Melhores momentos

AULA 20

Resumo

função	consumo de tempo	observação
bubble	$O(n^2)$	todos os casos
insercao	$O(n^2)$ $O(n)$	pior caso melhor caso
insercaoBinaria	$O(n^2)$ $O(n \lg n)$	pior caso melhor caso
selecao	$O(n^2)$	todos os casos
mergeSort	$O(n \lg n)$	todos os casos
quickSort	$O(n^2)$ $O(n \lg n)$	pior caso melhor caso

Divisão e conquista

Algoritmos por divisão-e-conquista têm três passos em cada nível da recursão:

Dividir: o problema é dividido em subproblemas de tamanho menor;

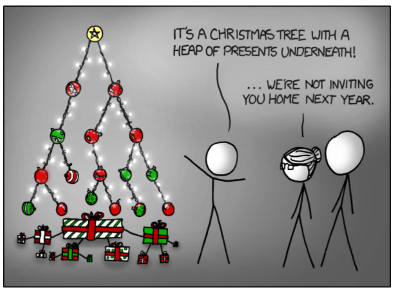
Conquistar: os subproblemas são resolvidos recursivamente e subproblemas "pequenos" são resolvidos diretamente;

Combinar: as soluções dos subproblemas são combinadas para obter uma solução do problema original.

Exemplo: ordenação por intercalação (`mergeSort`).

AULA 21

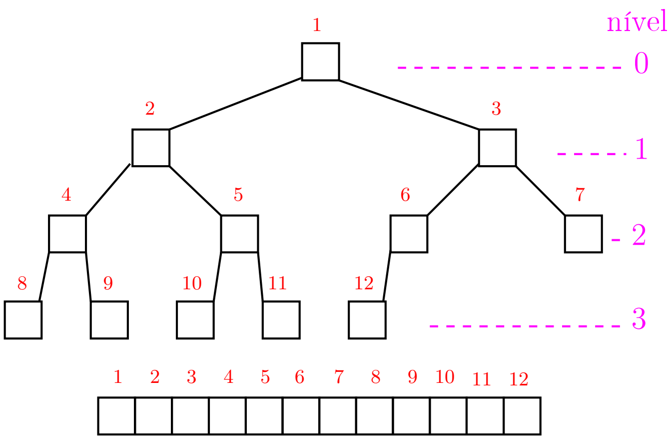
Árvores em vetores e heaps



Fonte: http://xkcd.com/835/

PF 10
http://www.ime.usp.br/~pf/algoritmos/aulas/hpsrt.html

Representação de árvores em vetores



Pais e filhos

$v[1\,..\,m]$ é um vetor representando uma árvore.

Diremos que para qualquer **índice** ou **nó** i,

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

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

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

Raiz e folhas

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

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

Todo nó i é raiz da subárvore formada por

$$v[i, 2i, 2i{+}1, 4i, 4i{+}1, 4i{+}2, 4i{+}3, 8i, \ldots, 8i{+}7, \ldots]$$

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 $\lfloor \lg i \rfloor$.

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

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

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

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

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

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

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

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

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

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

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

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

$$\langle \mathtt{filho}(i), \mathtt{filho}(\mathtt{filho}(i)), \mathtt{filho}(\mathtt{filho}(\mathtt{filho}(i))), \ldots$$

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

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

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

$$\langle \mathtt{filho}(i), \mathtt{filho}(\mathtt{filho}(i)), \mathtt{filho}(\mathtt{filho}(\mathtt{filho}(i))), \ldots$$

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

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

A altura de um nó i é $\lfloor \lg(m/i) \rfloor$ $(\ldots)$.

Resumão

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:	$\lfloor \lg(m/i) \rfloor$ $(\ldots)$
altura de uma folha:	0
total de nós de altura h	$\leq \lceil m/2^{h+1} \rceil$ $(\ldots)$

Heaps

Um vetor $v[1 \,..\, m]$ é um `max-heap` se

$$v[i/2] \geq v[i]$$

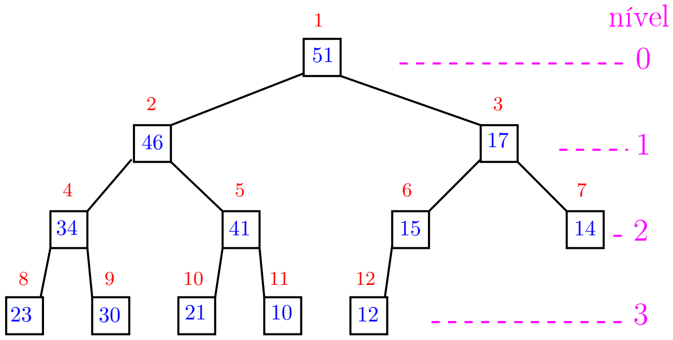
para todo $i = 2, 3, \ldots, m$.

De uma forma mais geral, $v[j \,..\, m]$ é um `max-heap` se

$$v[i/2] \geq v[i]$$

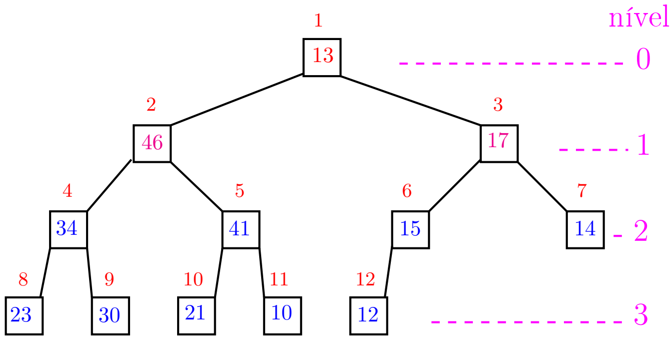
para todo
$i = 2j, 2j + 1, 4j, \ldots, 4j + 3, 8j, \ldots, 8j + 7, \ldots$.
Neste caso também diremos que a subárvore com raiz j é um `max-heap`.

max-heap



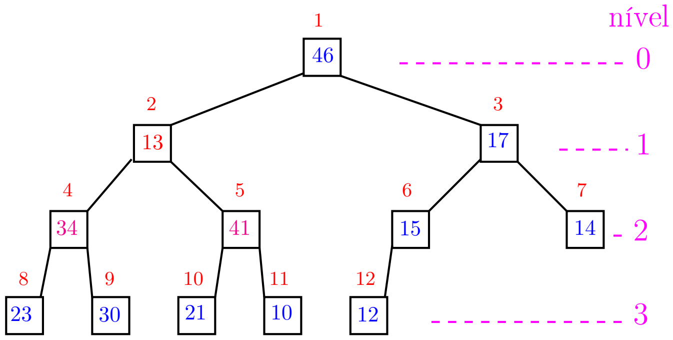
1	2	3	4	5	6	7	8	9	10	11	12
51	46	17	34	41	15	14	23	30	21	10	12

Função básica de manipulação de max-heap



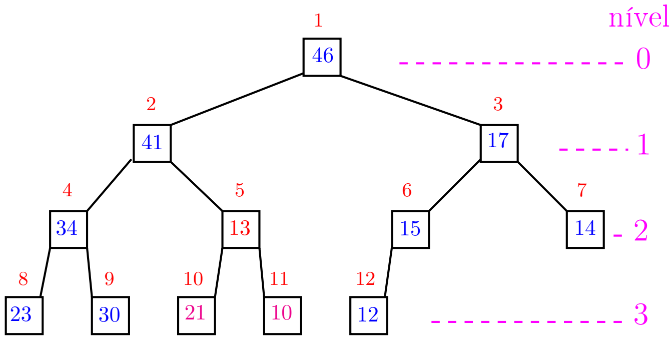
1	2	3	4	5	6	7	8	9	10	11	12
13	46	17	34	41	15	14	23	30	21	10	12

Função básica de manipulação de max-heap



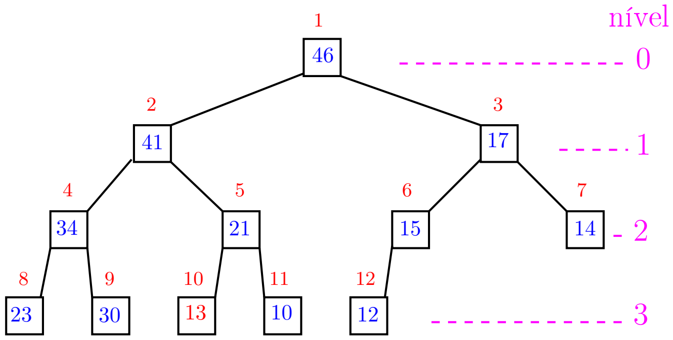
1	2	3	4	5	6	7	8	9	10	11	12
46	13	17	34	41	15	14	23	30	21	10	12

Função básica de manipulação de max-heap



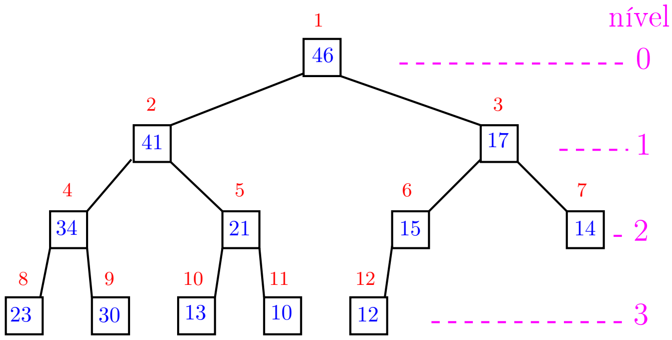
1	2	3	4	5	6	7	8	9	10	11	12
46	41	17	34	13	15	14	23	30	21	10	12

Função básica de 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

Função básica de 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

Função peneira

O coração de qualquer algoritmo que manipule um max-heap é uma função que recebe um vetor arbitrário v[1 . . m] e um índice i e faz v[i] "descer" para sua posição correta.

Função peneira

Rearranja o vetor v[1 . . m] de modo que o "subvetor" cuja raiz é i seja um max-heap.

```
void peneira (int i, int m, int v[]) {
1   int f = 2*i, x;
2   while (f <= m) {
3       if (f < m && v[f] < v[f+1]) f++;
4       if (v[i] >= v[f]) break;
5       x = v[i]; v[i] = v[f]; v[f] = x;
6       i = f; f = 2*i;
    }
}
```

Função peneira

Supõe que os "subvetores"cujas raízes são filhos de i já são max-heap.

```
void peneira (int i, int m, int v[]) {
1   int f = 2*i, x;
2   while (f <= m) {
3       if (f < m && v[f] < v[f+1]) f++;
4       if (v[i] >= v[f]) break;
5       x = v[i]; v[i] = v[f]; v[f] = x;
6       i = f; f = 2*i;
    }
}
```

Função peneira

A seguinte implementação é um pouco melhor pois em vez de trocas faz apenas deslocamentos (linha 5).

```
void peneira (int i, int m, int v[]) {
1   int f = 2*i, x = v[i];
2   while (f <= m) {
3       if (f < m && v[f] < v[f+1]) f++;
4       if (x >= v[f]) break;
5       v[i] = v[f];
6       i = f; f = 2*i;
    }
7   v[i] = x;
}
```

Consumo de tempo

linha	todas as execuções da linha	
1	$=$	1
2	$\leq$	$1 + \lg m$
3	$\leq$	$\lg m$
4	$\leq$	$\lg m$
5	$\leq$	$\lg m$
6	$\leq$	$\lg m$
7	$=$	1
total	$\leq$	$3 + 5 \lg m = O(\lg m)$

Conclusão

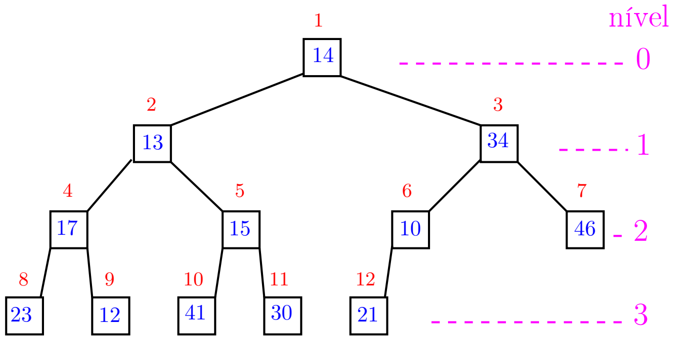
O consumo de tempo da função peneira é proporcional a $\lg m$.

O consumo de tempo da função peneira é $O(\lg m)$.

Verdade seja dita . . . (. . .)

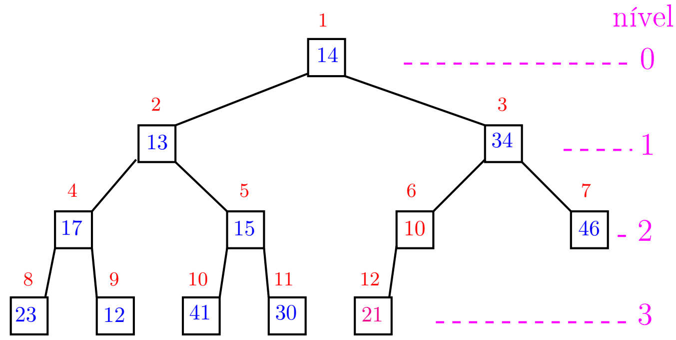
O consumo de tempo da função peneira é proporcional a $O(\lg m/i)$.

Construção de um max-heap



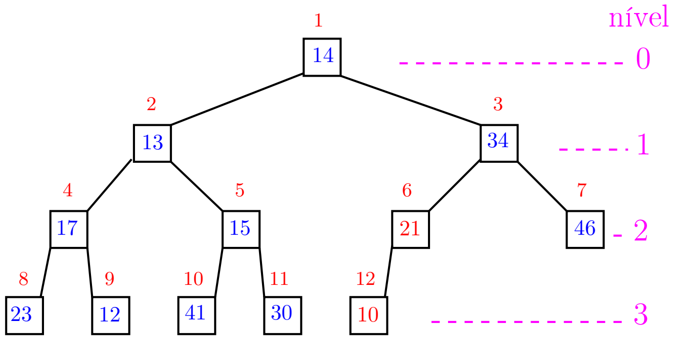
1	2	3	4	5	6	7	8	9	10	11	12
14	13	34	17	15	10	46	23	12	41	30	21

Construção de um max-heap



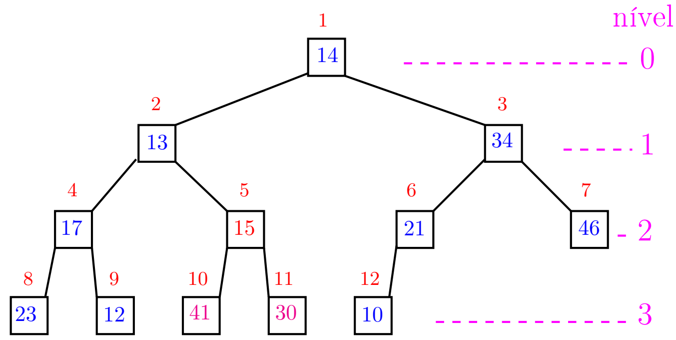
1	2	3	4	5	6	7	8	9	10	11	12
14	13	34	17	15	10	46	23	12	41	30	21

Construção de um max-heap



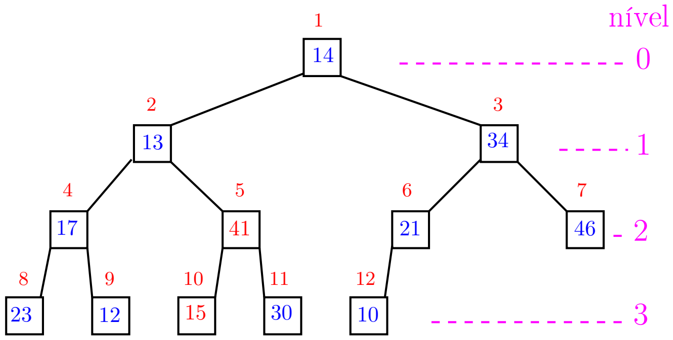
1	2	3	4	5	6	7	8	9	10	11	12
14	13	34	17	15	21	46	23	12	41	30	10

Construção de um max-heap



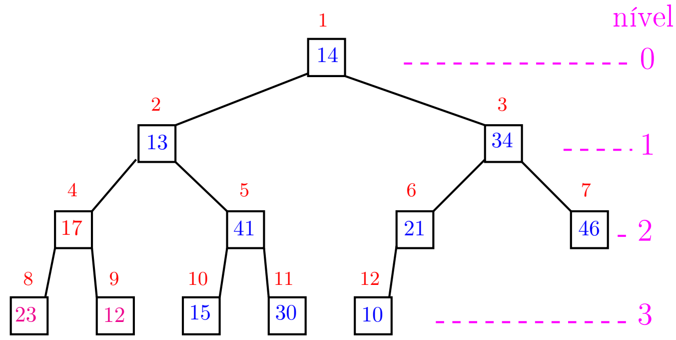
1	2	3	4	5	6	7	8	9	10	11	12
14	13	34	17	15	21	46	23	12	41	30	10

Construção de um max-heap



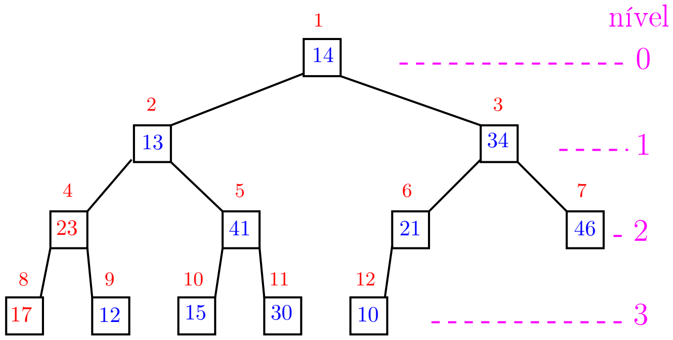
1	2	3	4	5	6	7	8	9	10	11	12
14	13	34	17	41	21	46	23	12	15	30	10

Construção de um max-heap



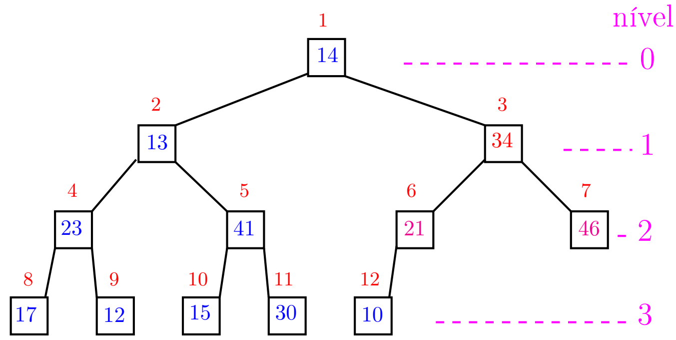
1	2	3	4	5	6	7	8	9	10	11	12
14	13	34	17	41	21	46	23	12	15	30	10

Construção de um max-heap



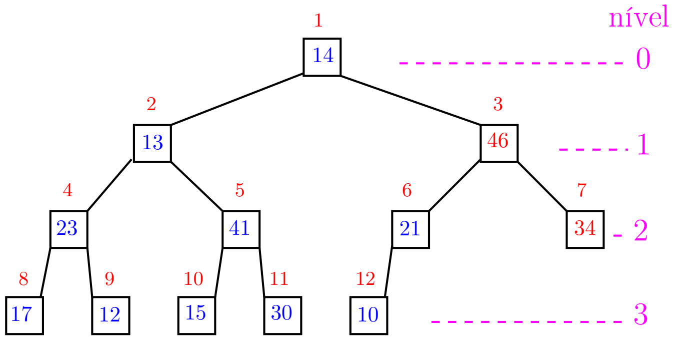
1	2	3	4	5	6	7	8	9	10	11	12
14	13	34	23	41	21	46	17	12	15	30	10

Construção de um max-heap



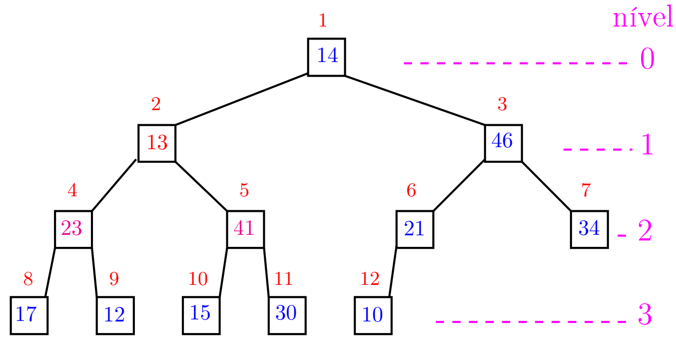
1	2	3	4	5	6	7	8	9	10	11	12
14	13	34	23	41	21	46	17	12	15	30	10

Construção de um max-heap



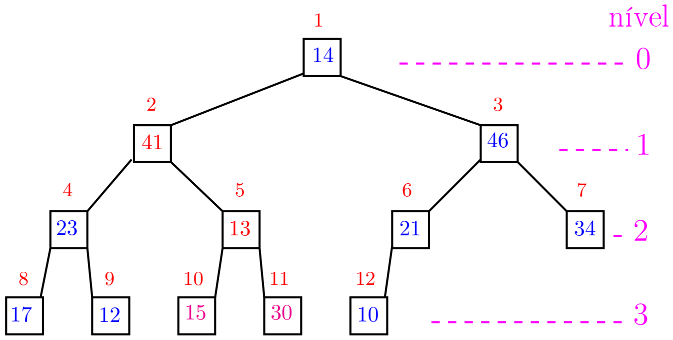
1	2	3	4	5	6	7	8	9	10	11	12
14	13	46	23	41	21	34	17	12	15	30	10

Construção de um max-heap



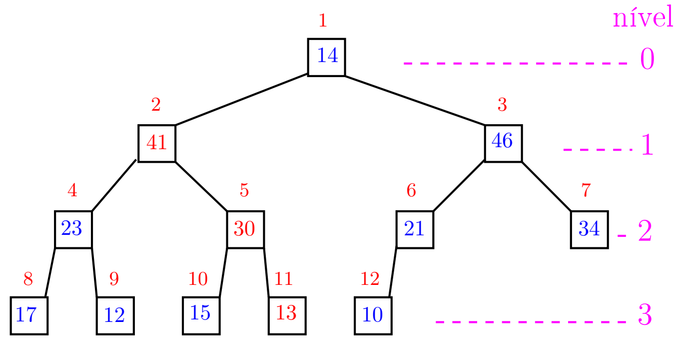
1	2	3	4	5	6	7	8	9	10	11	12
14	13	46	23	41	21	34	17	12	15	30	10

Construção de um max-heap



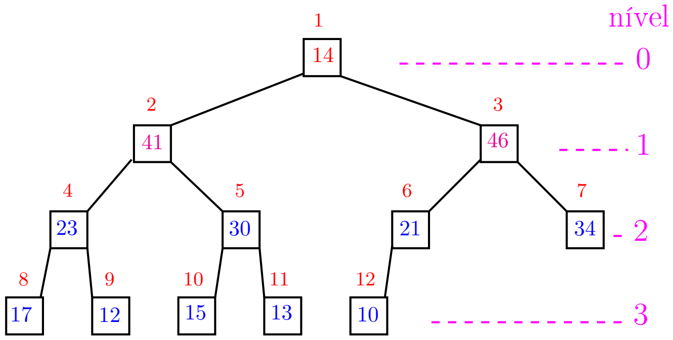
1	2	3	4	5	6	7	8	9	10	11	12
14	41	46	23	13	21	34	17	12	15	30	10

Construção de um max-heap



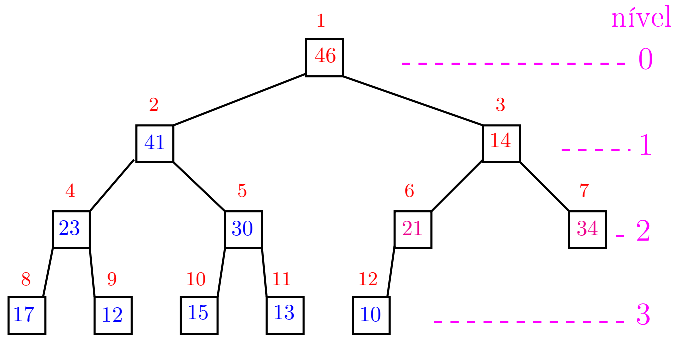
1	2	3	4	5	6	7	8	9	10	11	12
14	41	46	23	30	21	34	17	12	15	13	10

Construção de um max-heap



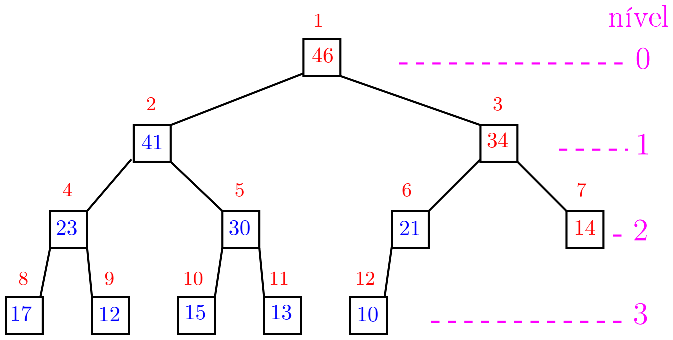
1	2	3	4	5	6	7	8	9	10	11	12
14	41	46	23	30	21	34	17	12	15	13	10

Construção de um max-heap



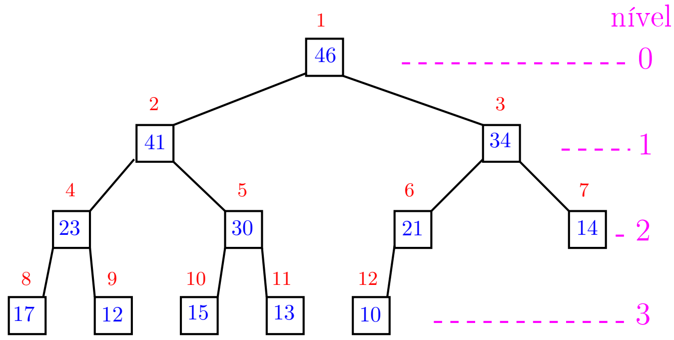
1	2	3	4	5	6	7	8	9	10	11	12
46	41	14	23	30	21	34	17	12	15	13	10

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 v[1 . . n] e rearranja v para que seja max-heap.

```
for (i = n/2; /*A*/ i >= 1; i--)
    peneira(i, n, v);
```

Relação invariante:

(i0) em /*A*/ vale que, i+1, . . . , n são raízes de max-heaps.

Consumo de tempo

Análise grosseira: consumo de tempo é

$$\frac{n}{2} \times \lg n = O(n \lg n).$$

Verdade seja dita . . . (. . .)

Análise mais cuidadosa: consumo de tempo é $O(n)$.

Conclusão

> O consumo de tempo para construir um `max-heap` é $O(n \lg n)$.

Verdade seja dita . . . (. . .)

> O consumo de tempo para construir um `max-heap` é $O(n)$.

Ordenação: algoritmo Heapsort

PF 10
http://www.ime.usp.br/~pf/algoritmos/aulas/hpsrt.html

Ordenação

$v[1..n]$ é crescente se $v[1] \le \cdots \le v[n]$.

Problema: Rearranjar um vetor $v[1..n]$ 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 um vetor $v[1..n]$ 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

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

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

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

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

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

Função selecao

Algoritmo rearranja v[0 . . n−1] em ordem crescente

```
void selecao (int n, int v[])
{
   int i, j, max, x;
   for (i = n-1; /*B*/ i > 0; i--) {
      max = i;
      for (j = i-1; j >= 0; j--)
         if (v[j] > v[max]) max = j;
      x=v[i]; v[i]=v[max]; v[max]=x;
   }
}
```

Função selecao

Algoritmo rearranja v[1 . . n] em ordem crescente

```
void selecao (int n, int v[])
{
   int i, j, max, x;
   for (i = n; /*B*/ i > 1; i--) {
      max = i;
      for (j = i-1; j >= 1; j--)
         if (v[j] > v[max]) max = j;
      x=v[i]; v[i]=v[max]; v[max]=x;
   }
}
```

Função selecao

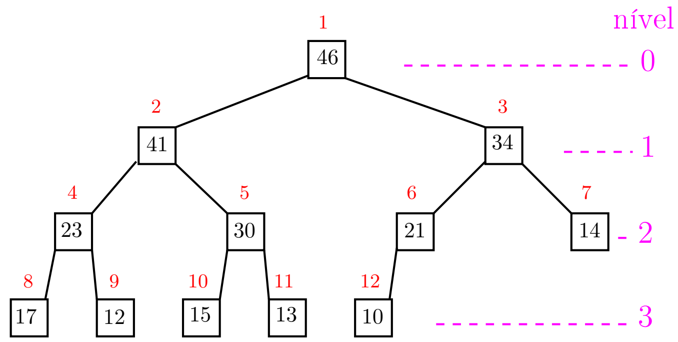
Relações invariantes: Em /*B*/ vale que:

(i0) v[i+1 . . n] é crescente;

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

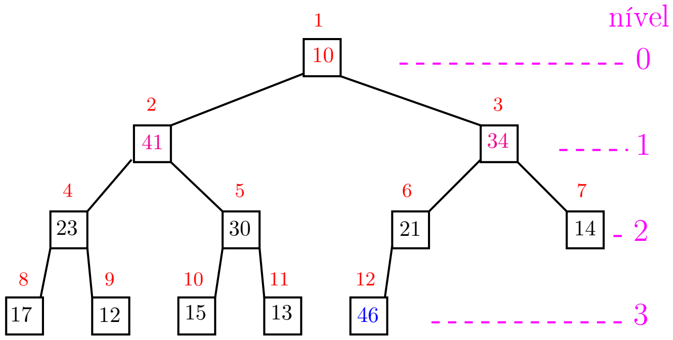
1			i							n
38	10	20	44	50	50	55	60	75	85	99

Heapsort



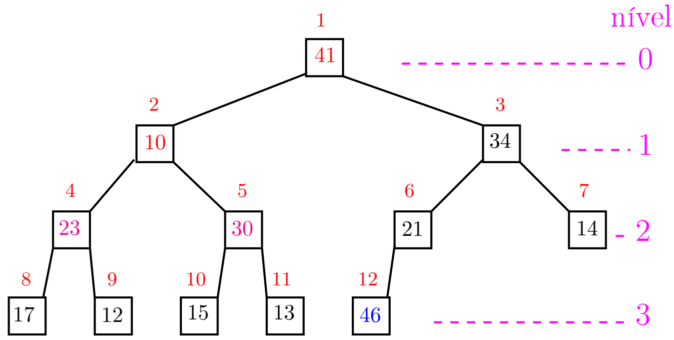
1	2	3	4	5	6	7	8	9	10	11	12
46	41	34	23	30	21	14	17	12	15	13	10

Heapsort



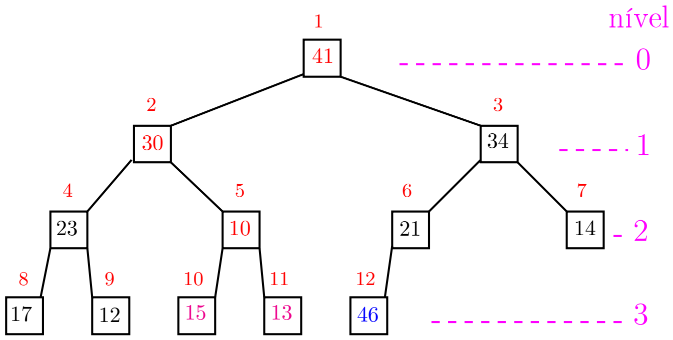
1	2	3	4	5	6	7	8	9	10	11	12
10	41	34	23	30	21	14	17	12	15	13	46

Heapsort



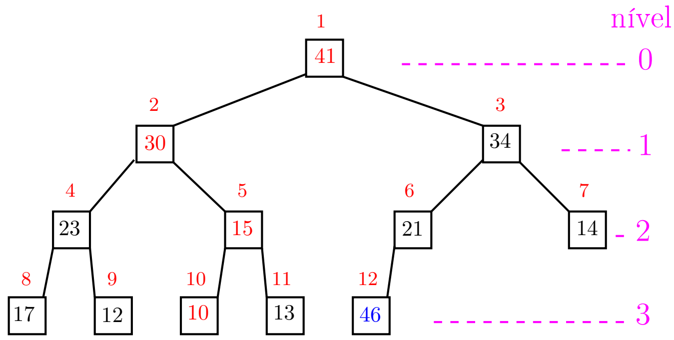
1	2	3	4	5	6	7	8	9	10	11	12
41	10	34	23	30	21	14	17	12	15	13	46

Heapsort



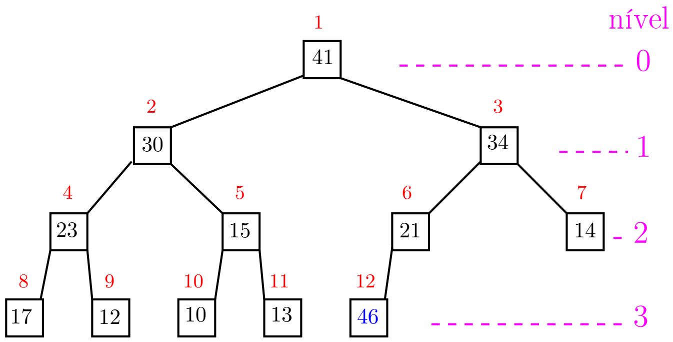
1	2	3	4	5	6	7	8	9	10	11	12
41	30	34	23	10	21	14	17	12	15	13	46

Heapsort



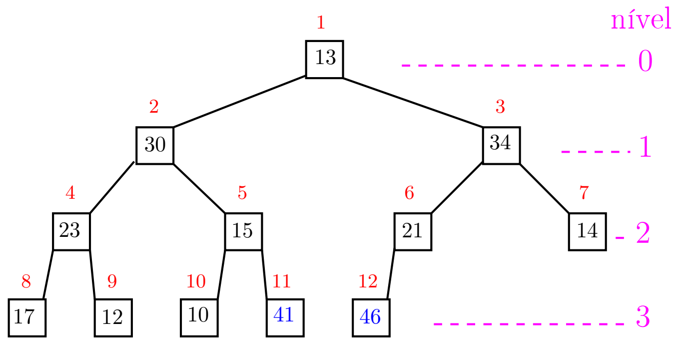
1	2	3	4	5	6	7	8	9	10	11	12
41	30	34	23	15	21	14	17	12	10	13	46

Heapsort



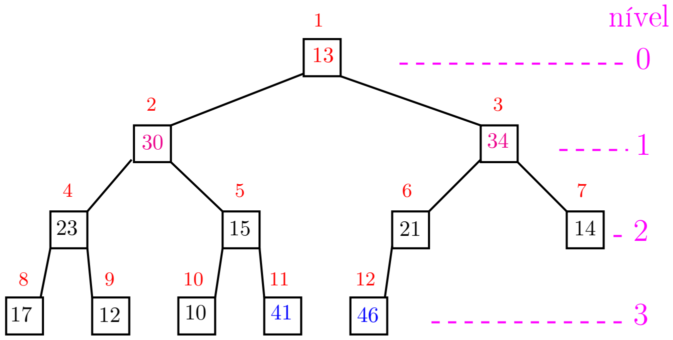
1	2	3	4	5	6	7	8	9	10	11	12
41	30	34	23	15	21	14	17	12	10	13	46

Heapsort



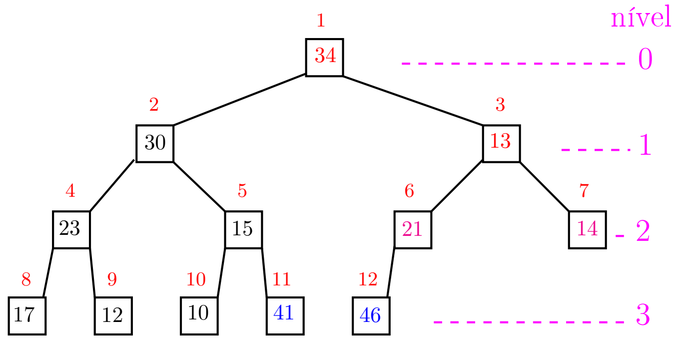
1	2	3	4	5	6	7	8	9	10	11	12
13	30	34	23	15	21	14	17	12	10	41	46

Heapsort



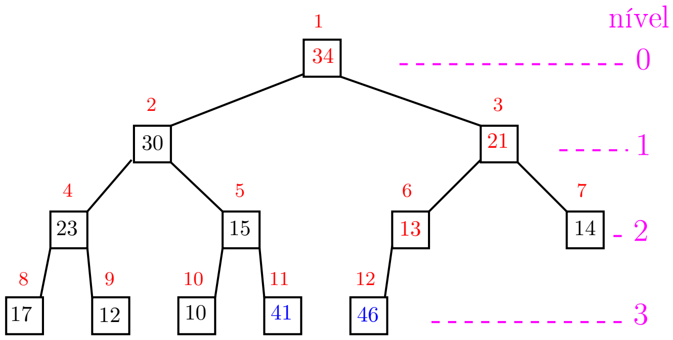
1	2	3	4	5	6	7	8	9	10	11	12
13	30	34	23	15	21	14	17	12	10	41	46

Heapsort



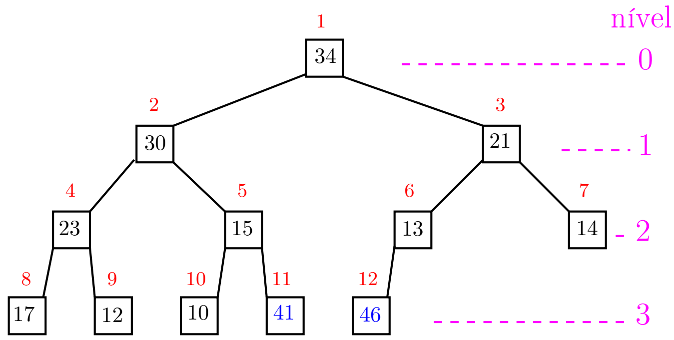
1	2	3	4	5	6	7	8	9	10	11	12
34	30	13	23	15	21	14	17	12	10	41	46

Heapsort



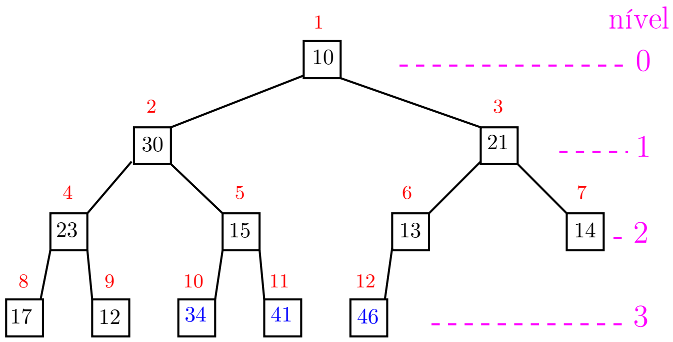
1	2	3	4	5	6	7	8	9	10	11	12
34	30	21	23	15	13	14	17	12	10	41	46

Heapsort



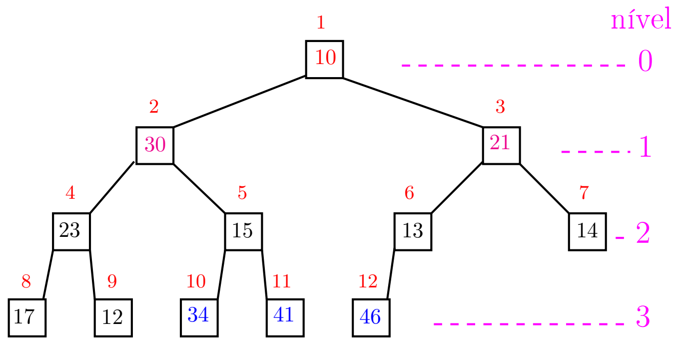
1	2	3	4	5	6	7	8	9	10	11	12
34	30	21	23	15	13	14	17	12	10	41	46

Heapsort



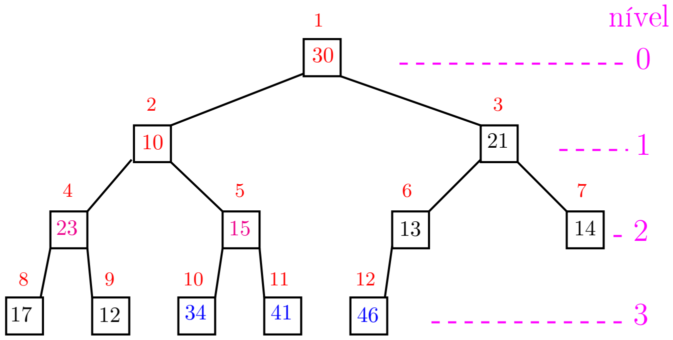
1	2	3	4	5	6	7	8	9	10	11	12
10	30	21	23	15	13	14	17	12	34	41	46

Heapsort



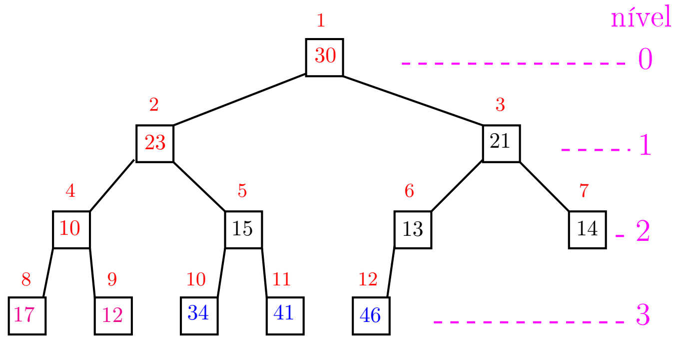
1	2	3	4	5	6	7	8	9	10	11	12
10	30	21	23	15	13	14	17	12	34	41	46

Heapsort



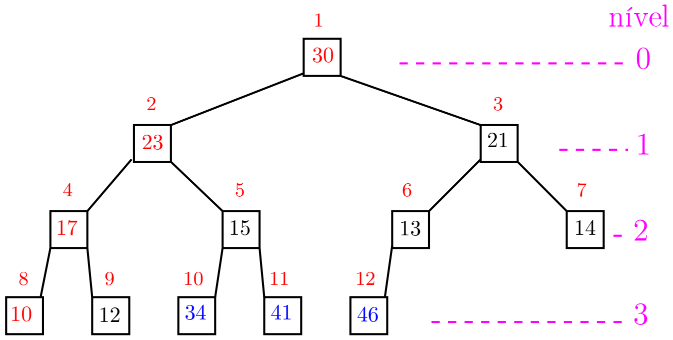
1	2	3	4	5	6	7	8	9	10	11	12
30	10	21	23	15	13	14	17	12	34	41	46

Heapsort



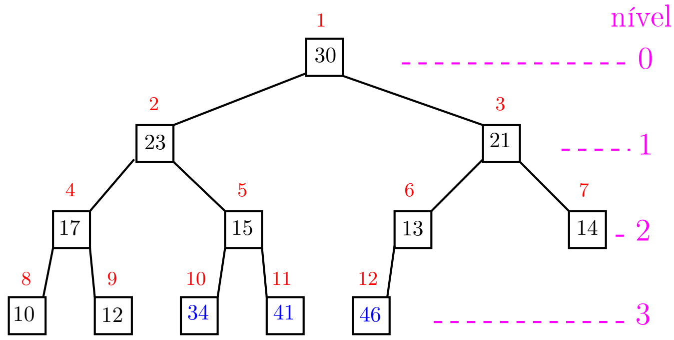
1	2	3	4	5	6	7	8	9	10	11	12
30	23	21	10	15	13	14	17	12	34	41	46

Heapsort



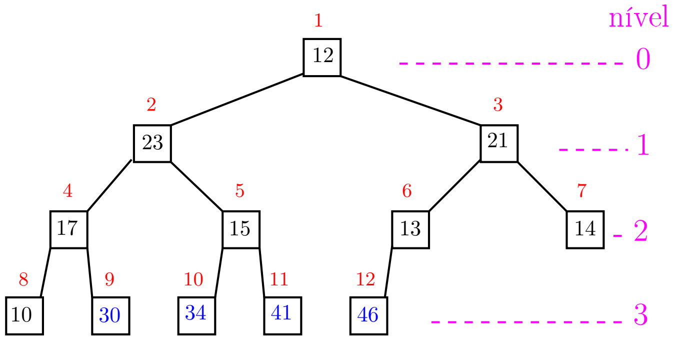
1	2	3	4	5	6	7	8	9	10	11	12
30	23	21	17	15	13	14	10	12	34	41	46

Heapsort



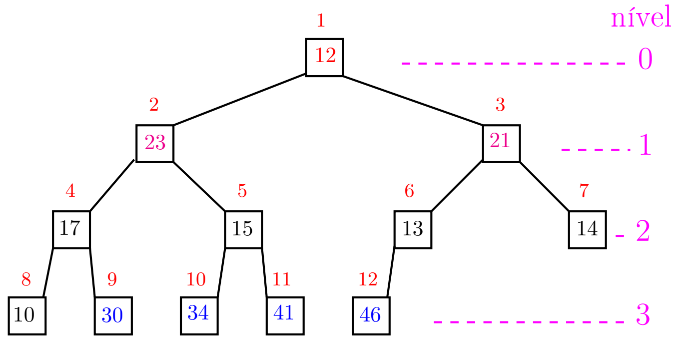
1	2	3	4	5	6	7	8	9	10	11	12
30	23	21	17	15	13	14	10	12	34	41	46

Heapsort



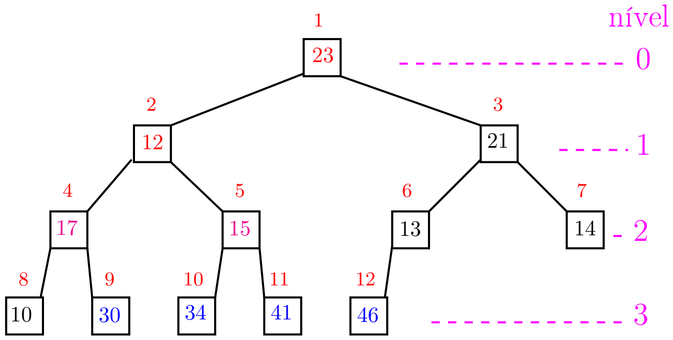
1	2	3	4	5	6	7	8	9	10	11	12
12	23	21	17	15	13	14	10	30	34	41	46

Heapsort



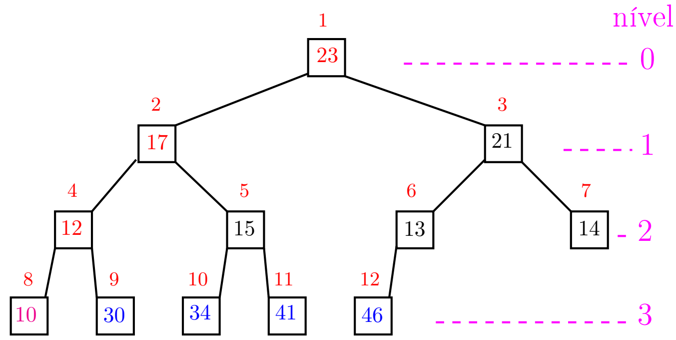
1	2	3	4	5	6	7	8	9	10	11	12
12	23	21	17	15	13	14	10	30	34	41	46

Heapsort



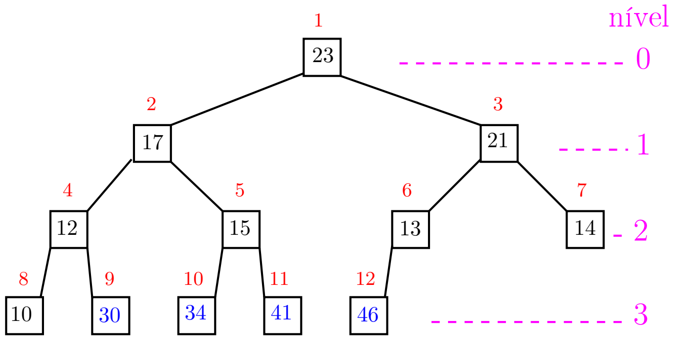
1	2	3	4	5	6	7	8	9	10	11	12
23	12	21	17	15	13	14	10	30	34	41	46

Heapsort



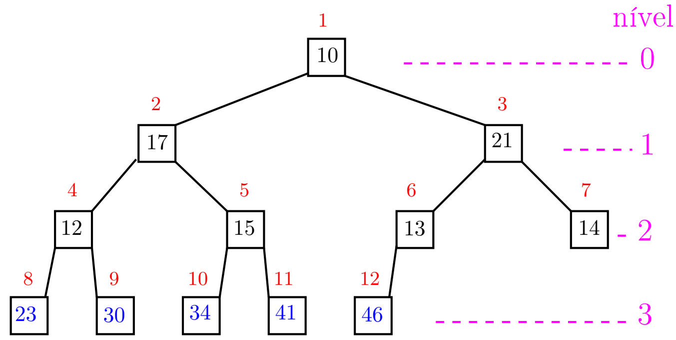
1	2	3	4	5	6	7	8	9	10	11	12
23	17	21	12	15	13	14	10	30	34	41	46

Heapsort



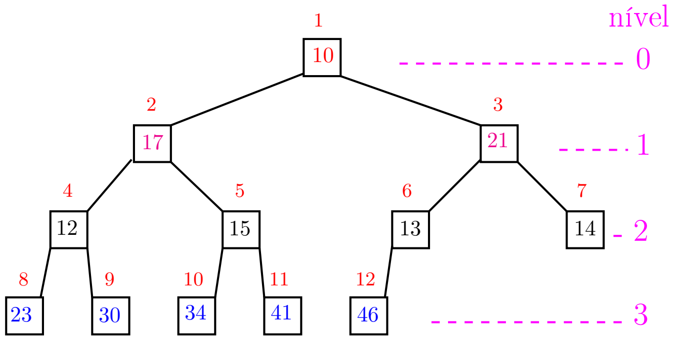
1	2	3	4	5	6	7	8	9	10	11	12
23	17	21	12	15	13	14	10	30	34	41	46

Heapsort



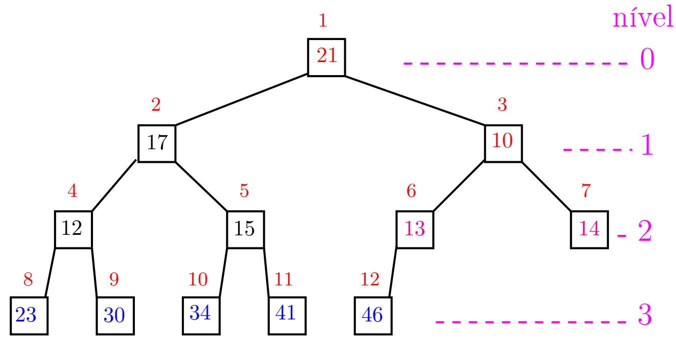
1	2	3	4	5	6	7	8	9	10	11	12
10	17	21	12	15	13	14	23	30	34	41	46

Heapsort



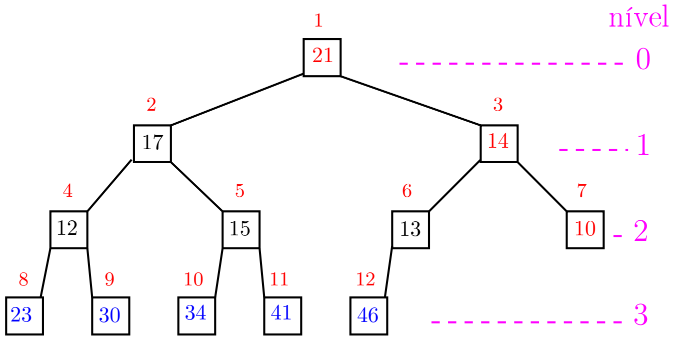
1	2	3	4	5	6	7	8	9	10	11	12
10	17	21	12	15	13	14	23	30	34	41	46

Heapsort



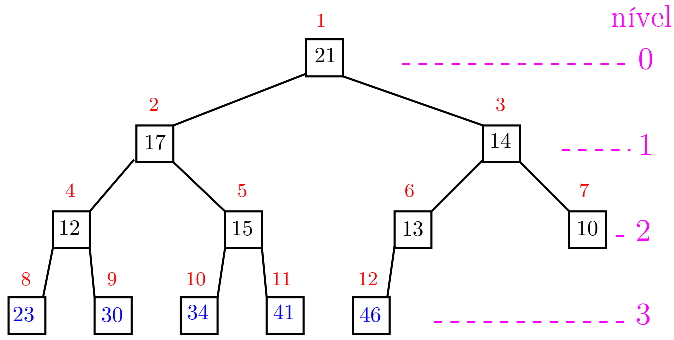
1	2	3	4	5	6	7	8	9	10	11	12
21	17	10	12	15	13	14	23	30	34	41	46

Heapsort



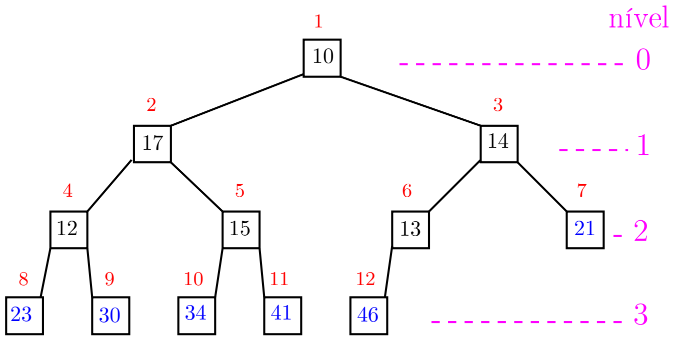
1	2	3	4	5	6	7	8	9	10	11	12
21	17	14	12	15	13	10	23	30	34	41	46

Heapsort



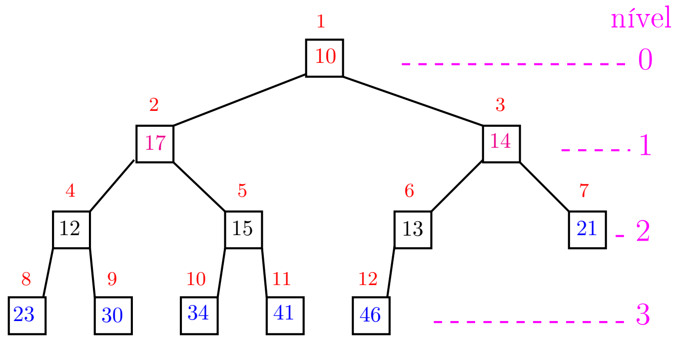
1	2	3	4	5	6	7	8	9	10	11	12
21	17	14	12	15	13	10	23	30	34	41	46

Heapsort



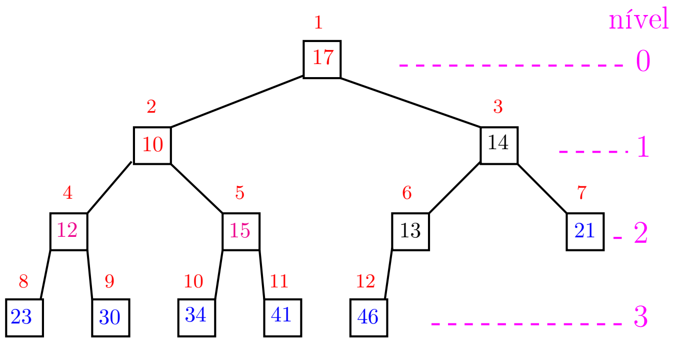
1	2	3	4	5	6	7	8	9	10	11	12
10	17	14	12	15	13	21	23	30	34	41	46

Heapsort



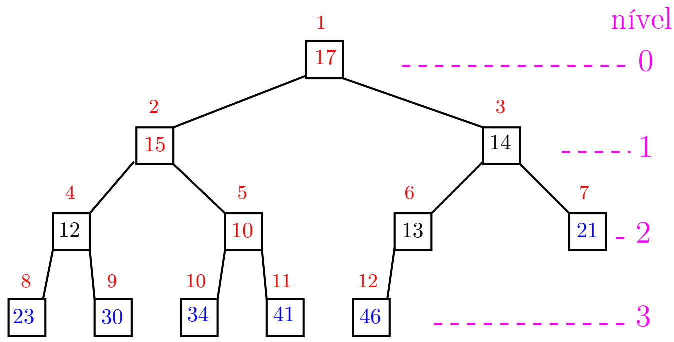
1	2	3	4	5	6	7	8	9	10	11	12
10	17	14	12	15	13	21	23	30	34	41	46

Heapsort



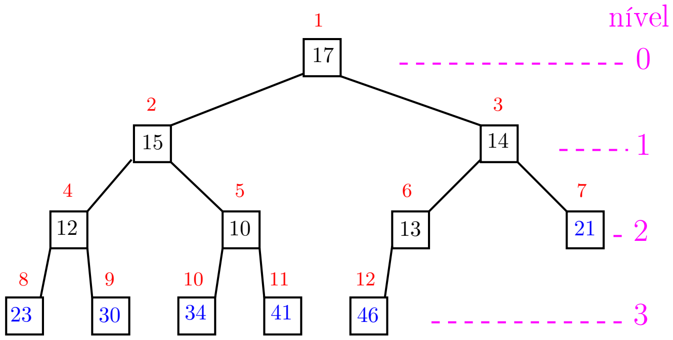
1	2	3	4	5	6	7	8	9	10	11	12
17	10	14	12	15	13	21	23	30	34	41	46

Heapsort



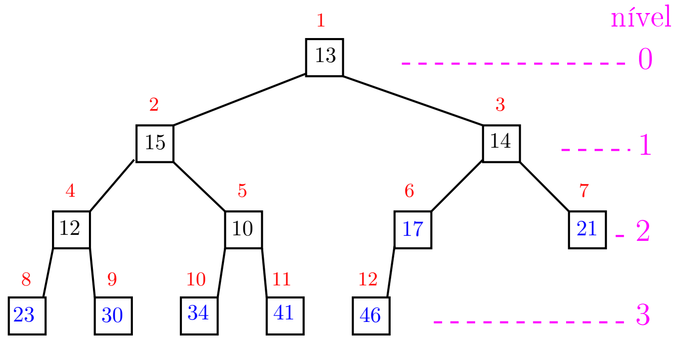
1	2	3	4	5	6	7	8	9	10	11	12
17	15	14	12	10	13	21	23	30	34	41	46

Heapsort



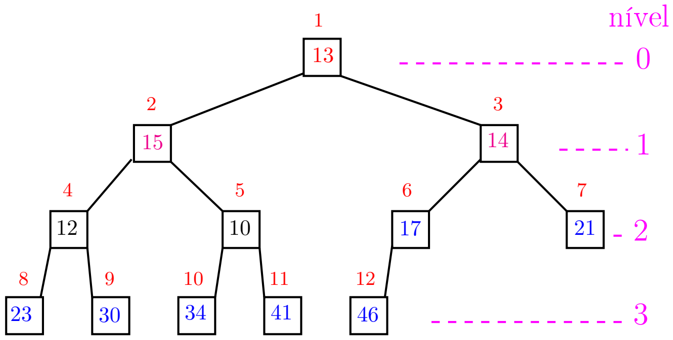
1	2	3	4	5	6	7	8	9	10	11	12
17	15	14	12	10	13	21	23	30	34	41	46

Heapsort



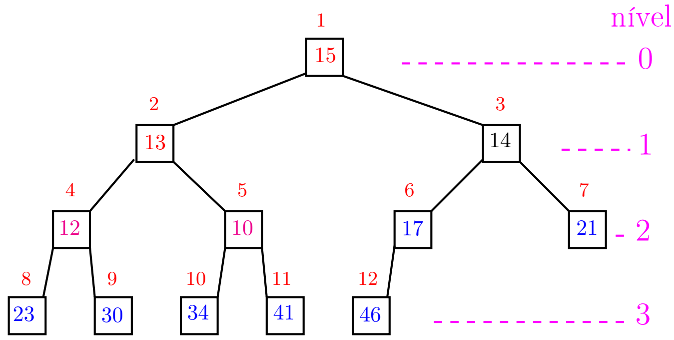
1	2	3	4	5	6	7	8	9	10	11	12
13	15	14	12	10	17	21	23	30	34	41	46

Heapsort



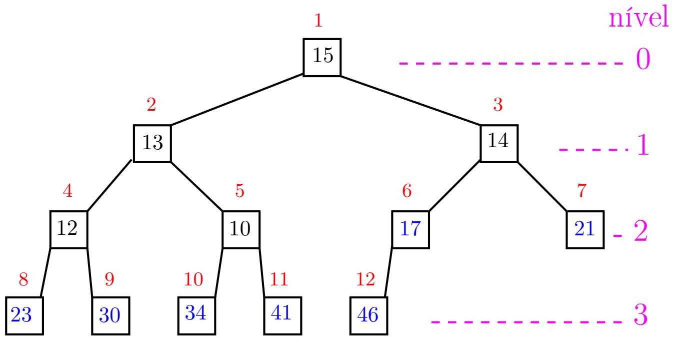
1	2	3	4	5	6	7	8	9	10	11	12
13	15	14	12	10	17	21	23	30	34	41	46

Heapsort



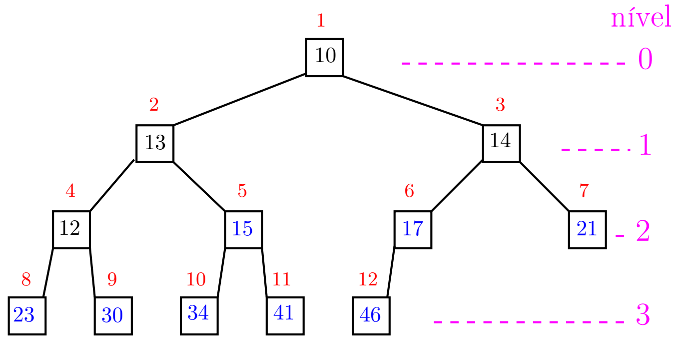
1	2	3	4	5	6	7	8	9	10	11	12
15	13	14	12	10	17	21	23	30	34	41	46

Heapsort



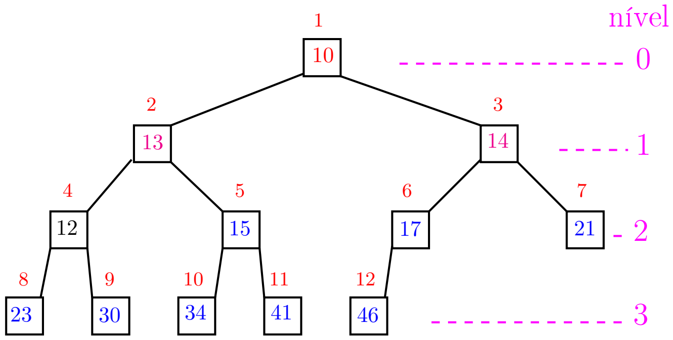
1	2	3	4	5	6	7	8	9	10	11	12
15	13	14	12	10	17	21	23	30	34	41	46

Heapsort



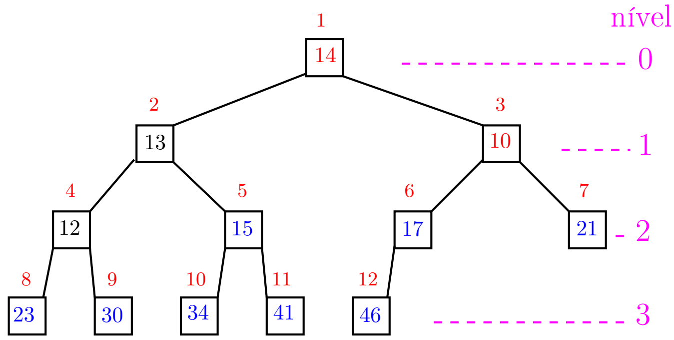
1	2	3	4	5	6	7	8	9	10	11	12
10	13	14	12	15	17	21	23	30	34	41	46

Heapsort



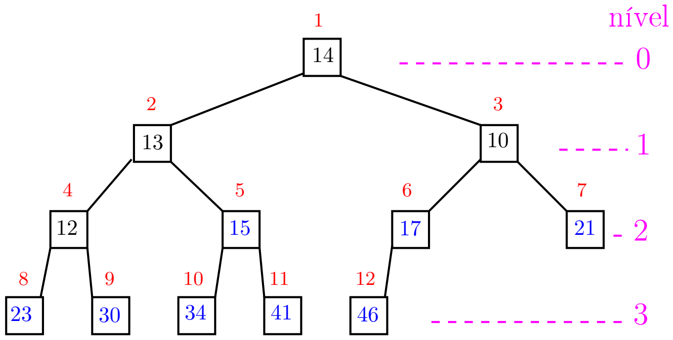
1	2	3	4	5	6	7	8	9	10	11	12
10	13	14	12	15	17	21	23	30	34	41	46

Heapsort



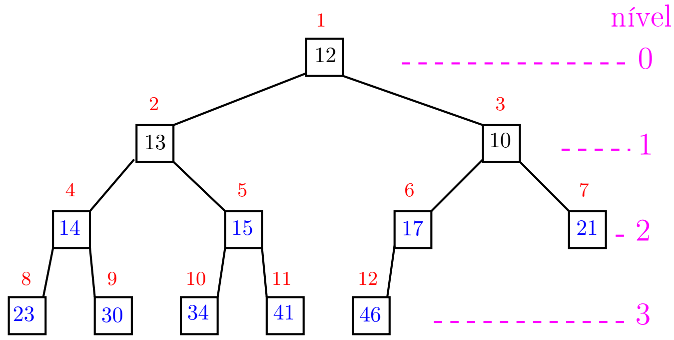
1	2	3	4	5	6	7	8	9	10	11	12
14	13	10	12	15	17	21	23	30	34	41	46

Heapsort



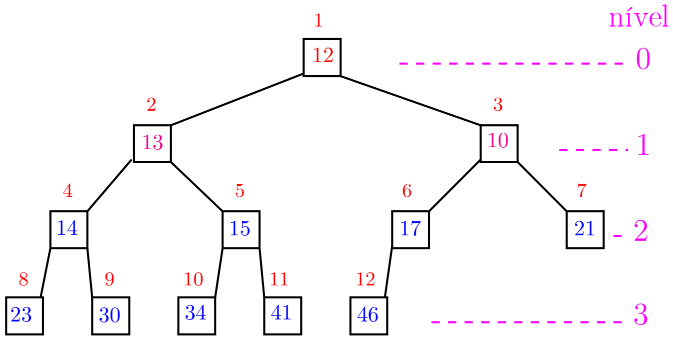
1	2	3	4	5	6	7	8	9	10	11	12
14	13	10	12	15	17	21	23	30	34	41	46

Heapsort



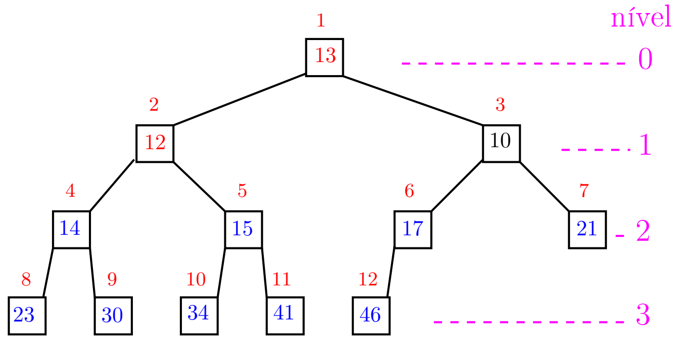
1	2	3	4	5	6	7	8	9	10	11	12
12	13	10	14	15	17	21	23	30	34	41	46

Heapsort



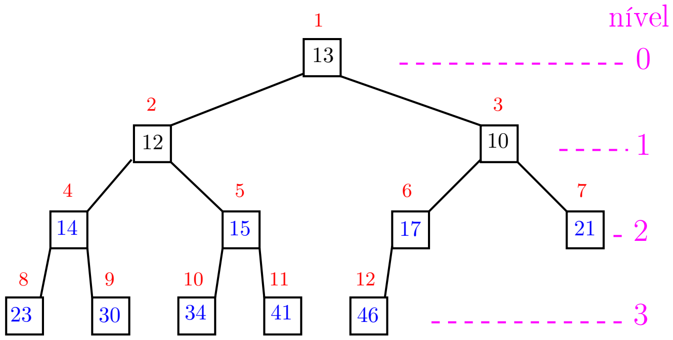
1	2	3	4	5	6	7	8	9	10	11	12
12	13	10	14	15	17	21	23	30	34	41	46

Heapsort



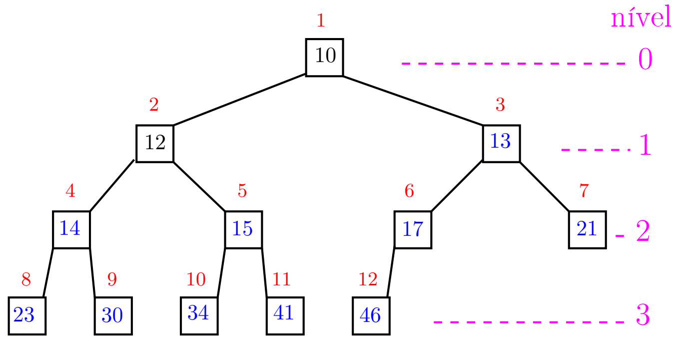
1	2	3	4	5	6	7	8	9	10	11	12
13	12	10	14	15	17	21	23	30	34	41	46

Heapsort



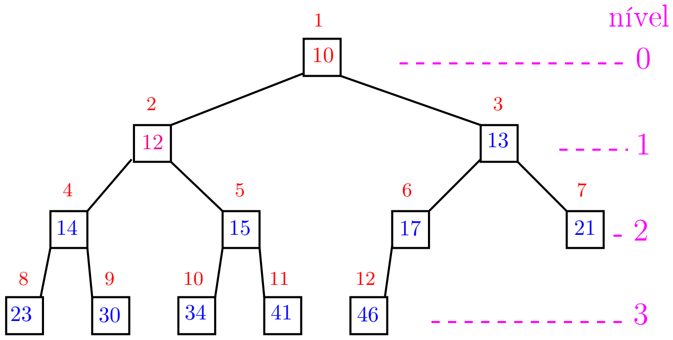
1	2	3	4	5	6	7	8	9	10	11	12
13	12	10	14	15	17	21	23	30	34	41	46

Heapsort



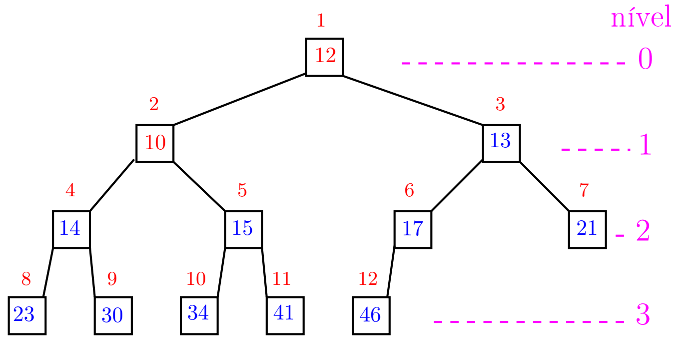
1	2	3	4	5	6	7	8	9	10	11	12
10	12	13	14	15	17	21	23	30	34	41	46

Heapsort



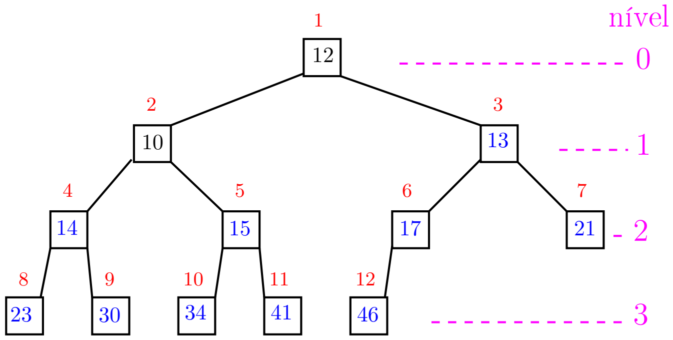
1	2	3	4	5	6	7	8	9	10	11	12
10	12	13	14	15	17	21	23	30	34	41	46

Heapsort



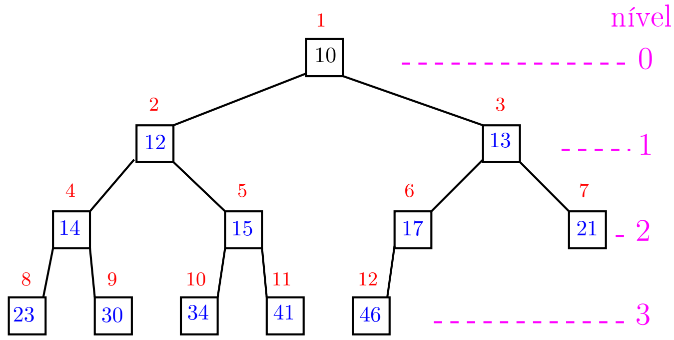
1	2	3	4	5	6	7	8	9	10	11	12
12	10	13	14	15	17	21	23	30	34	41	46

Heapsort



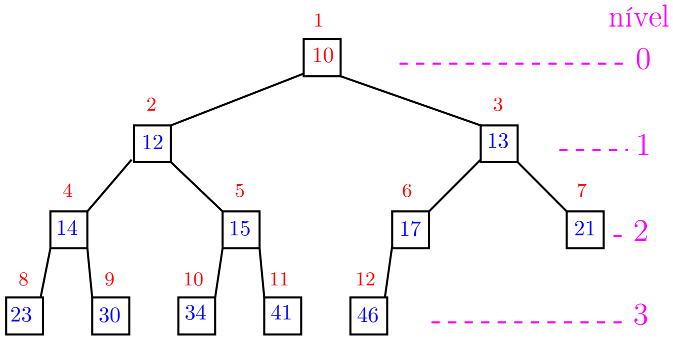
1	2	3	4	5	6	7	8	9	10	11	12
12	10	13	14	15	17	21	23	30	34	41	46

Heapsort



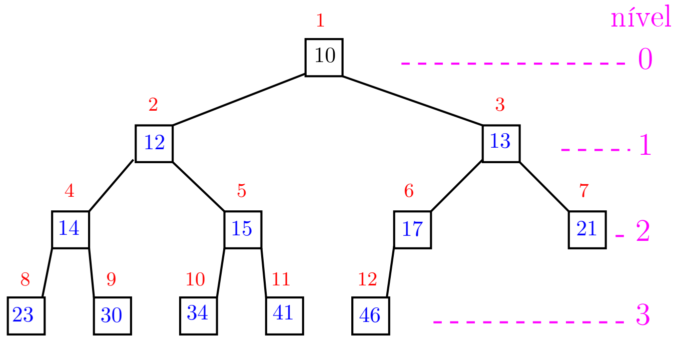
1	2	3	4	5	6	7	8	9	10	11	12
10	12	13	14	15	17	21	23	30	34	41	46

Heapsort



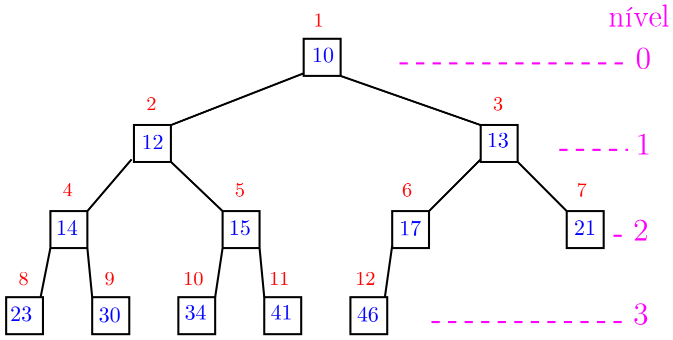
1	2	3	4	5	6	7	8	9	10	11	12
10	12	13	14	15	17	21	23	30	34	41	46

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 heapSort

Algoritmo rearranja v[1 . . n] em ordem crescente

```
void heapSort (int n, int v[])
{
     int i, x;
     /* pre-processamento */
1    for (i = n/2; i >= 1; i--)
2        peneira(i, n, v);

3    for (i = n; /*C*/ i > 1; i--) {
4        x=v[i]; v[i]=v[1]; v[1]=x;
5        peneira(1,i-1,v);
     }
 }
```

Função heapSort

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

(i0) v[i+1 . . n] é crescente;

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

(i2) v[1 . . i] 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 `heapSort` é proporcional a $n \lg n$.

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

Função `insereHeap`

Inseção de um elemento `x` em um `max-heap` $v[1\,.\,.\,n]$

```
void insereHeap (int x, int *n, int v[]) {
   int f /* filho */, p/* pai */, t;
   *n += 1; f = *n; p = f / 2; v[f] = x;
   while/*D*/ (f > 1 && v[p] < v[f]) {
      t = v[p];
      v[p] = v[f];
      v[f] = t;
      /* pai no papel de filho */
      f = p; p = f / 2;
   }
}
```

Função `insereHeap`

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

(i0) $v[1\,.\,.\,*n]$ é uma permutação do vetor original

(i1) $v[i/2] \geq v[i]$ para todo $i = 2, \ldots, *n$ diferente de `f`.

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

Conclusão

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

O consumo de tempo da função `heapSort` é $O(n)$, onde n é o número de elementos no `max-heap`.

Mais análise experimental

Algoritmos implementados:

- `mergeR` `mergeSort` recursivo.
- `mergeI` `mergeSort` iterativo.
- `quick` `quickSort` recursivo.
- `heap` `heapSort`.

Mais análise experimental

A plataforma utilizada nos experimentos foi um computador rodando Ubuntu GNU/Linux 3.5.0-17

Compilador:

`gcc -Wall -ansi -O2 -pedantic -Wno-unused-result`.

Computador:

```
 model name:  Intel(R) Core(TM)2 Quad CPU Q6600 @
2.40GHz
cpu MHz :  1596.000
cache size:  4096 KB
MemTotal :  3354708 kB
```

Aleatório: média de 10

n	mergeR	mergeI	quick	heap
8192	0.00	0.00	0.00	0.00
16384	0.00	0.00	0.00	0.00
32768	0.01	0.01	0.01	0.00
65536	0.01	0.01	0.01	0.01
131072	0.02	0.02	0.02	0.03
262144	0.05	0.04	0.04	0.06
524288	0.10	0.08	0.08	0.12
1048576	0.21	0.20	0.17	0.28
2097152	0.44	0.43	0.35	0.70
4194304	0.92	0.90	0.73	1.73
8388608	1.90	1.87	1.51	4.13

Tempos em segundos.

Decrescente

n	mergeR	mergeI	quick	heap
1024	0.00	0.00	0.00	0.00
2048	0.00	0.00	0.00	0.00
4096	0.01	0.00	0.01	0.00
8192	0.00	0.00	0.03	0.00
16384	0.00	0.00	0.14	0.00
32768	0.00	0.01	0.57	0.00
65536	0.01	0.01	2.27	0.01
131072	0.02	0.01	9.06	0.02
262144	0.03	0.03	36.31	0.04

Tempos em segundos.

Para n=524288 `quickSort` dá `Segmentation fault (core dumped)`

Crescente

n	mergeR	mergeI	quick	heap
1024	0.00	0.00	0.00	0.00
2048	0.00	0.00	0.00	0.00
4096	0.00	0.00	0.00	0.00
8192	0.00	0.00	0.03	0.00
16384	0.00	0.00	0.14	0.01
32768	0.01	0.00	0.57	0.01
65536	0.00	0.01	2.26	0.01
131072	0.02	0.02	9.05	0.02
262144	0.03	0.02	36.21	0.04

Tempos em segundos.

Para n=524288 `quickSort` dá `Segmentation fault (core dumped)`

Resumo

função	consumo de tempo	observação
bubble	$O(n^2)$	todos os casos
insercao	$O(n^2)$ $O(n)$	pior caso melhor caso
insercaoBinaria	$O(n^2)$ $O(n \lg n)$	pior caso melhor caso
selecao	$O(n^2)$	todos os casos
mergeSort	$O(n \lg n)$	todos os casos
quickSort	$O(n^2)$ $O(n \lg n)$	pior caso melhor caso
heapSort	$O(n \lg n)$	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/UCIqiLefbVHsOAXDAxQJH7I