Melhores momentos

AULA 17

Resumo

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

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;

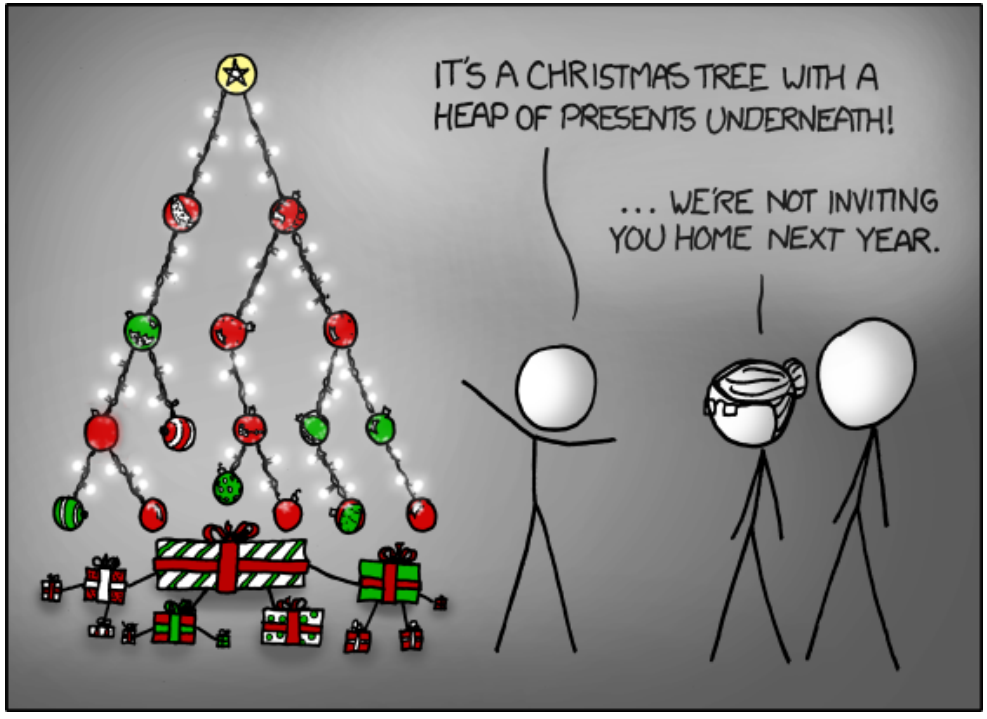
Conquistar: os subproblemas são resolvidos recursivamente e subproblemas "pequenos" são resolvidos diretamente;

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

Exemplos: `mergeSort` e `quickSort`.

AULA 18

Árvores em vetores e heaps

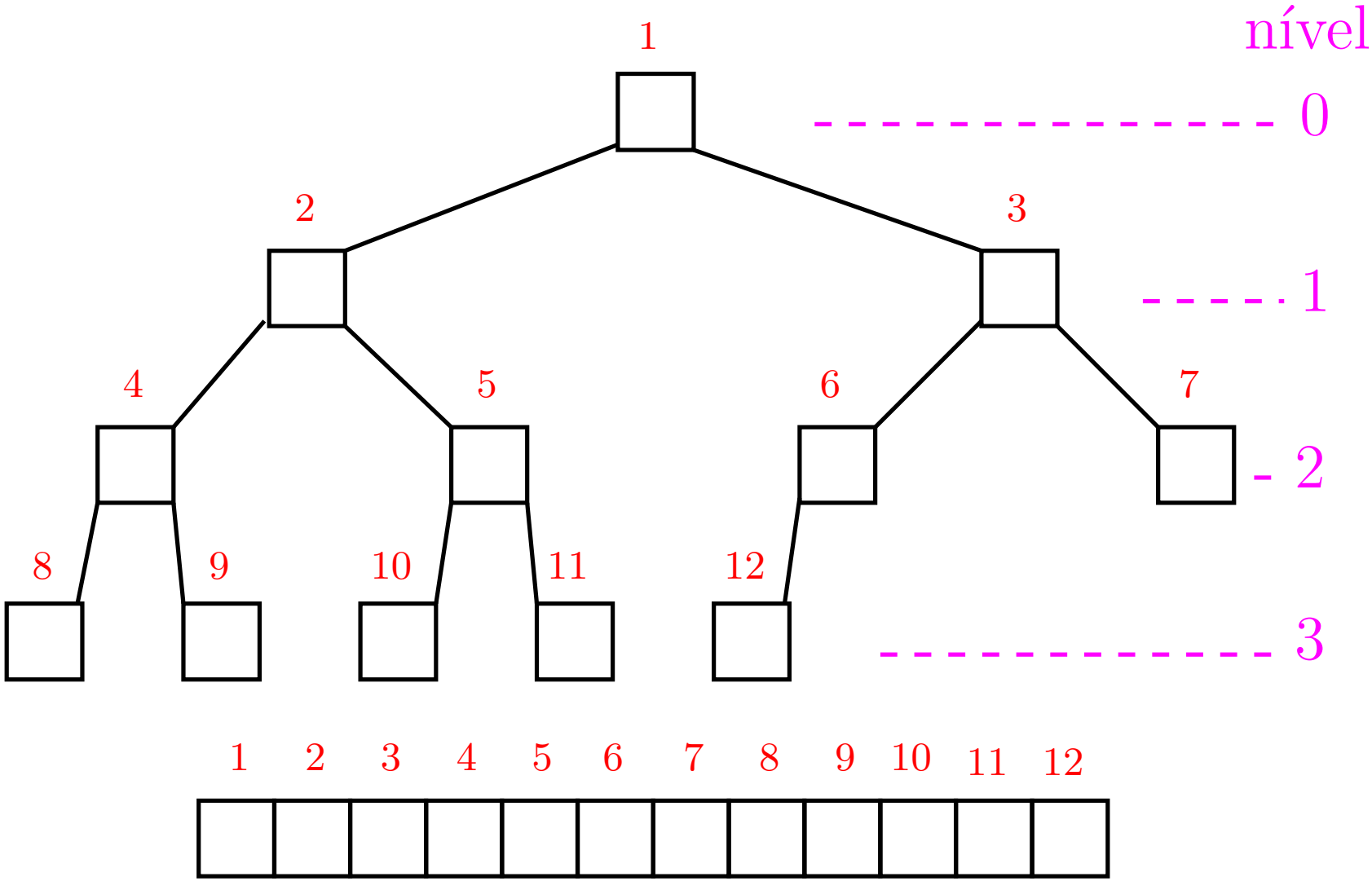


Fonte: http://xkcd.com/835/

PF 10

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

Representação de árvores em vetores



Pais e filhos

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

Diremos que para qualquer **índice** ou **nó** $\mathtt{i}$,

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

Um nó $\mathtt{i}$ tem filho esquerdo se $2\,\mathtt{i} \leq \mathtt{m}$.

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

Raiz e folhas

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

Um nó $\mathtt{i}$ é uma **folha** se não tem **filhos**,
ou seja, $2\,\mathtt{i} > \mathtt{m}$.

Raiz e folhas

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

Um nó $\mathtt{i}$ é uma **folha** se não tem **filhos**,
ou seja, $2\,\mathtt{i} > \mathtt{m}$.

Níveis

Cada nível $\mathtt{p}$, exceto talvez o último,
tem exatamente $2^{\mathtt{p}}$ nós e esses são

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

Raiz e folhas

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

Um nó $\mathtt{i}$ é uma **folha** se não tem **filhos**,
ou seja, $2\,\mathtt{i} > \mathtt{m}$.

Níveis

Cada nível $\mathtt{p}$, exceto talvez o último,
tem exatamente $2^{\mathtt{p}}$ nós e esses são

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

O nó $\mathtt{i}$ pertence ao nível

Raiz e folhas

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

Um nó $\mathtt{i}$ é uma **folha** se não tem **filhos**,
ou seja, $2\,\mathtt{i} > \mathtt{m}$.

Níveis

Cada nível $\mathtt{p}$, exceto talvez o último,
tem exatamente $2^{\mathtt{p}}$ nós e esses são

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

O nó $\mathtt{i}$ pertence ao nível $\lfloor \lg \mathtt{i} \rfloor$.

Raiz e folhas

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

Um nó $\mathtt{i}$ é uma **folha** se não tem **filhos**,
ou seja, $2\,\mathtt{i} > \mathtt{m}$.

Níveis

Cada nível $\mathtt{p}$, exceto talvez o último,
tem exatamente $2^{\mathtt{p}}$ nós e esses são

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

O nó $\mathtt{i}$ pertence ao nível $\lfloor \lg \mathtt{i} \rfloor$.

Portanto, o número total de níveis é

Raiz e folhas

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

Um nó $\mathtt{i}$ é uma **folha** se não tem **filhos**,
ou seja, $2\,\mathtt{i} > \mathtt{m}$.

Níveis

Cada nível $\mathtt{p}$, exceto talvez o último,
tem exatamente $2^{\mathtt{p}}$ nós e esses são

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

O nó $\mathtt{i}$ pertence ao nível $\lfloor \lg \mathtt{i} \rfloor$.

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

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 sequência da forma

$$\langle \texttt{filho}(\texttt{i}), \texttt{filho}(\texttt{filho}(\texttt{i})), \texttt{filho}(\texttt{filho}(\texttt{filho}(\texttt{i}))), \ldots \rangle$$

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

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

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:	$\lfloor \lg m \rfloor$

Heaps

Um vetor $\mathtt{v}[1\,..\,\mathtt{m}]$ é um **max-heap** se

$$\mathtt{v}[\mathtt{i}/2] \geq \mathtt{v}[\mathtt{i}]$$

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

Heaps

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

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

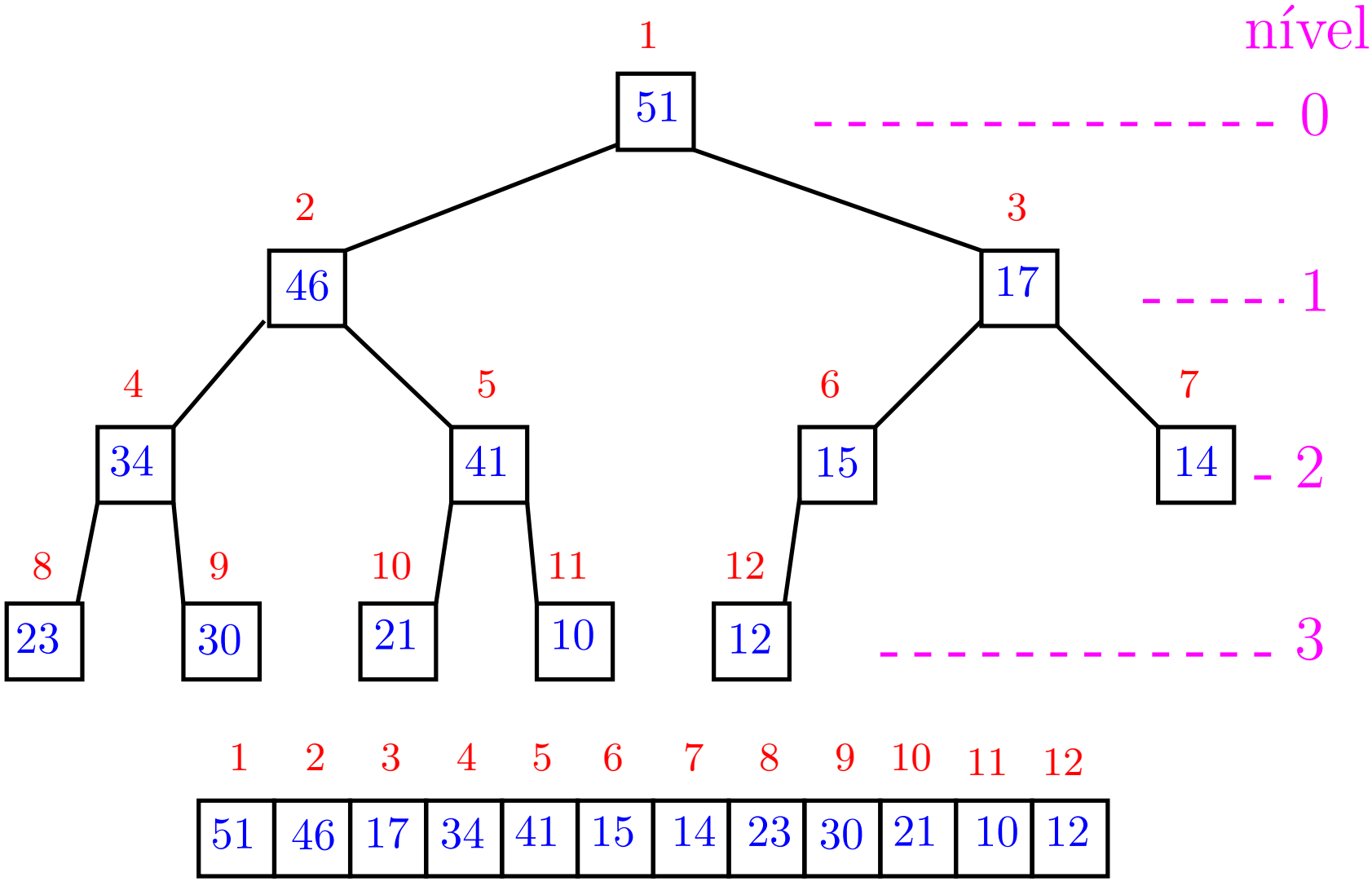
para todo $i = 2, 3, \ldots, m$.

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

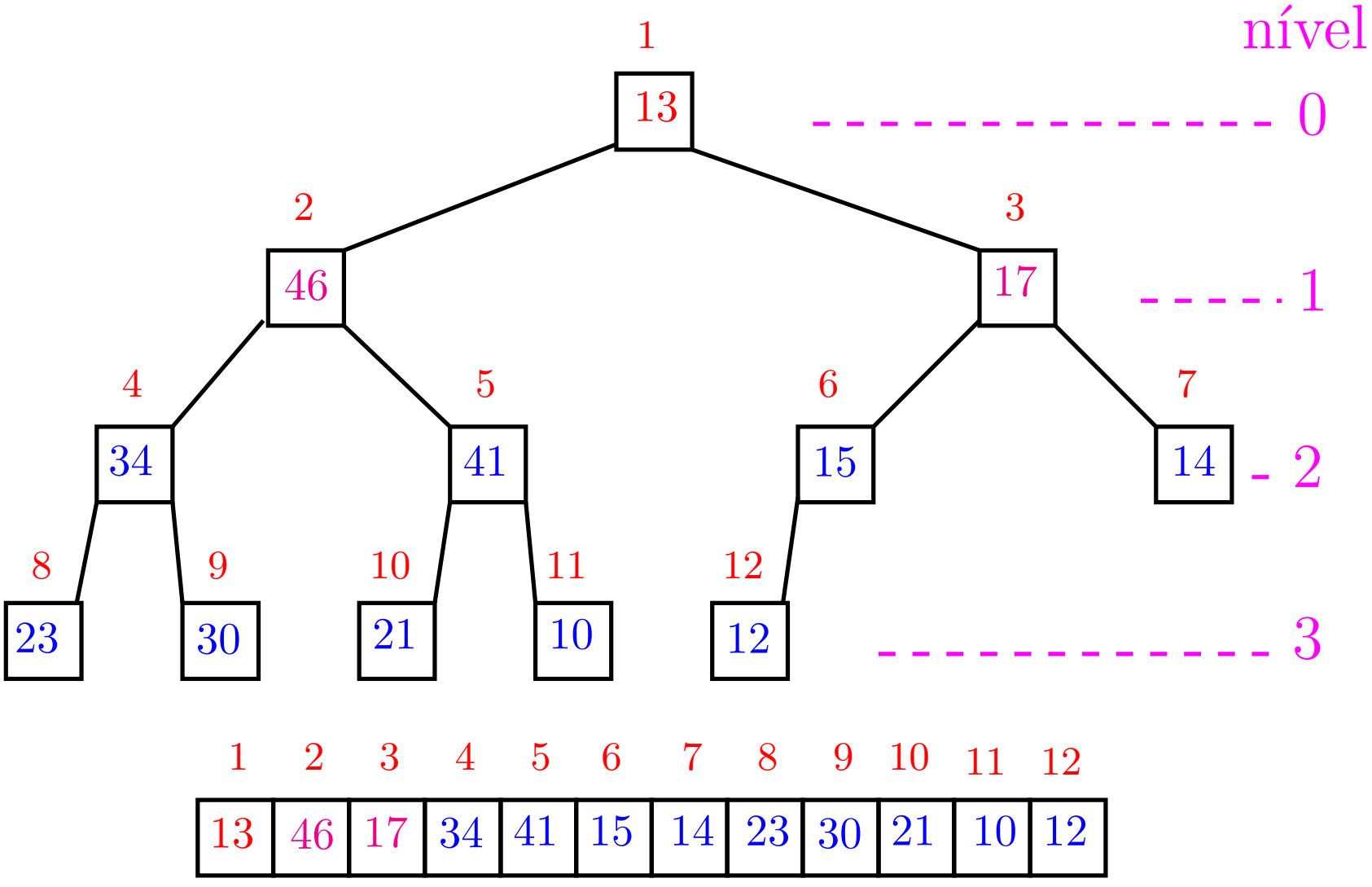
$$v[i/2] \geq v[i]$$

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**.

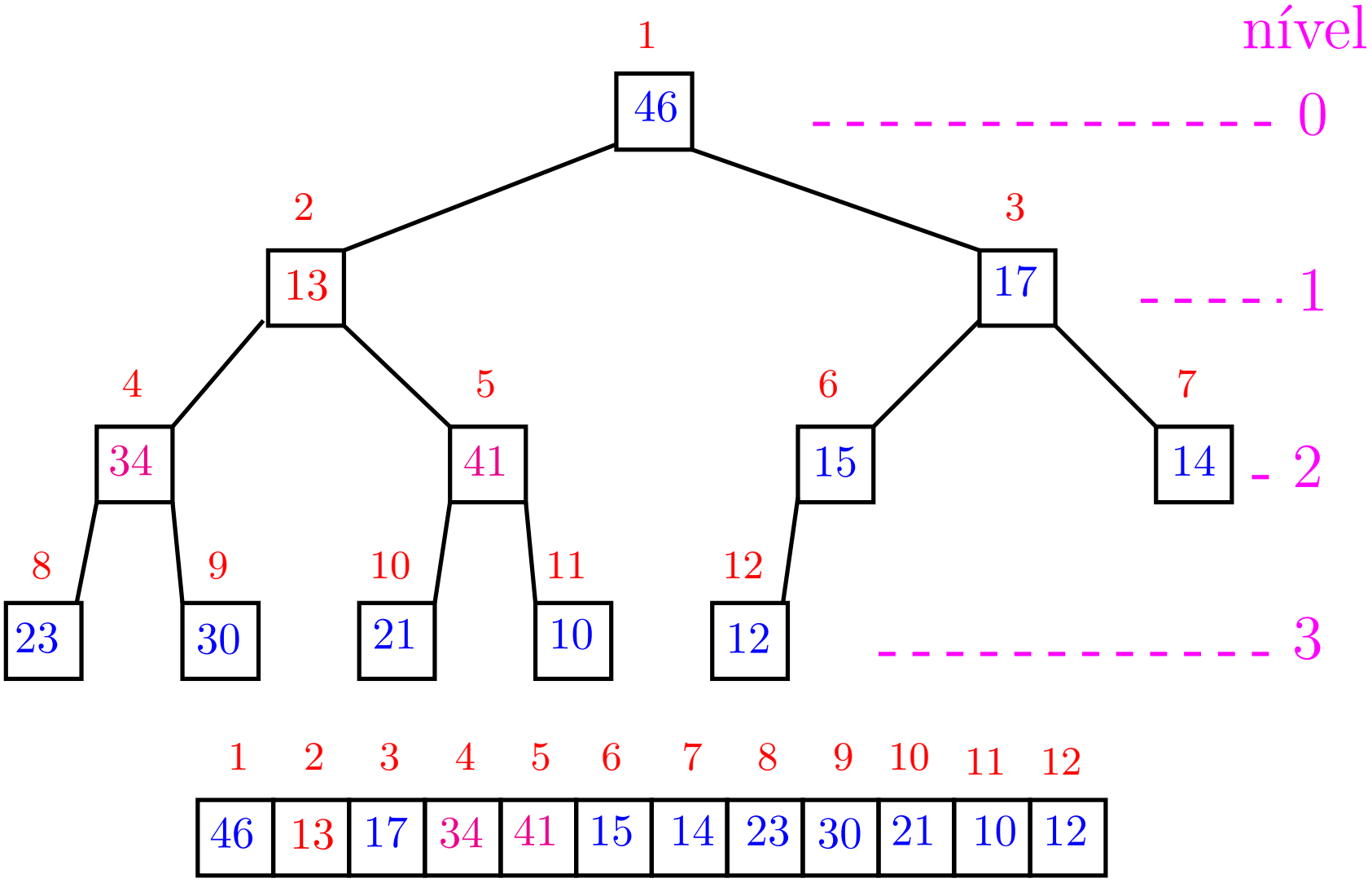
max-heap


nível
1
51
0
2
46
3
17
1
4
34
5
41
6
15
7
14
2
8
23
9
30
10
21
11
10
12
12
3
1 2 3 4 5 6 7 8 9 10 11 12
51 46 17 34 41 15 14 23 30 21 10 12

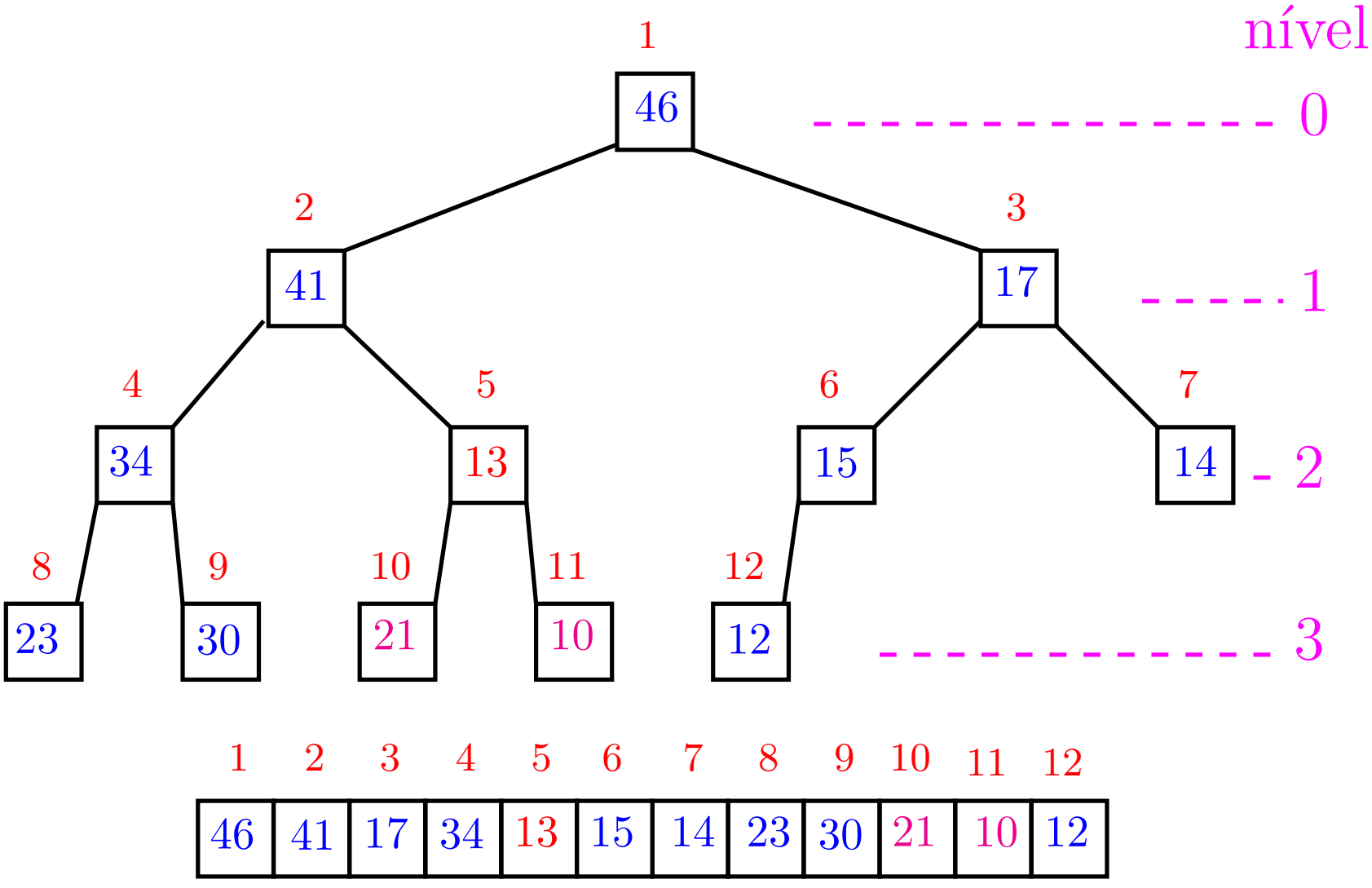
Função básica de manipulação de `max-heap`



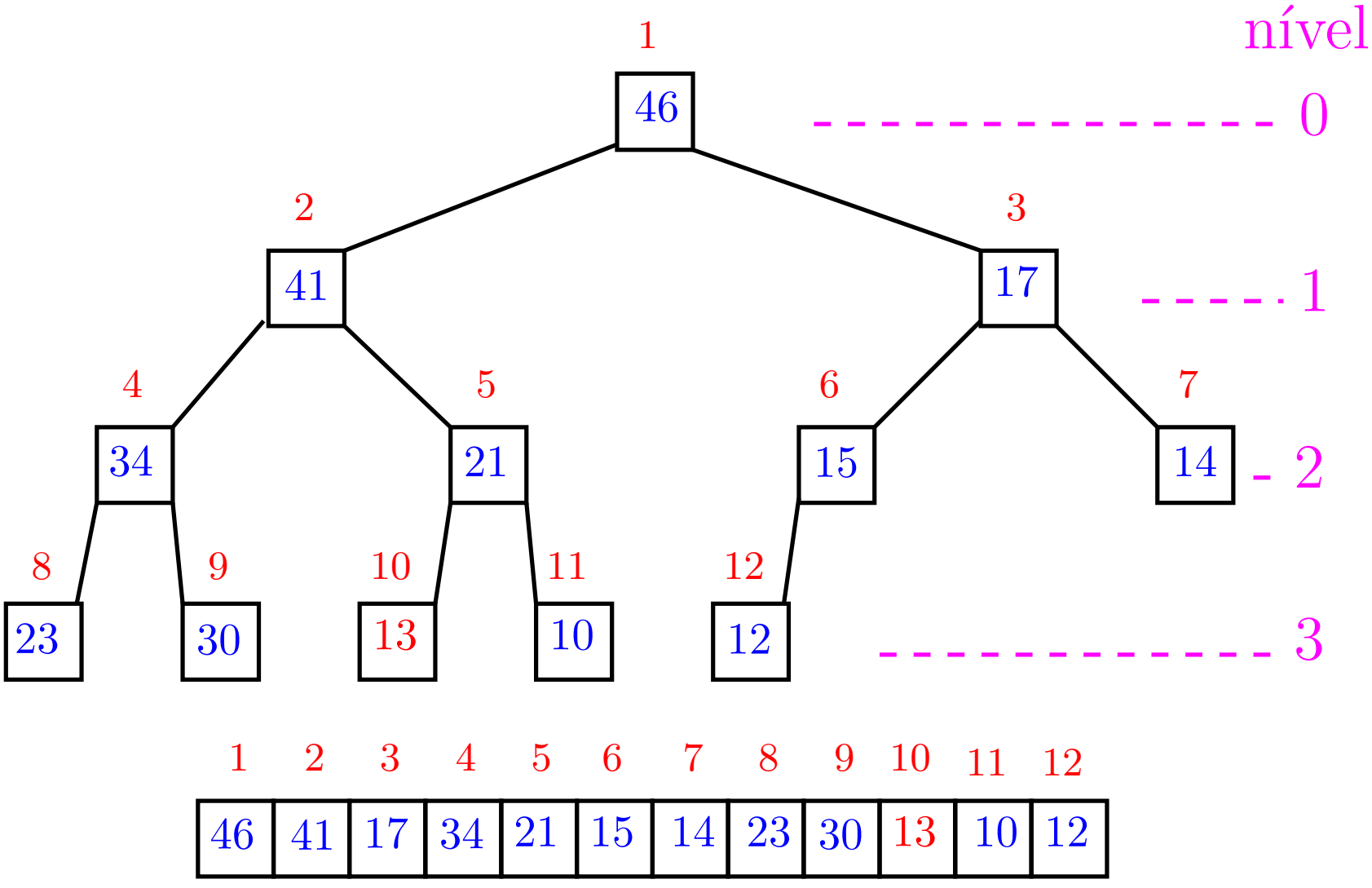
Função básica de manipulação de `max-heap`



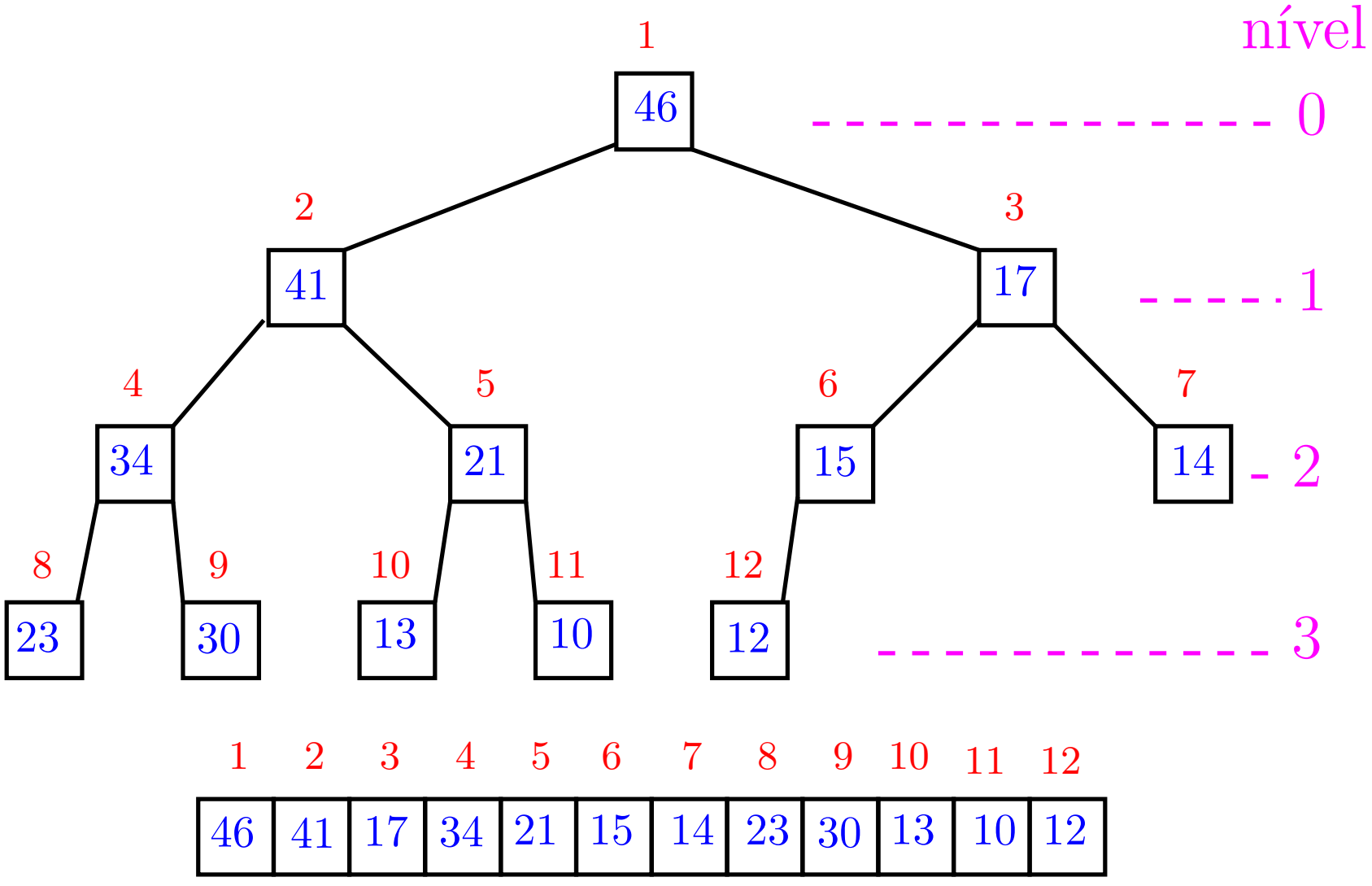
Função básica de manipulação de `max-heap`



Função básica de manipulação de `max-heap`



Função básica de manipulação de `max-heap`



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

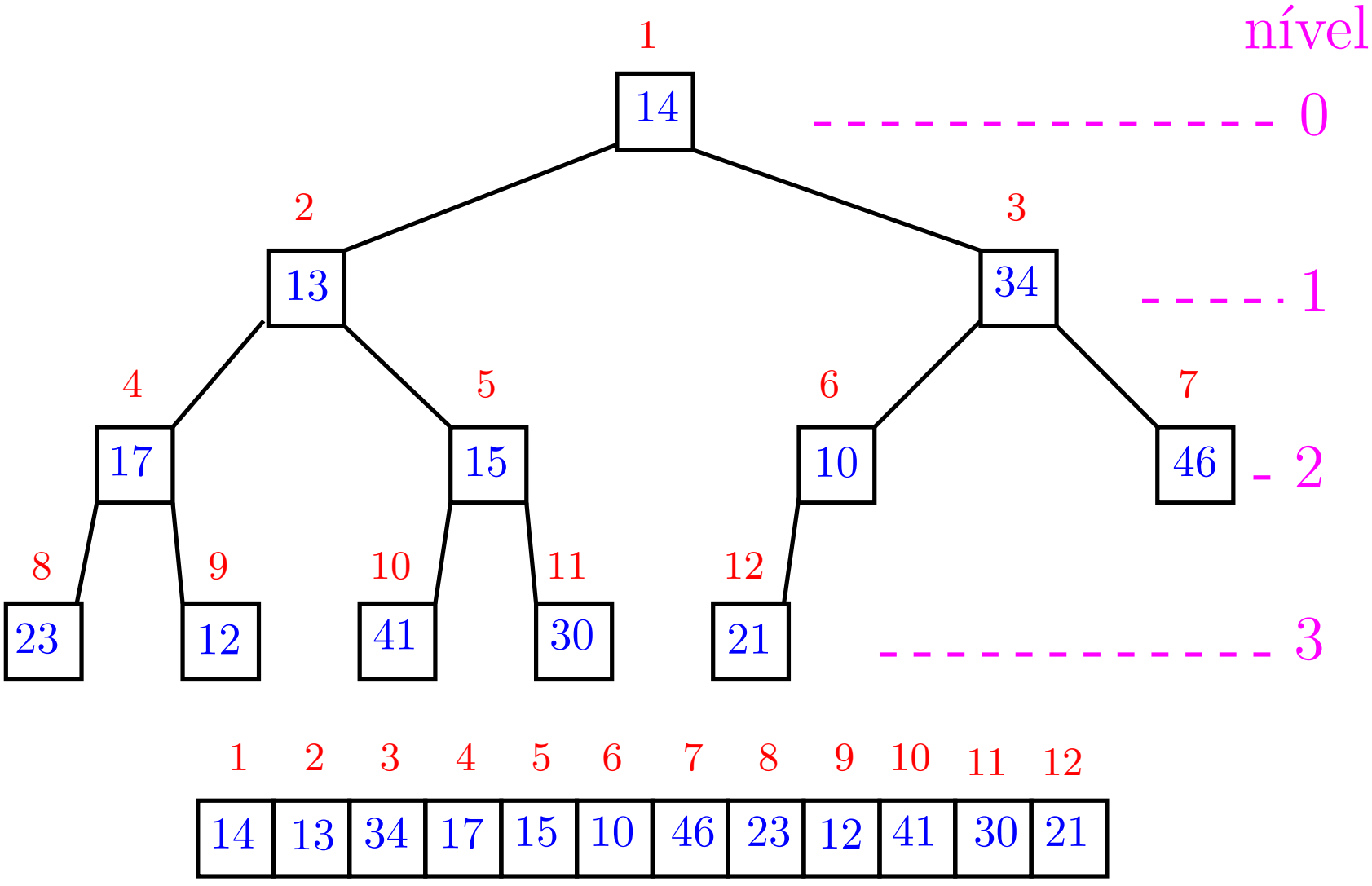
linha		todas as execuções da linha
1	$=$	1
2	$\leq$	$1 + \lg \mathtt{m}$
3	$\leq$	$\lg \mathtt{m}$
4	$\leq$	$\lg \mathtt{m}$
5	$\leq$	$\lg \mathtt{m}$
6	$\leq$	$\lg \mathtt{m}$
7	$=$	1
total	$\leq$	$3 + 5 \lg \mathtt{m} = \mathrm{O}(\lg \mathtt{m})$

Conclusão

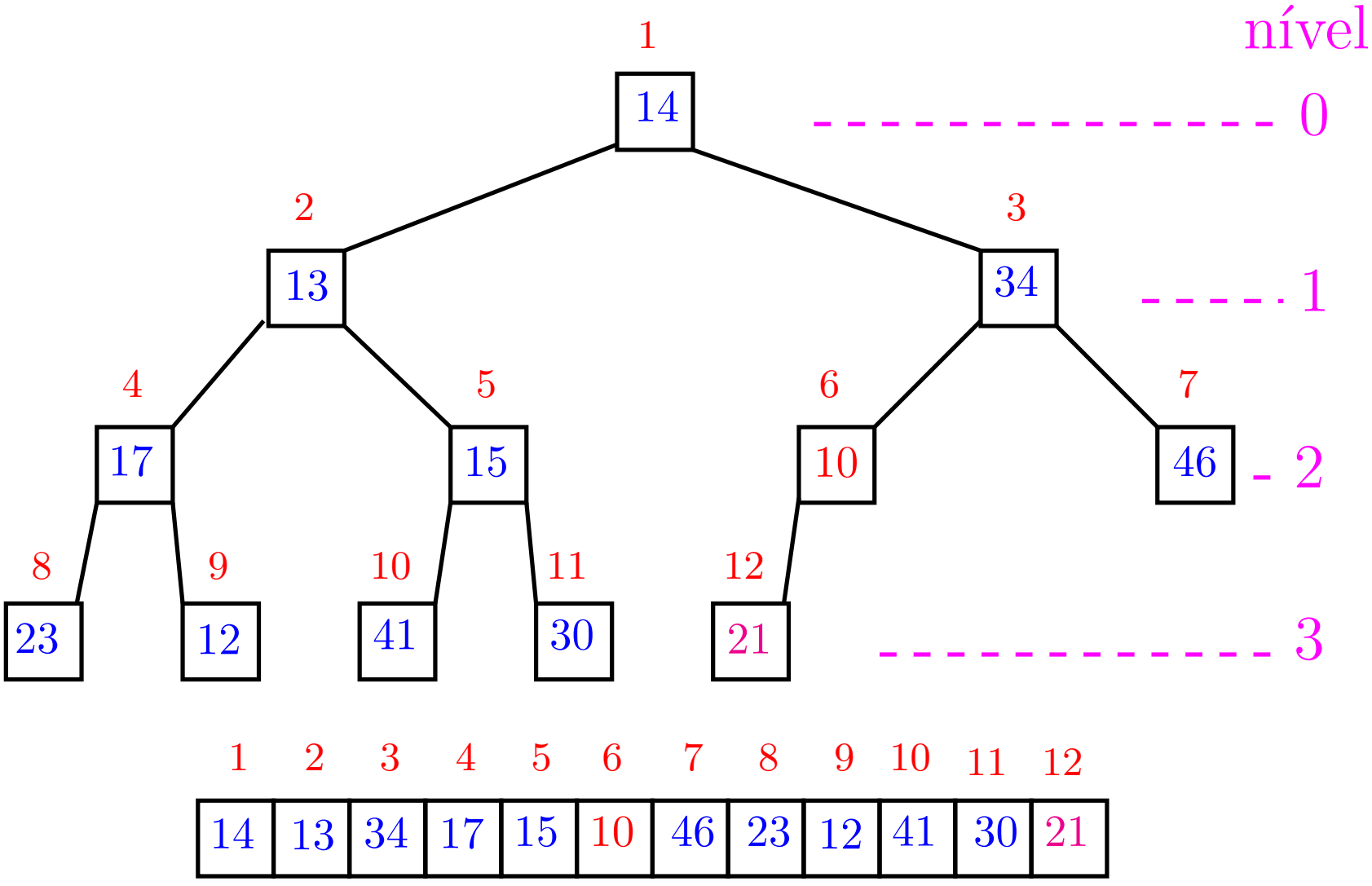
O consumo de tempo da função `peneira` é proporcional a $\lg$ `m`.

O consumo de tempo da função `peneira` é $\mathrm{O}(\lg$ `m`$)$.

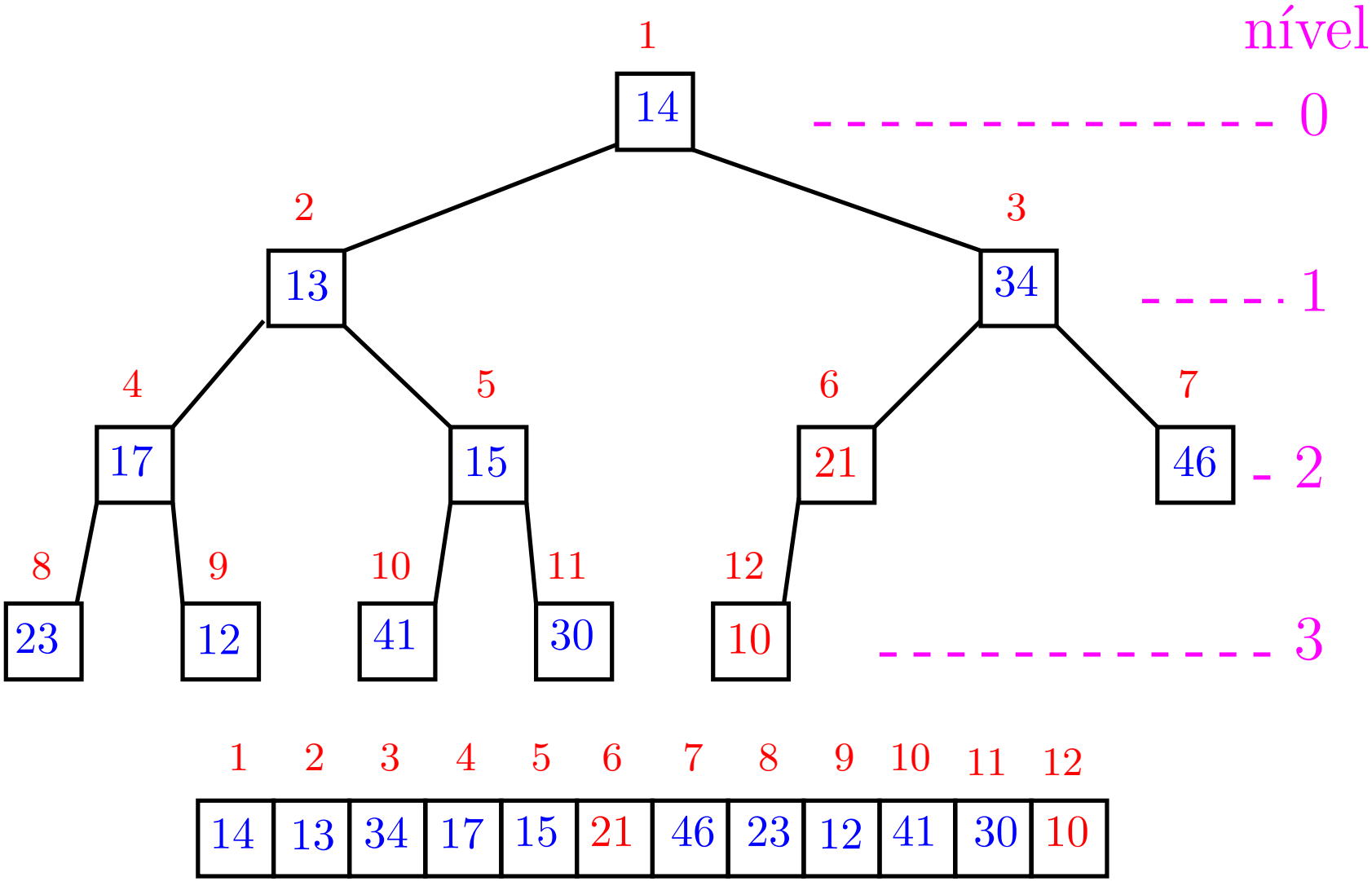
Construção de um max-heap



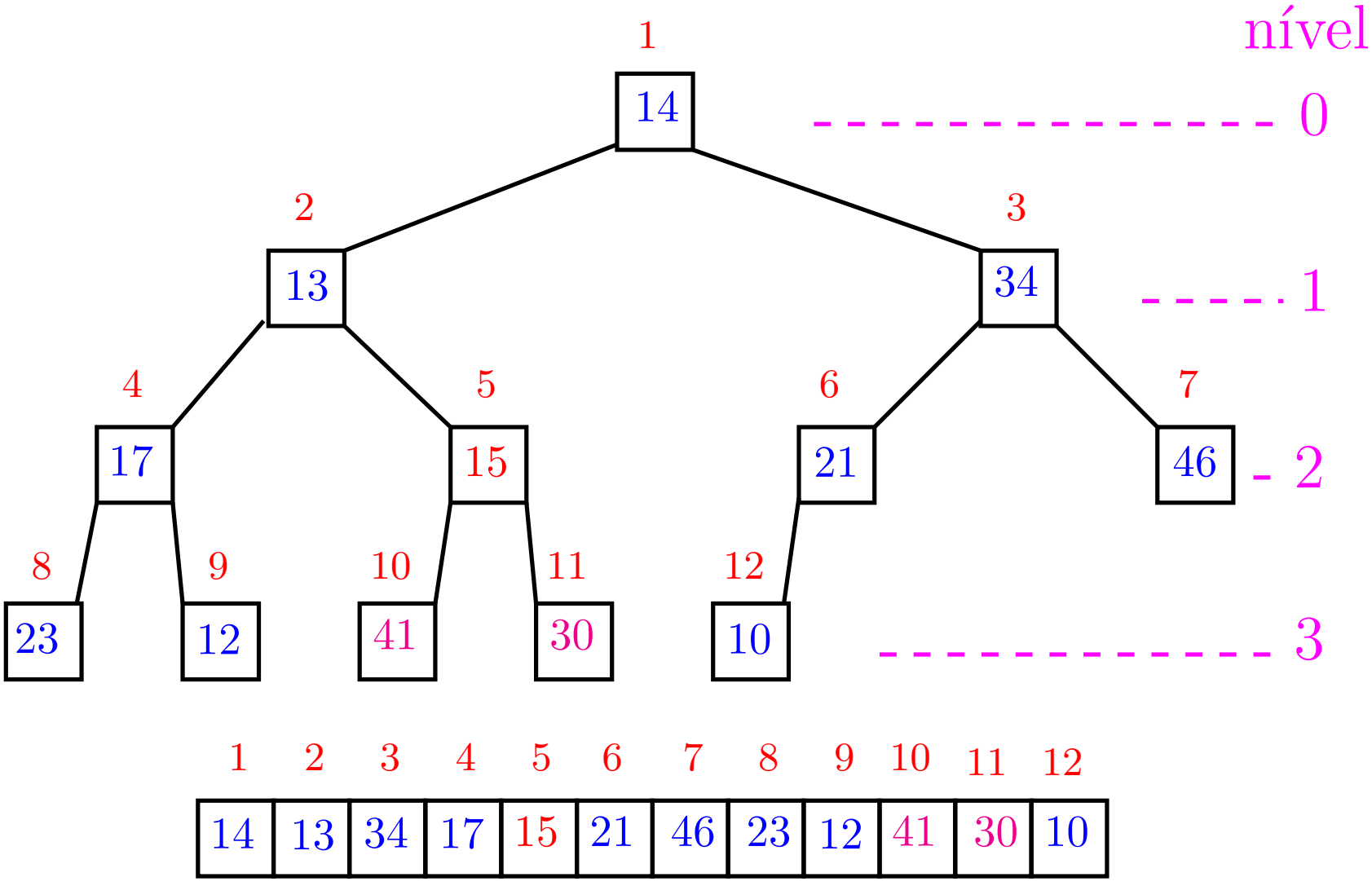
Construção de um max-heap



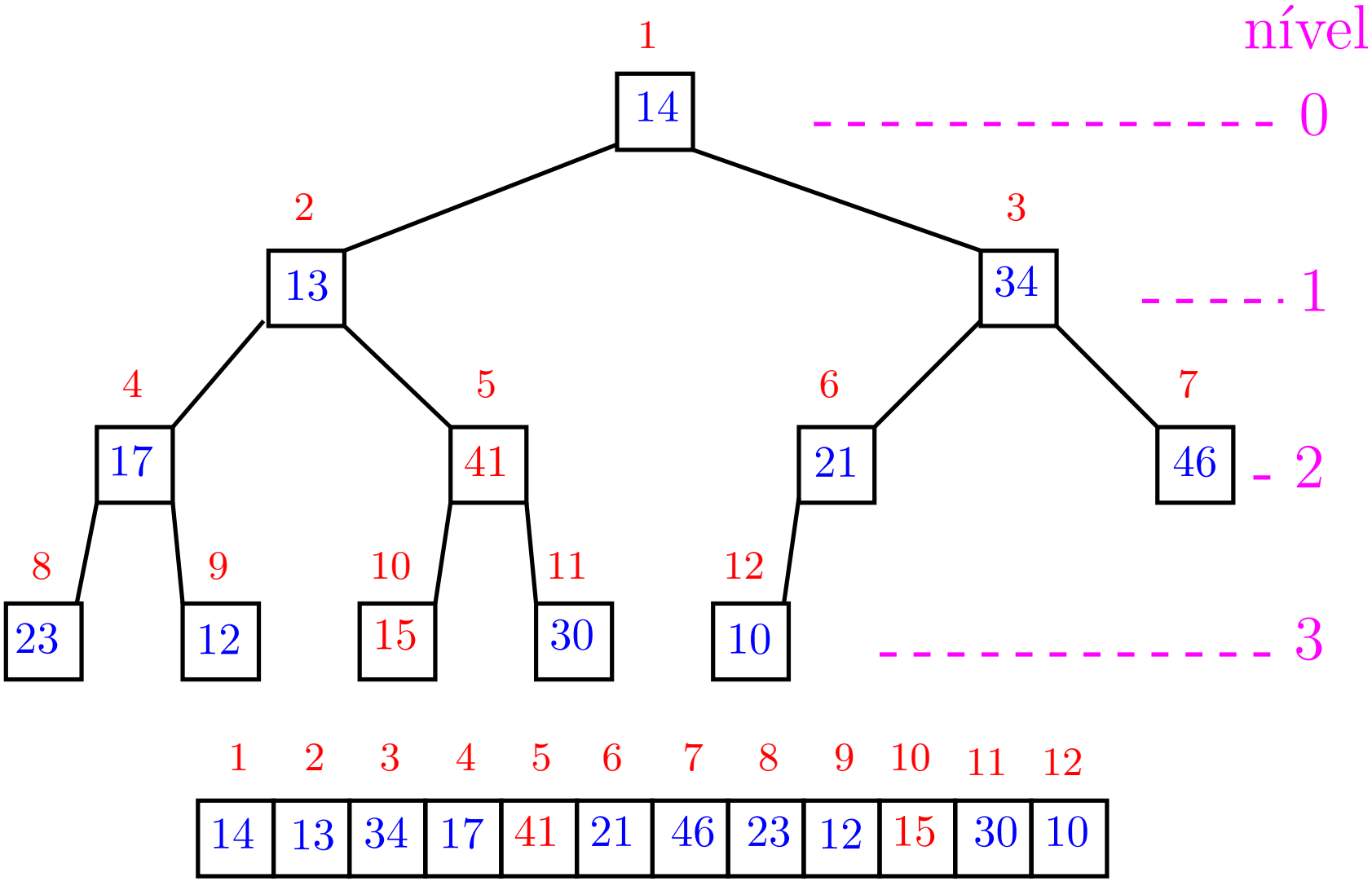
Construção de um max-heap



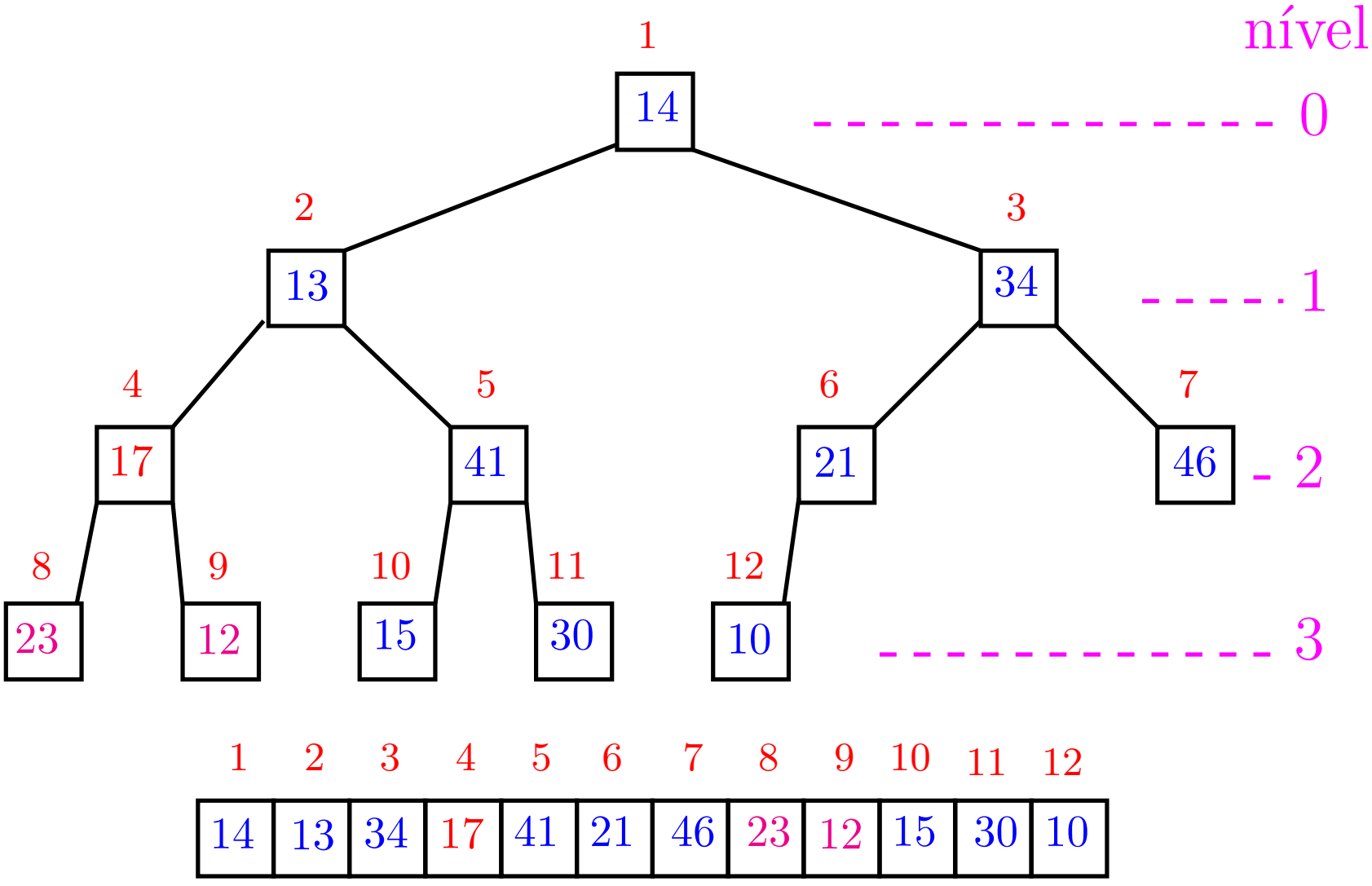
Construção de um max-heap



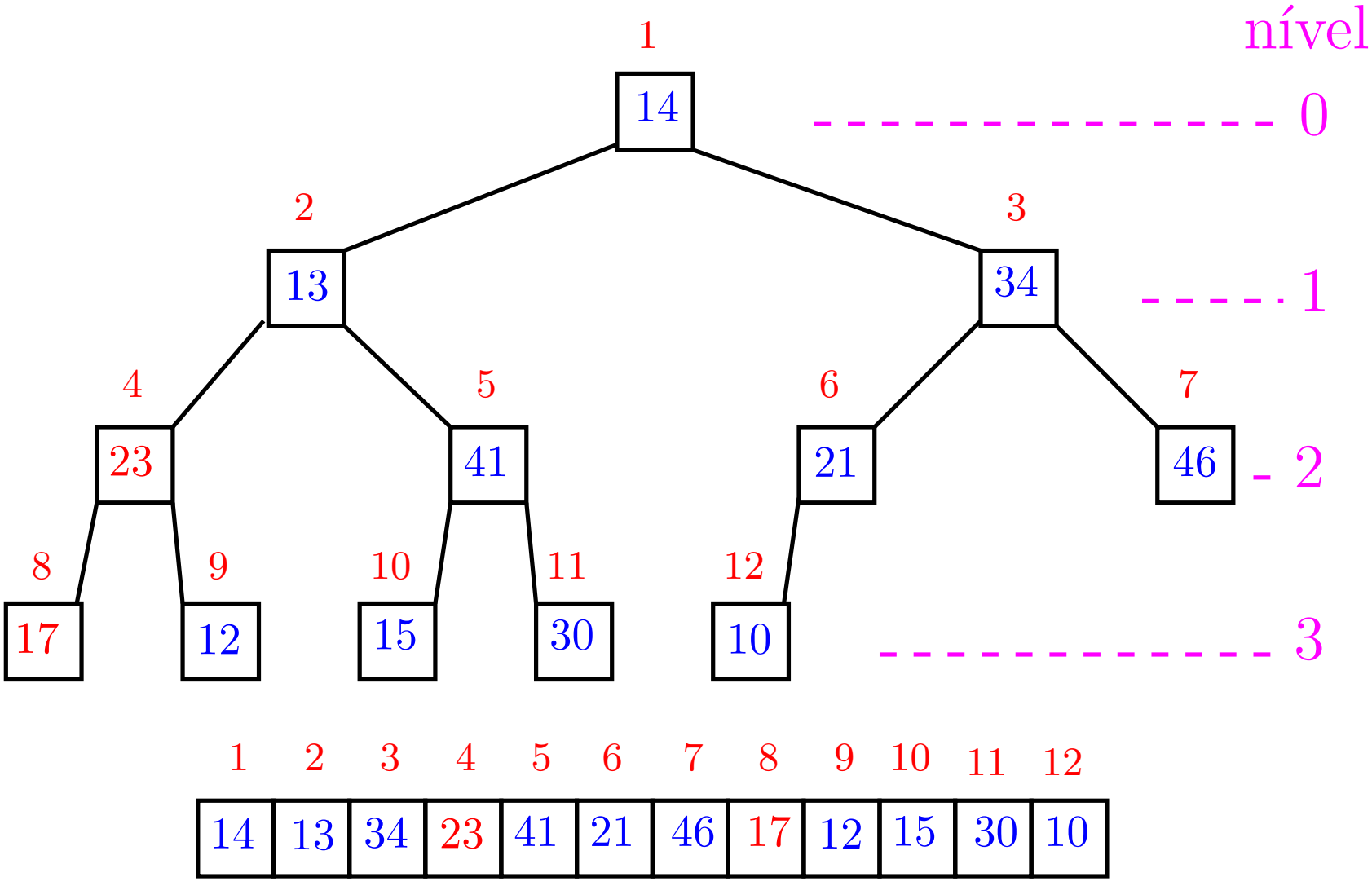
Construção de um max-heap



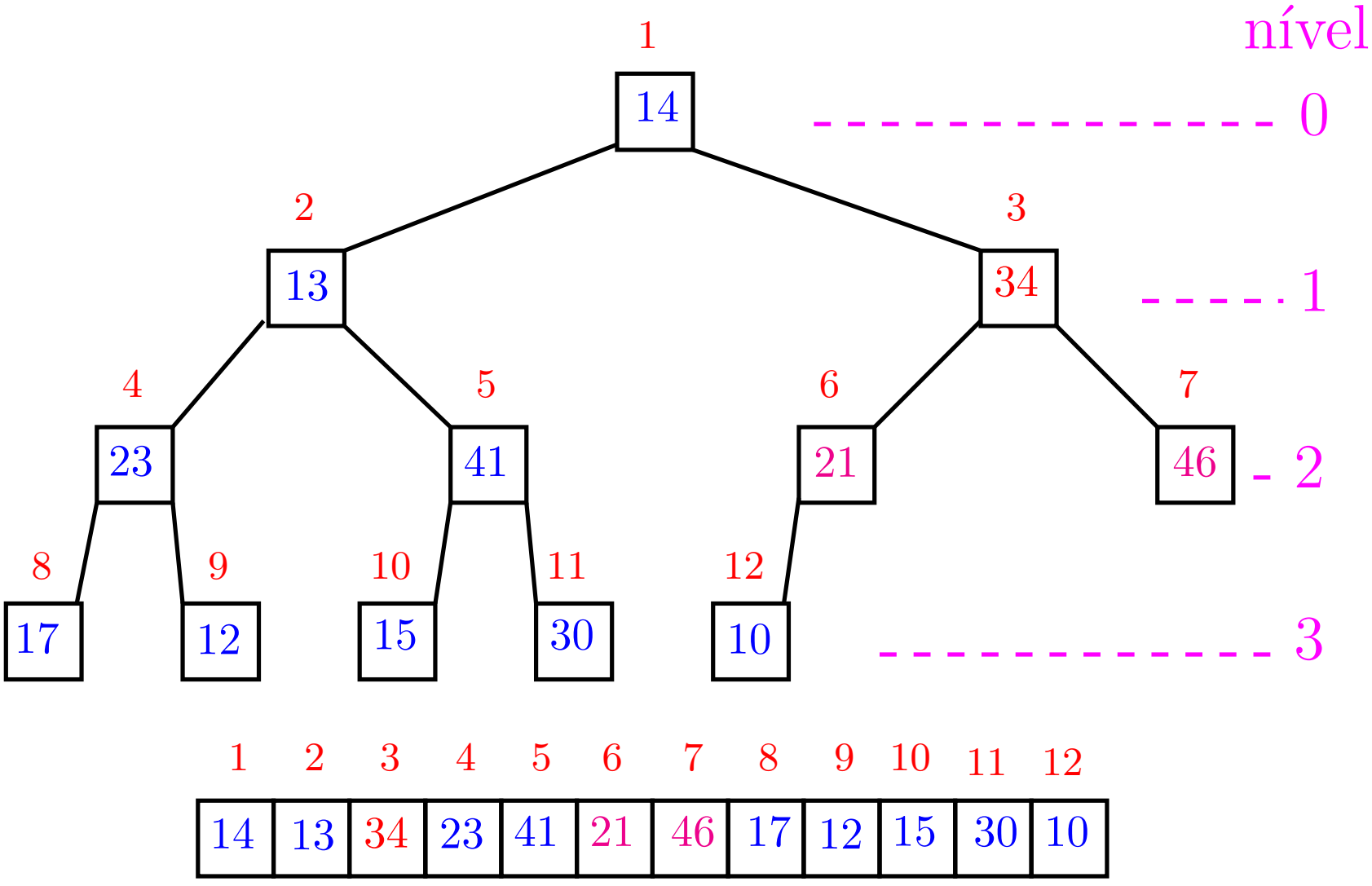
Construção de um max-heap



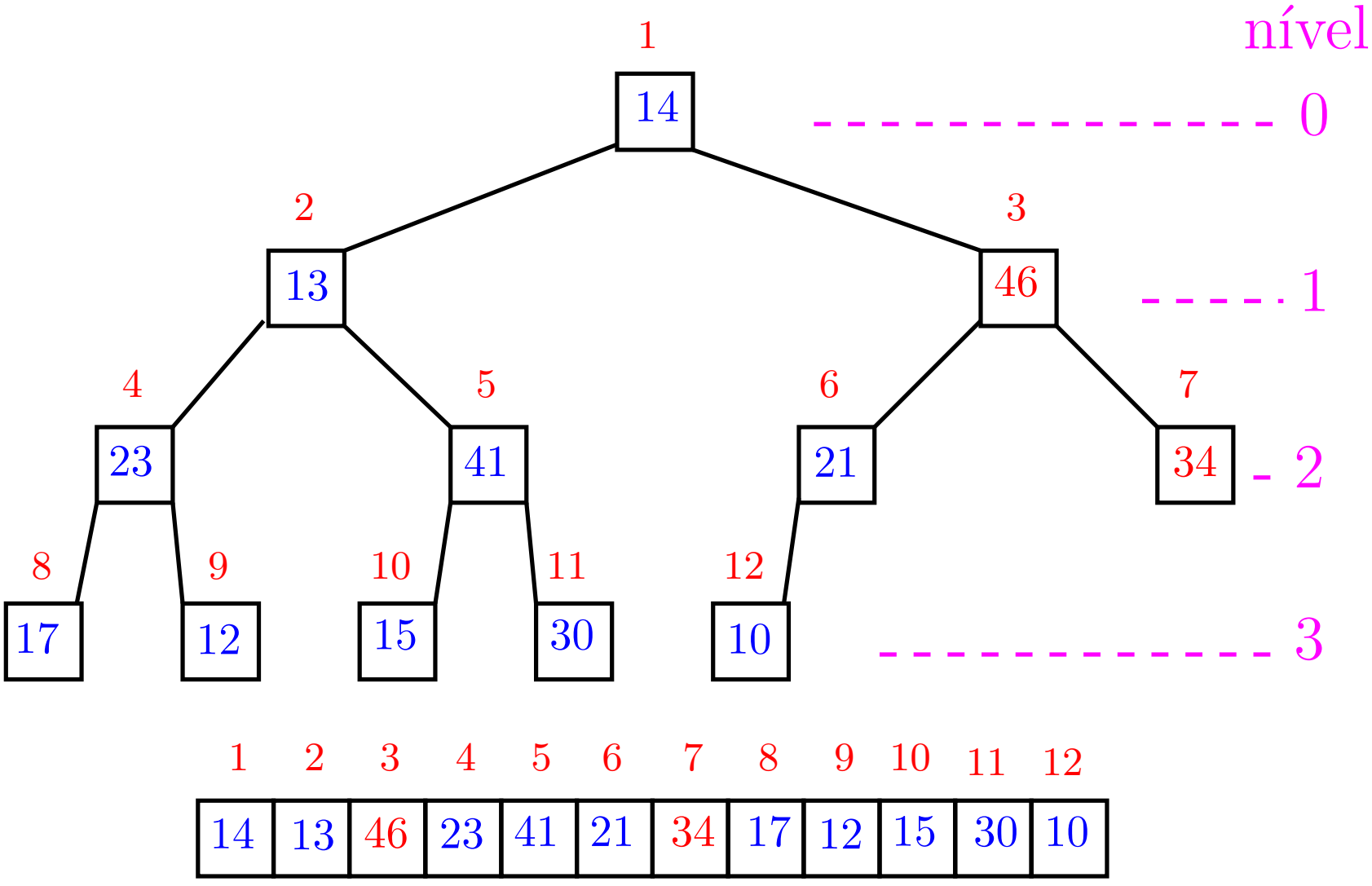
Construção de um max-heap



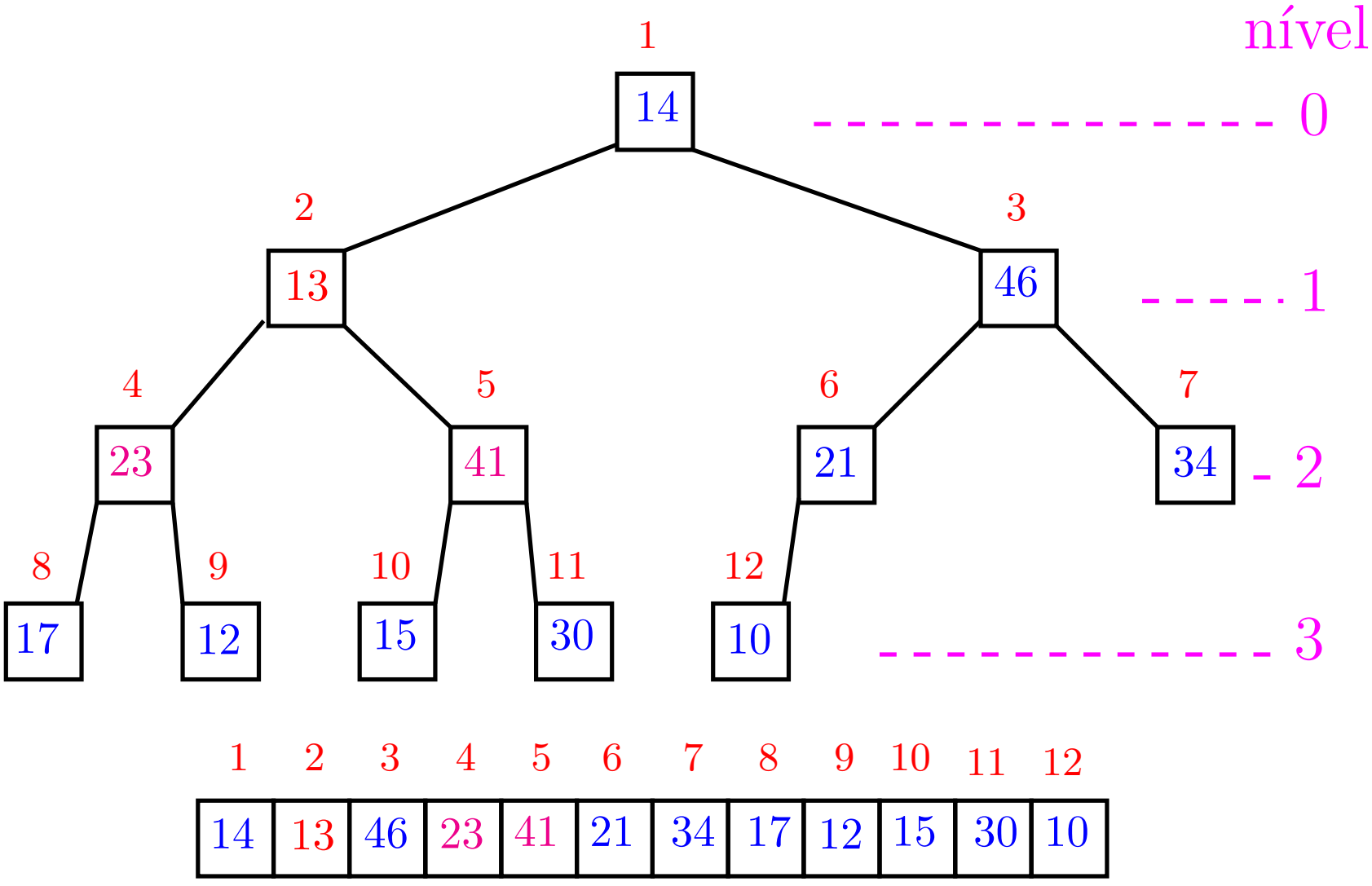
Construção de um max-heap



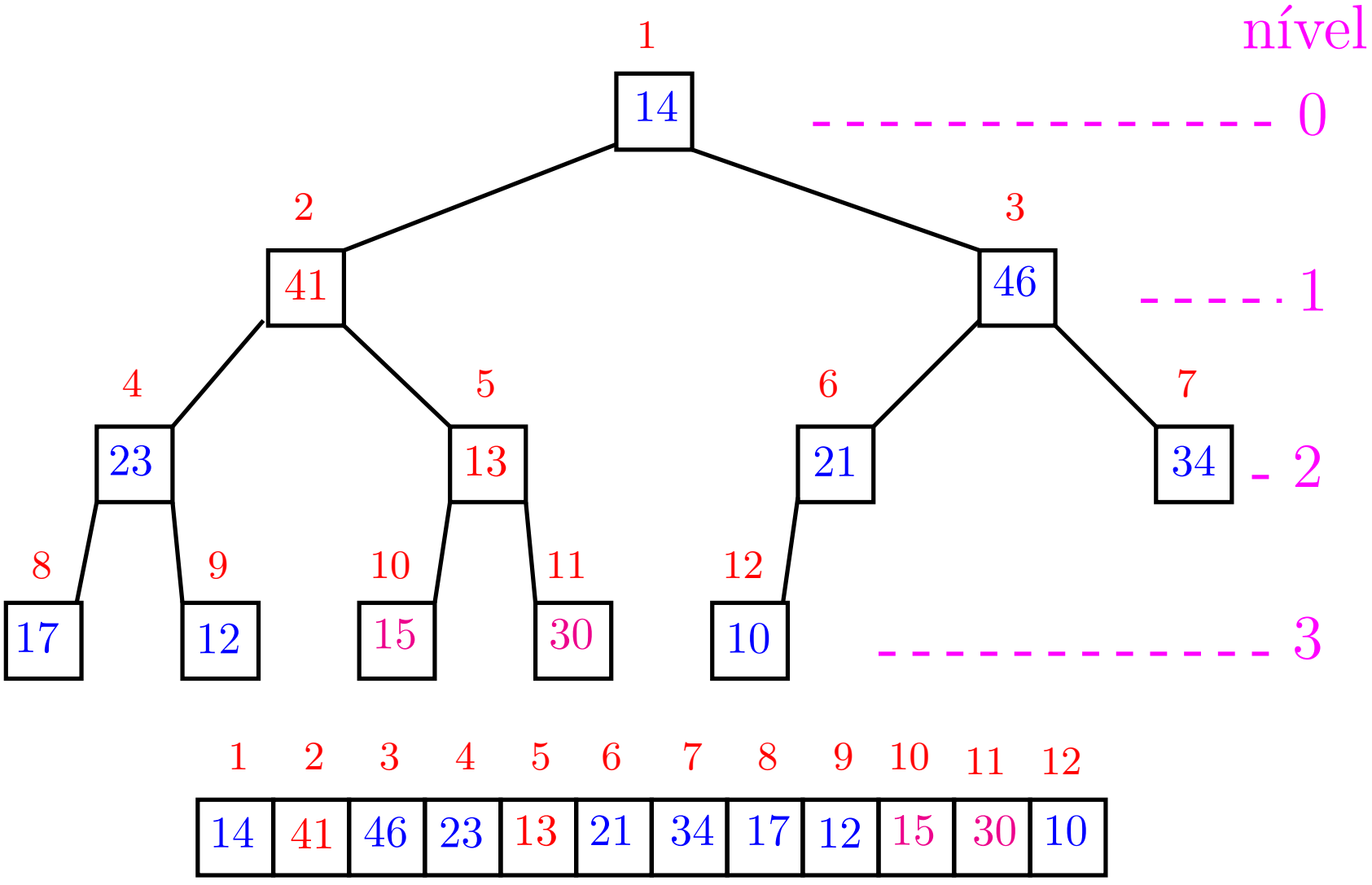
Construção de um max-heap



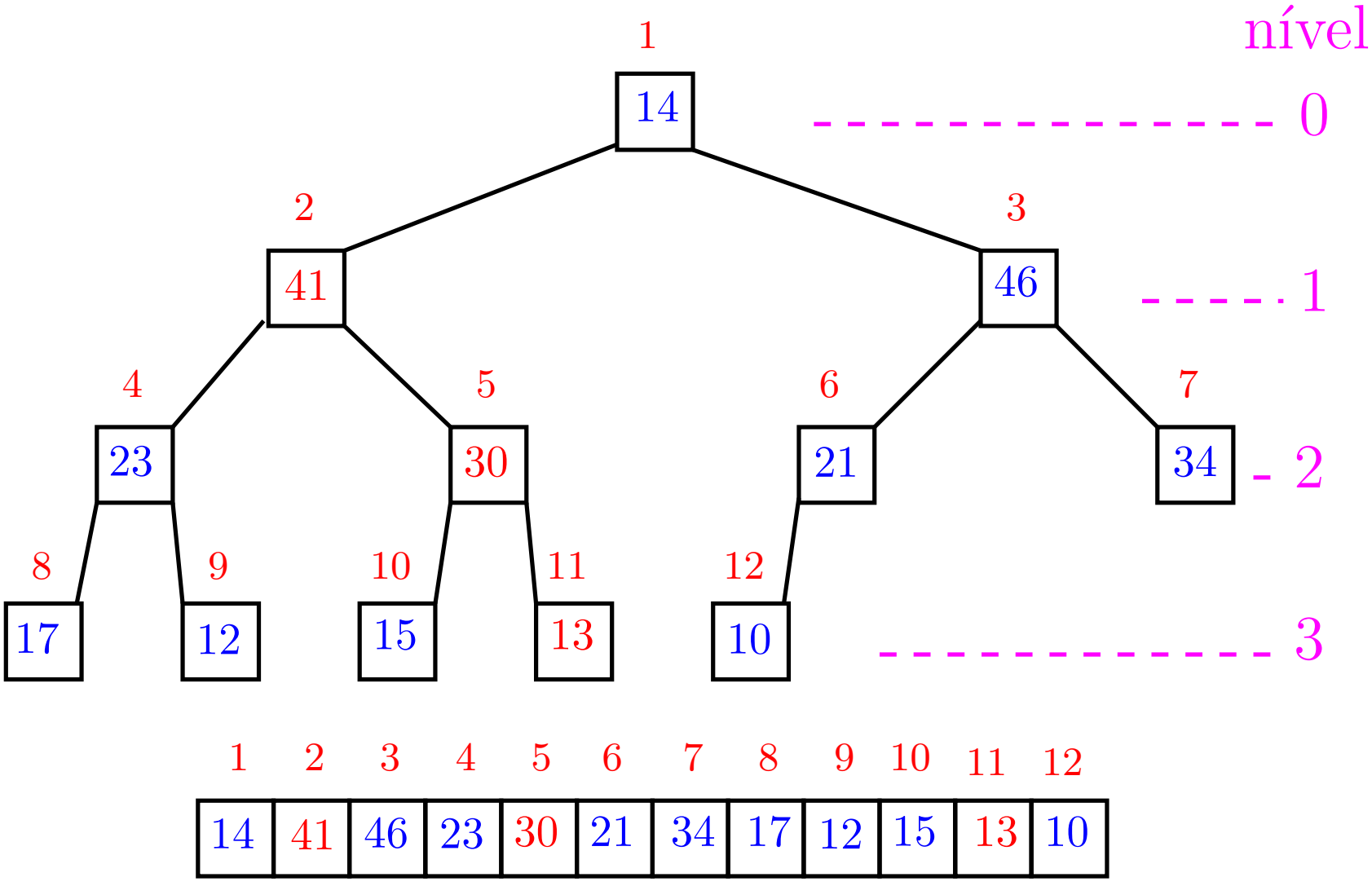
Construção de um max-heap



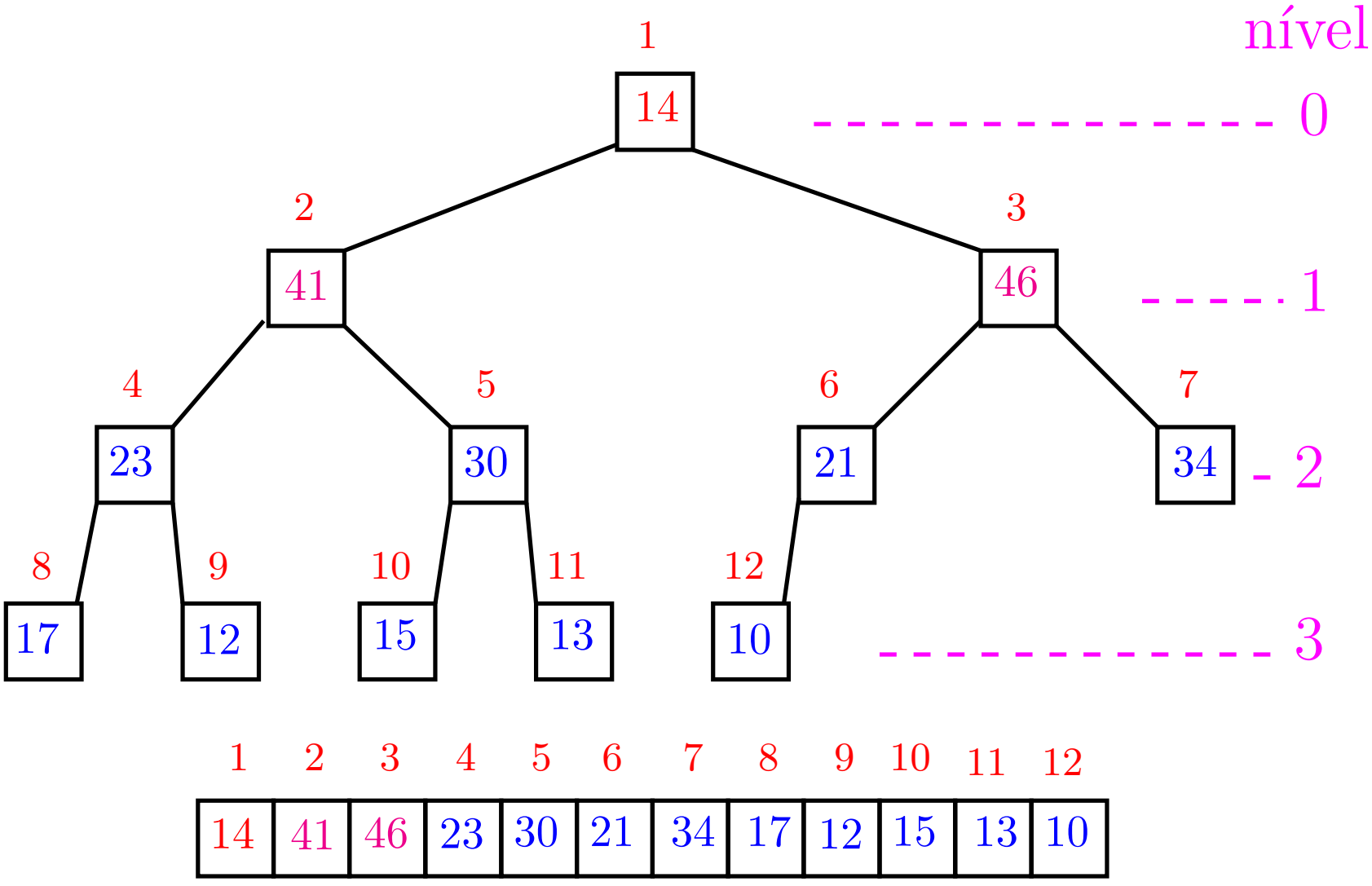
Construção de um max-heap



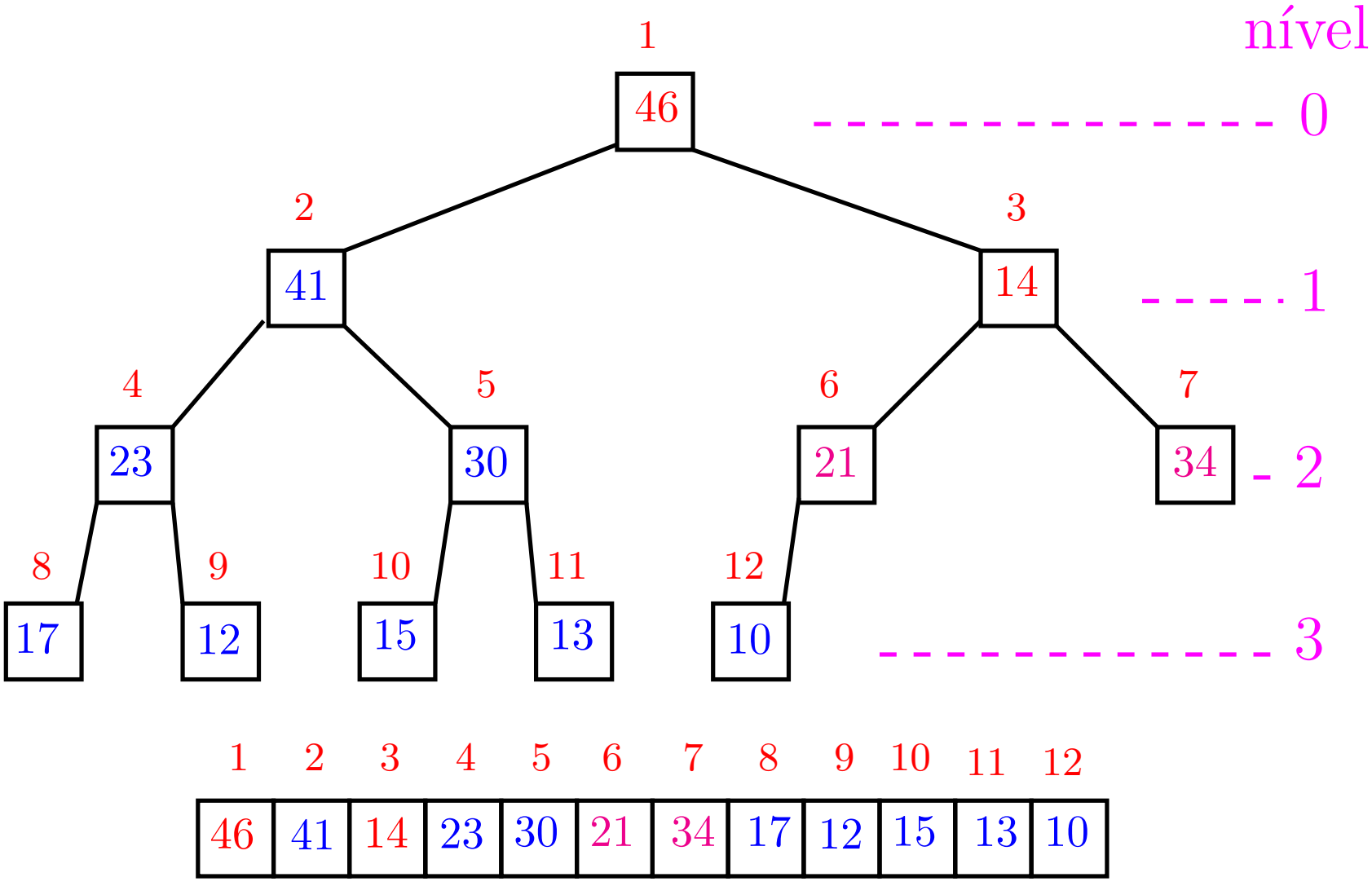
Construção de um max-heap



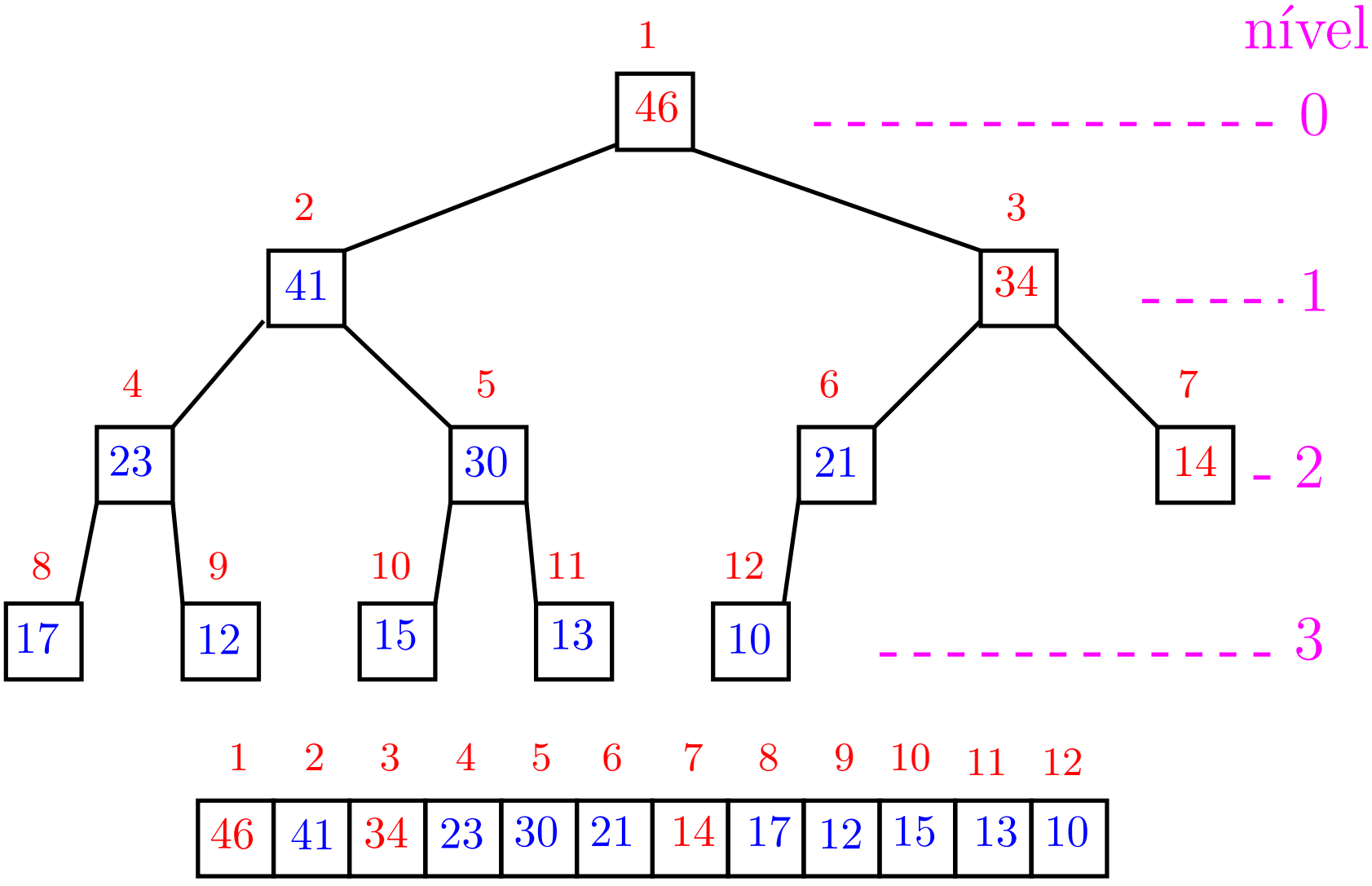
Construção de um max-heap



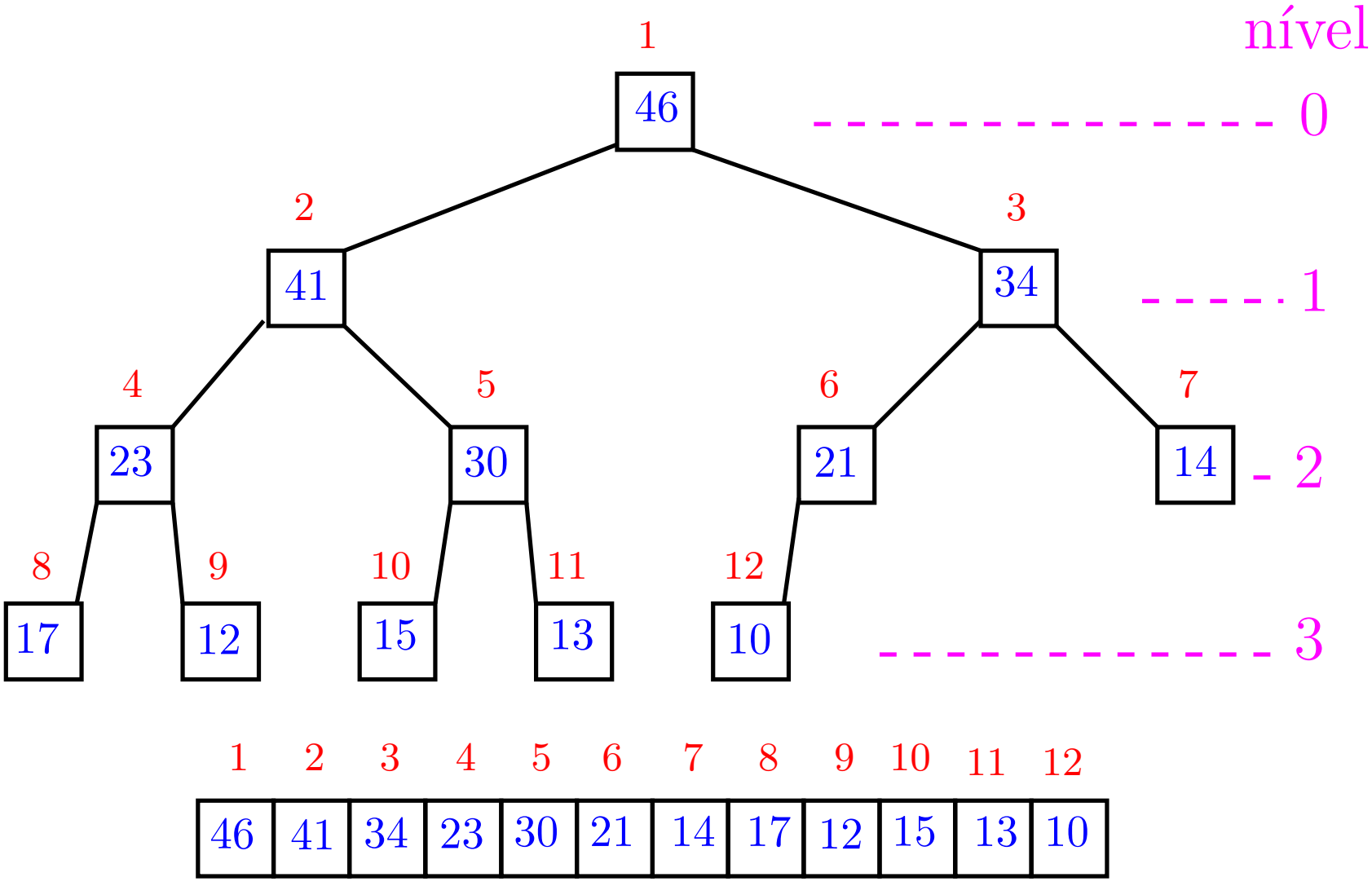
Construção de um max-heap



Construção de um max-heap



Construção de um max-heap



Construção de um max-heap

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

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

Relação invariante:

(i0) em /*A*/ vale que
$i+1, \ldots, n$ 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 $\mathrm{O}(\mathrm{n} \lg \mathrm{n})$.

Verdade seja dita . . .

O consumo de tempo para construir um `max-heap` é $\mathrm{O}(\mathrm{n})$.

Ordenação: algoritmo Heapsort

PF 10

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

Ordenação

$\mathtt{v}[1\,..\,\mathtt{n}]$ é crescente se $\mathtt{v}[1] \leq \cdots \leq \mathtt{v}[\mathtt{n}]$.

Problema: Rearranjar um vetor $\mathtt{v}[1\,..\,\mathtt{n}]$
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:

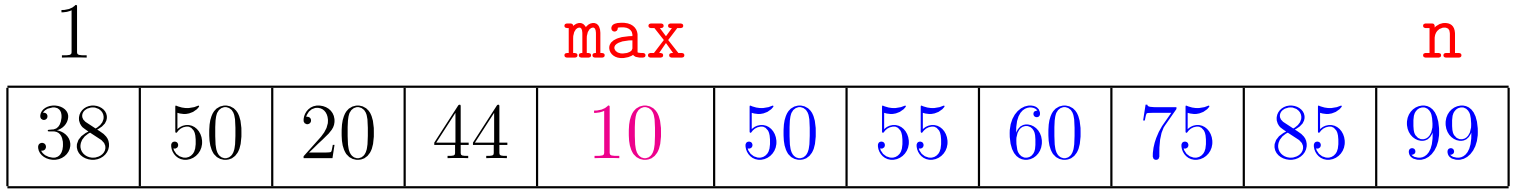
1										n
33	55	33	44	33	22	11	99	22	55	77

Sai:

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

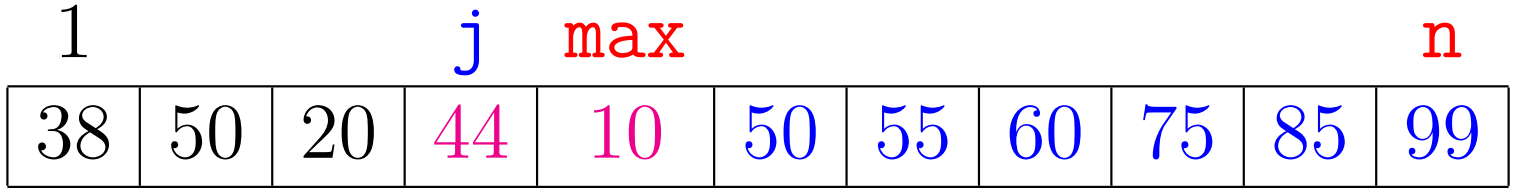
Ordenação por seleção

$\texttt{i} = 5$



Ordenação por seleção

i = 5



Ordenação por seleção

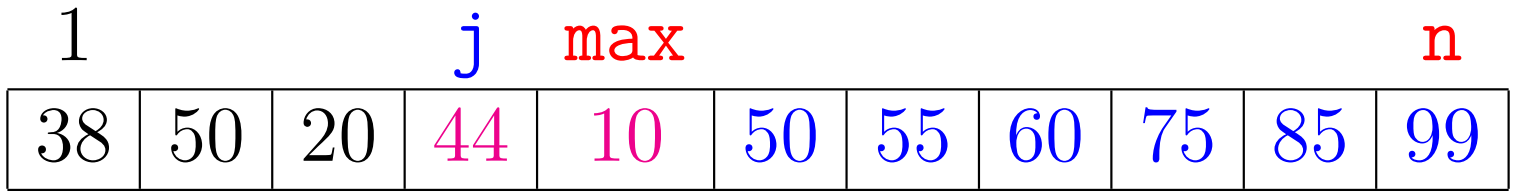
$\texttt{i} = 5$

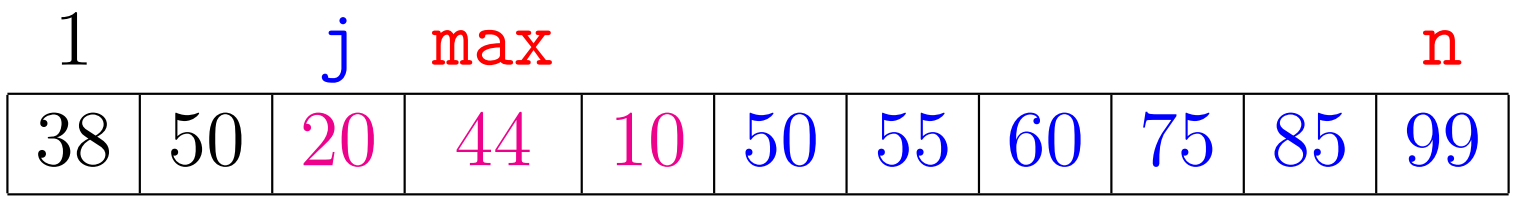
1			j	max						n
38	50	20	44	10	50	55	60	75	85	99

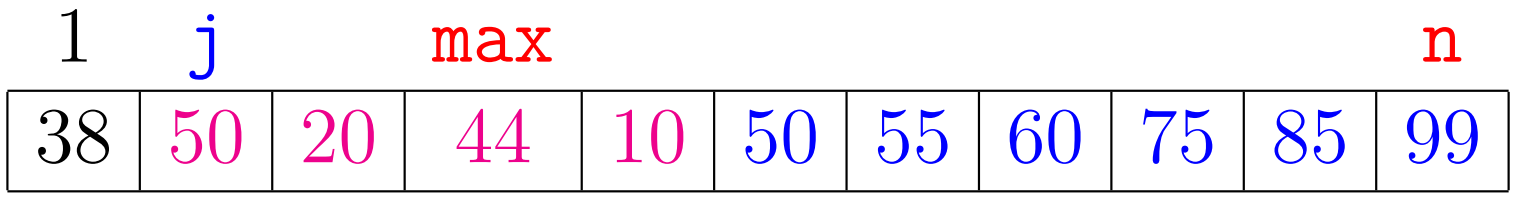
1		j	max							n
38	50	20	44	10	50	55	60	75	85	99

Ordenação por seleção

i = 5

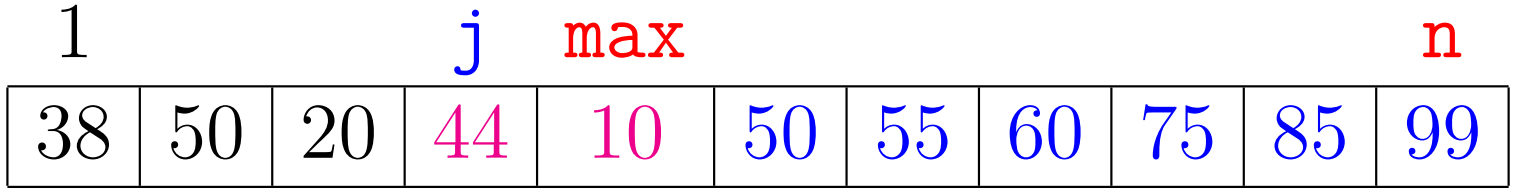


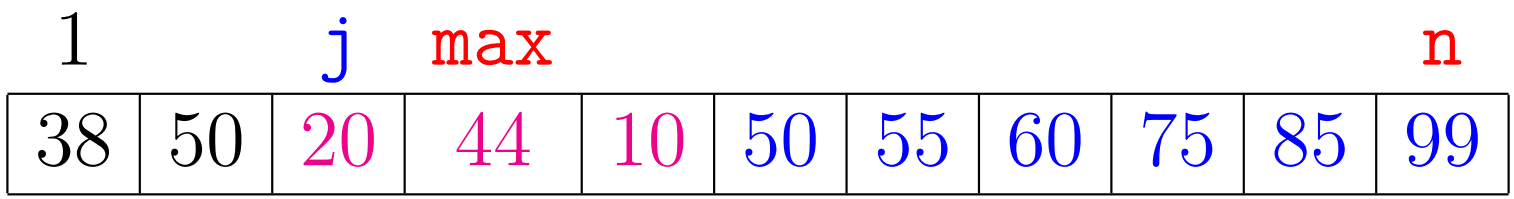


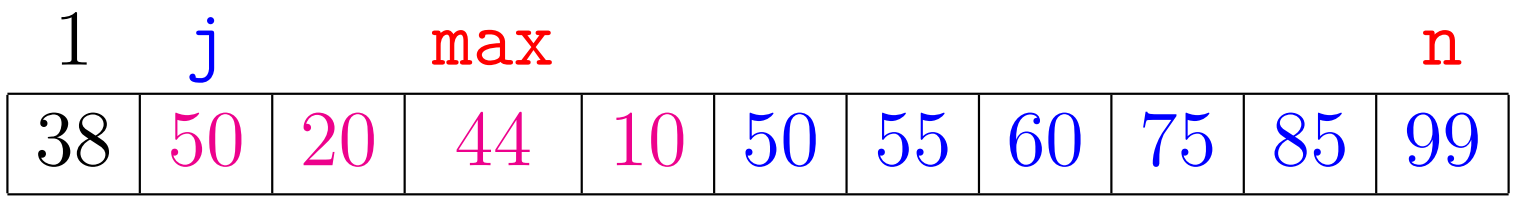


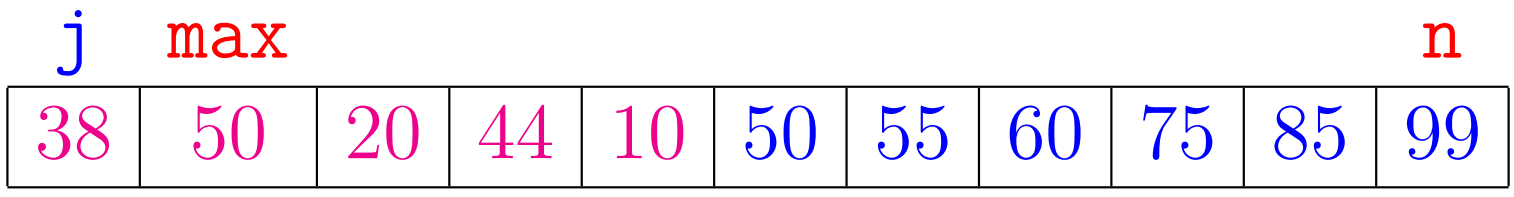
Ordenação por seleção

$\texttt{i} = 5$



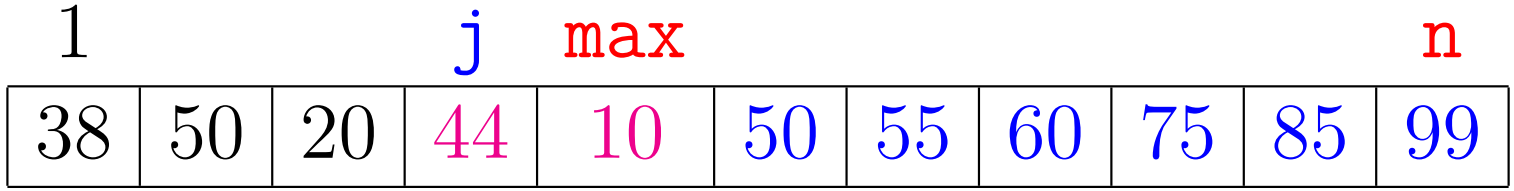


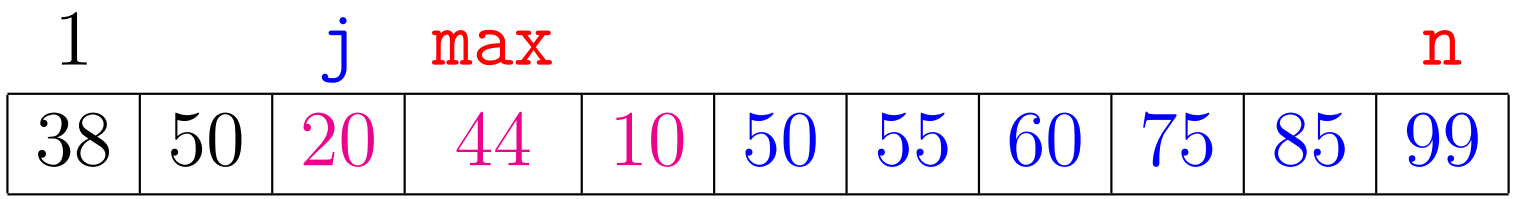


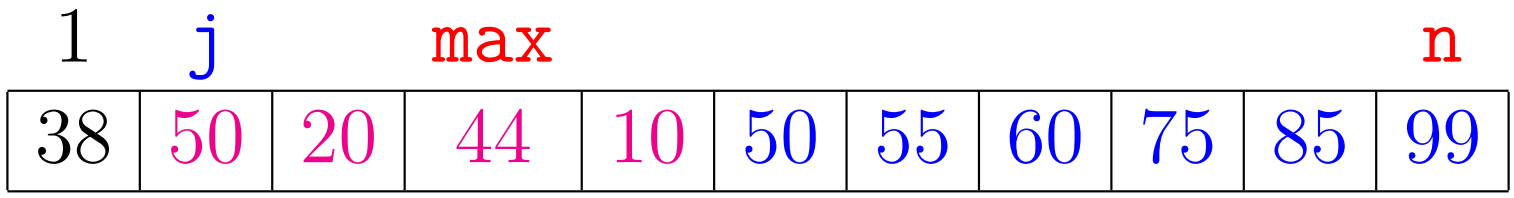


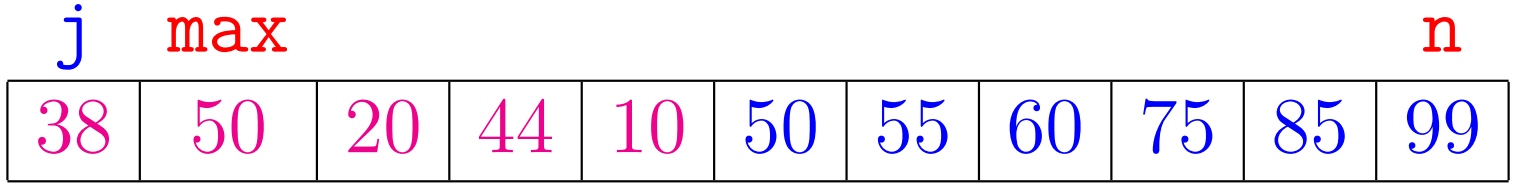
Ordenação por seleção

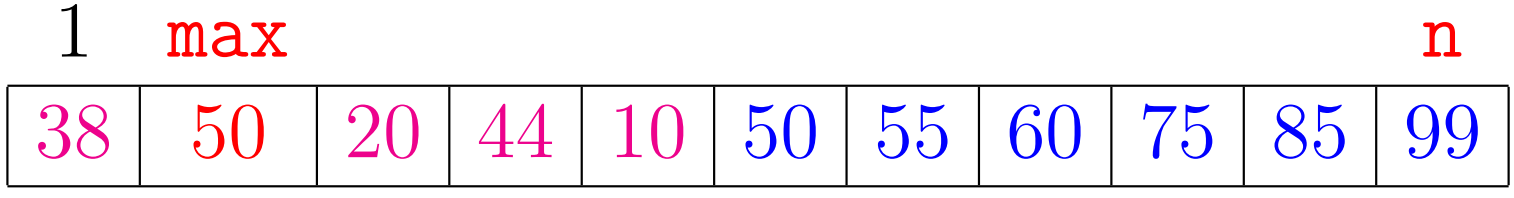
$\texttt{i} = 5$



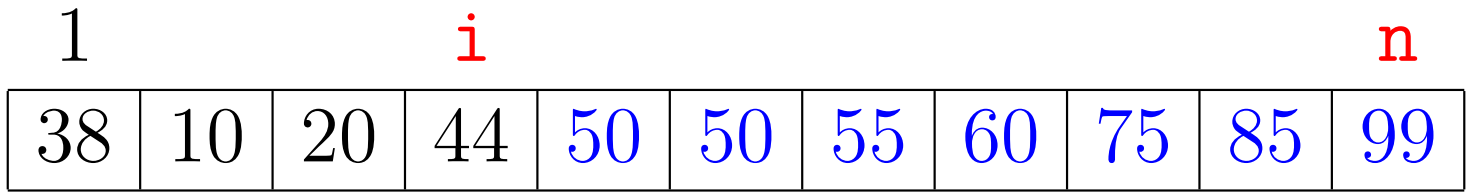




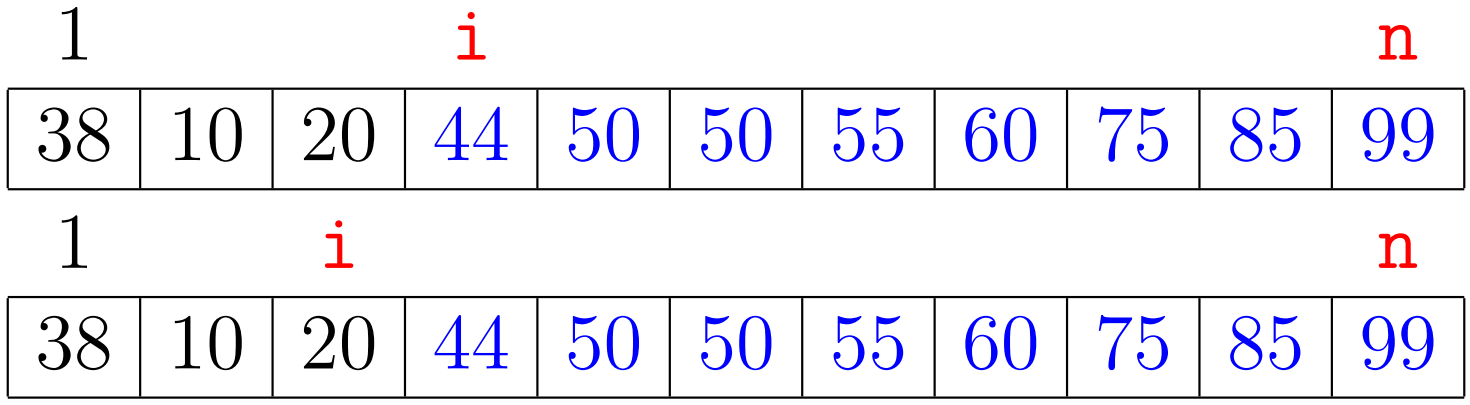




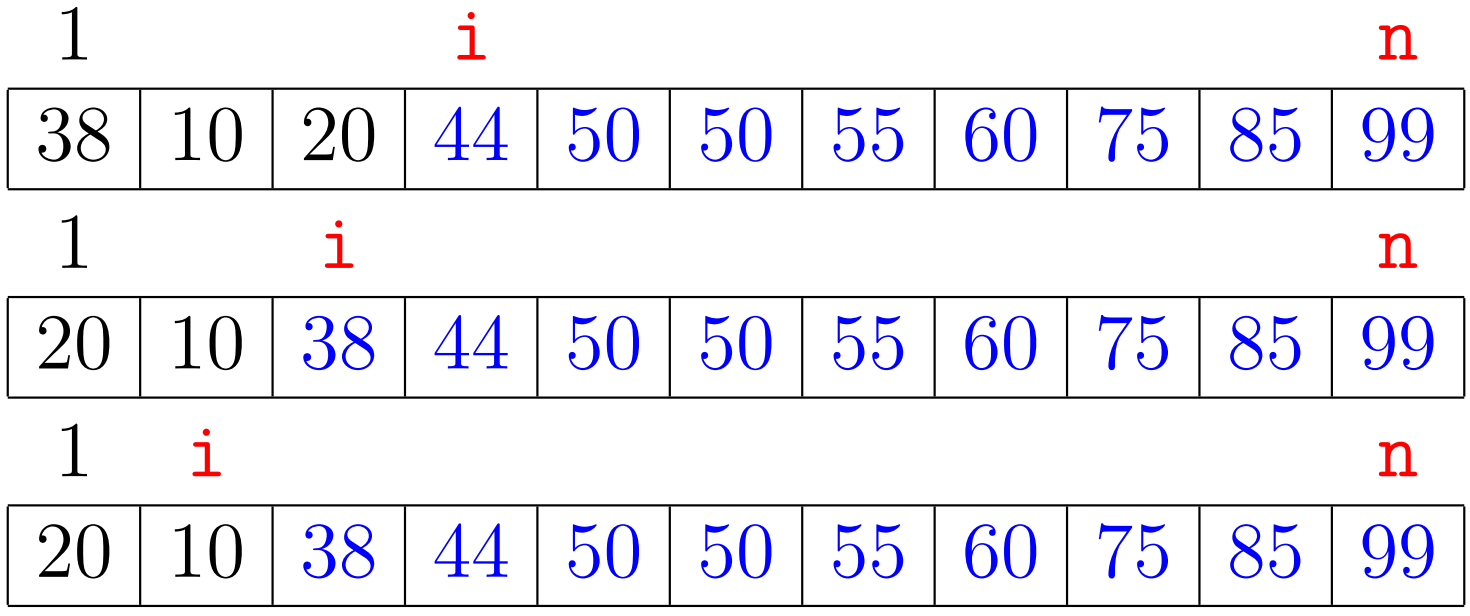
Ordenação por seleção



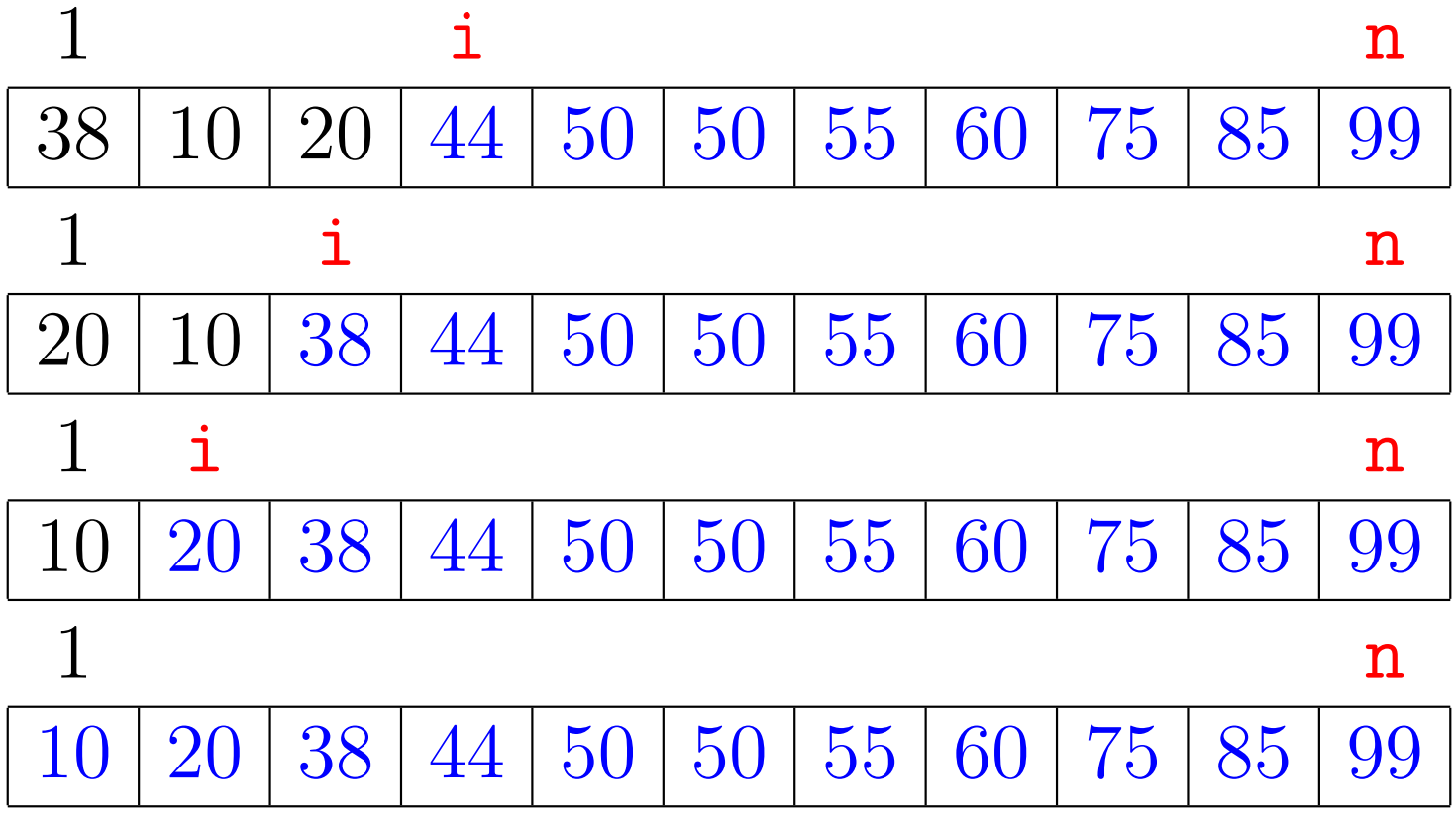
Ordenação por seleção



Ordenação por seleção



Ordenação por seleção



Função selecao

Algoritmo rearranja $v[1\,..\,n]$ em ordem crescente.

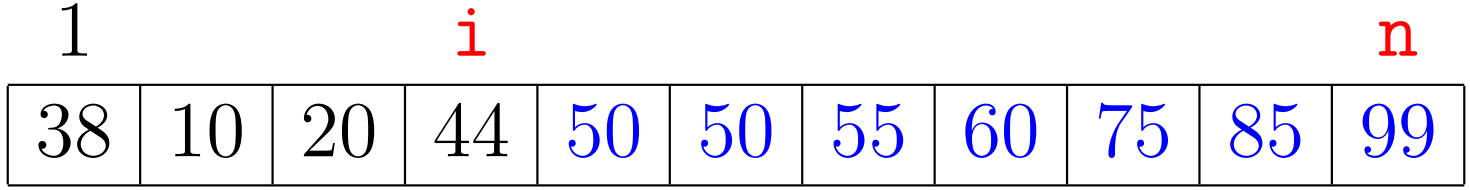
```
void selecao (int n, int v[]) {
    int i, j, max, x;
1   for (i = n; /*B*/ i > 1; i--) {
2      max = i;
3      for (j = i-1; j >= 1; j--)
4         if (v[j] > v[max]) max = j;
5      x=v[i]; v[i]=v[max]; v[max]=x;
    }
}
```

Função `selecao`

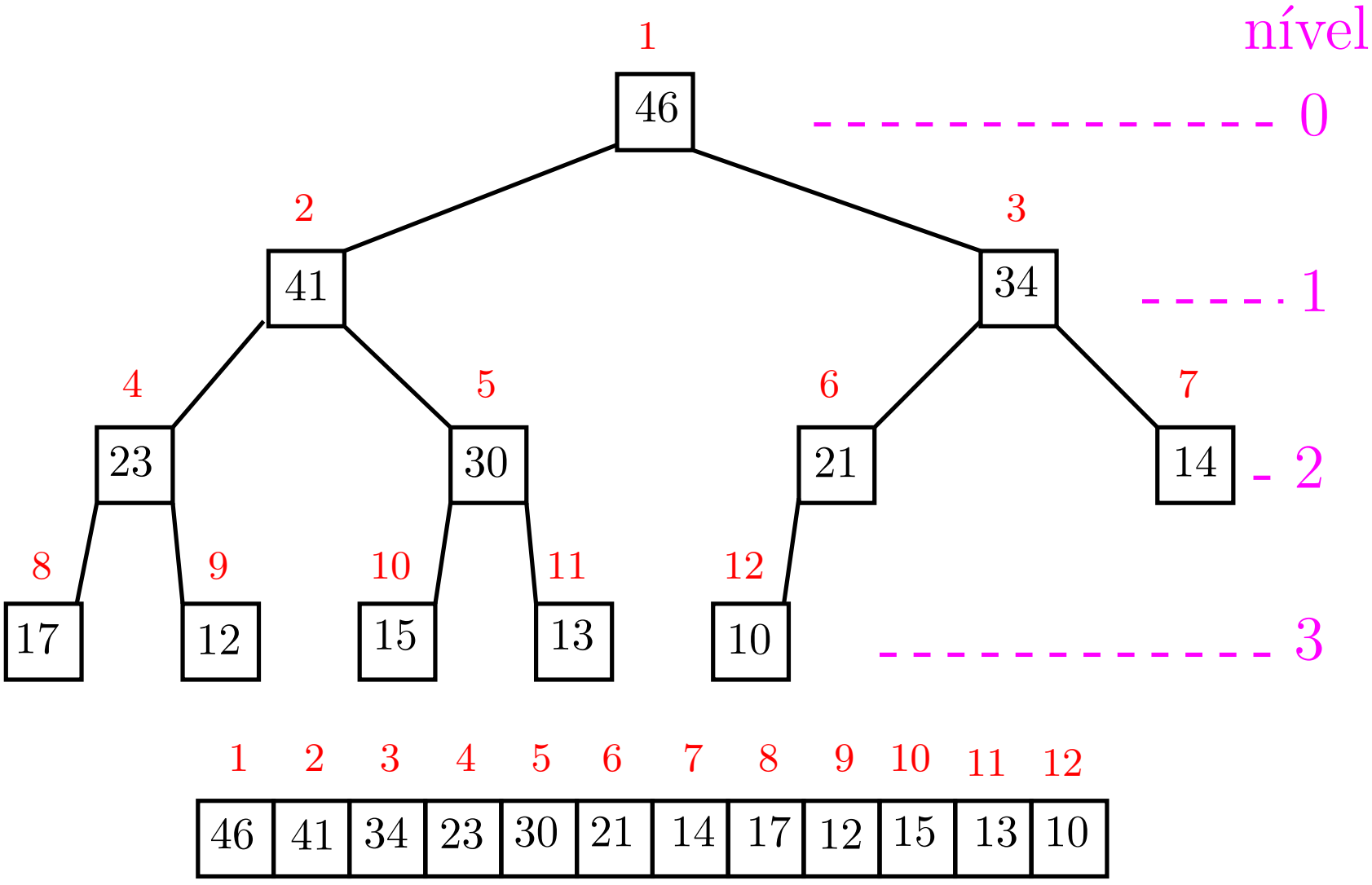
Relações invariantes: Em `/*B*/` vale que:

(i0) $\texttt{v}[\texttt{i}+1\,..\,\texttt{n}]$ é crescente;

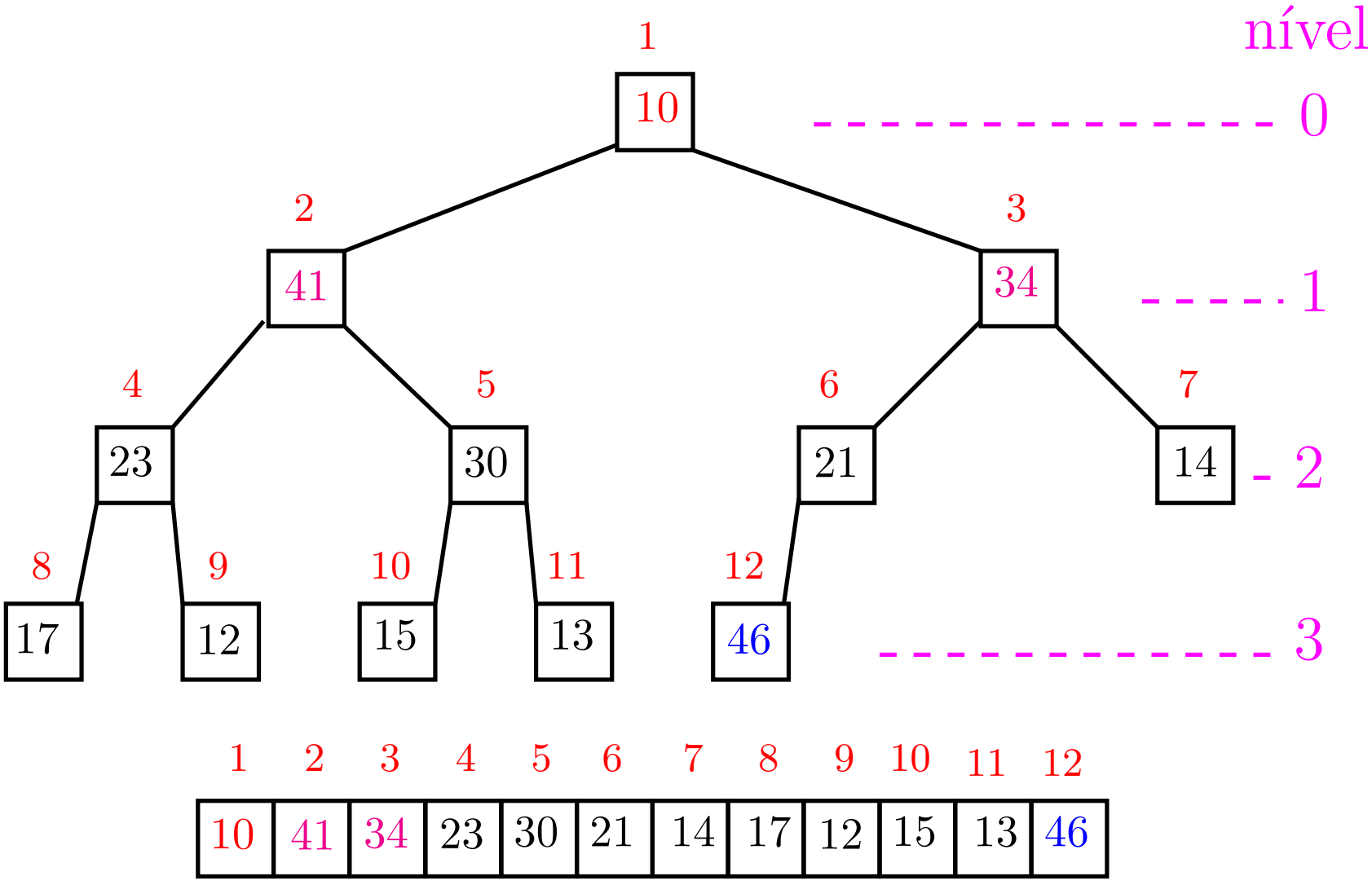
(i1) $\texttt{v}[1\,..\,\texttt{i}] \leq \texttt{v}[\texttt{i}+1]$;



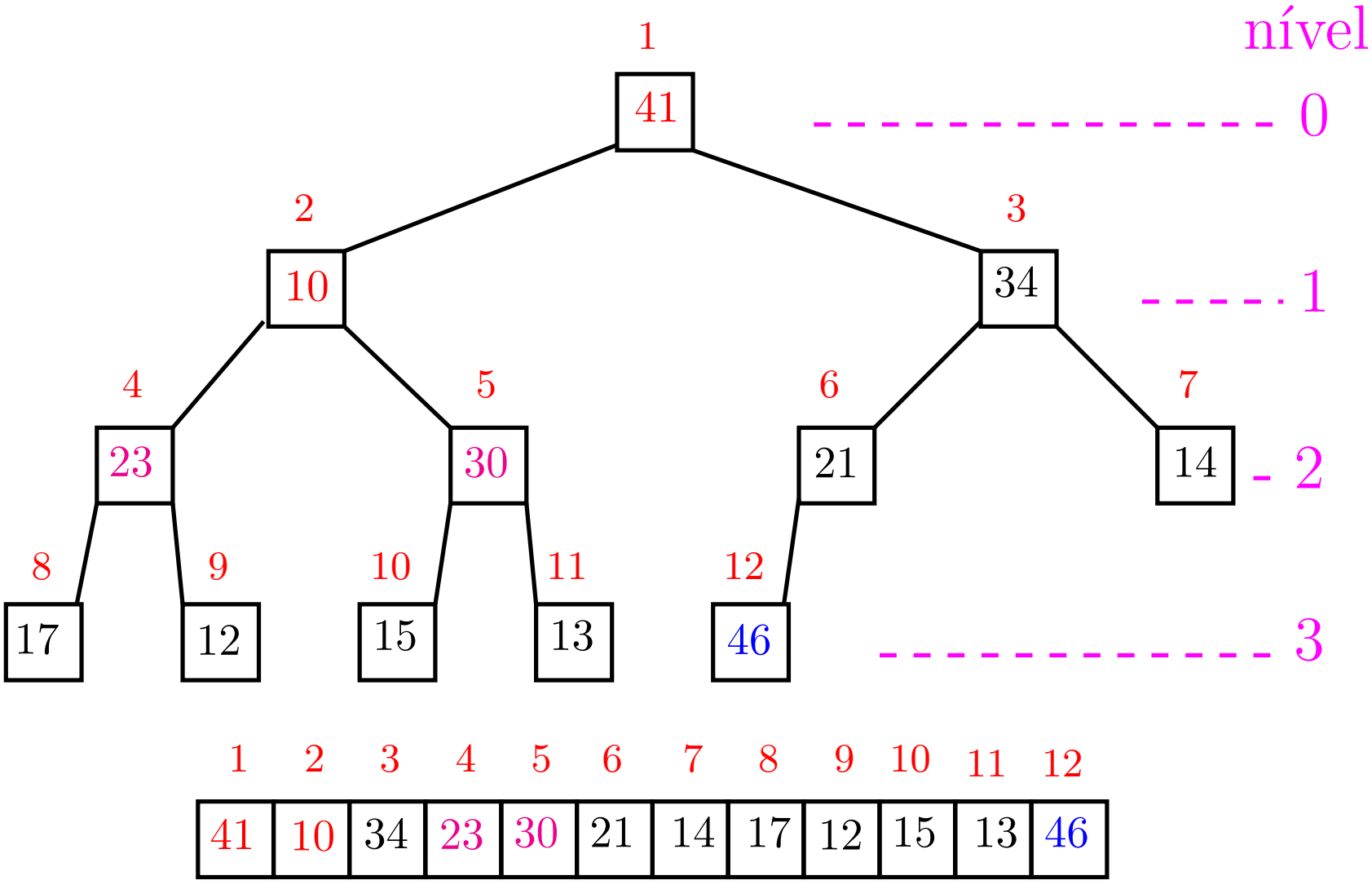
Heapsort


nível
1
46
0
2
41
3
34
1
4
23
5
30
6
21
7
14
2
8
17
9
12
10
15
11
13
12
10
3
1 2 3 4 5 6 7 8 9 10 11 12
46 41 34 23 30 21 14 17 12 15 13 10

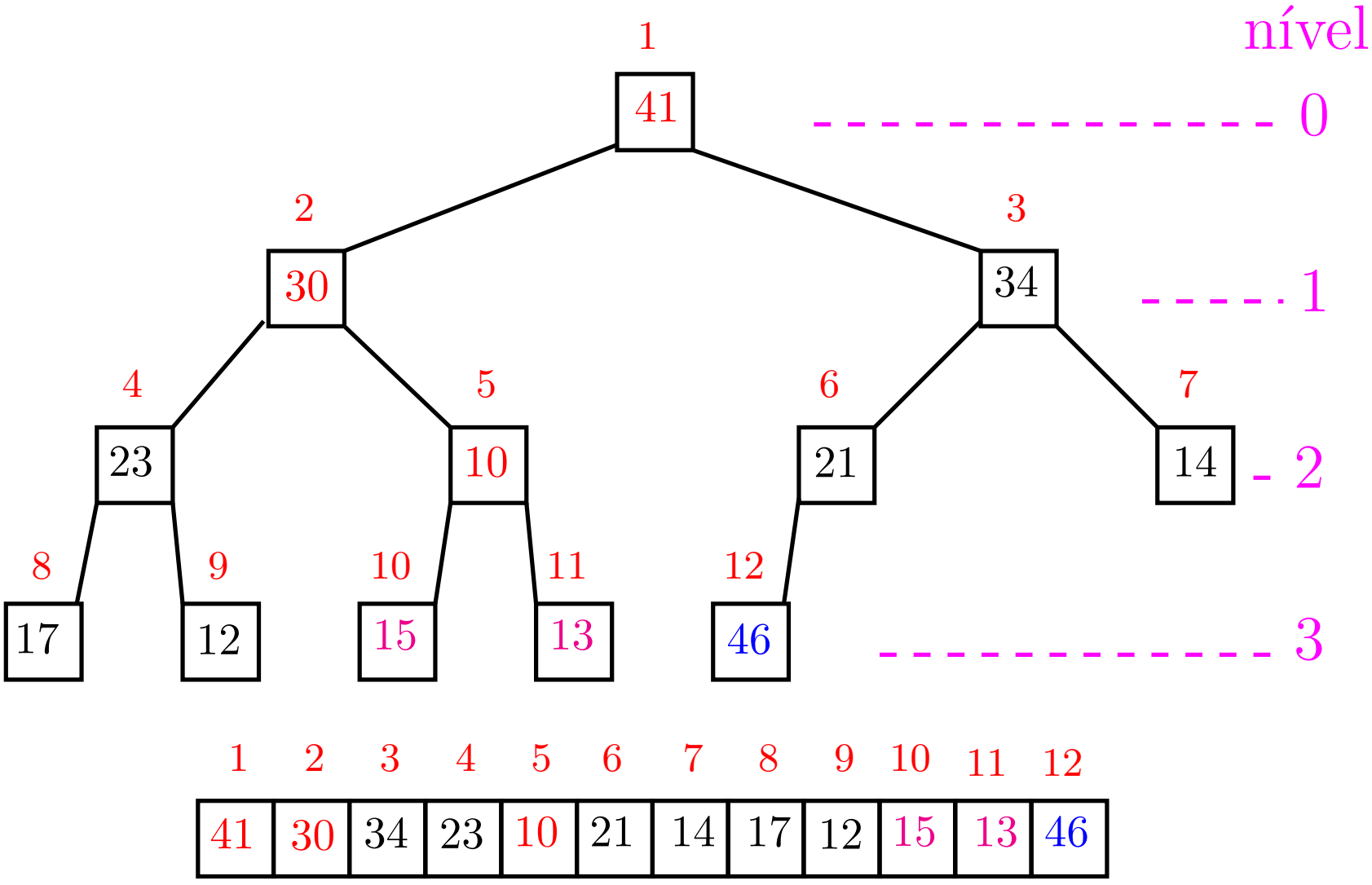
Heapsort


nível
1
10
0
2
41
3
34
1
4
23
5
30
6
21
7
14
2
8
17
9
12
10
15
11
13
12
46
3
1 2 3 4 5 6 7 8 9 10 11 12
10 41 34 23 30 21 14 17 12 15 13 46

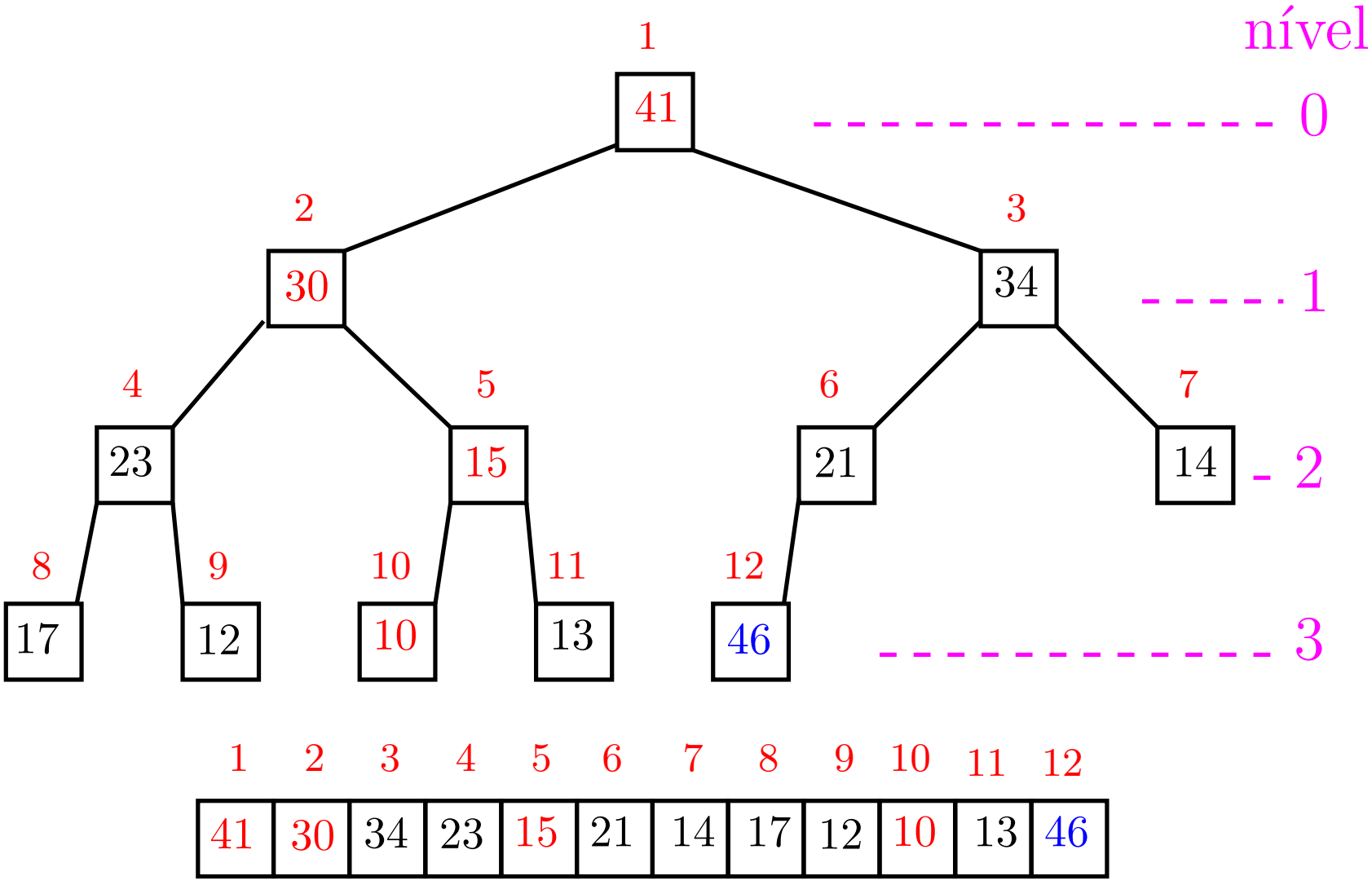
Heapsort


nível
1
41
0
2
10
3
34
1
4
23
5
30
6
21
7
14
2
8
17
9
12
10
15
11
13
12
46
3
1 2 3 4 5 6 7 8 9 10 11 12
41 10 34 23 30 21 14 17 12 15 13 46

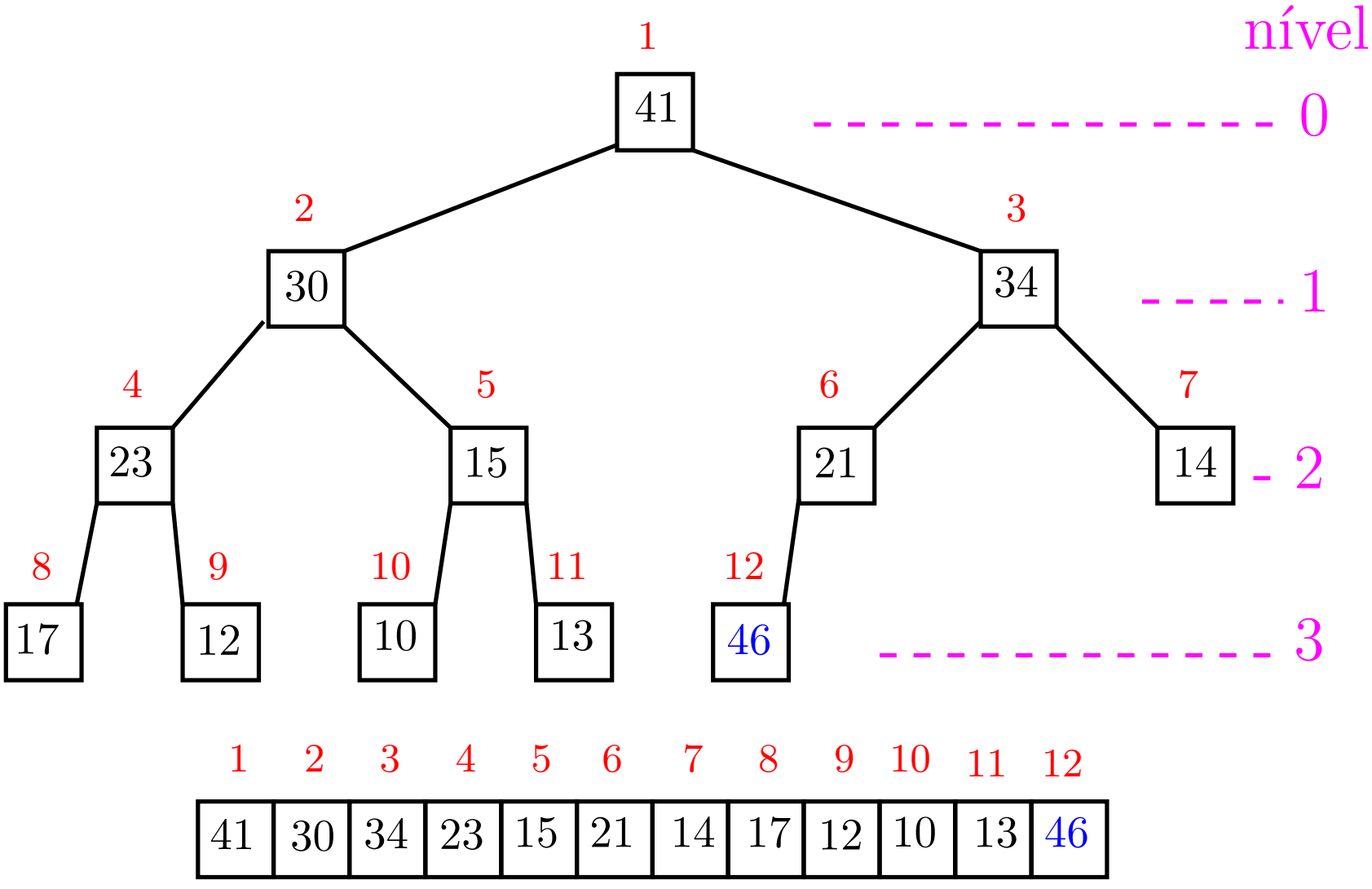
Heapsort


nível
1
41
0
2
30
3
34
1
4
23
5
10
6
21
7
14
2
8
17
9
12
10
15
11
13
12
46
3
1 2 3 4 5 6 7 8 9 10 11 12
41 30 34 23 10 21 14 17 12 15 13 46

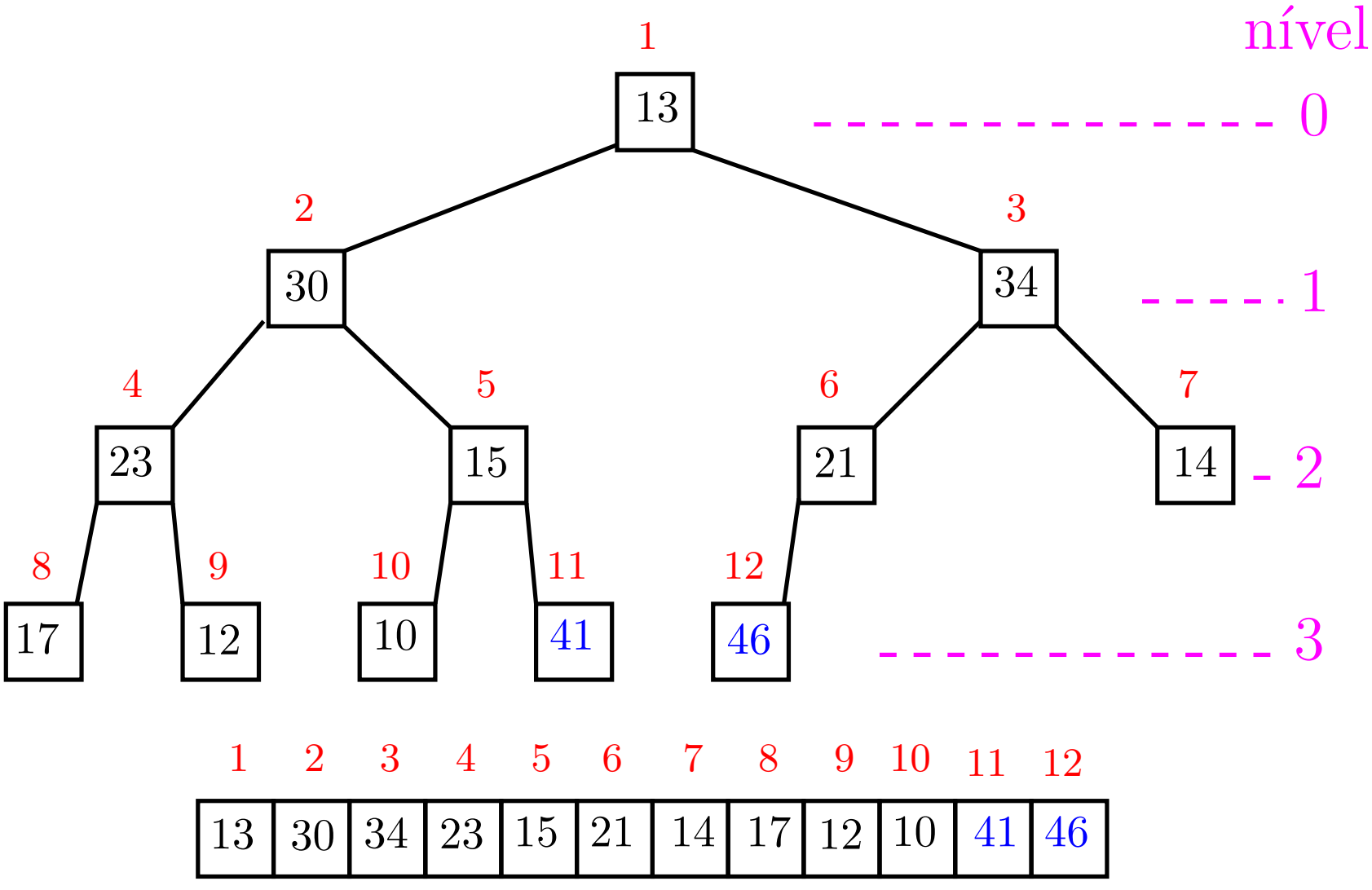
Heapsort


nível
1
41
0
2
30
3
34
1
4
23
5
15
6
21
7
14
2
8
17
9
12
10
10
11
13
12
46
3
1 2 3 4 5 6 7 8 9 10 11 12
41 30 34 23 15 21 14 17 12 10 13 46

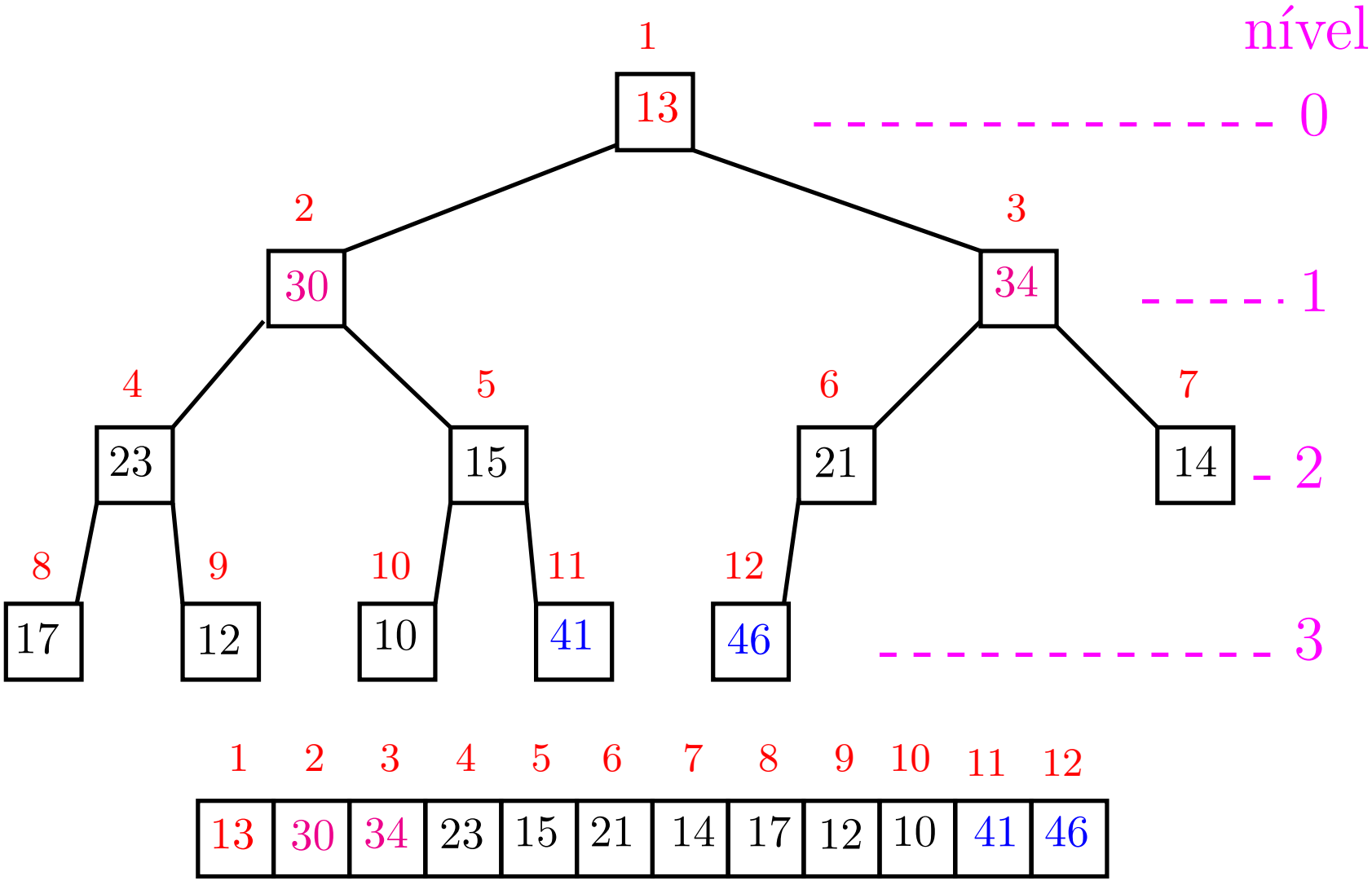
Heapsort


nível
1
41
0
2
30
3
34
1
4
23
5
15
6
21
7
14
2
8
17
9
12
10
10
11
13
12
46
3
1 2 3 4 5 6 7 8 9 10 11 12
41 30 34 23 15 21 14 17 12 10 13 46

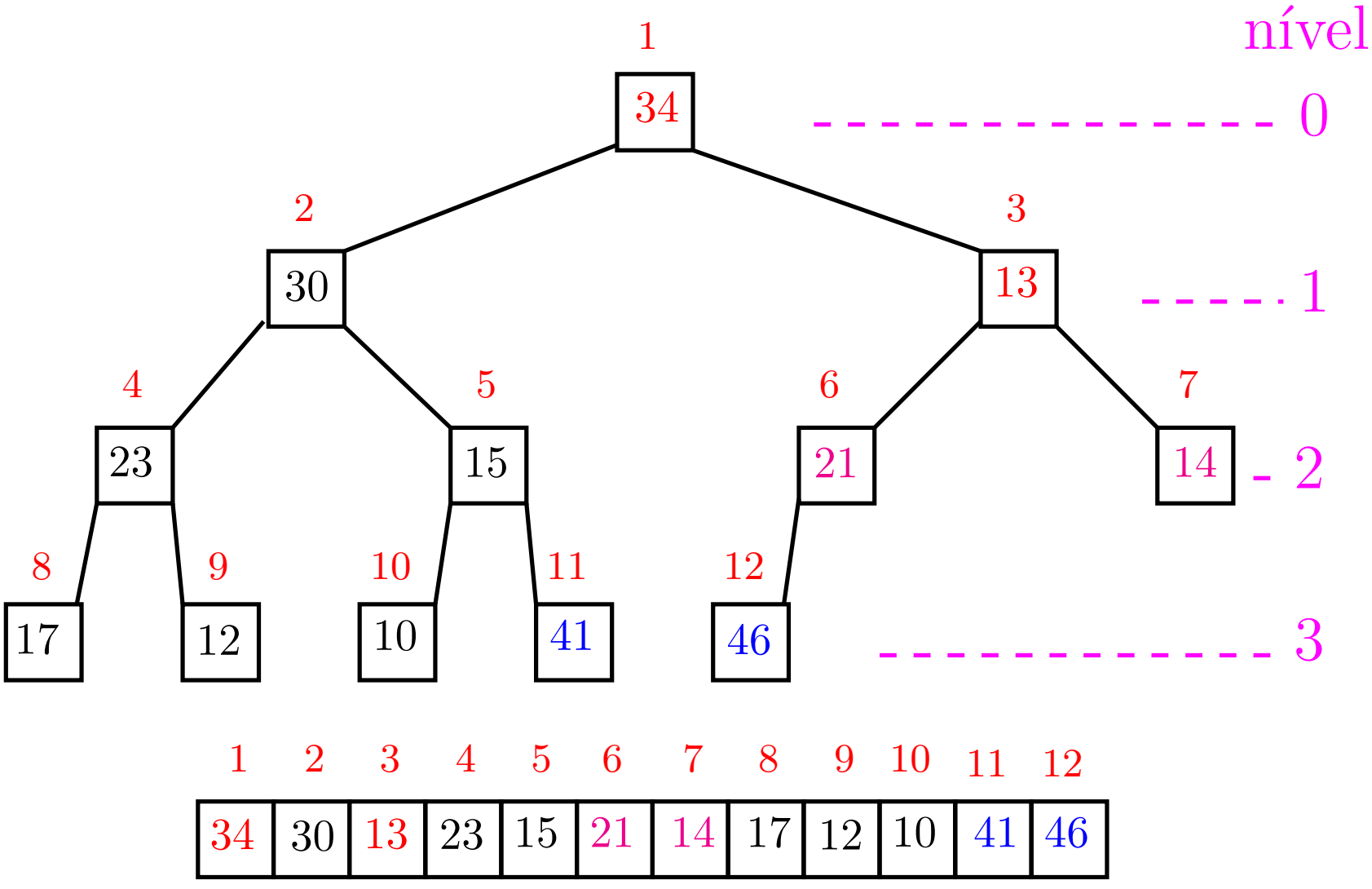
Heapsort


nível
1
13
0
2
30
3
34
1
4
23
5
15
6
21
7
14
2
8
17
9
12
10
10
11
41
12
46
3
1 2 3 4 5 6 7 8 9 10 11 12
13 30 34 23 15 21 14 17 12 10 41 46

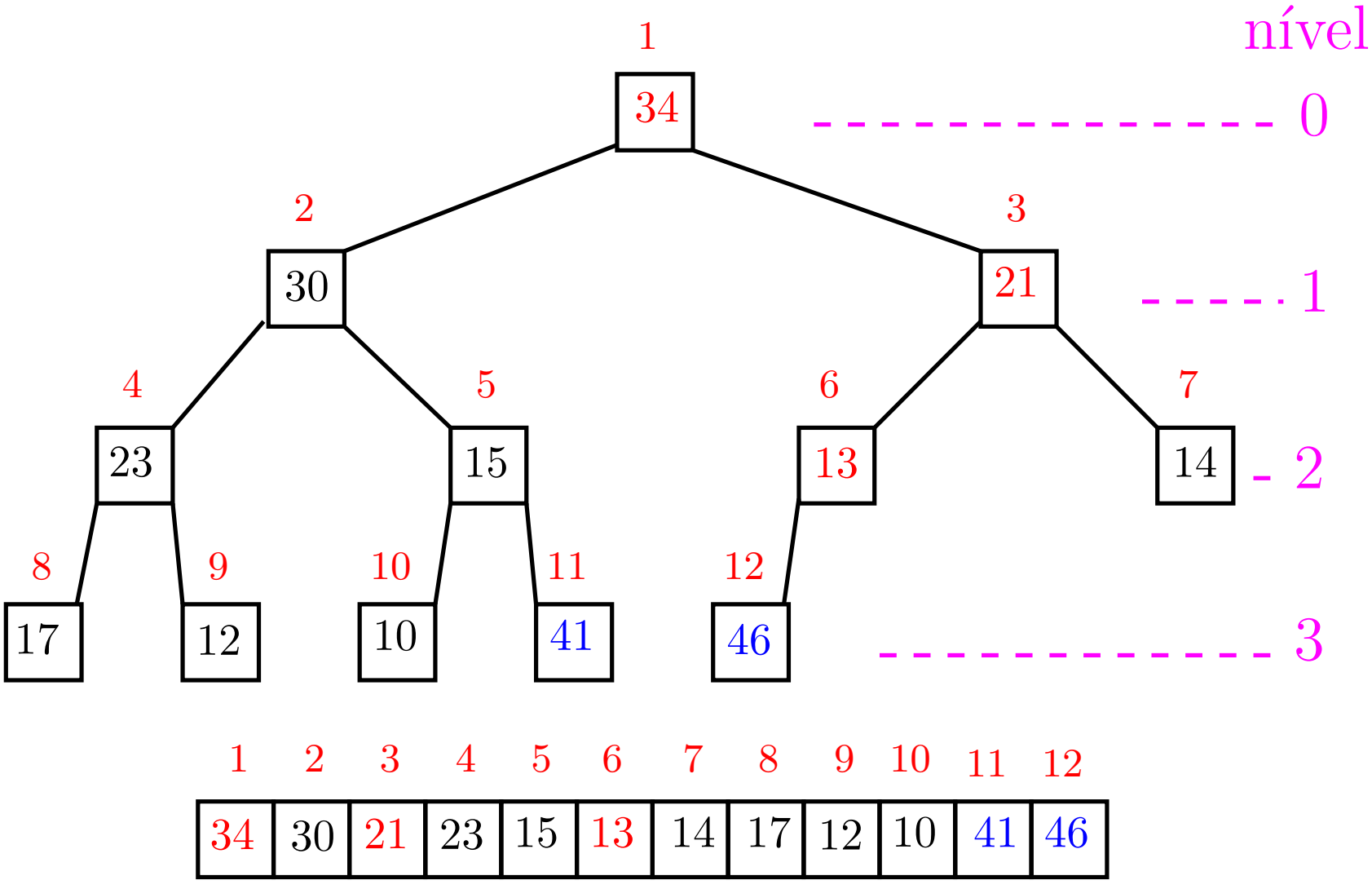
Heapsort


nível
1
13
0
2
30
3
34
1
4
23
5
15
6
21
7
14
2
8
17
9
12
10
10
11
41
12
46
3
1 2 3 4 5 6 7 8 9 10 11 12
13 30 34 23 15 21 14 17 12 10 41 46

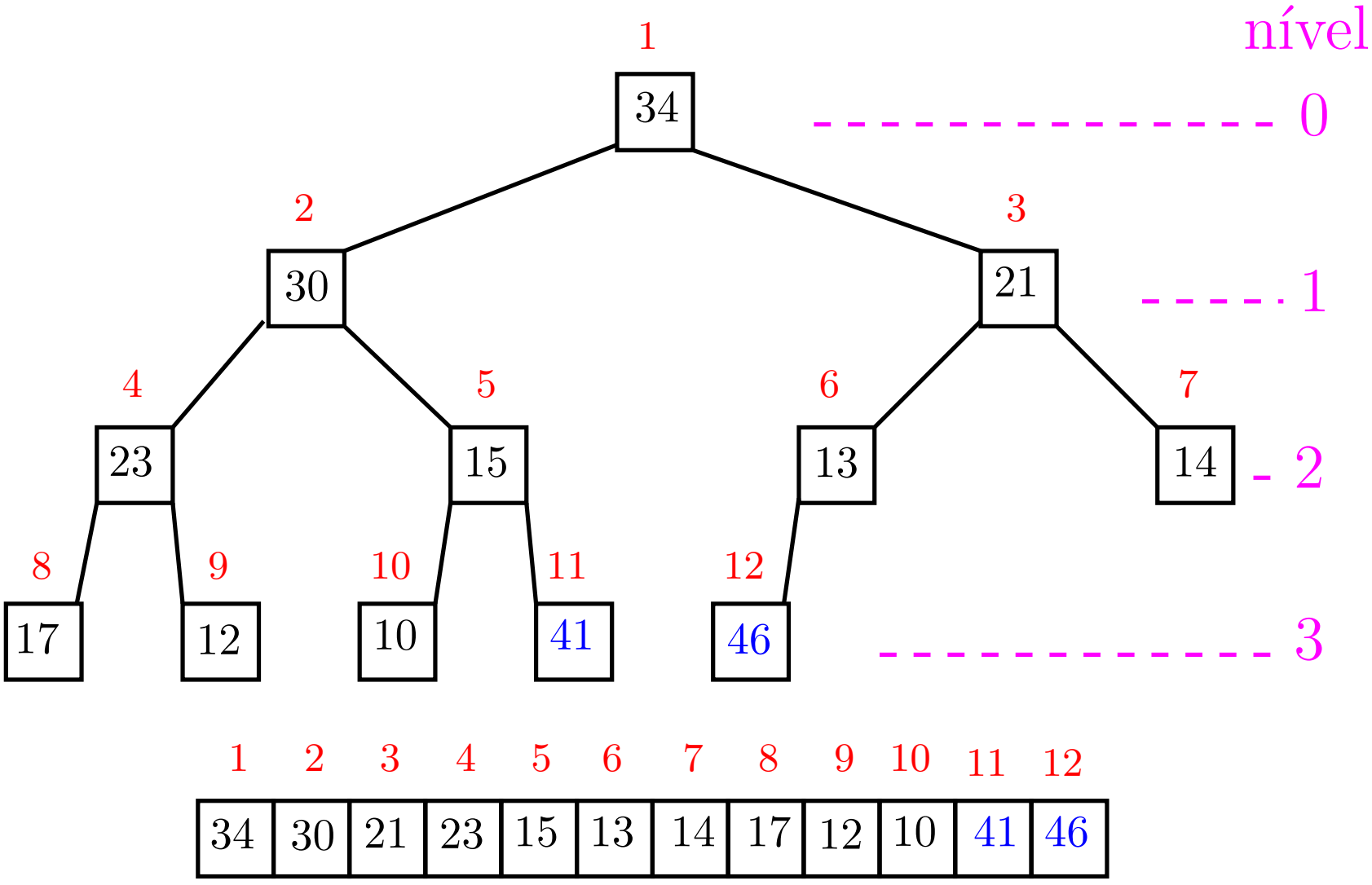
Heapsort


nível
1
34
0
2
30
3
13
1
4
23
5
15
6
21
7
14
2
8
17
9
12
10
10
11
41
12
46
3
1 2 3 4 5 6 7 8 9 10 11 12
34 30 13 23 15 21 14 17 12 10 41 46

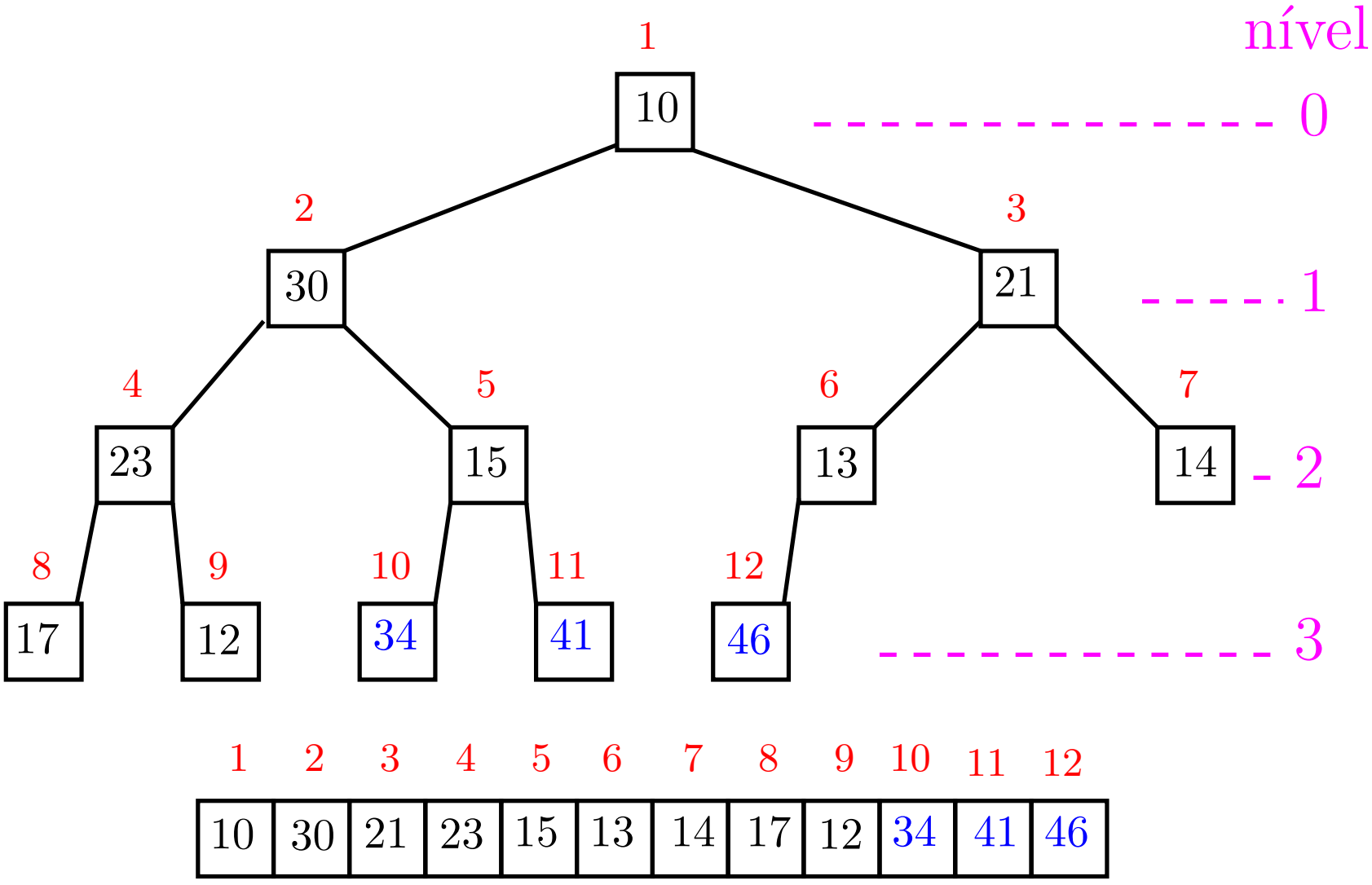
Heapsort


nível
1
34
0
2
30
3
21
1
4
23
5
15
6
13
7
14
2
8
17
9
12
10
10
11
41
12
46
3
1 2 3 4 5 6 7 8 9 10 11 12
34 30 21 23 15 13 14 17 12 10 41 46

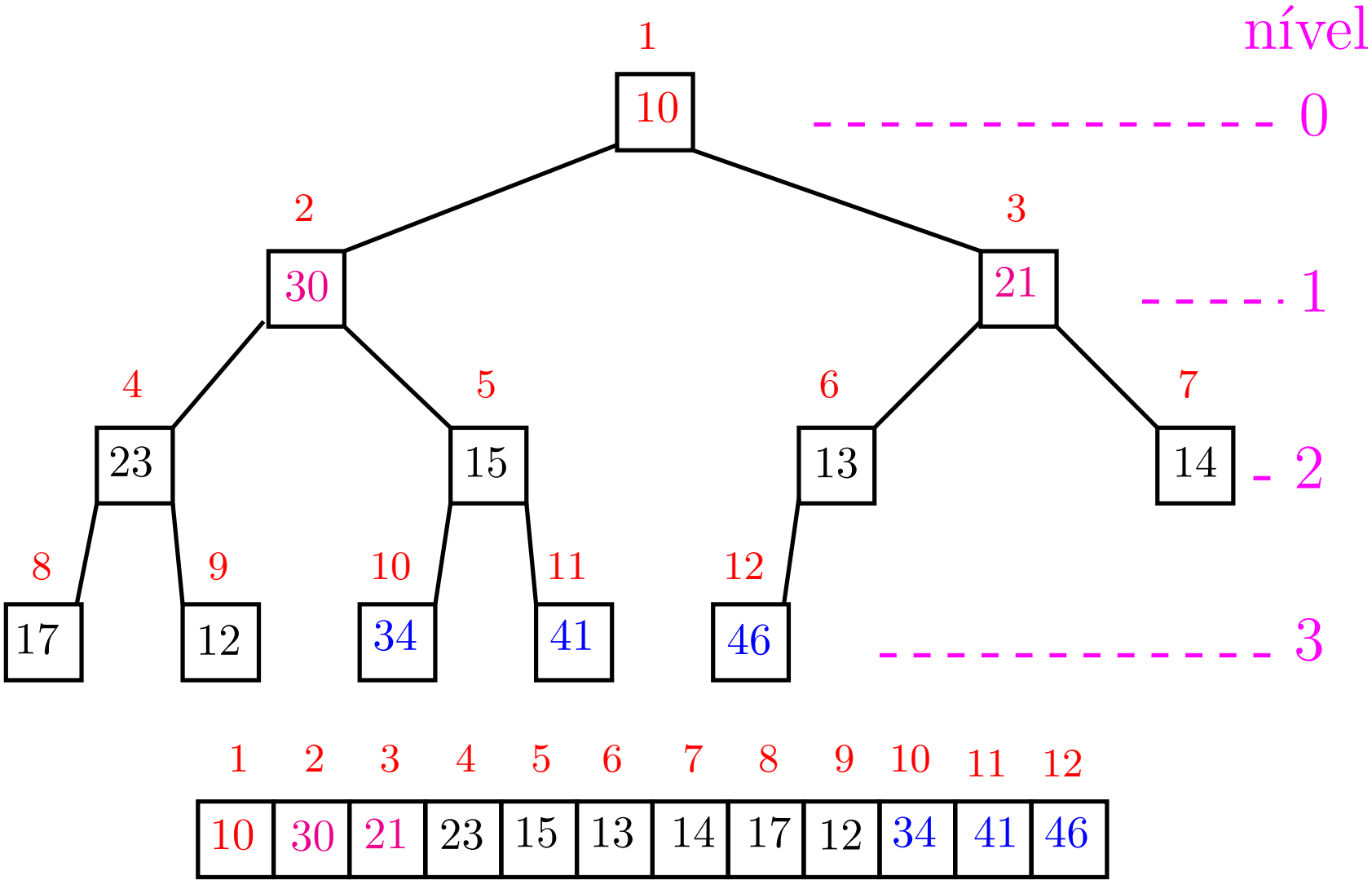
Heapsort


nível
1
34
0
2
3
30
21
1
4
5
6
7
23
15
13
14
2
8
9
10
11
12
17
12
10
41
46
3
1 2 3 4 5 6 7 8 9 10 11 12
34 30 21 23 15 13 14 17 12 10 41 46

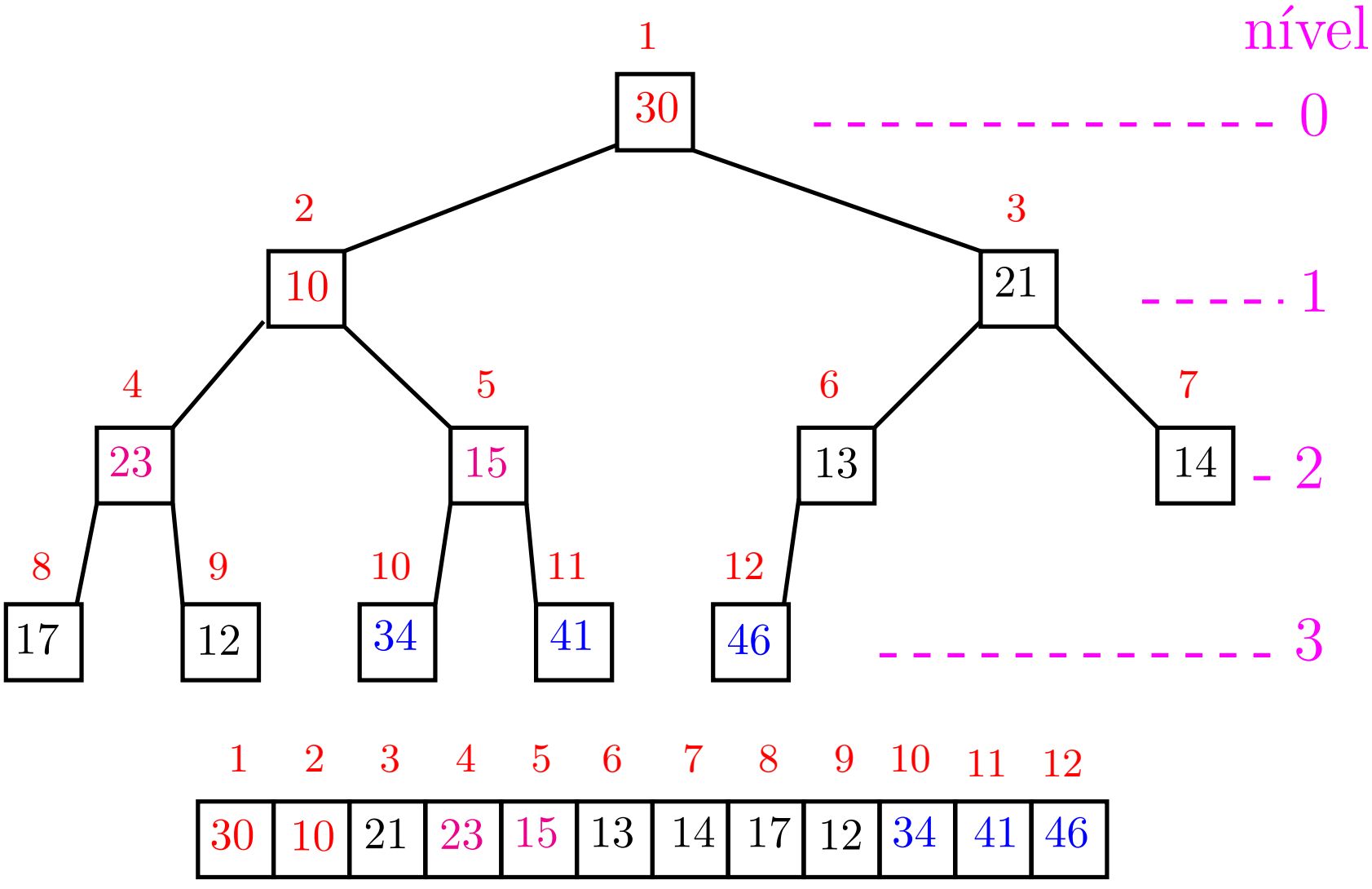
Heapsort


nível
1
10
0
2
30
3
21
1
4
23
5
15
6
13
7
14
2
8
17
9
12
10
34
11
41
12
46
3
1 2 3 4 5 6 7 8 9 10 11 12
10 30 21 23 15 13 14 17 12 34 41 46

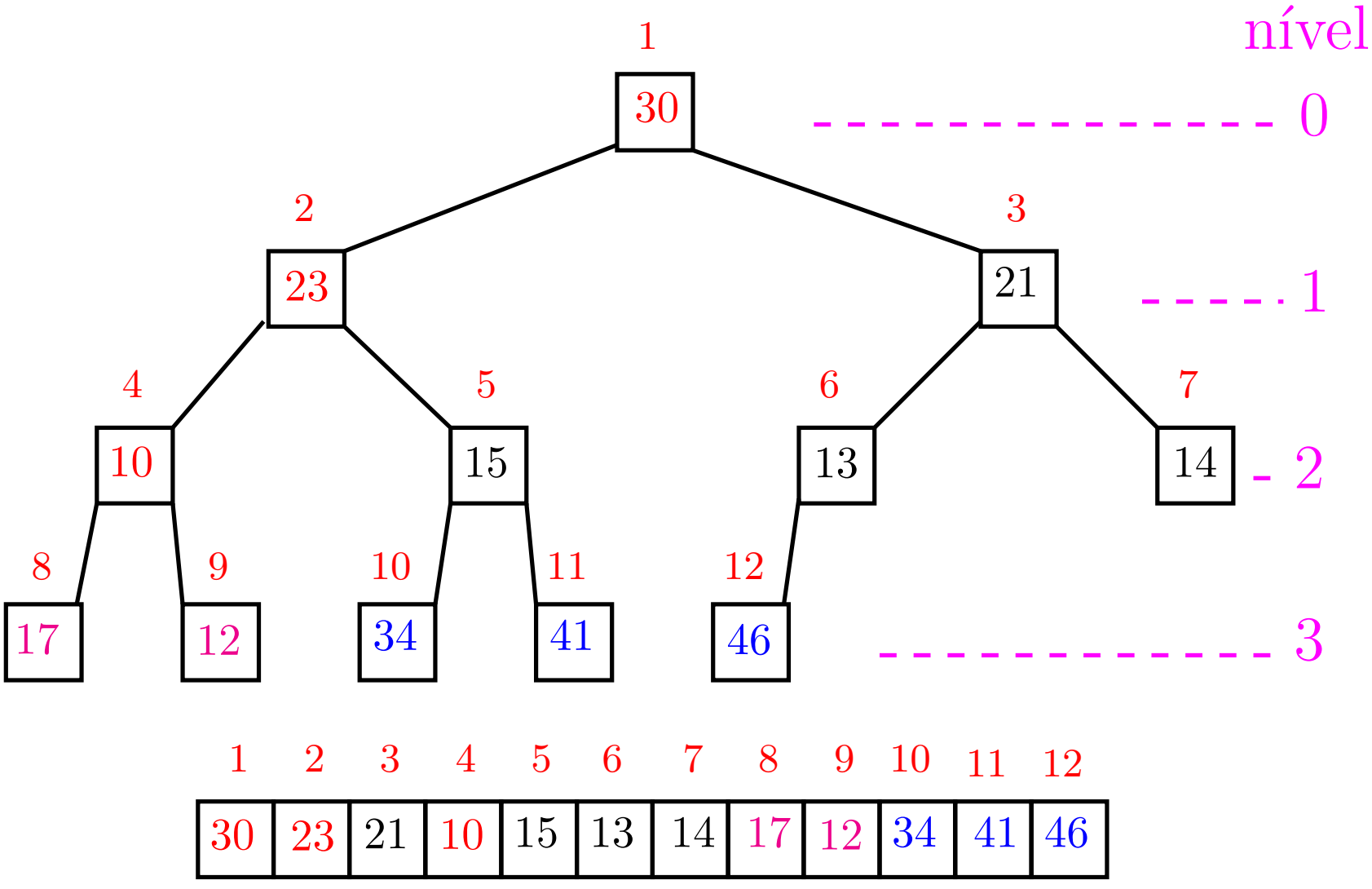
Heapsort


nível
1
10
0
2
30
3
21
1
4
23
5
15
6
13
7
14
2
8
17
9
12
10
34
11
41
12
46
3
1 2 3 4 5 6 7 8 9 10 11 12
10 30 21 23 15 13 14 17 12 34 41 46

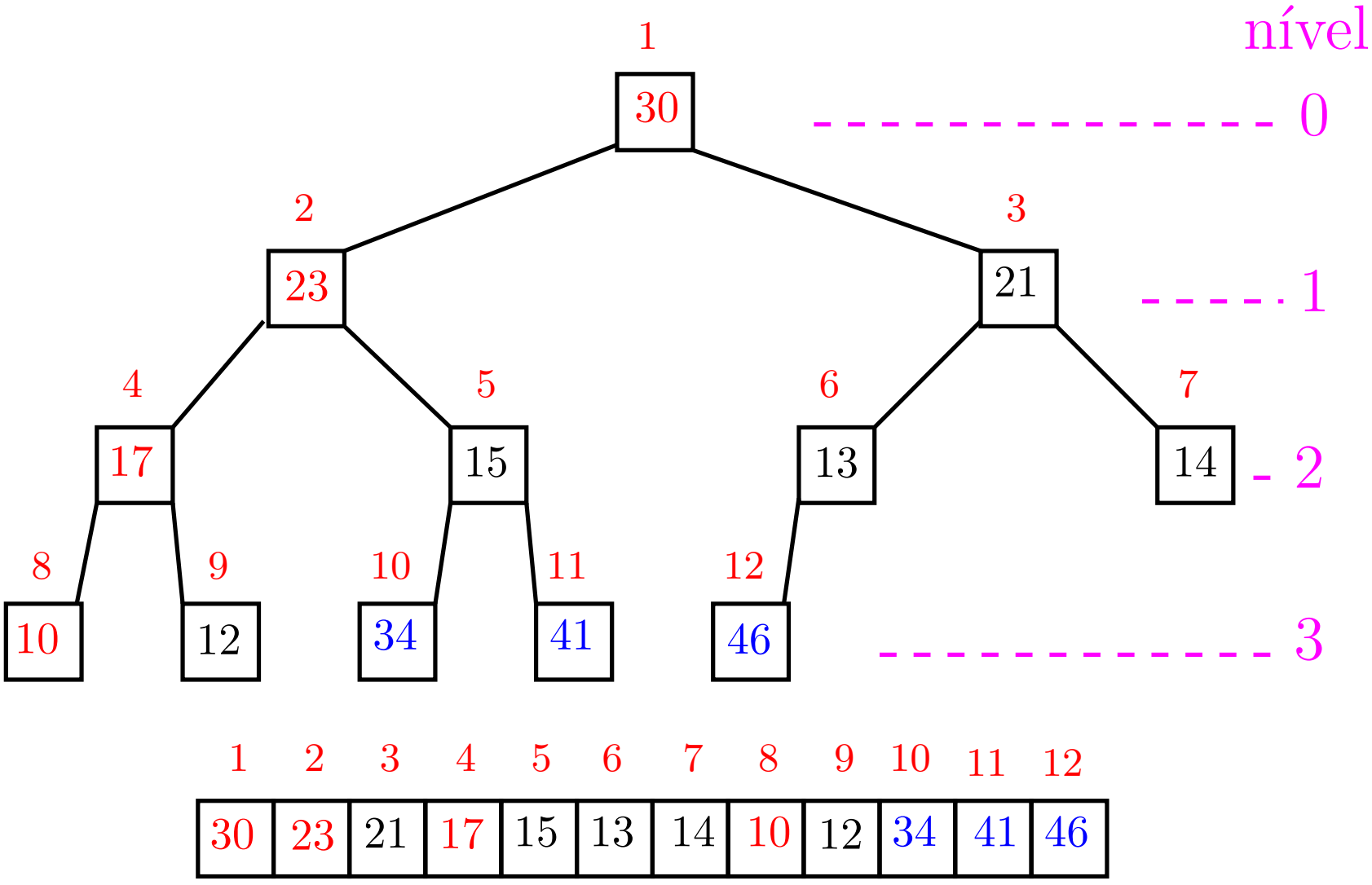
Heapsort


nível
1
30
0
2
10
3
21
1
4
23
5
15
6
13
7
14
2
8
17
9
12
10
34
11
41
12
46
3
1 2 3 4 5 6 7 8 9 10 11 12
30 10 21 23 15 13 14 17 12 34 41 46

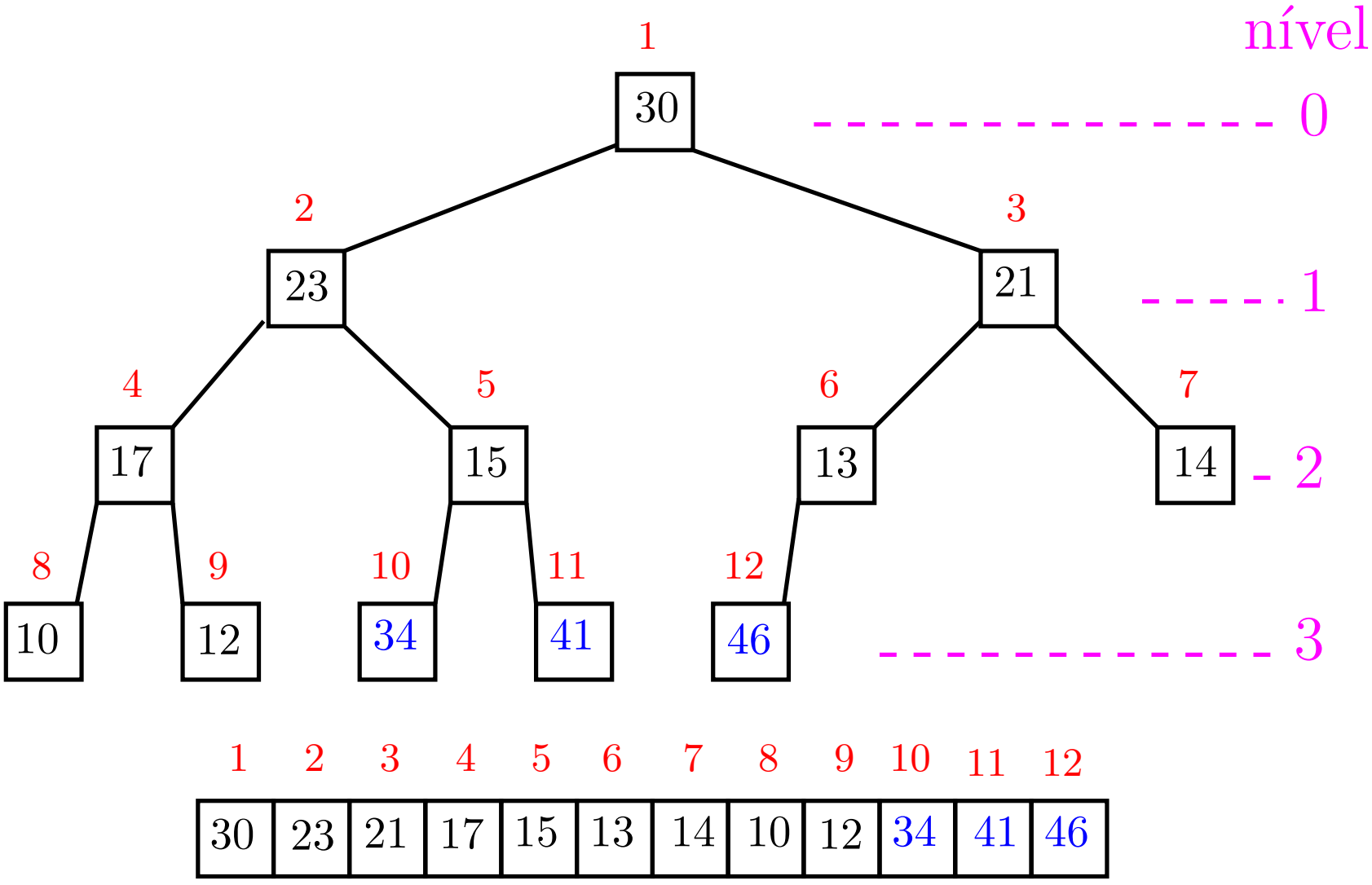
Heapsort


nível
1
30
0
2
23
3
21
1
4
10
5
15
6
13
7
14
2
8
17
9
12
10
34
11
41
12
46
3
1 2 3 4 5 6 7 8 9 10 11 12
30 23 21 10 15 13 14 17 12 34 41 46

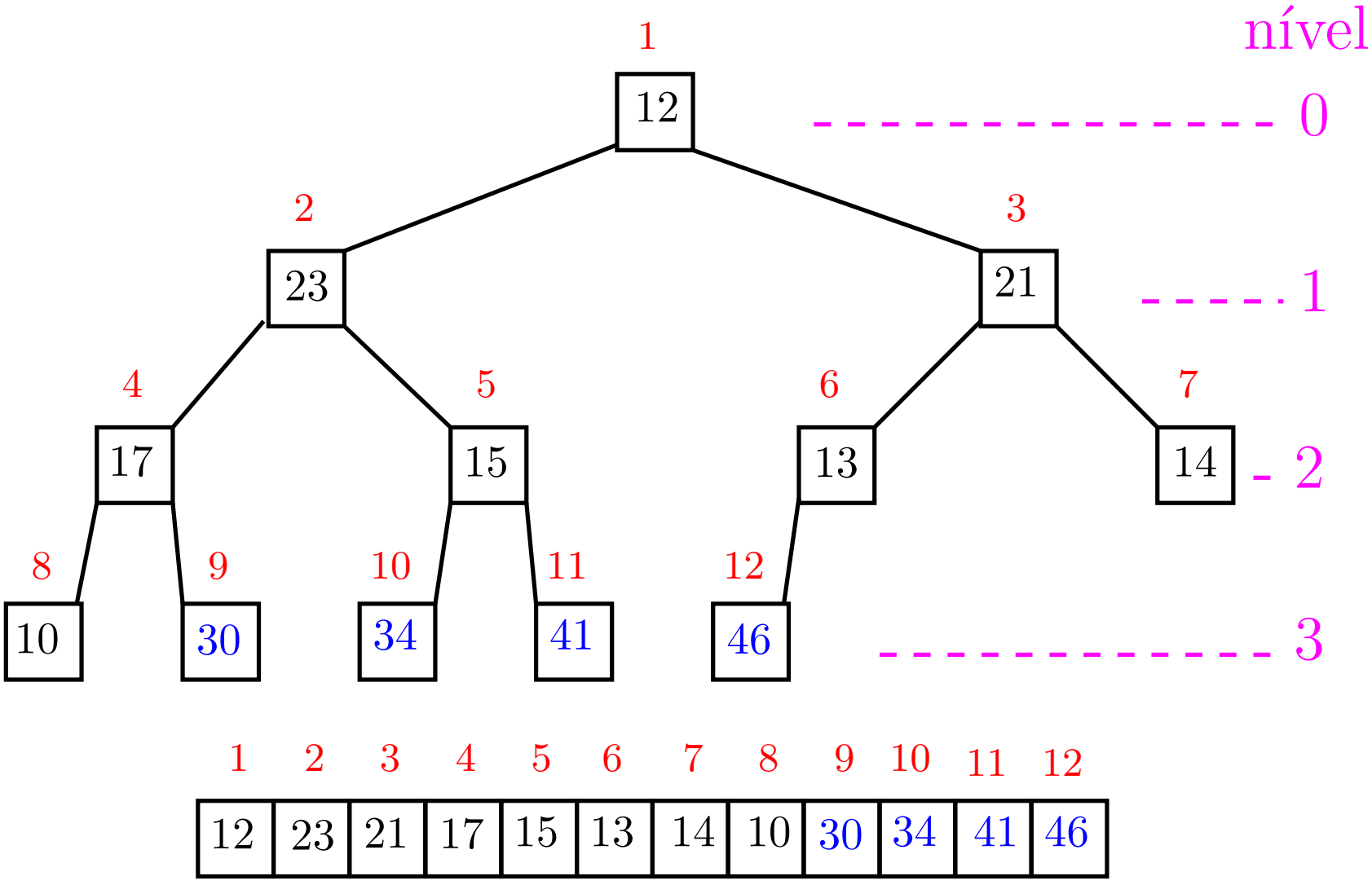
Heapsort


nível
1
30
0
2
23
3
21
1
4
17
5
15
6
13
7
14
2
8
10
9
12
10
34
11
41
12
46
3
1 2 3 4 5 6 7 8 9 10 11 12
30 23 21 17 15 13 14 10 12 34 41 46

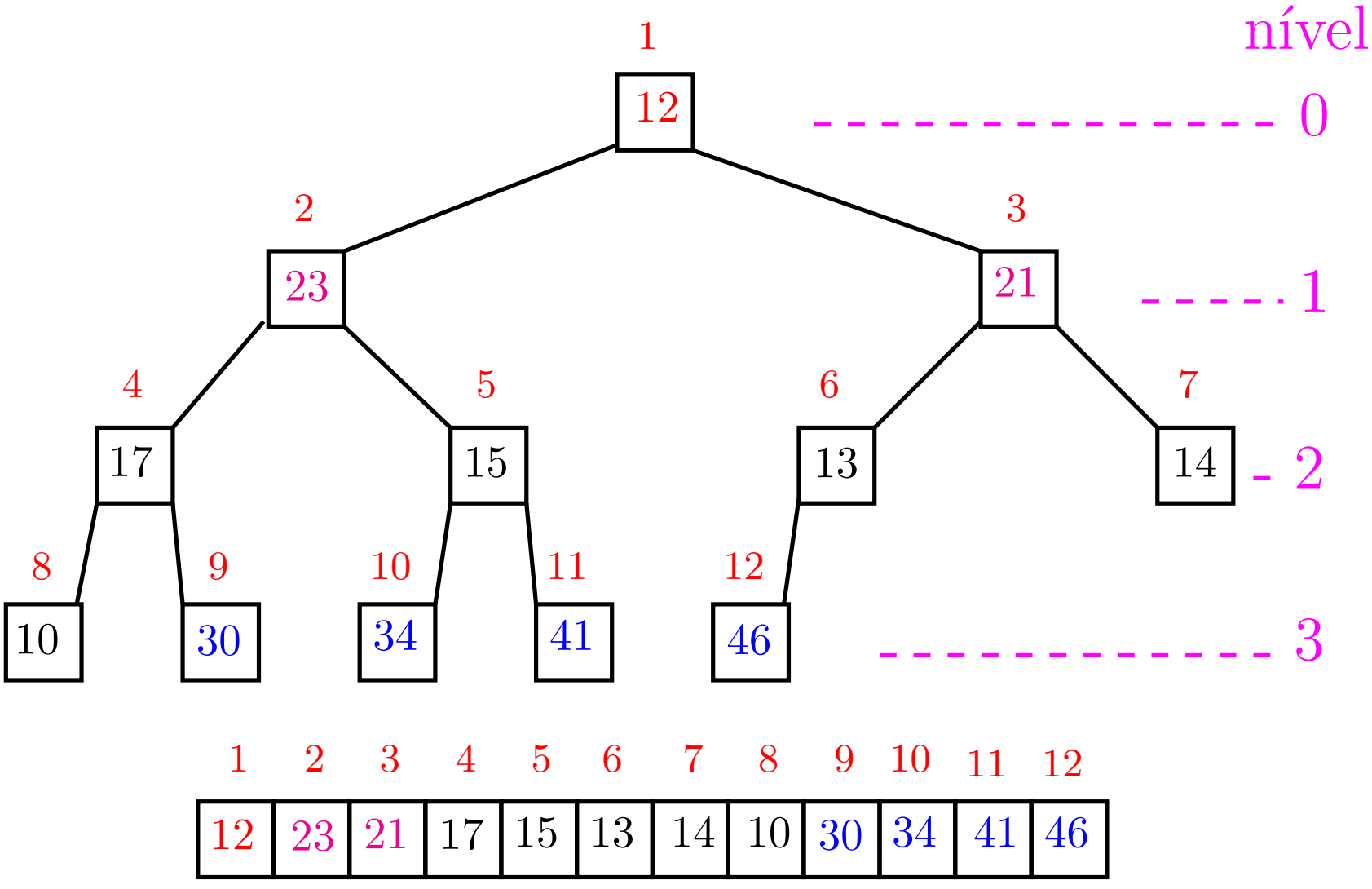
Heapsort


nível
1
30
0
2
23
3
21
1
4
17
5
15
6
13
7
14
2
8
10
9
12
10
34
11
41
12
46
3
1 2 3 4 5 6 7 8 9 10 11 12
30 23 21 17 15 13 14 10 12 34 41 46

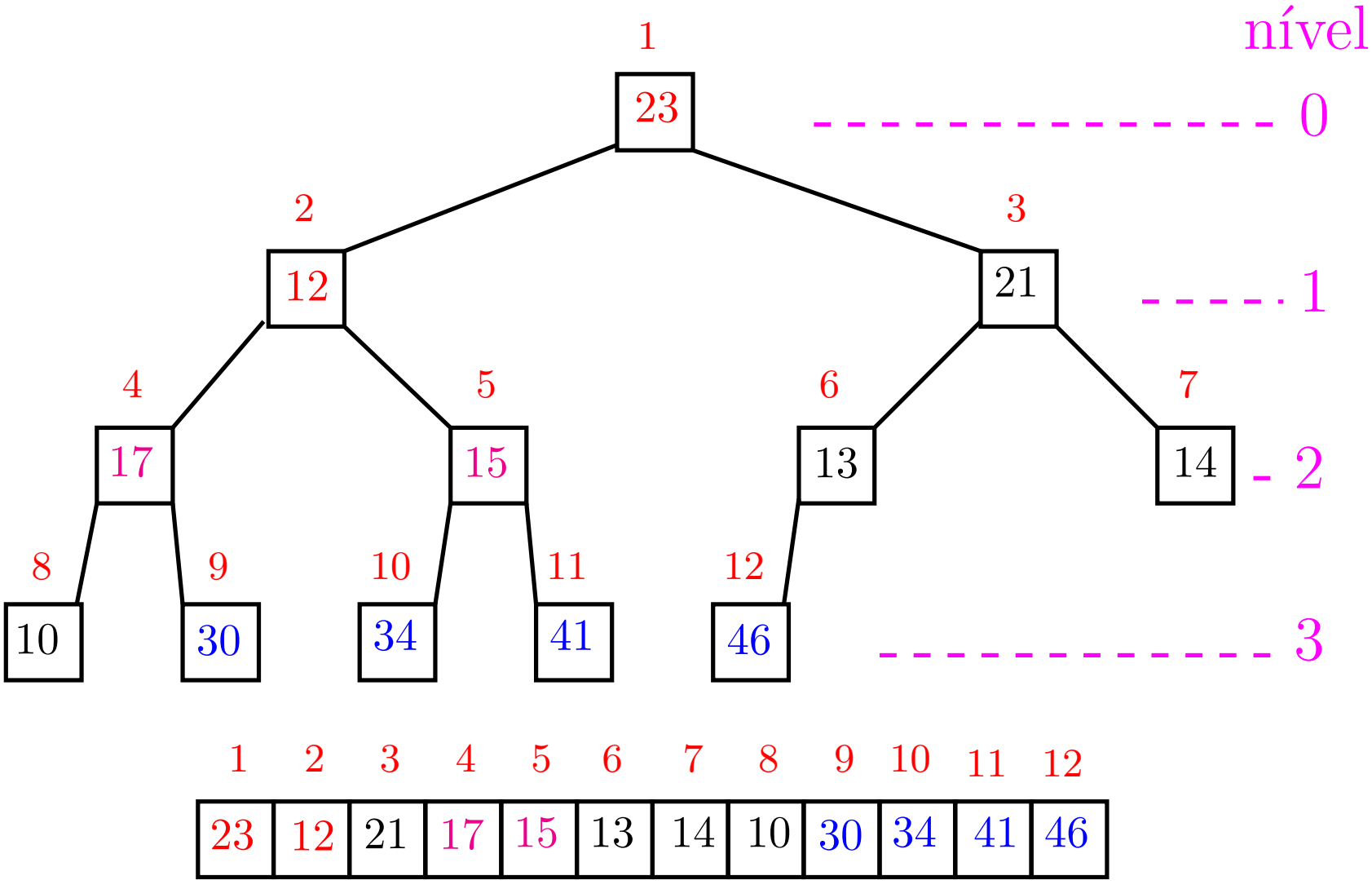
Heapsort


nível
1
12
0
2
23
3
21
1
4
17
5
15
6
13
7
14
2
8
10
9
30
10
34
11
41
12
46
3
1 2 3 4 5 6 7 8 9 10 11 12
12 23 21 17 15 13 14 10 30 34 41 46

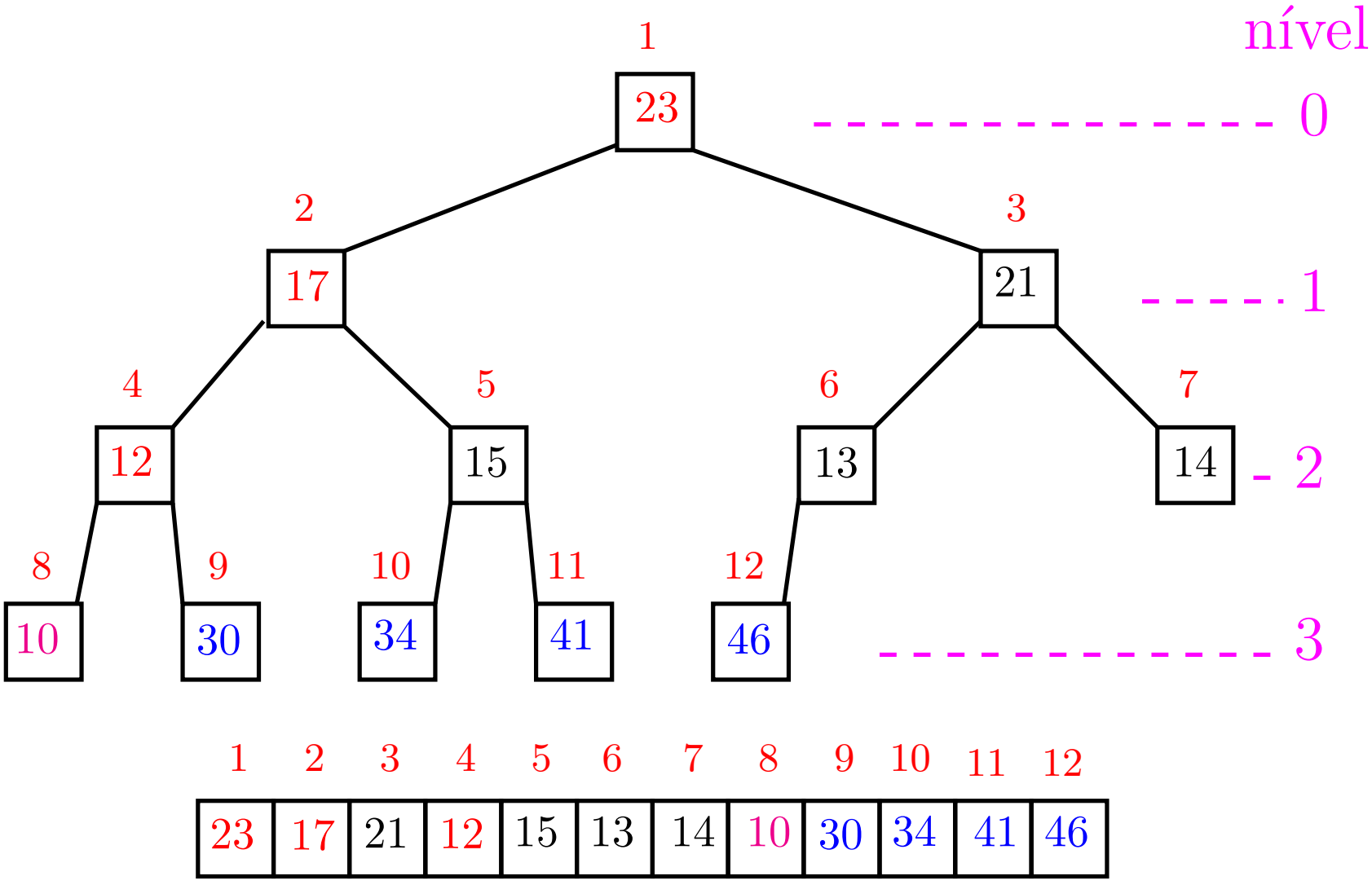
Heapsort


nível
1
12
0
2
23
3
21
1
4
17
5
15
6
13
7
14
2
8
10
9
30
10
34
11
41
12
46
3
1 2 3 4 5 6 7 8 9 10 11 12
12 23 21 17 15 13 14 10 30 34 41 46

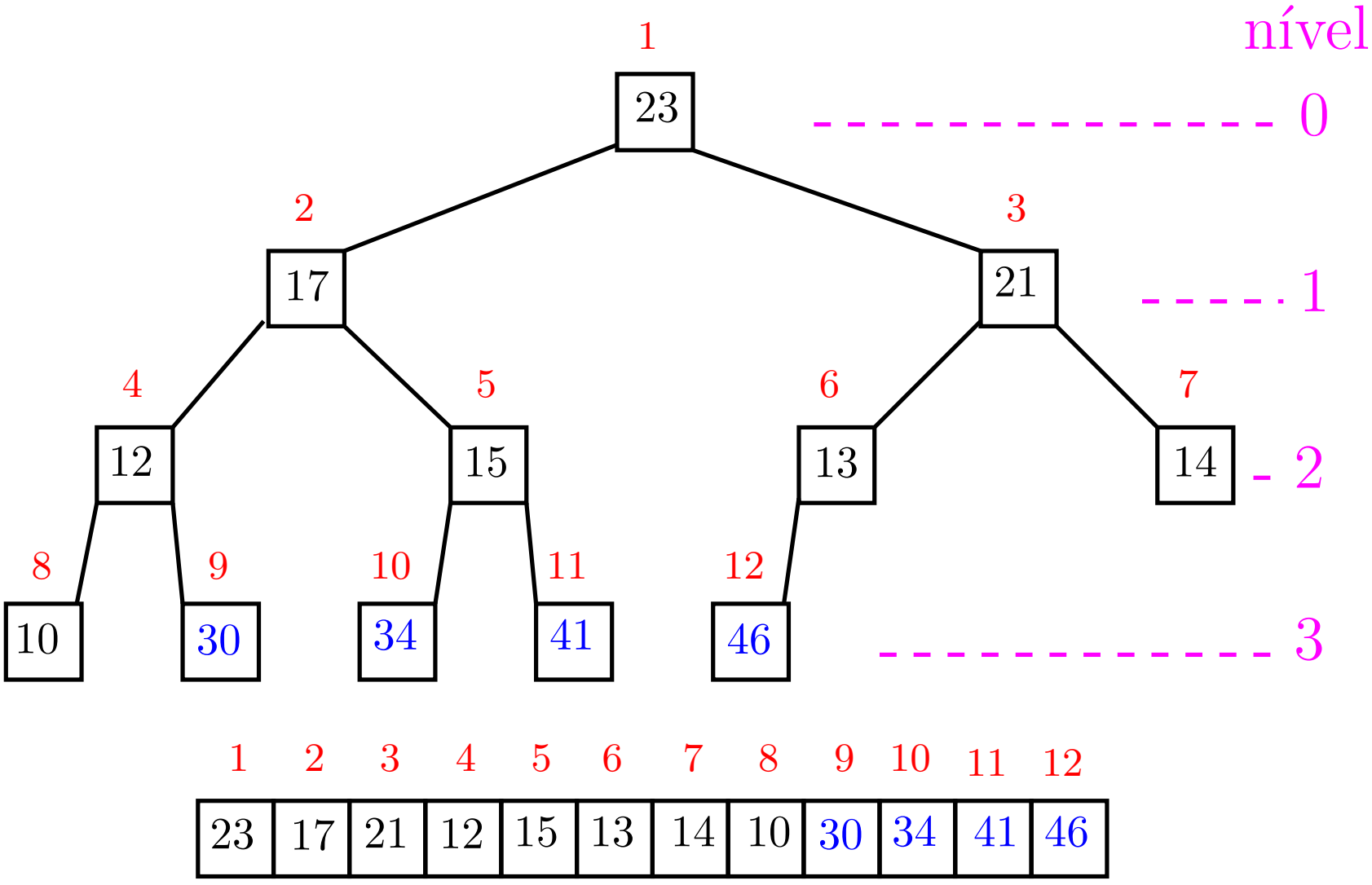
Heapsort


nível
1
23
0
2
12
3
21
1
4
17
5
15
6
13
7
14
2
8
10
9
30
10
34
11
41
12
46
3
1 2 3 4 5 6 7 8 9 10 11 12
23 12 21 17 15 13 14 10 30 34 41 46

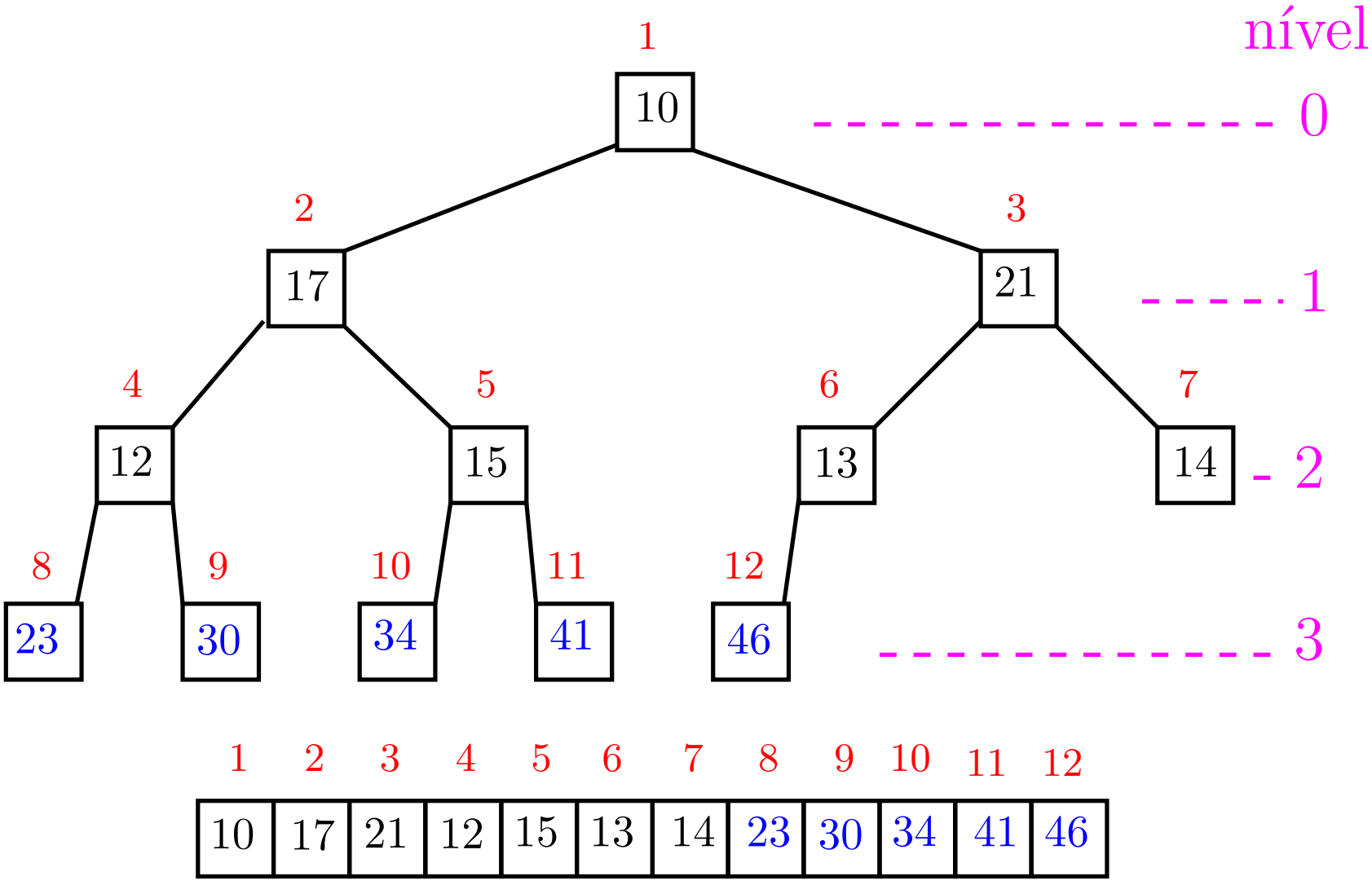
Heapsort


nível
1
23
0
2
17
3
21
1
4
12
5
15
6
13
7
14
2
8
10
9
30
10
34
11
41
12
46
3
1 2 3 4 5 6 7 8 9 10 11 12
23 17 21 12 15 13 14 10 30 34 41 46

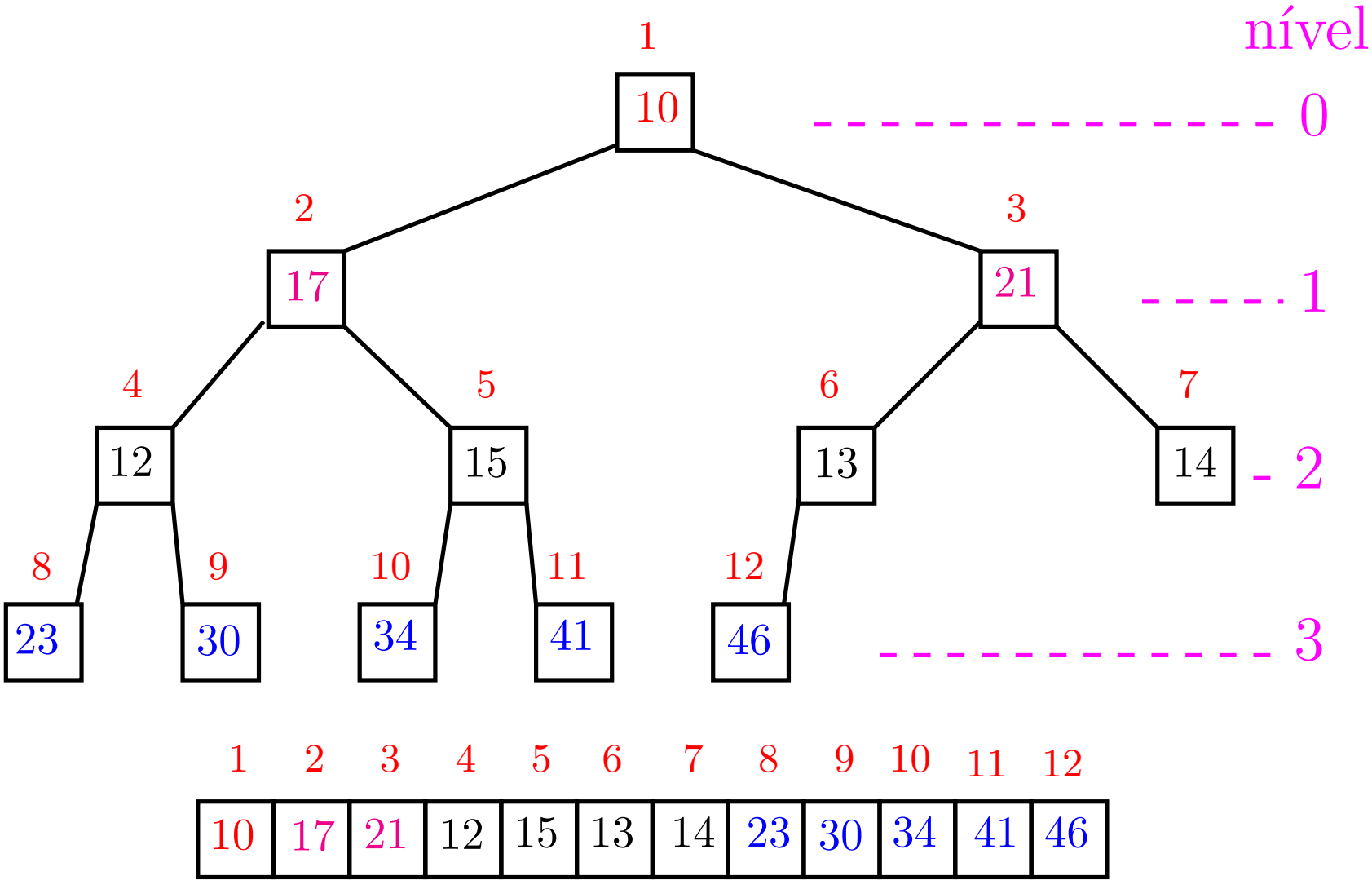
Heapsort


nível
1
23
0
2
17
3
21
1
4
12
5
15
6
13
7
14
2
8
10
9
30
10
34
11
41
12
46
3
1 2 3 4 5 6 7 8 9 10 11 12
23 17 21 12 15 13 14 10 30 34 41 46

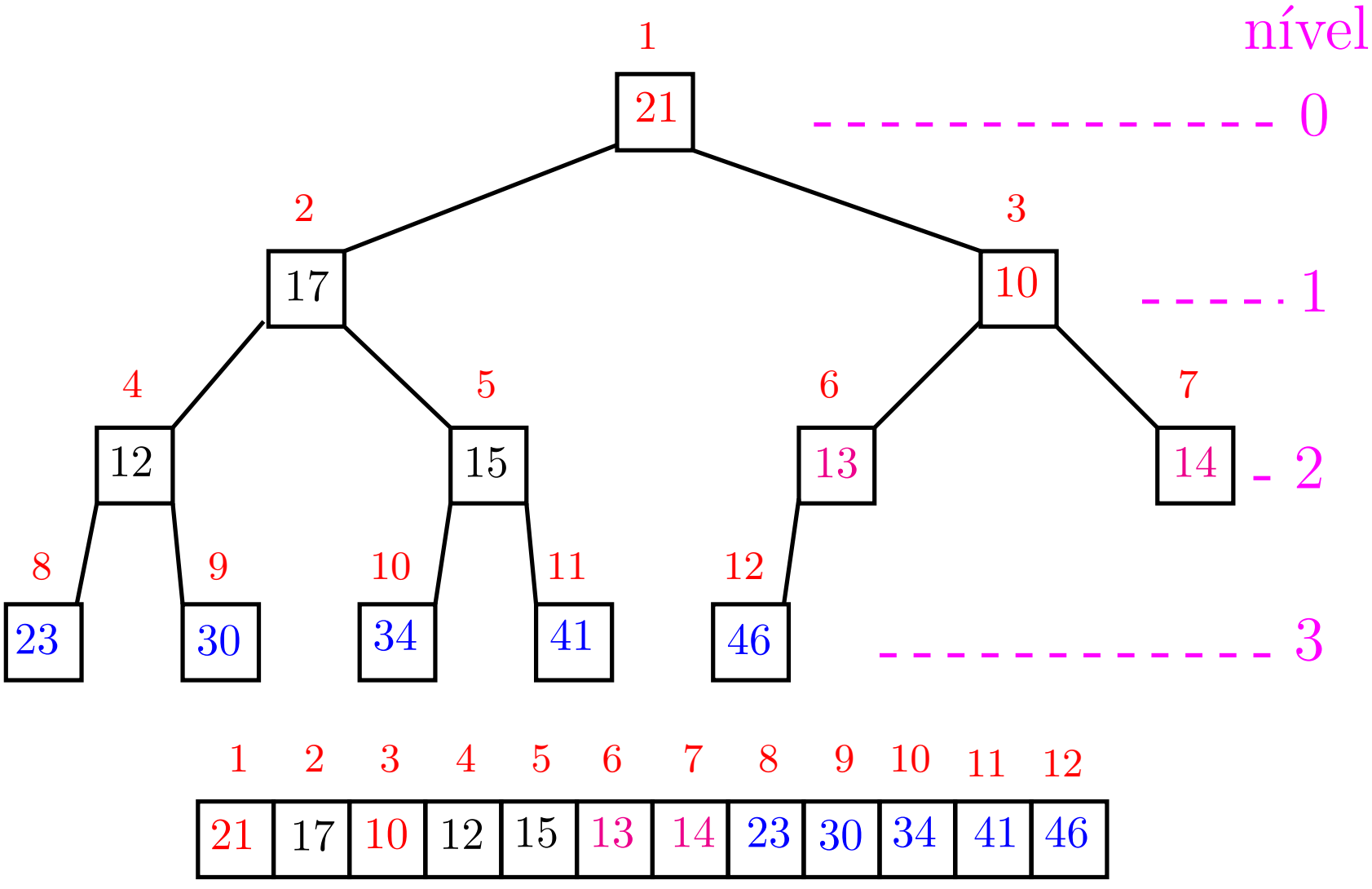
Heapsort


nível
1
10
0
2
17
3
21
1
4
12
5
15
6
13
7
14
2
8
23
9
30
10
34
11
41
12
46
3
1 2 3 4 5 6 7 8 9 10 11 12
10 17 21 12 15 13 14 23 30 34 41 46

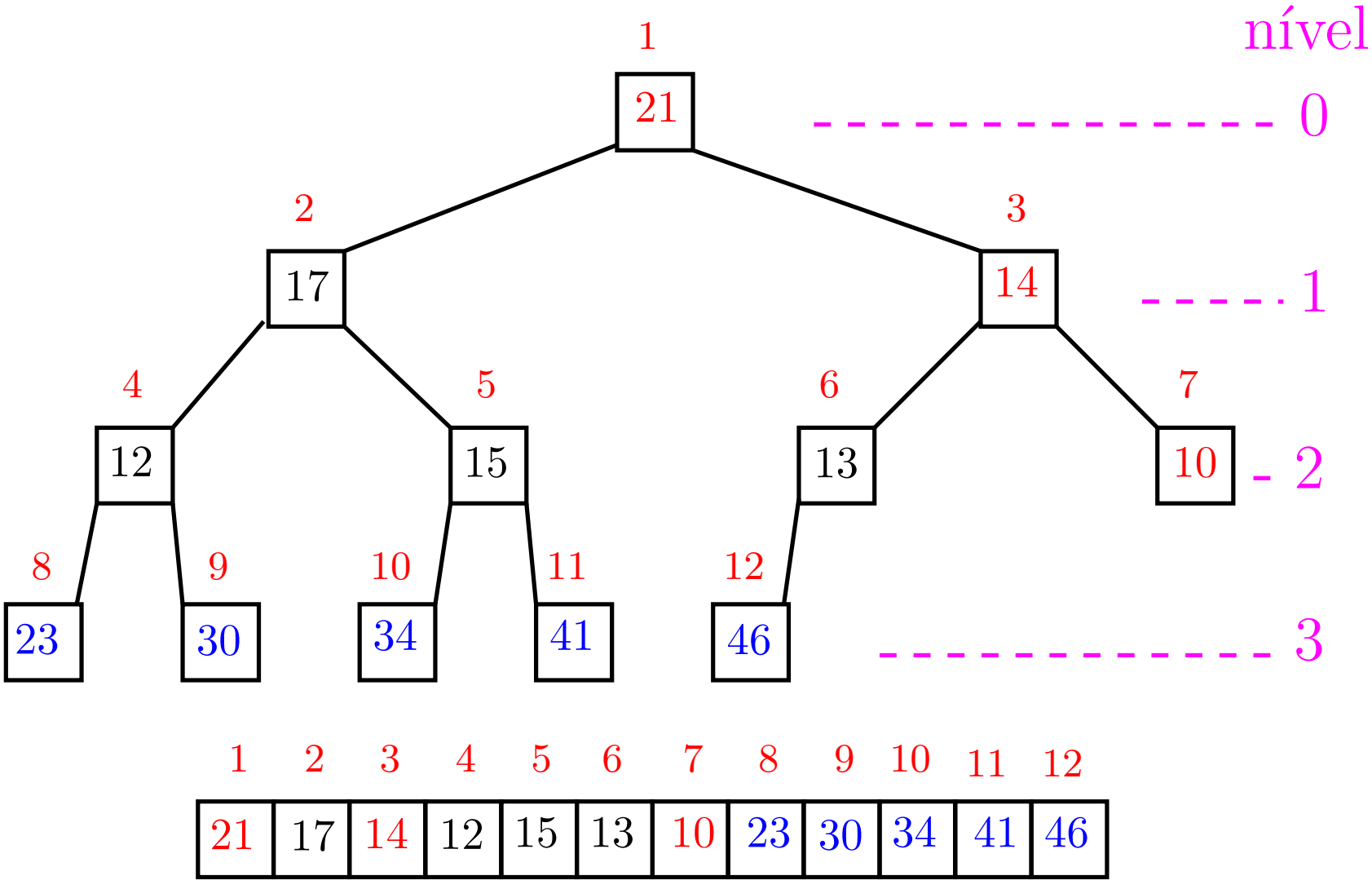
Heapsort


nível
1
10
0
2
17
3
21
1
4
12
5
15
6
13
7
14
2
8
23
9
30
10
34
11
41
12
46
3
1 2 3 4 5 6 7 8 9 10 11 12
10 17 21 12 15 13 14 23 30 34 41 46

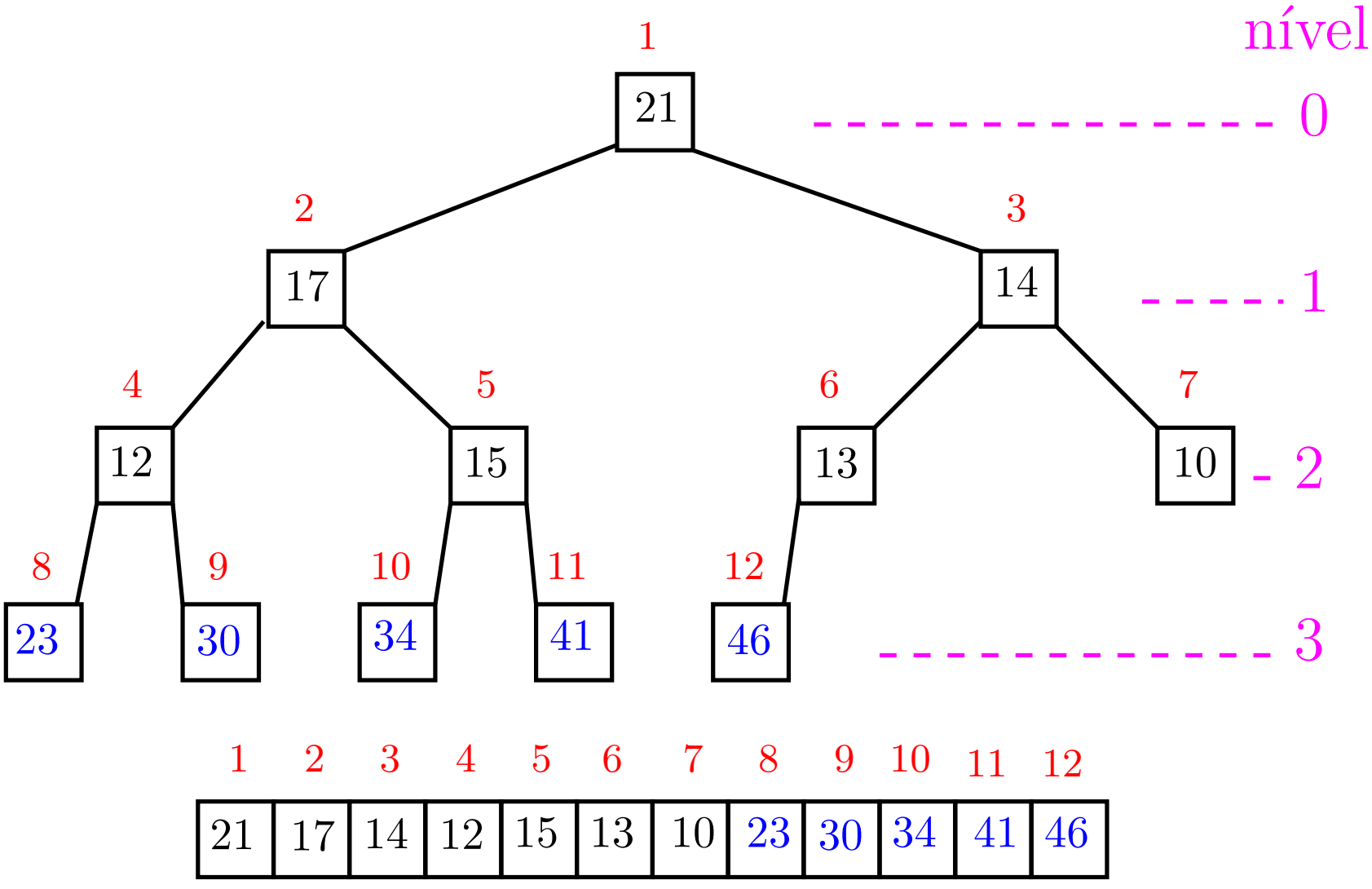
Heapsort


nível
1
21
0
2
17
3
10
1
4
12
5
15
6
13
7
14
2
8
23
9
30
10
34
11
41
12
46
3
1 2 3 4 5 6 7 8 9 10 11 12
21 17 10 12 15 13 14 23 30 34 41 46

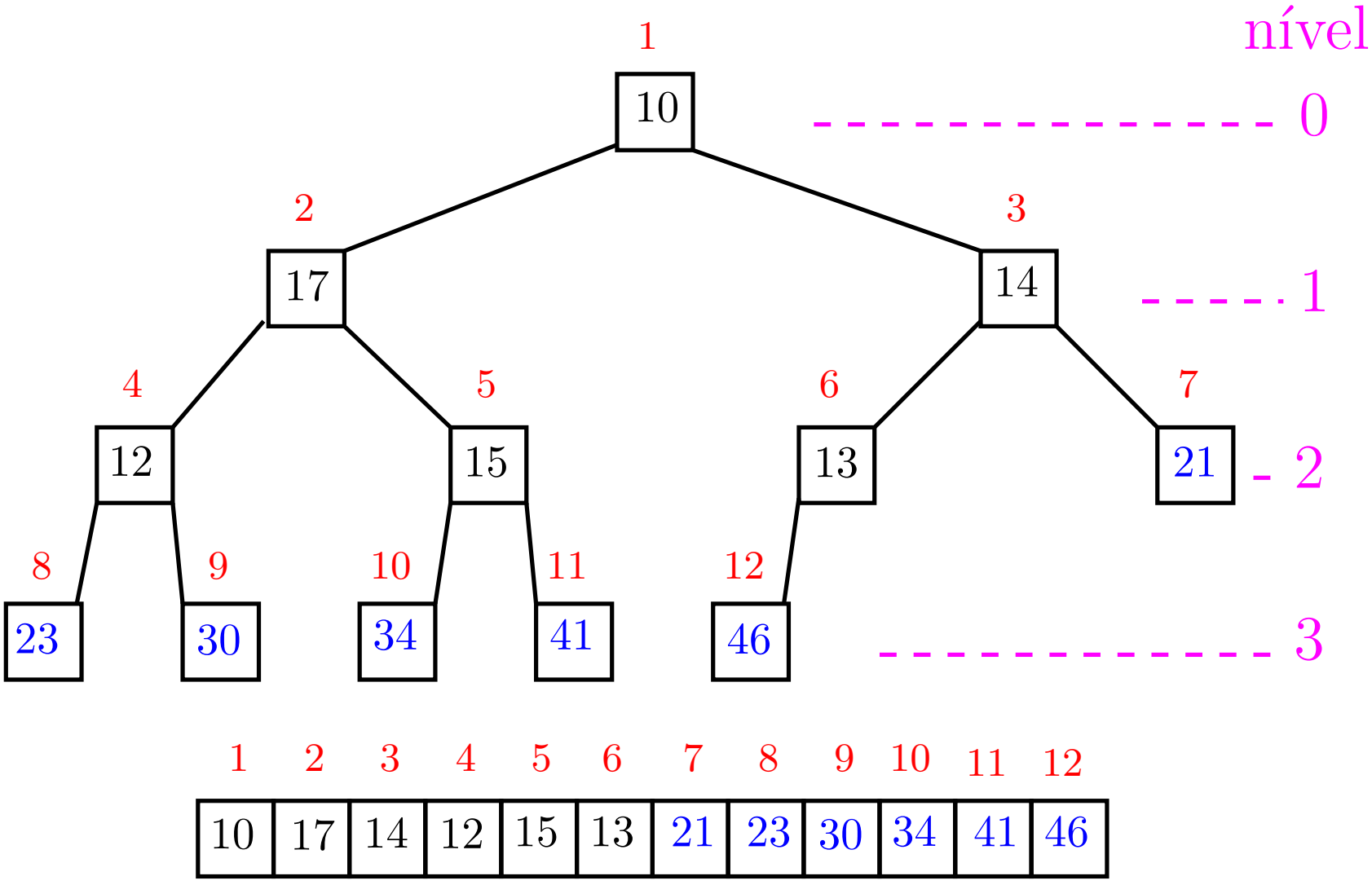
Heapsort


nível
1
21
0
2
17
3
14
1
4
12
5
15
6
13
7
10
2
8
23
9
30
10
34
11
41
12
46
3
1 2 3 4 5 6 7 8 9 10 11 12
21 17 14 12 15 13 10 23 30 34 41 46

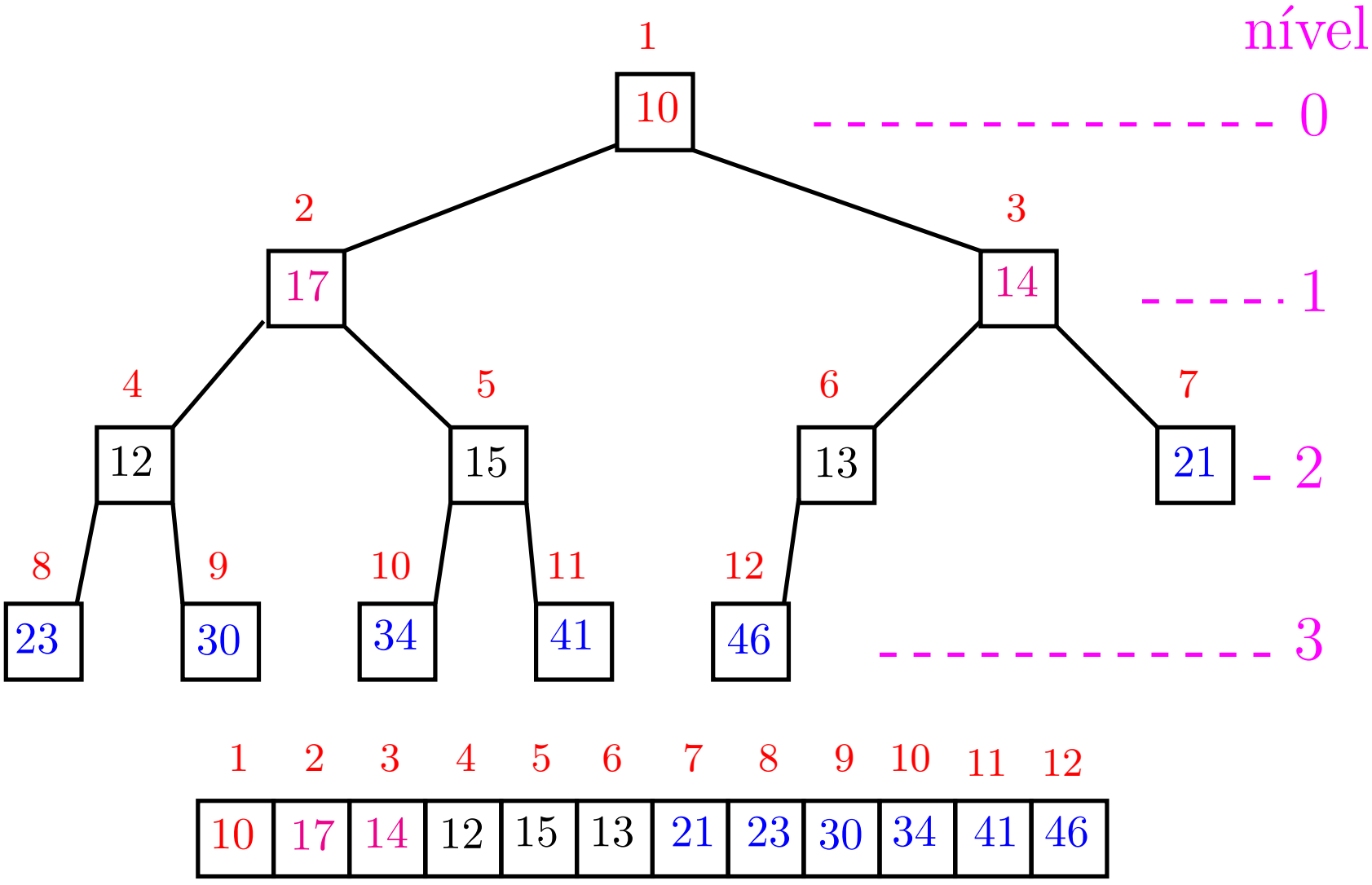
Heapsort


nível
1
21
0
2
17
3
14
1
4
12
5
15
6
13
7
10
2
8
23
9
30
10
34
11
41
12
46
3
1 2 3 4 5 6 7 8 9 10 11 12
21 17 14 12 15 13 10 23 30 34 41 46

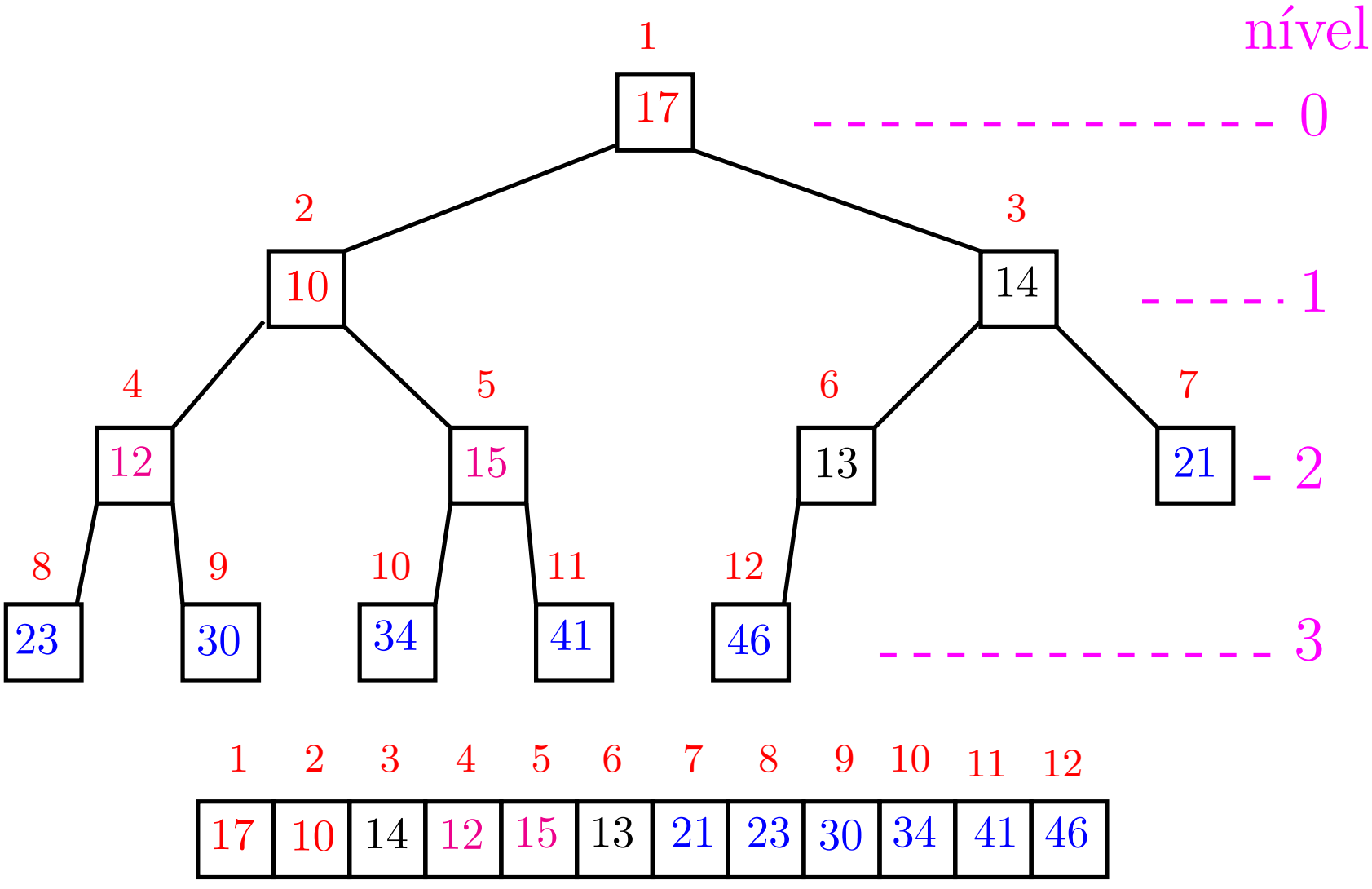
Heapsort


nível
1
10
0
2
17
3
14
1
4
12
5
15
6
13
7
21
2
8
23
9
30
10
34
11
41
12
46
3
1 2 3 4 5 6 7 8 9 10 11 12
10 17 14 12 15 13 21 23 30 34 41 46

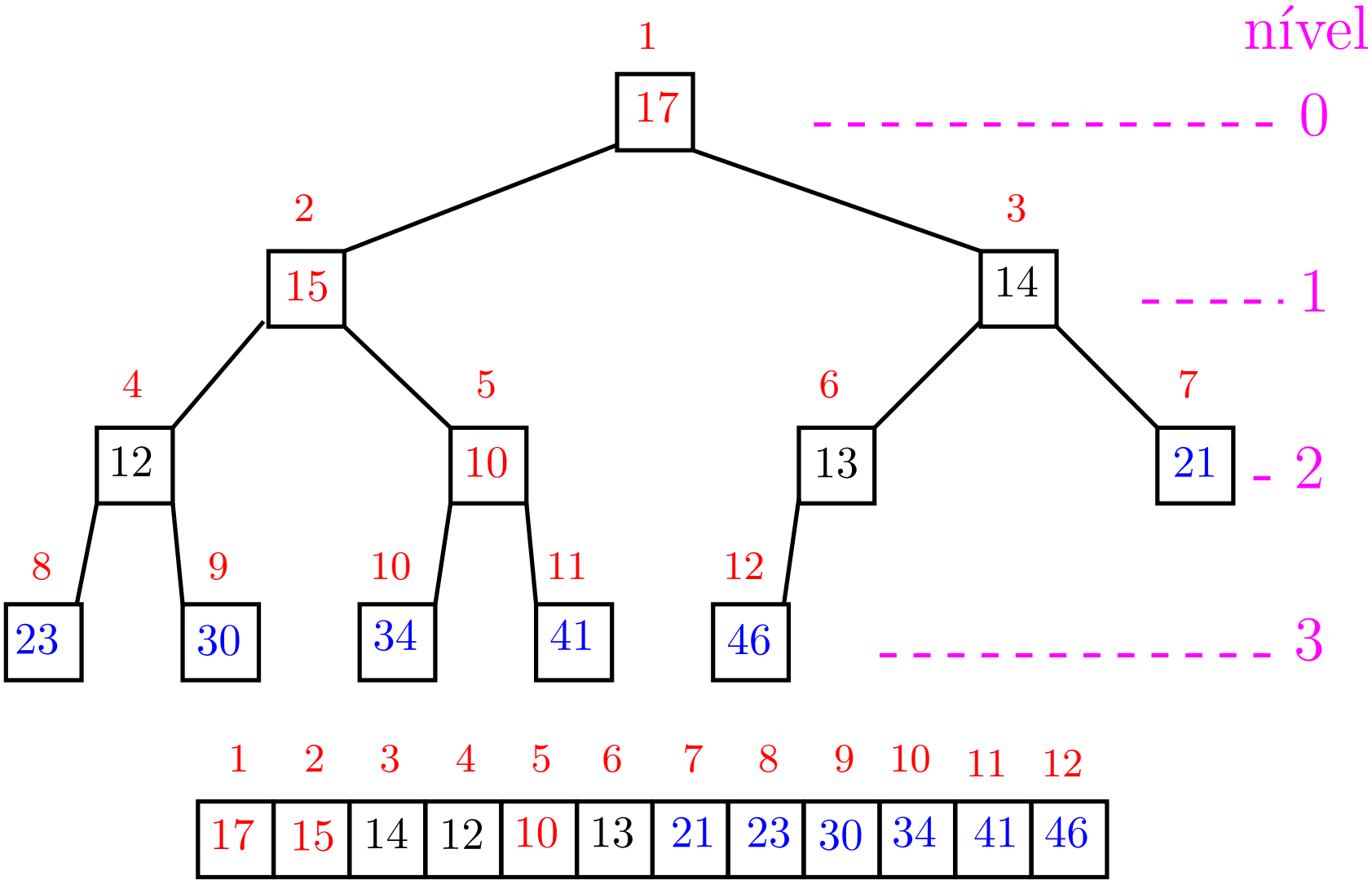
Heapsort


nível
1
10
0
2
17
3
14
1
4
12
5
15
6
13
7
21
2
8
23
9
30
10
34
11
41
12
46
3
1 2 3 4 5 6 7 8 9 10 11 12
10 17 14 12 15 13 21 23 30 34 41 46

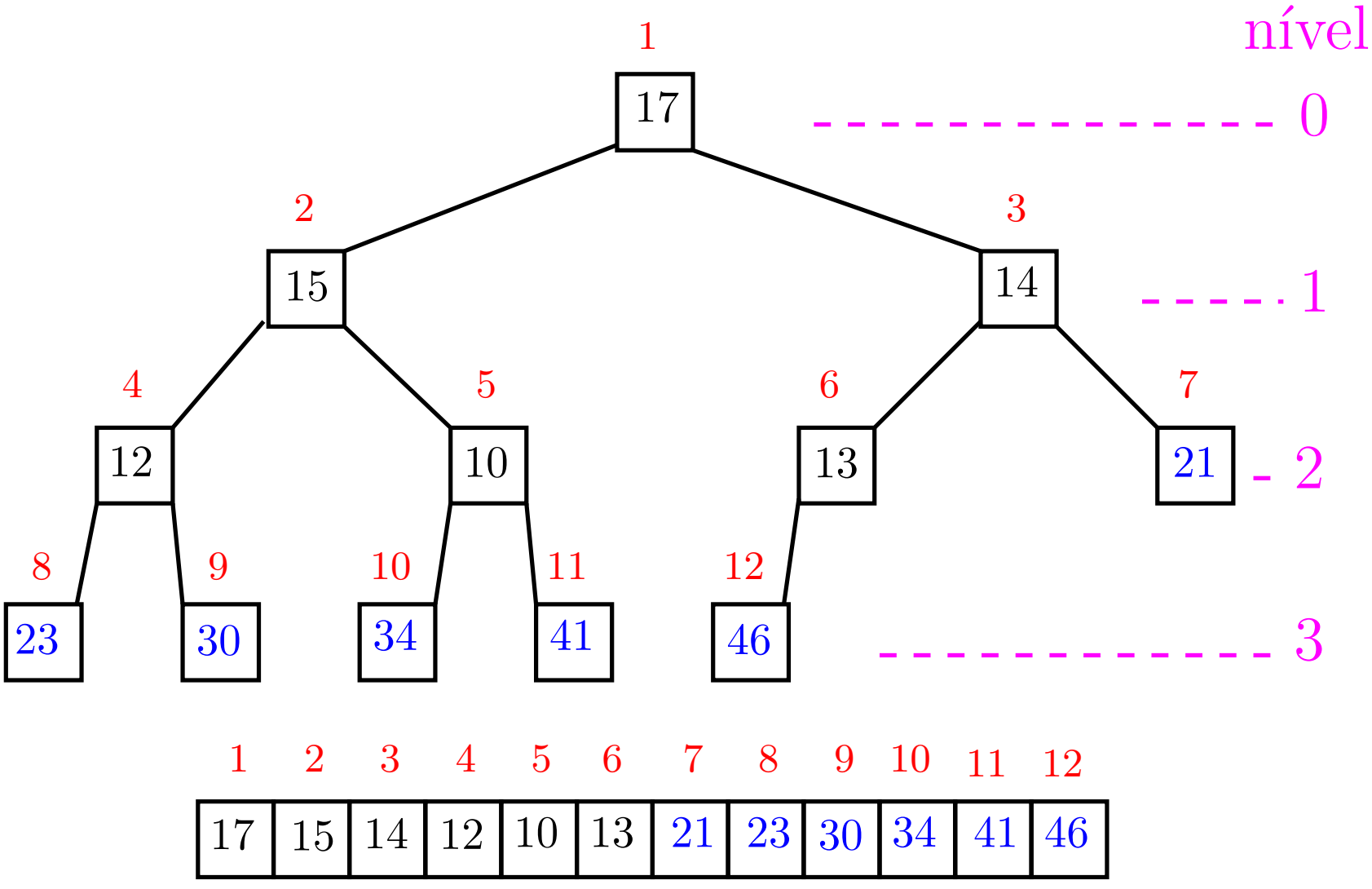
Heapsort


nível
1
17
0
2
10
3
14
1
4
12
5
15
6
13
7
21
2
8
23
9
30
10
34
11
41
12
46
3
1 2 3 4 5 6 7 8 9 10 11 12
17 10 14 12 15 13 21 23 30 34 41 46

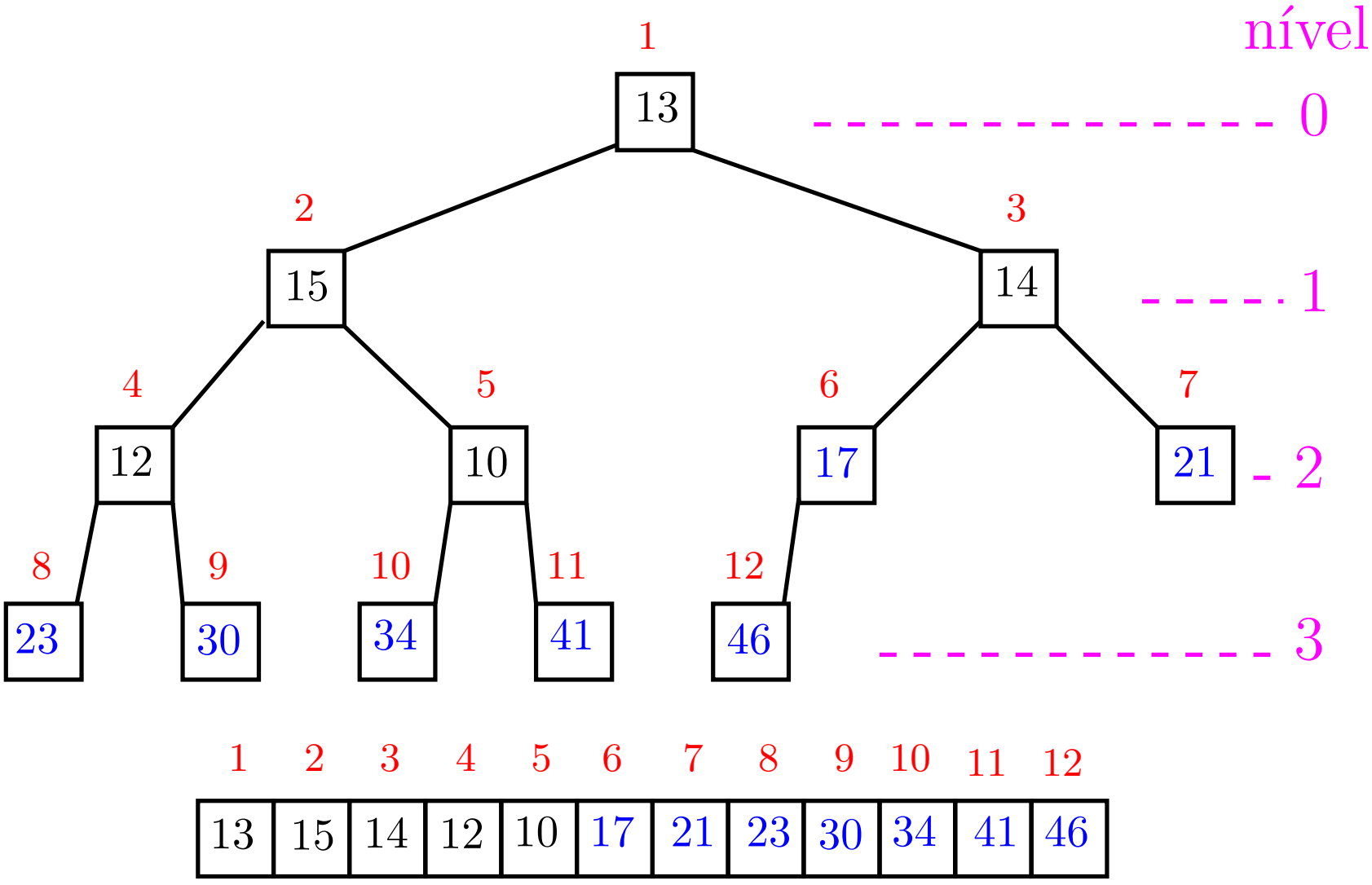
Heapsort


nível
1
17
0
2
15
3
14
1
4
12
5
10
6
13
7
21
2
8
23
9
30
10
34
11
41
12
46
3
1 2 3 4 5 6 7 8 9 10 11 12
17 15 14 12 10 13 21 23 30 34 41 46

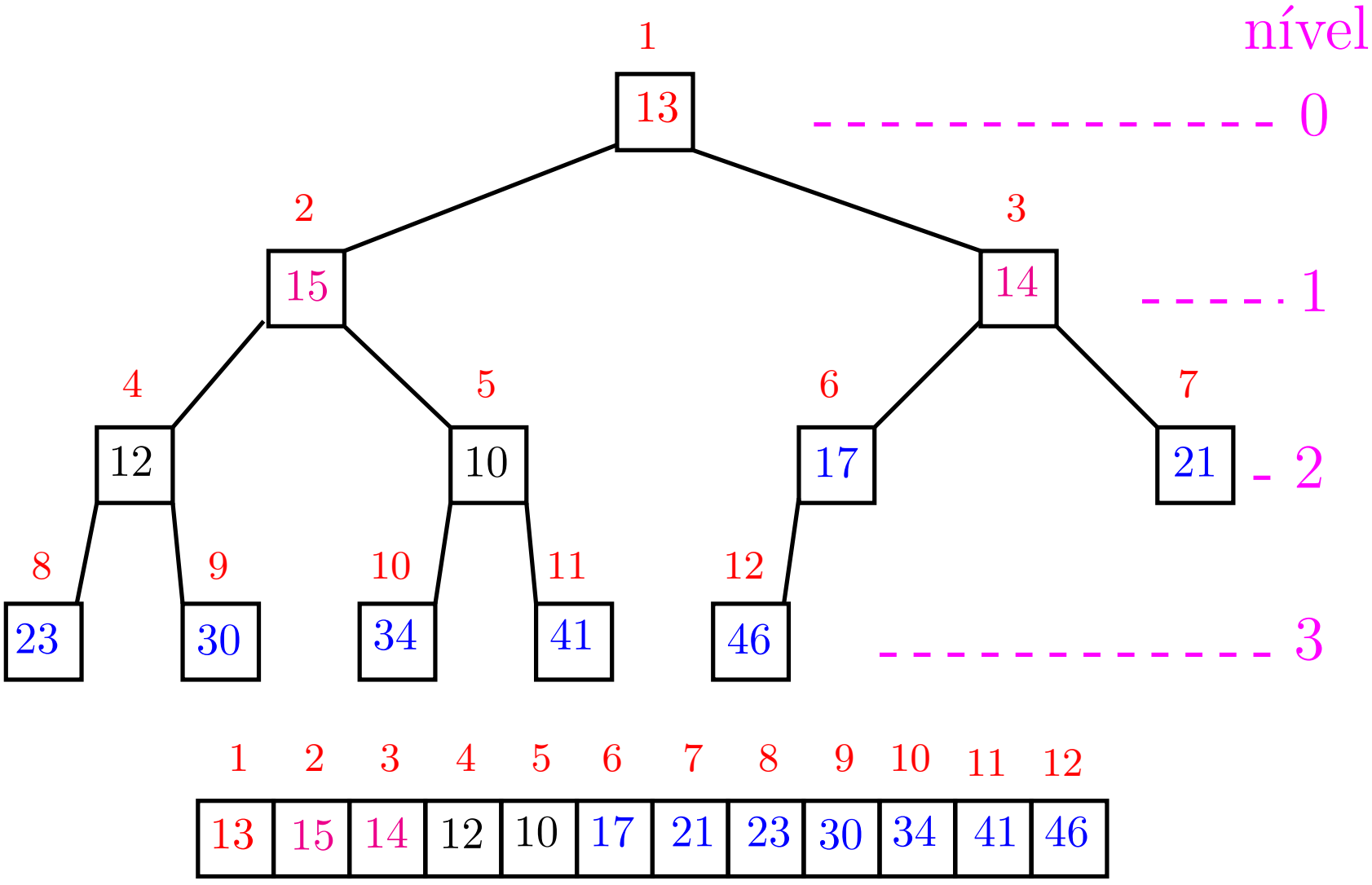
Heapsort


nível
1
17
0
2
3
15
14
1
4
5
6
7
12
10
13
21
2
8
9
10
11
12
23
30
34
41
46
3
1 2 3 4 5 6 7 8 9 10 11 12
17 15 14 12 10 13 21 23 30 34 41 46

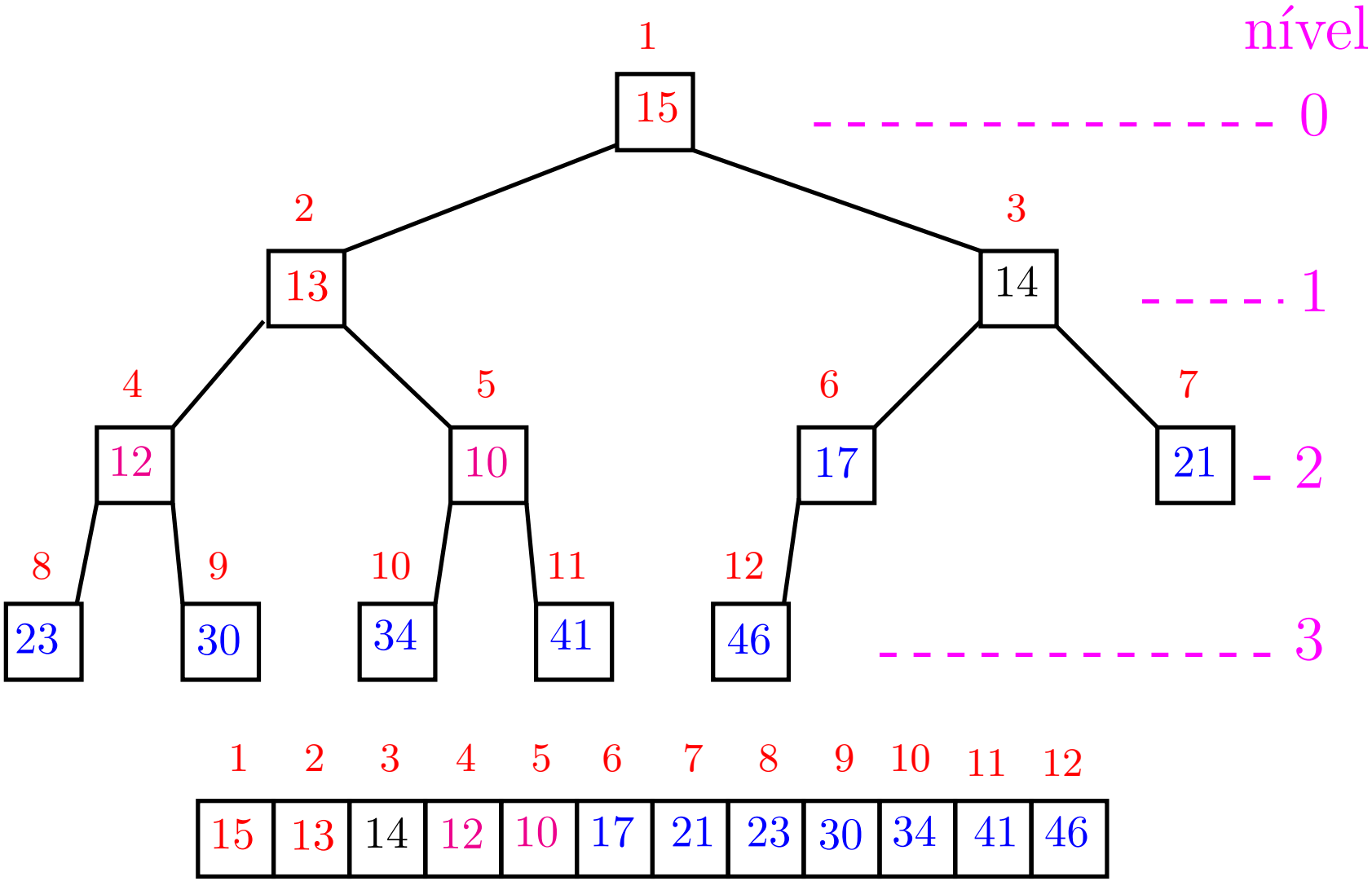
Heapsort


nível
1
13
0
2
15
3
14
1
4
12
5
10
6
17
7
21
2
8
23
9
30
10
34
11
41
12
46
3
1 2 3 4 5 6 7 8 9 10 11 12
13 15 14 12 10 17 21 23 30 34 41 46

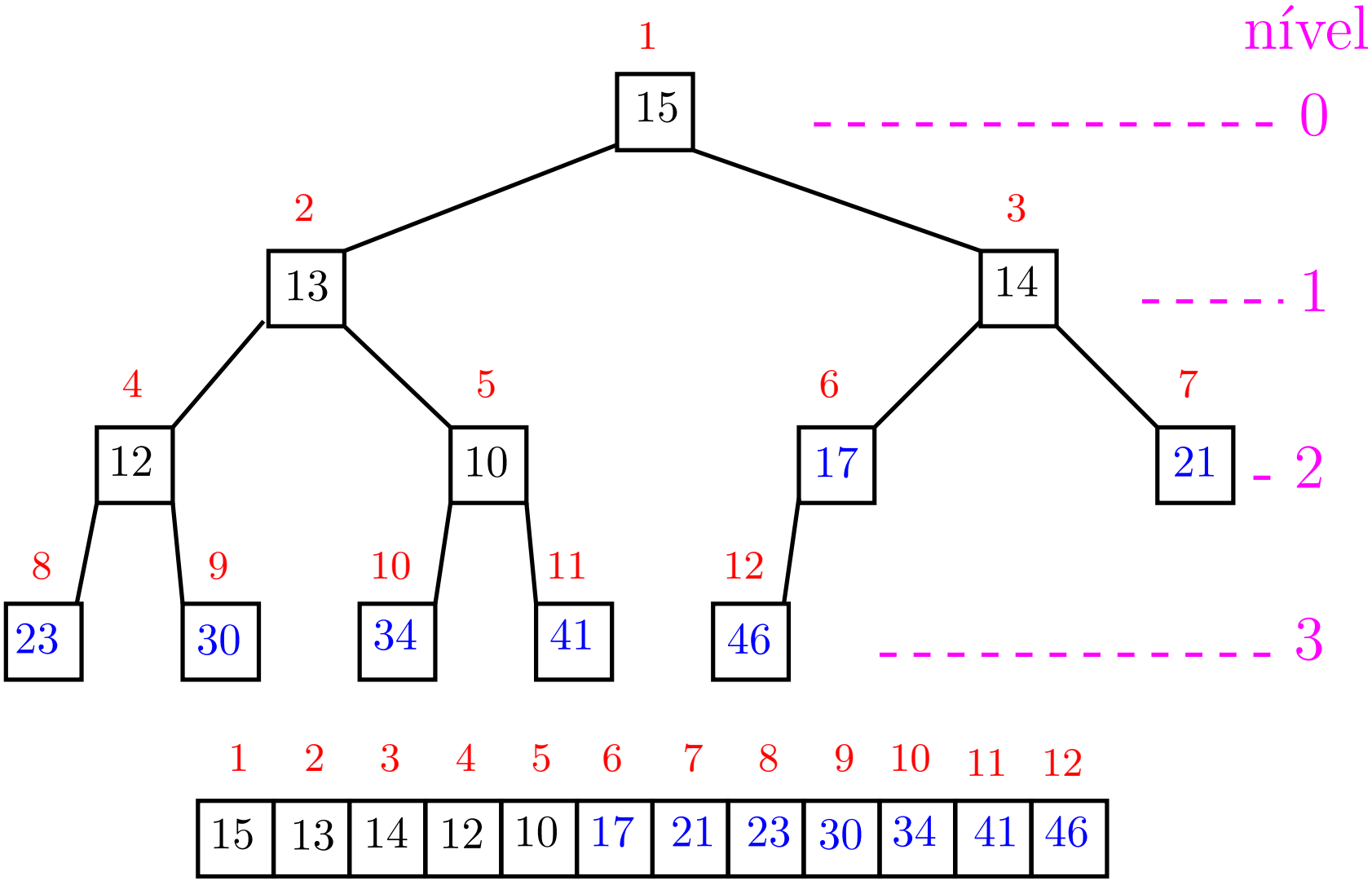
Heapsort


nível
1
13
0
2
15
3
14
1
4
12
5
10
6
17
7
21
2
8
23
9
30
10
34
11
41
12
46
3
1 2 3 4 5 6 7 8 9 10 11 12
13 15 14 12 10 17 21 23 30 34 41 46

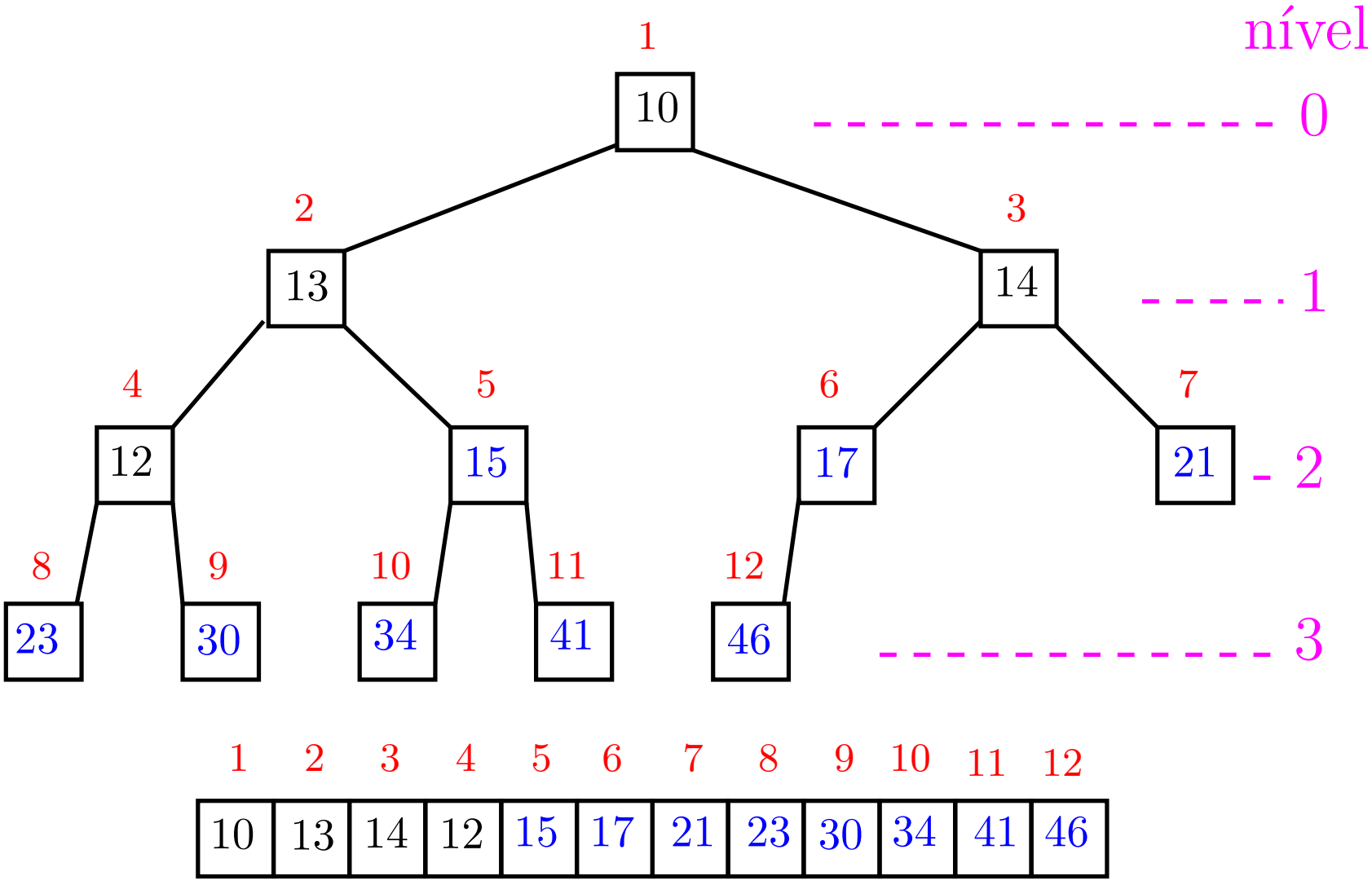
Heapsort


nível
1
15
0
2
13
3
14
1
4
12
5
10
6
17
7
21
2
8
23
9
30
10
34
11
41
12
46
3
1 2 3 4 5 6 7 8 9 10 11 12
15 13 14 12 10 17 21 23 30 34 41 46

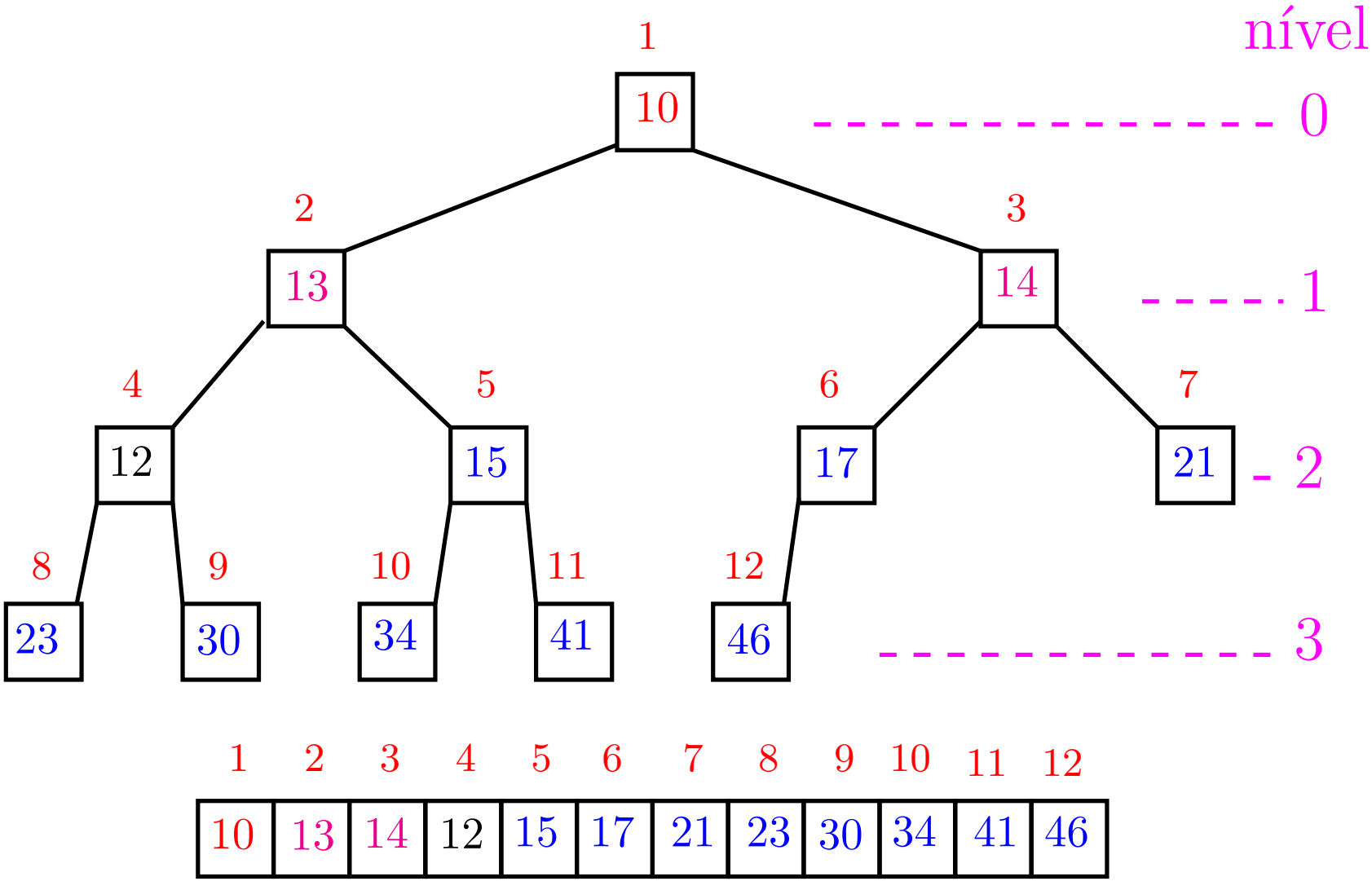
Heapsort


nível
1
15
0
2
13
3
14
1
4
12
5
10
6
17
7
21
2
8
23
9
30
10
34
11
41
12
46
3
1 2 3 4 5 6 7 8 9 10 11 12
15 13 14 12 10 17 21 23 30 34 41 46

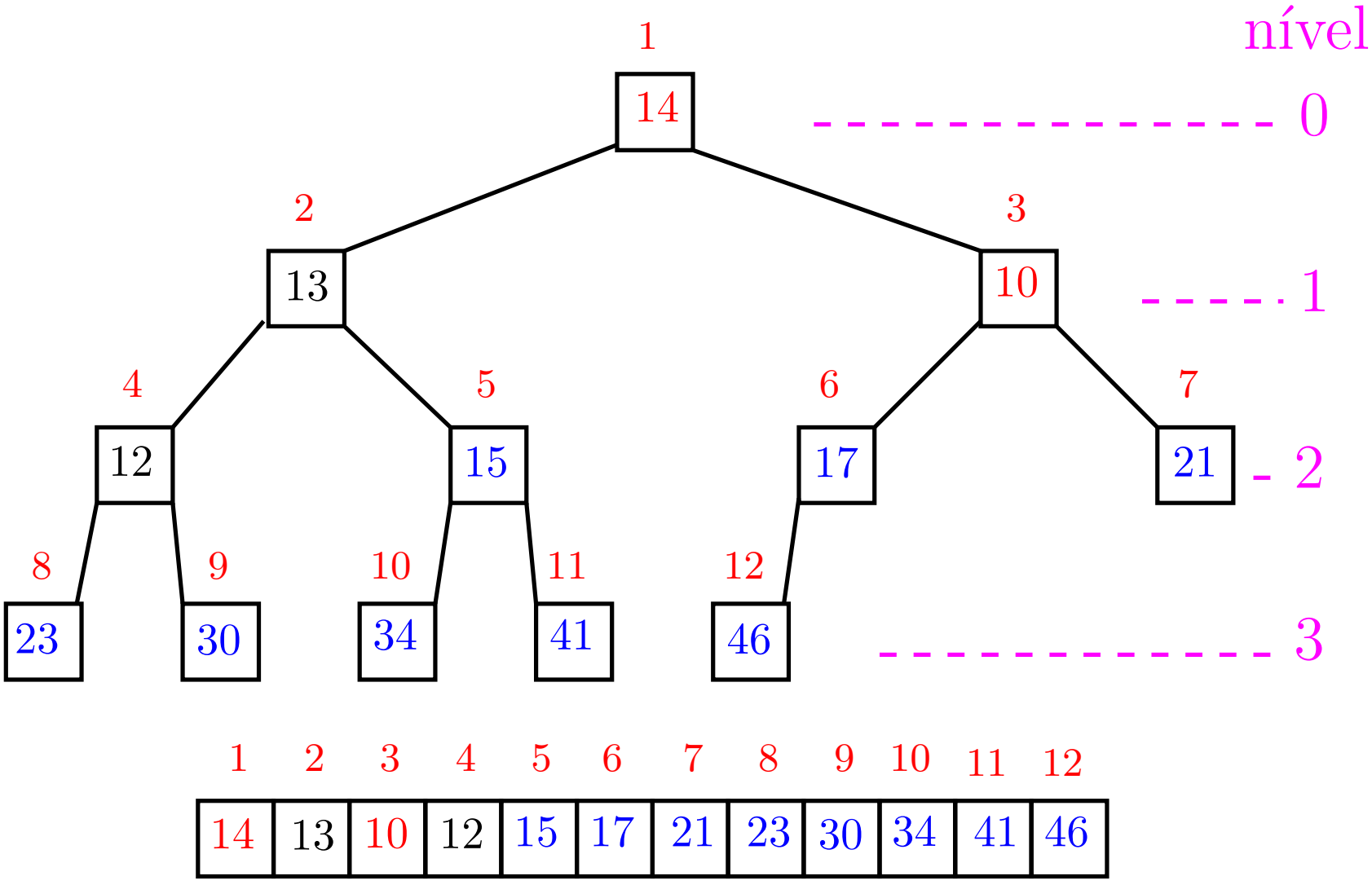
Heapsort


nível
1
10
0
2
13
3
14
1
4
12
5
15
6
17
7
21
2
8
23
9
30
10
34
11
41
12
46
3
1 2 3 4 5 6 7 8 9 10 11 12
10 13 14 12 15 17 21 23 30 34 41 46

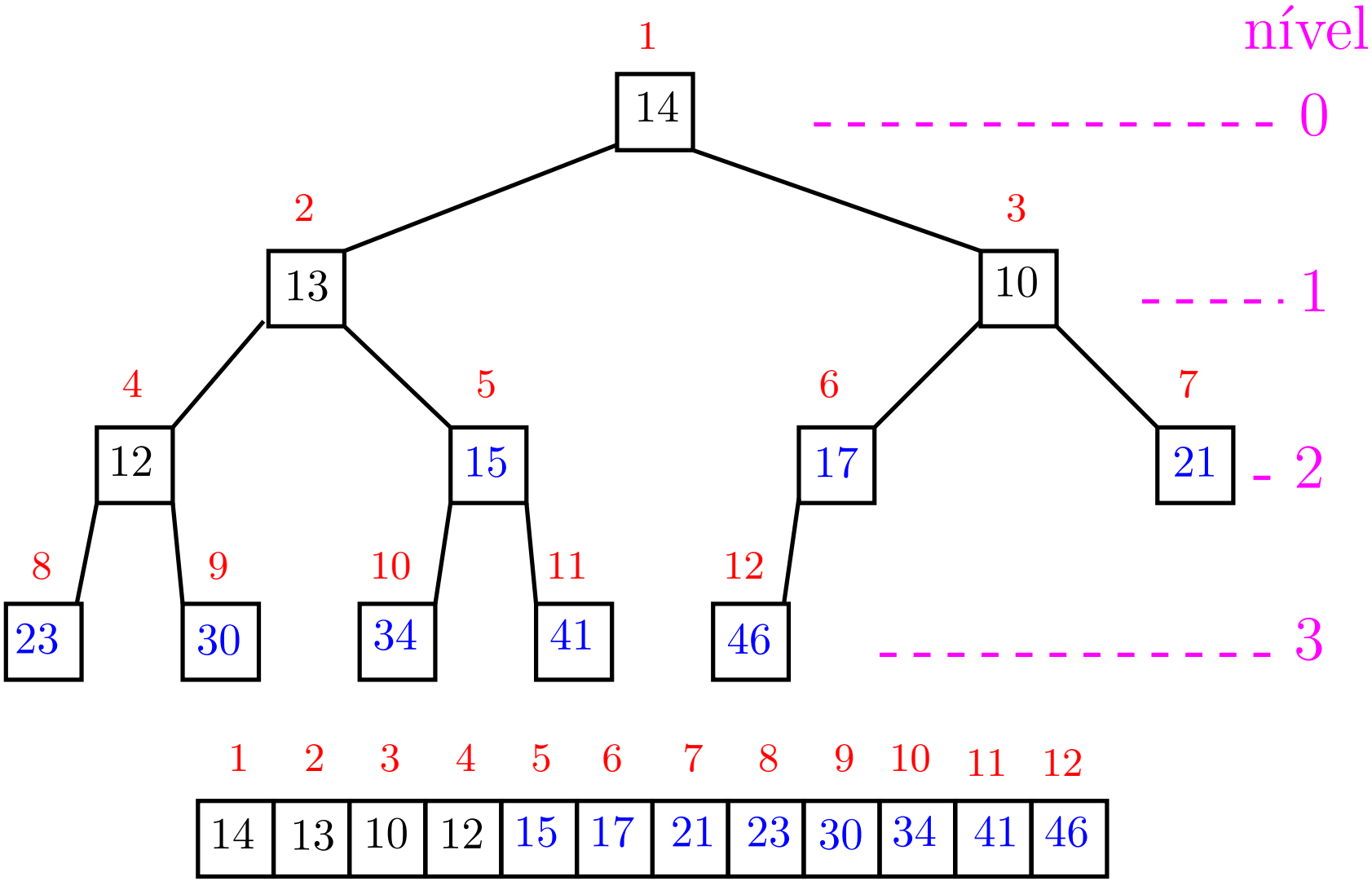
Heapsort


nível
1
10
0
2
13
3
14
1
4
12
5
15
6
17
7
21
2
8
23
9
30
10
34
11
41
12
46
3
1 2 3 4 5 6 7 8 9 10 11 12
10 13 14 12 15 17 21 23 30 34 41 46

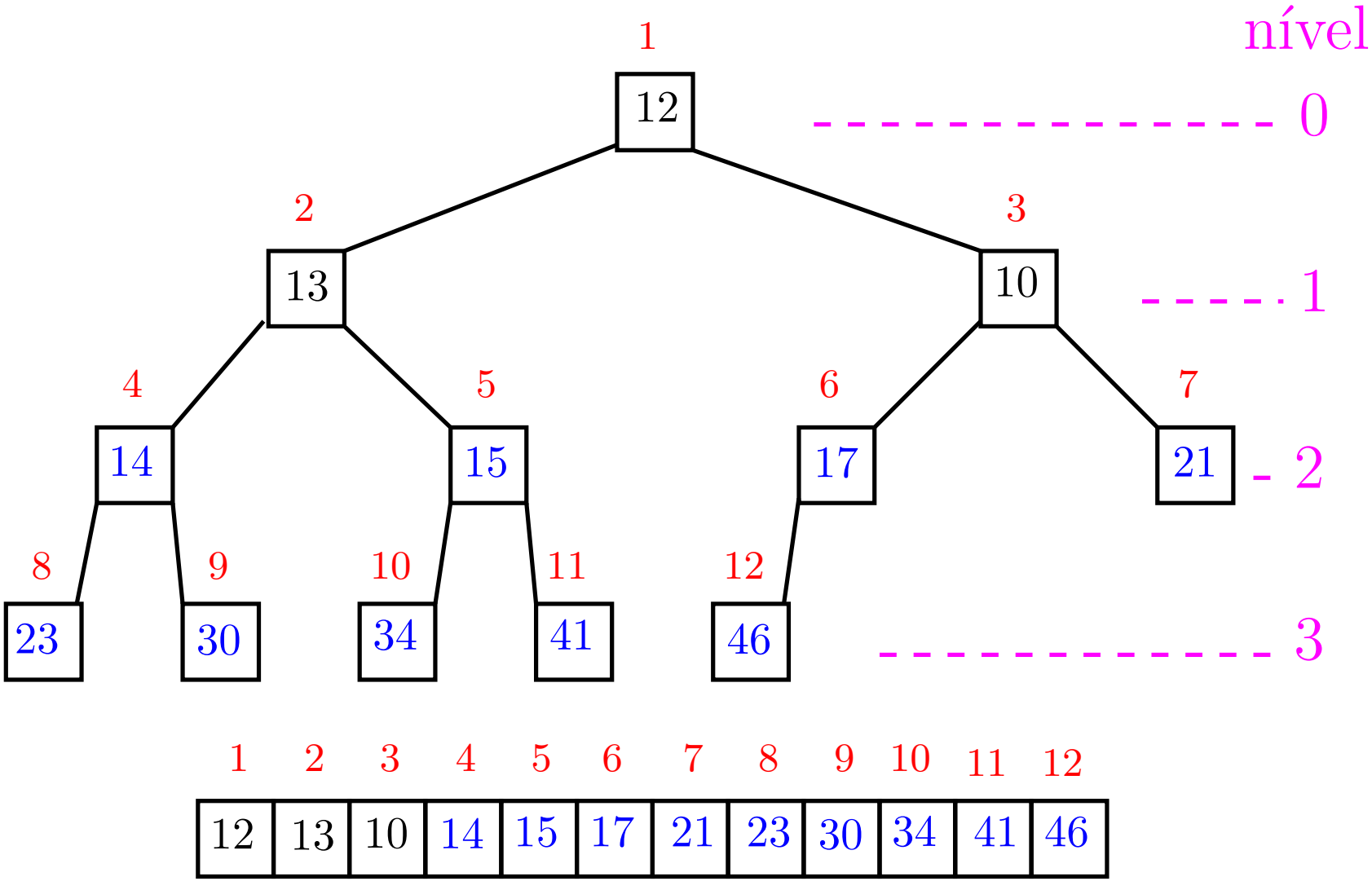
Heapsort


nível
1
14
0
2
13
3
10
1
4
12
5
15
6
17
7
21
2
8
23
9
30
10
34
11
41
12
46
3
1 2 3 4 5 6 7 8 9 10 11 12
14 13 10 12 15 17 21 23 30 34 41 46

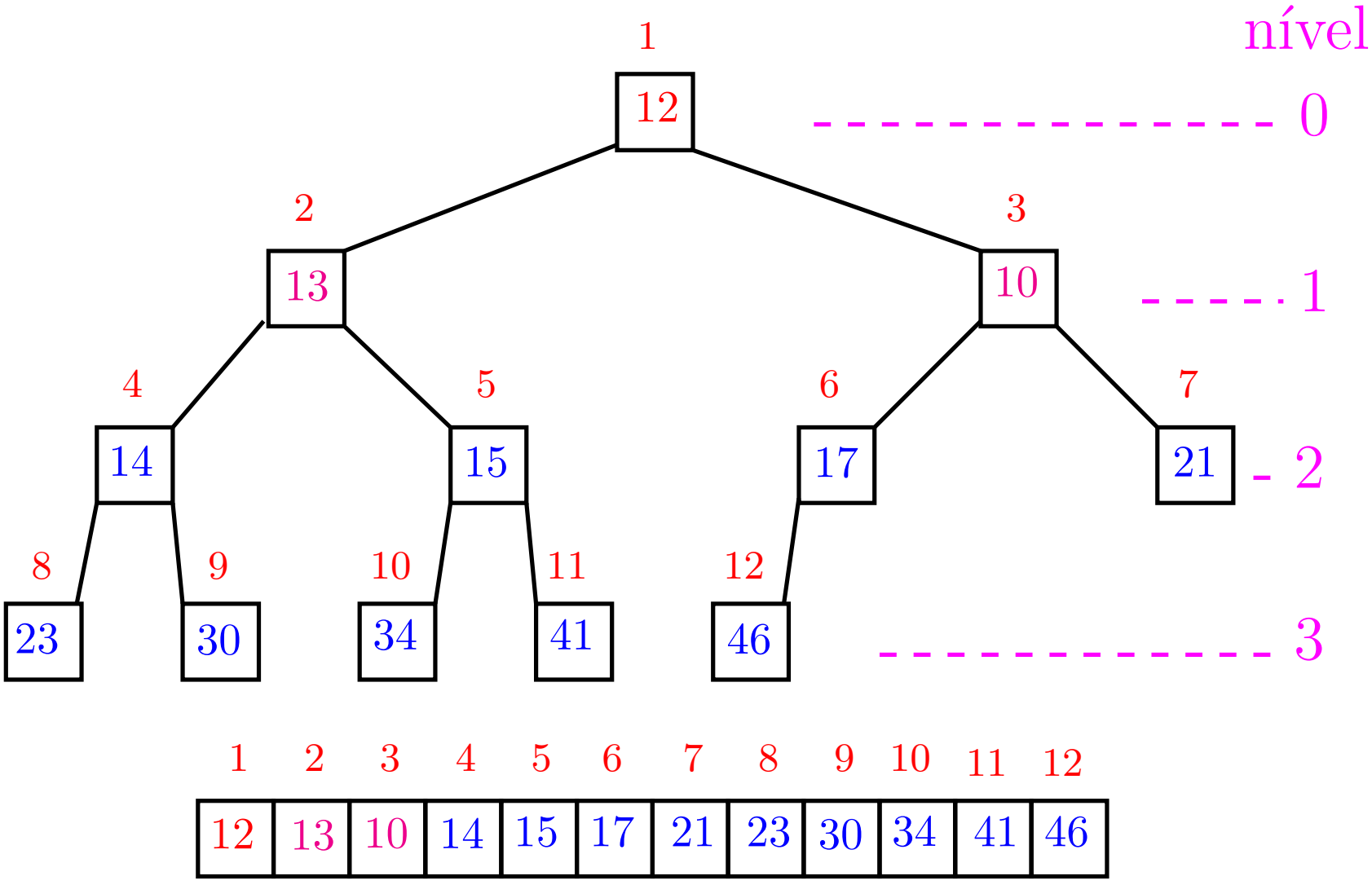
Heapsort


nível
1
14
0
2
3
13
10
1
4
5
6
7
12
15
17
21
2
8
9
10
11
12
23
30
34
41
46
3
1 2 3 4 5 6 7 8 9 10 11 12
14 13 10 12 15 17 21 23 30 34 41 46

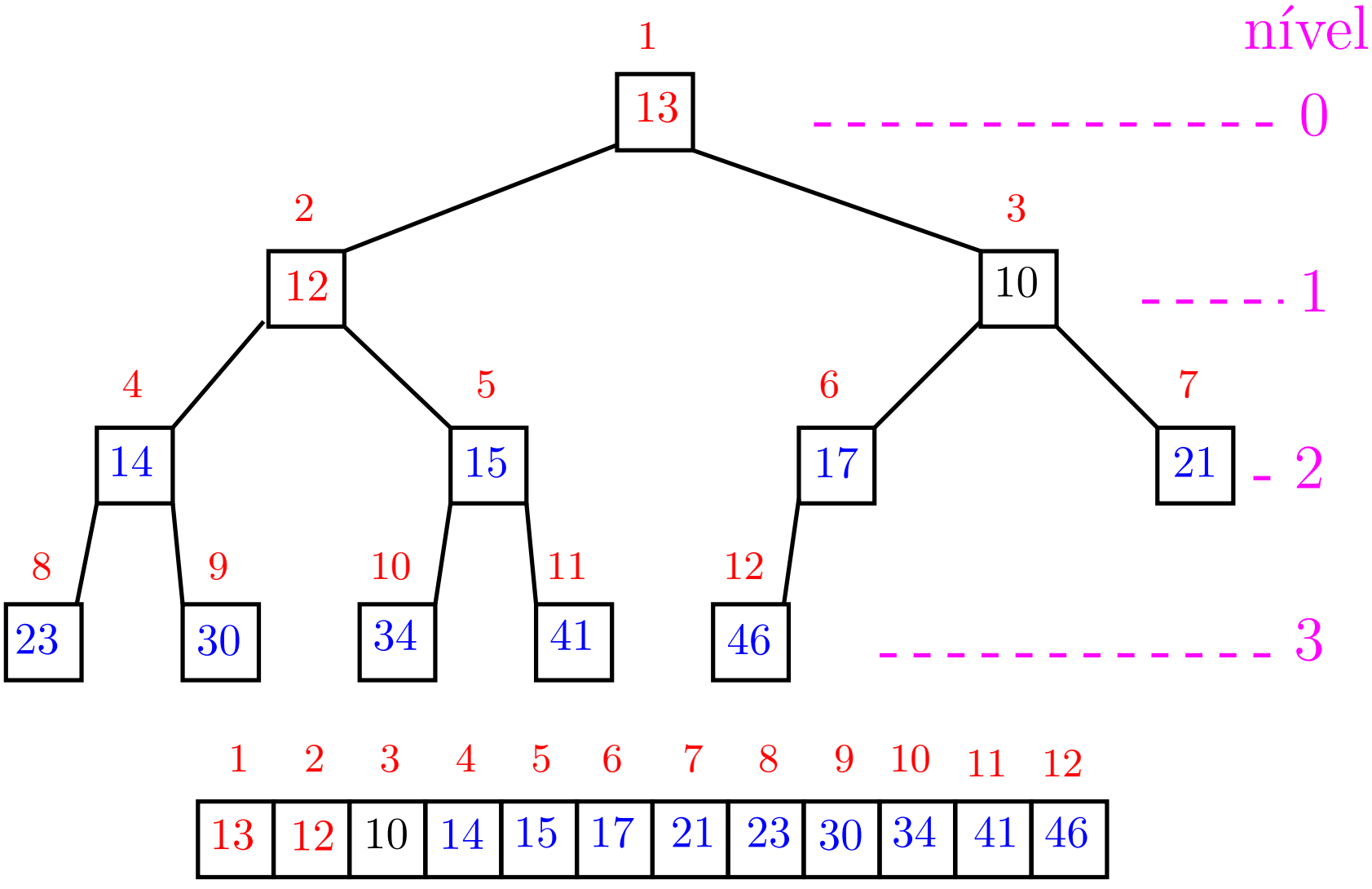
Heapsort


nível
1
12
0
2
13
3
10
1
4
14
5
15
6
17
7
21
2
8
23
9
30
10
34
11
41
12
46
3
1 2 3 4 5 6 7 8 9 10 11 12
12 13 10 14 15 17 21 23 30 34 41 46

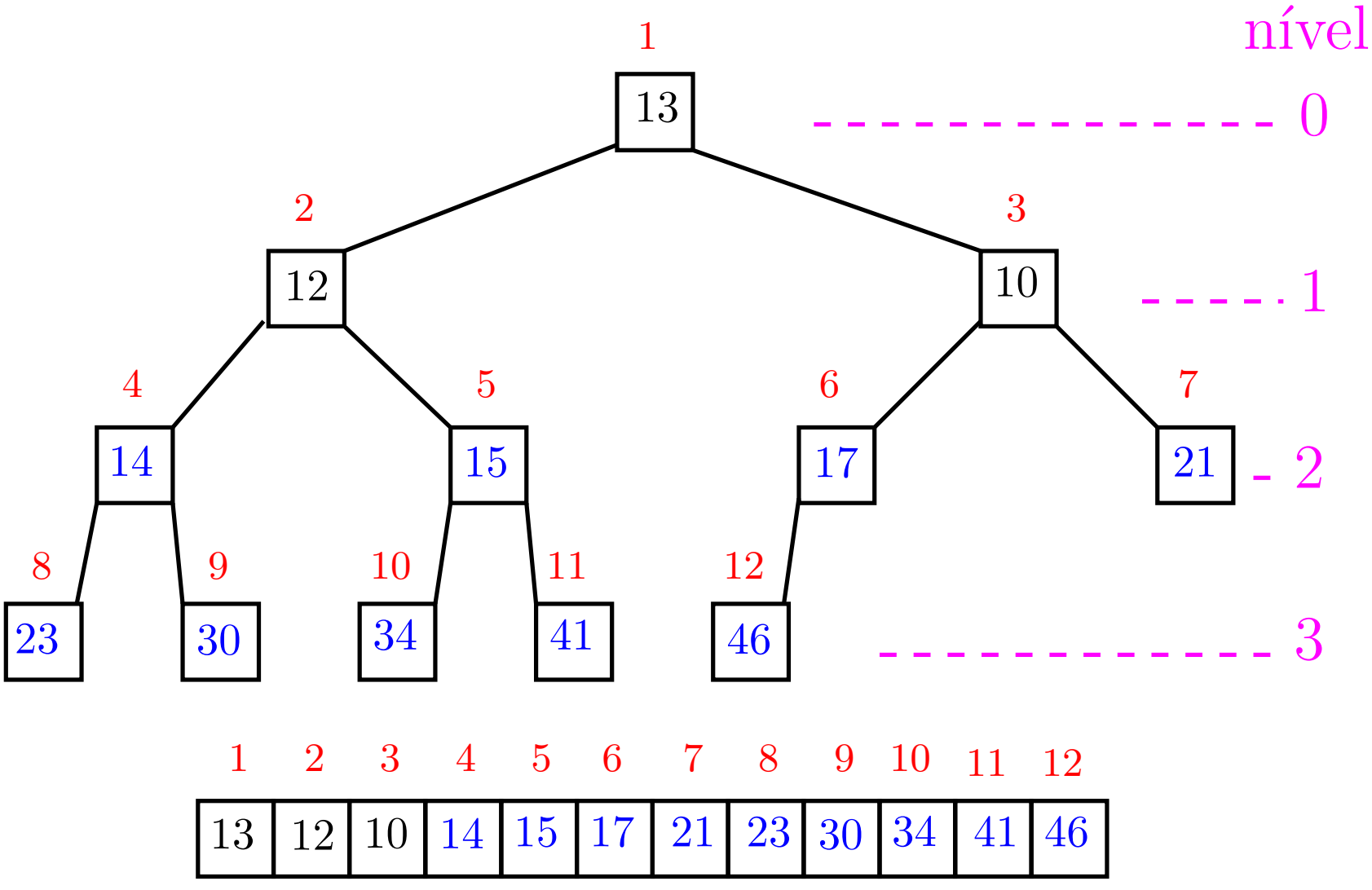
Heapsort


nível
1
12
0
2
13
3
10
1
4
14
5
15
6
17
7
21
2
8
23
9
30
10
34
11
41
12
46
3
1 2 3 4 5 6 7 8 9 10 11 12
12 13 10 14 15 17 21 23 30 34 41 46

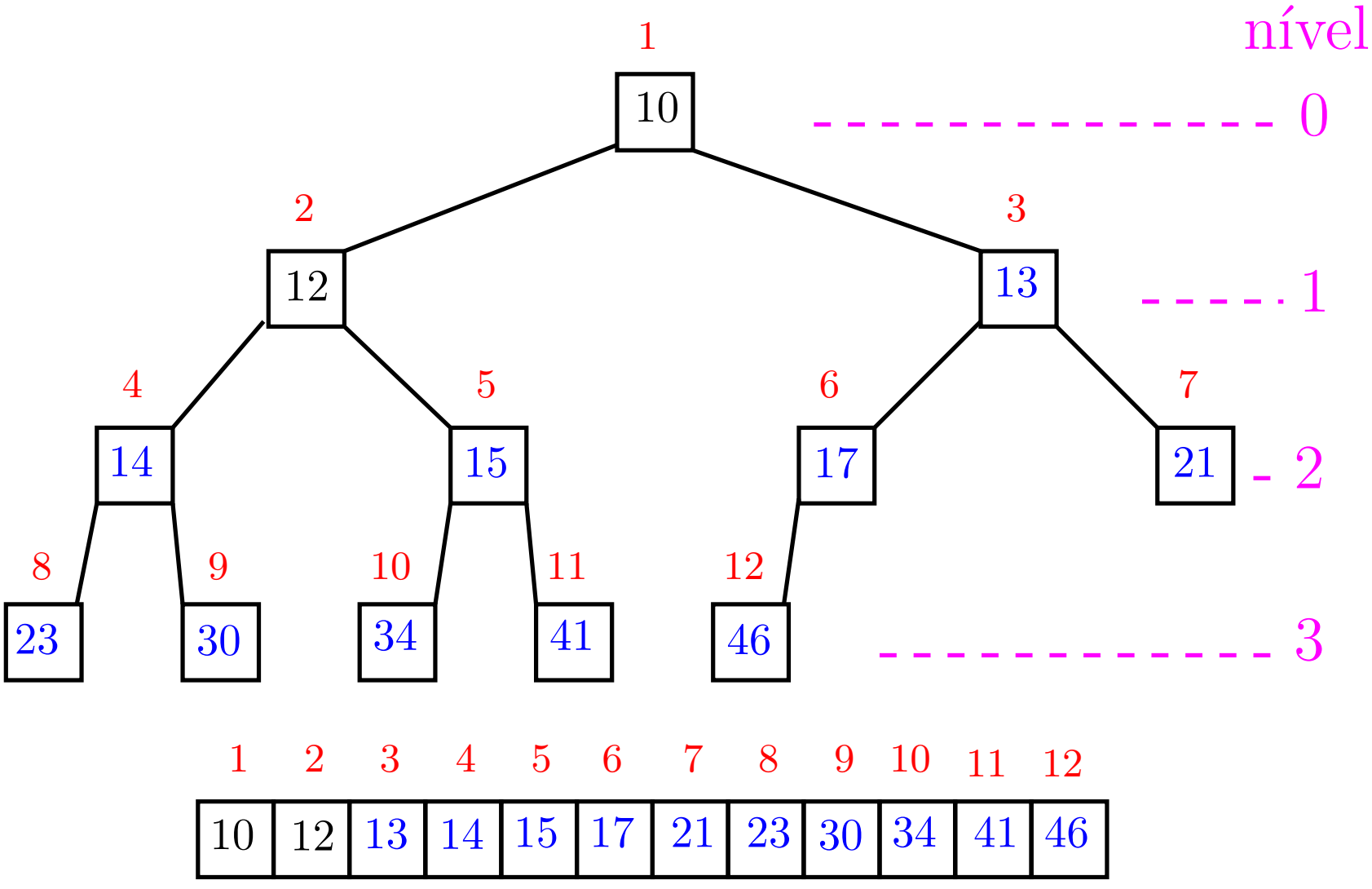
Heapsort


nível
1
13
0
2
12
3
10
1
4
14
5
15
6
17
7
21
2
8
23
9
30
10
34
11
41
12
46
3
1 2 3 4 5 6 7 8 9 10 11 12
13 12 10 14 15 17 21 23 30 34 41 46

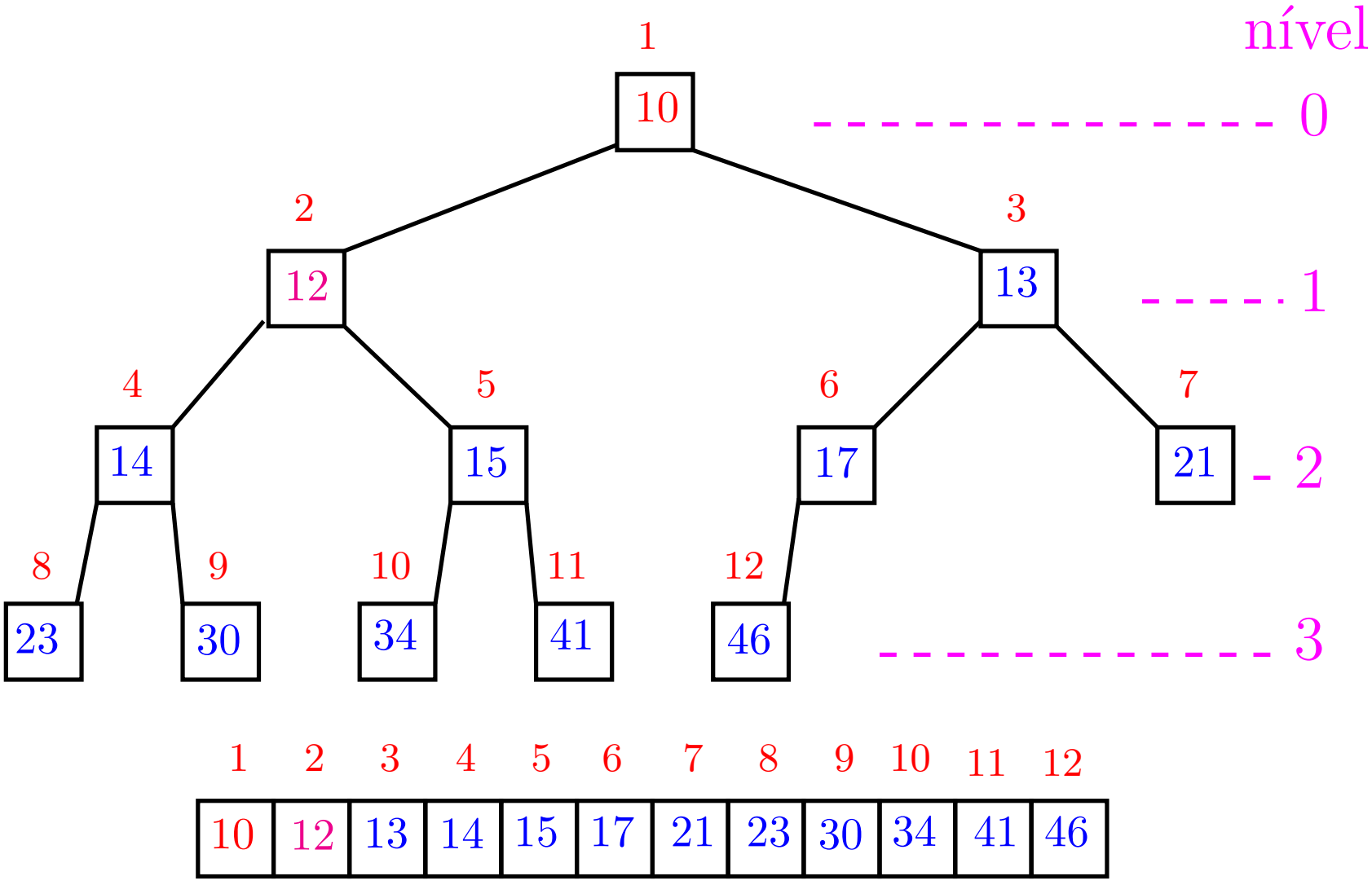
Heapsort


nível
1
13
0
2
12
3
10
1
4
14
5
15
6
17
7
21
2
8
23
9
30
10
34
11
41
12
46
3
1 2 3 4 5 6 7 8 9 10 11 12
13 12 10 14 15 17 21 23 30 34 41 46

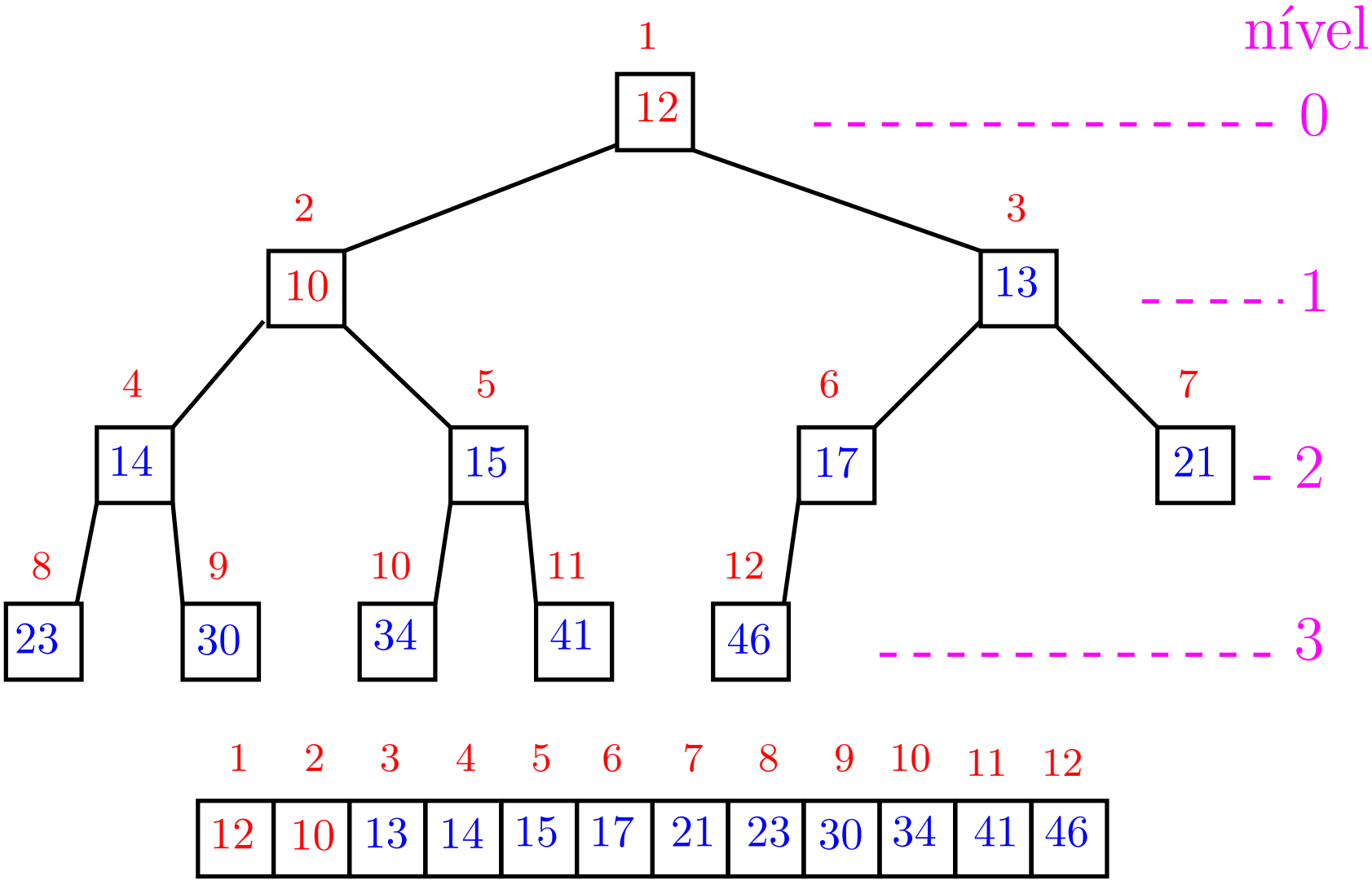
Heapsort


nível
1
10
0
2
12
3
13
1
4
14
5
15
6
17
7
21
2
8
23
9
30
10
34
11
41
12
46
3
1 2 3 4 5 6 7 8 9 10 11 12
10 12 13 14 15 17 21 23 30 34 41 46

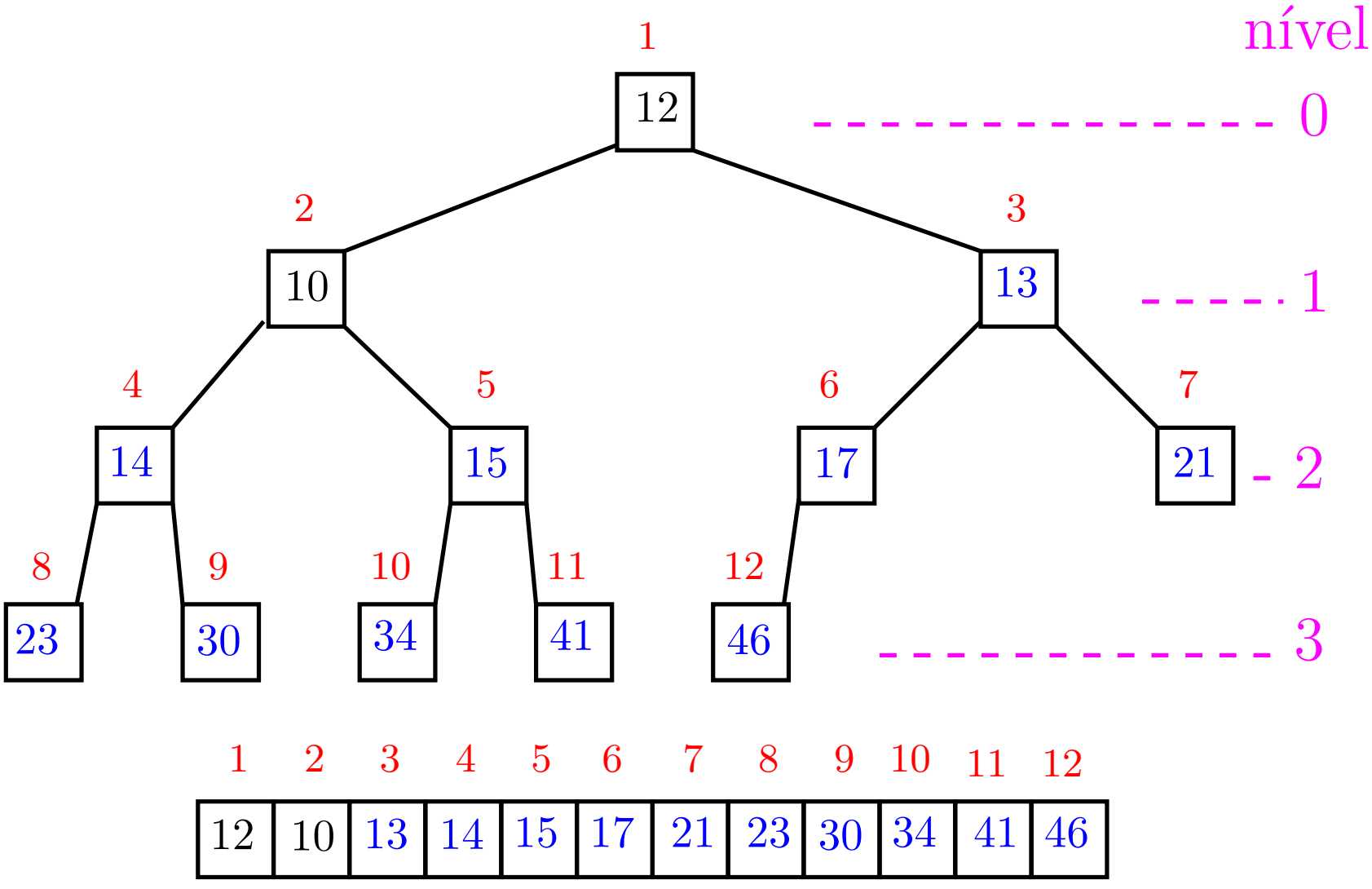
Heapsort


nível
1
10
0
2
12
3
13
1
4
14
5
15
6
17
7
21
2
8
23
9
30
10
34
11
41
12
46
3
1 2 3 4 5 6 7 8 9 10 11 12
10 12 13 14 15 17 21 23 30 34 41 46

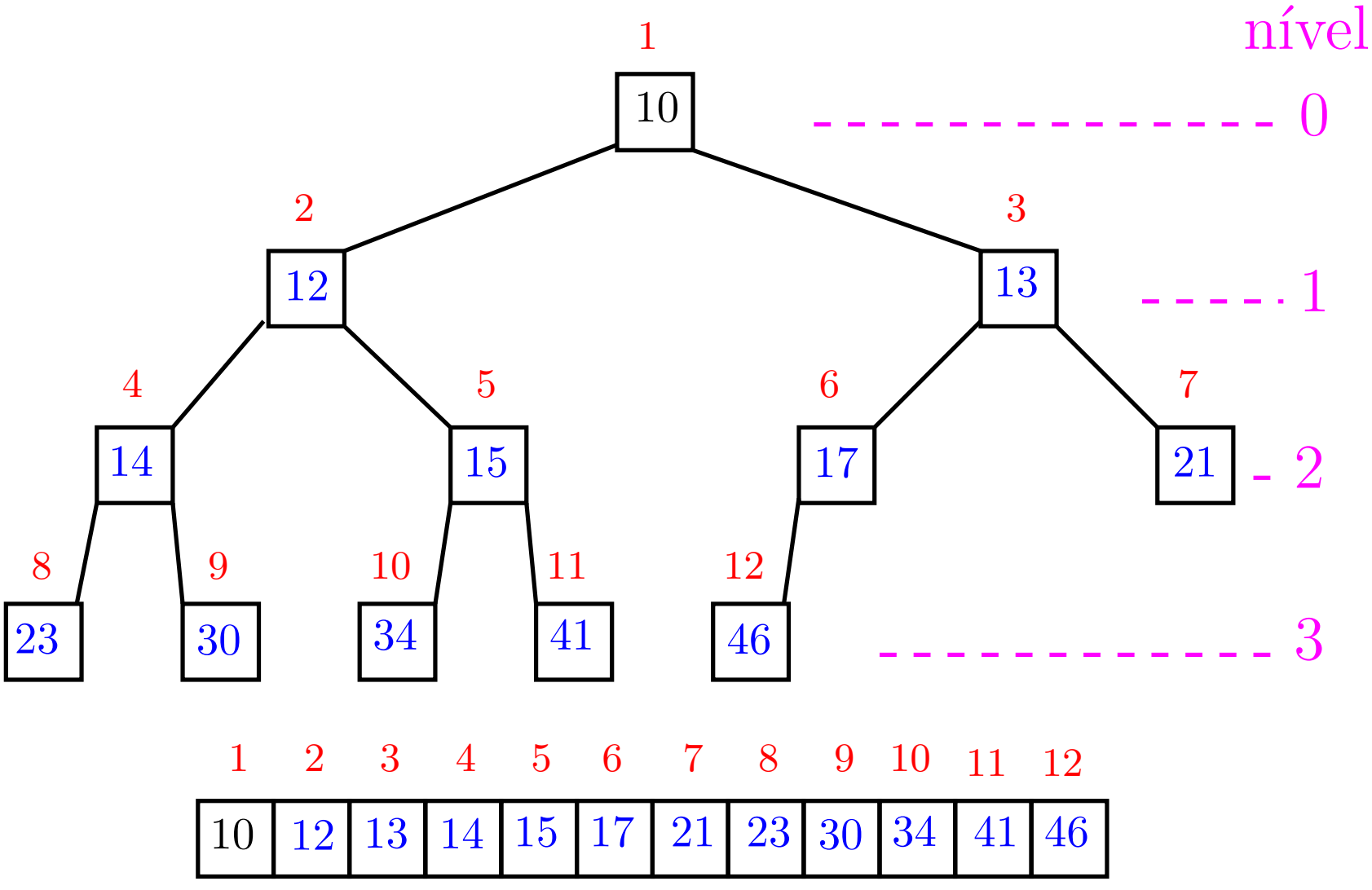
Heapsort


nível
1
12
0
2
10
3
13
1
4
14
5
15
6
17
7
21
2
8
23
9
30
10
34
11
41
12
46
3
1 2 3 4 5 6 7 8 9 10 11 12
12 10 13 14 15 17 21 23 30 34 41 46

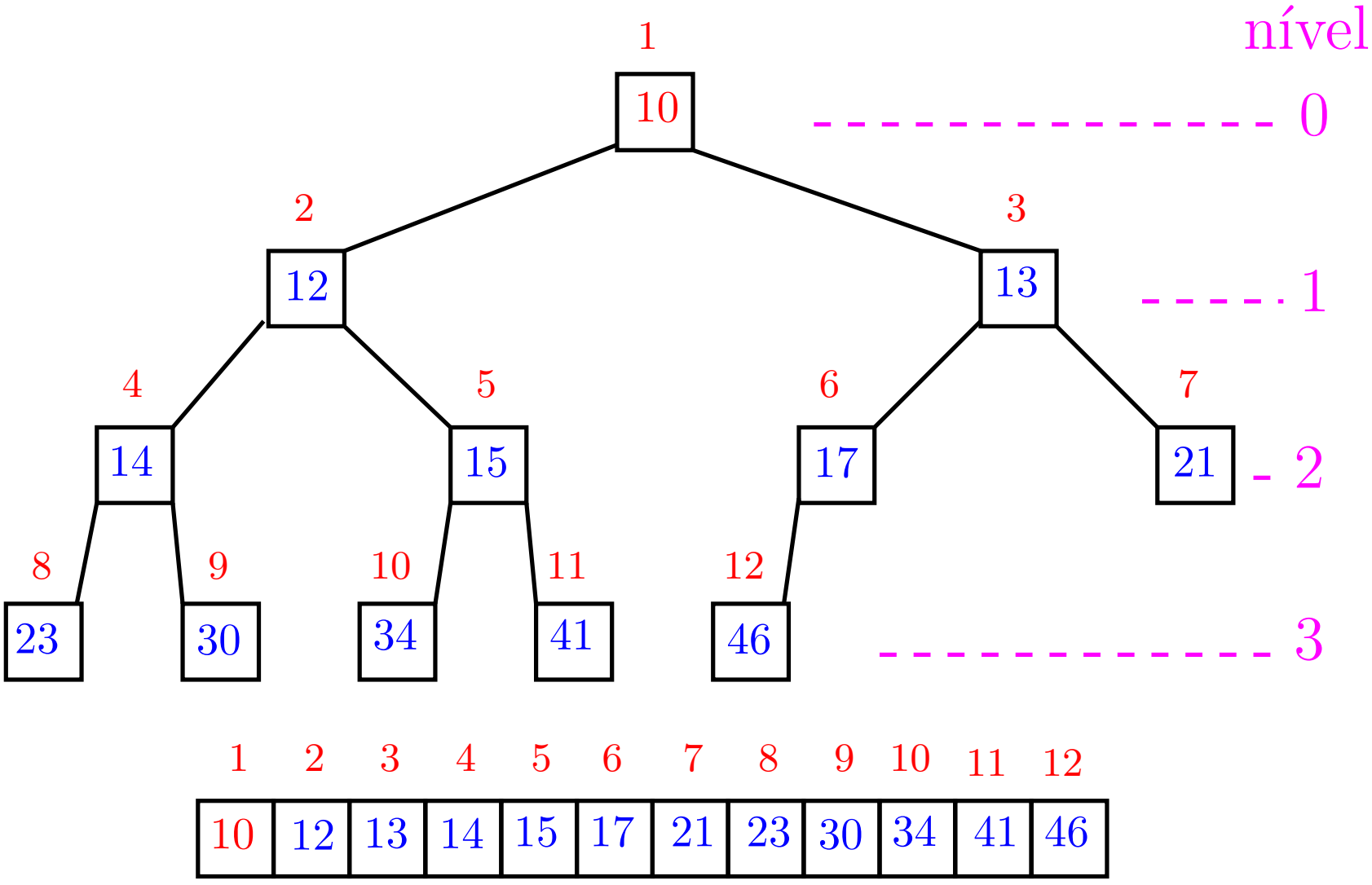
Heapsort


nível
1
12
0
2
10
3
13
1
4
14
5
15
6
17
7
21
2
8
23
9
30
10
34
11
41
12
46
3
1 2 3 4 5 6 7 8 9 10 11 12
12 10 13 14 15 17 21 23 30 34 41 46

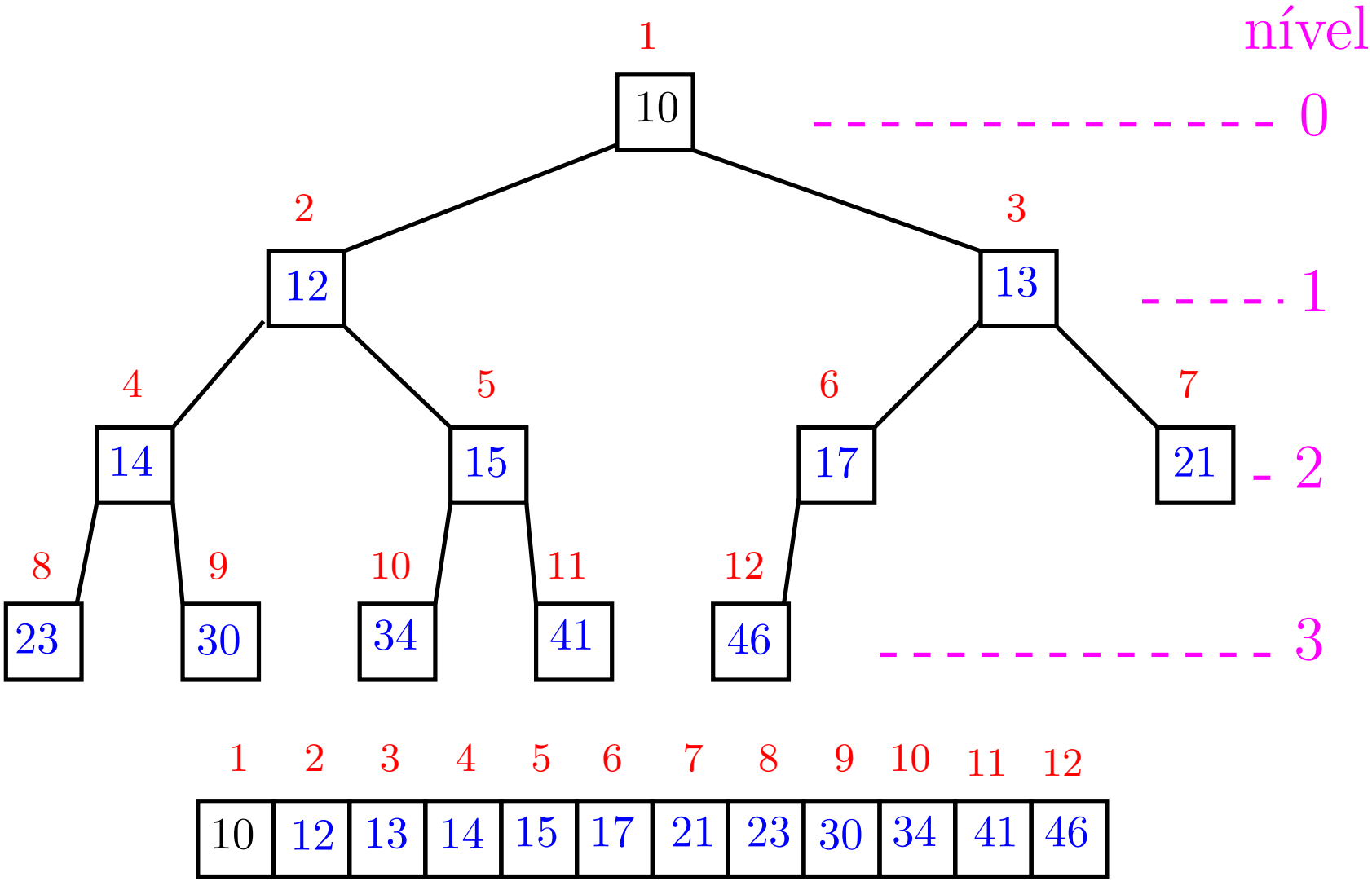
Heapsort


nível
1
10
0
2
12
3
13
1
4
14
5
15
6
17
7
21
2
8
23
9
30
10
34
11
41
12
46
3
1 2 3 4 5 6 7 8 9 10 11 12
10 12 13 14 15 17 21 23 30 34 41 46

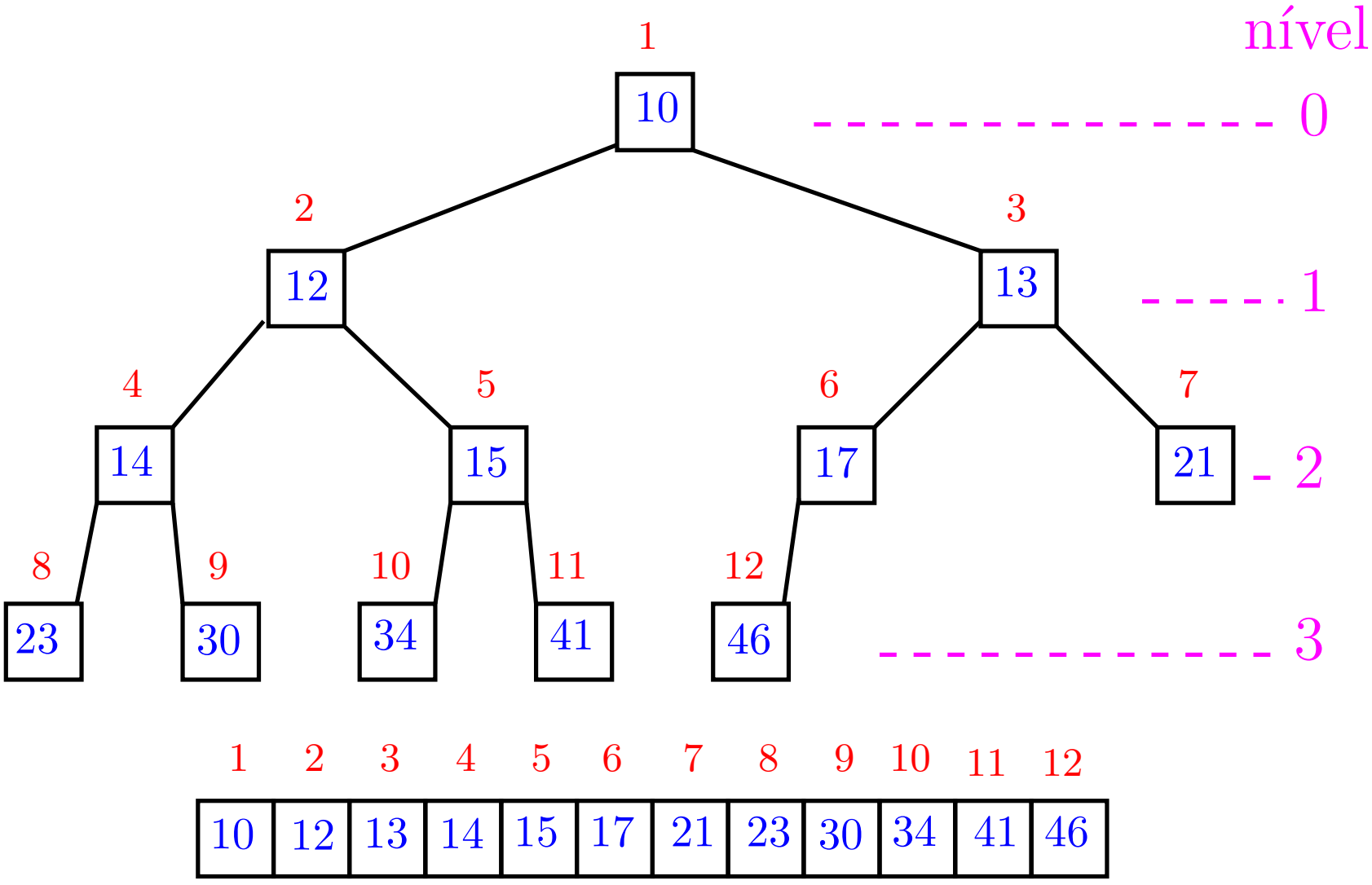
Heapsort


nível
1
10
0
2
12
3
13
1
4
14
5
15
6
17
7
21
2
8
23
9
30
10
34
11
41
12
46
3
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


nível
1
10
0
2
12
3
13
1
4
14
5
15
6
17
7
21
2
8
23
9
30
10
34
11
41
12
46
3
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


nível
1
10
0
2
12
3
13
1
4
14
5
15
6
17
7
21
2
8
23
9
30
10
34
11
41
12
46
3
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);
    }
 }
```

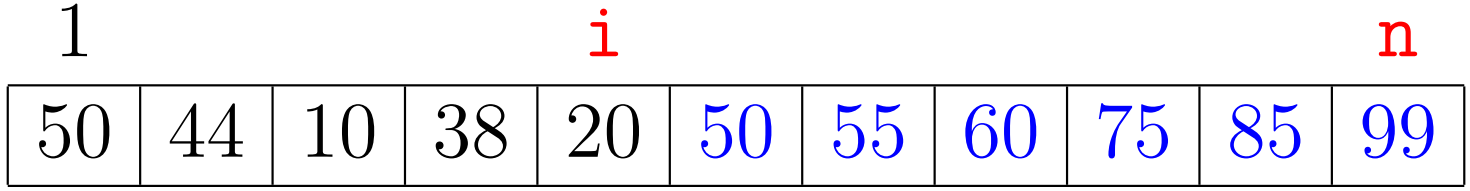
Função heapSort

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.



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 $\mathtt{n} \lg \mathtt{n}$.

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

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 na Sapientia University (Romania):

https://www.youtube.com/channel/UCIqiLefbVHsOAXDAxQJH7Xw