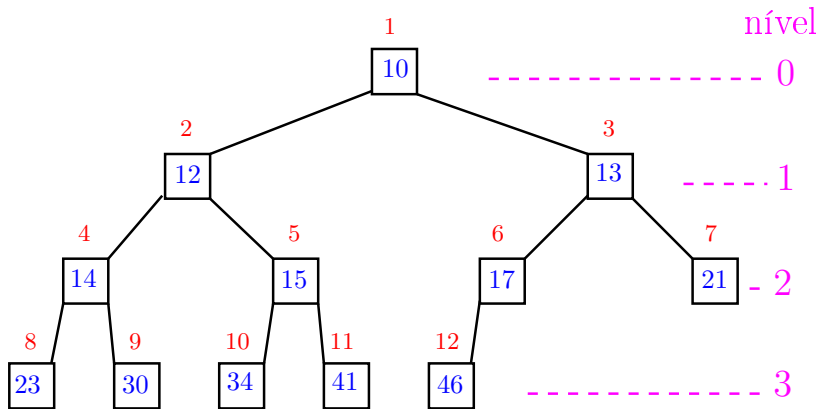
1	2	3	4	5	6	7	8	9	10	11	12
10	12	13	14	15	17	21	23	30	34	41	46

Heapsort



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

Heapsort



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

Função heap_sort

Algoritmo rearranja $v[1 : n]$ em ordem crescente

```
def heap_sort(v):
0     n = len(v)
      # pre-processamento
1     for i in range((n-1)//2, 0, -1):
2         peneire(i,n,v)

3     for i in range(n-1, 1, -1): #C#
4         v[i], v[1] = v[1], v[i]
5         peneire(1,i,v)
```

Função heap_sort

Relações invariantes: Em #C# vale que:

- (i0) $v[i+1 : n]$ é crescente;
- (i1) $v[1 : i+1] \leq v[i+1]$;
- (i2) $v[1 : i+1]$ é um max-heap.

1				i						n
50	44	10	38	20	50	55	60	75	85	99

Consumo de tempo

linha	consumo de tempo das execuções da linha	
1-2	$\approx n \lg n$	$= O(n \lg n)$
3	$\approx n$	$= O(n)$
4	$\approx n$	$= O(n)$
5	$\approx n \lg n$	$= O(n \lg n)$
total	$= 2n \lg n + 2n$	$= O(n \lg n)$

Conclusão

O consumo de tempo da função `heap_sort` é proporcional a $n \lg n$.

O consumo de tempo da função `heap_sort` é $O(n \lg n)$.

Mais análise experimental

Algoritmos implementados:

mergeR `merge_sort` recursivo.

mergeI `merge_sort` iterativo.

quick `quick_sort` recursivo.

heap `heap_sort`.

Mais análise experimental

A **plataforma utilizada** nos experimentos foi um computador rodando Ubuntu GNU/Linux 4.4.0-40

Python: Python 3.5.2.

Computador:

```
model name:  AMD FX(tm)-4300 Quad-Core Processor
cpu MHz :    3800.000
cache size:  2048 KB
MemTotal :   4095920 kB
```

Aleatório: média de 10

n	mergeR	mergeI	quick	heap
512	0.00	0.00	0.00	0.00
1024	0.01	0.01	0.00	0.01
2048	0.01	0.01	0.01	0.01
4096	0.03	0.02	0.02	0.03
8192	0.05	0.05	0.04	0.06
16384	0.11	0.11	0.09	0.13
32768	0.24	0.24	0.18	0.27
65536	0.50	0.50	0.40	0.60
131072	1.07	1.06	0.88	1.34
262144	2.27	2.27	1.87	2.96
524288	4.85	4.85	4.08	6.45
1048576	10.31	10.43	8.63	14.05

Tempos em segundos.

Decrescente

n	mergeR	mergeI	quick	heap
512	0.00	0.00	0.03	0.00
1024	0.01	0.00	*	0.01
2048	0.01	0.01	*	0.01
4096	0.02	0.02	*	0.02
8192	0.05	0.05	*	0.06
16384	0.11	0.10	*	0.12
32768	0.22	0.21	*	0.26
65536	0.48	0.46	*	0.56
131072	1.00	0.97	*	1.20
262144	2.10	2.05	*	2.57
524288	4.36	4.26	*	5.51

Tempos em segundos.

Para $n=1024$ `quick_sort` dá `RecursionError:`
`maximum recursion depth exceeded in`
`comparison`

Crescente

n	mergeR	mergeI	quick	heap
512	0.00	0.00	0.04	0.00
1024	0.01	0.00	*	0.01
2048	0.01	0.01	*	0.01
4096	0.02	0.02	*	0.03
8192	0.05	0.05	*	0.06
16384	0.11	0.10	*	0.13
32768	0.23	0.22	*	0.30
65536	0.49	0.47	*	0.62
131072	1.07	1.02	*	1.37
262144	2.17	2.09	*	2.77
524288	4.54	4.52	*	6.07

Tempos em segundos.

Para $n=1024$ `quick_sort` dá `RecursionError:`
`maximum recursion depth exceeded in`
`comparison`

Resumo

função	consumo de tempo	observação
bubble	$O(n^2)$	todos os casos
insercao	$O(n^2)$ $O(n)$	pior caso melhor caso
insercao_binaria	$O(n^2)$ $O(n \lg n)$	pior caso melhor caso
selecao	$O(n^2)$	todos os casos
merge_sort	$O(n \lg n)$	todos os casos
quick_sort	$O(n^2)$ $O(n \lg n)$	pior caso melhor caso
heap_sort	$O(n \lg n)$	todos os casos

Animação de algoritmos de ordenação

Criados por **Nicholas André Pinho de Oliveira**:
<http://nicholasandre.com.br/sorting/>

Criados na **Sapientia University** (Romania):
<https://www.youtube.com/channel/UCIqiLefbVHs0AXDaxQJH7X>

Fila de prioridades

PF 10

<http://www.ime.usp.br/~pf/algoritmos/aulas/hpsrt.html>

Fila de prioridades

Uma **fila de prioridades** (= *priority queue*) é qualquer tipo-de-dados abstrato dotado de duas operações:

- ▶ **retirar** um elemento de valor máximo;
- ▶ **acrescentar** um novo elemento.

É fácil implementar a fila de prioridades utilizando um **max-heap**.

A definição acima é a de uma fila de prioridades "de máximo". Não é difícil adaptar essa definição para filas de prioridades "de mínimo".

Função `remove_heap`

Remoção de um elemento x de um `max-heap` $v[1 : n]$

```
def remove_heap(v):  
1   x = v.pop()  
2   peneire(1, len(v), v)  
3   return x
```

Função insira_heap

Inseção de um elemento x em um **max-heap** $v[1 : n]$

```
def insira_heap(x, v):
0   n = f = len(v) # f = filho
1   v.append(x)
2   p = f // 2 # p = pai
3   while f > 1 and v[p] < v[f]: #D#
        # troca o filho com o pai
4       v[p], v[f] = v[f], v[p]
        # pai no papel de filho
5       f = p
6       p = f // 2
```

Função insira_heap

Relações invariantes: Em */*D*/* vale que:

- (i0) $v[1 : n]$ é uma permutação da lista original
- (i1) $v[i//2] \geq v[i]$ para todo $i = 2, \dots, n-1$ diferente de f .

1				f						n
83	75	25	68	99	15	10	60	57	65	79

Conclusões

O consumo de tempo da função `remove_heap` é proporcional a $\lg n$, onde n é o número de elementos no `max-heap`.

O consumo de tempo da função `insira_heap` é proporcional a $\lg n$, onde n é o número de elementos no `max-heap`.