MAC0323 Algoritmos e Estruturas de Dados II

Edição 2020 – 2

Fonte: ash.atozviews.com

Compacto dos melhores momentos

AULA 4

Union-find

1.5 Case Study: Union-Find

Interface uf.h

Arquivo	uf.h	
`void`	`ufInit(int n)`	inicializa `n` sites com nomes inteiros `0,...,n-1`
`void`	`ufUnion(int p, int q)`	acrescenta ligação entre `p` e `q`
`int`	`ufFind(int p)`	retorna `id` do componente de `p`
`bool`	`ufConnected(int p,int q)`	`true` se `p` e `q` estão no mesmo componente
`int`	`ufCount()`	número de componentes
`void`	`ufFree()`	destroi a ED

Estratégias

`QuickFindUF`:
Cada elemento mantém o `id` do seu conjunto.
`ufFind` é rápido, mas `ufUnion` pode ser lento.

`QuickUnionUF`:
Cada conjunto é uma árvore enraizada.
A raiz da árvore é o representante do conjunto.
`ufUnion` é rápido (descontados os `ufFind`s),
mas `ufFind` pode ser lento.

`WeightedQuickUnionUF`: No `ufUnion`,
pendurar sempre a árvore mais baixa na mais alta.
O `ufFind` e o `ufUnion` ficam bem mais rápidos!

Experimentos

```
% time client < tinyUF.txt
```
2 components
0.0003seg

```
% time client < mediumUF.txt
```
3 components
0.005seg

```
% time client < largeUF.txt
```
6 components
3.726seg

Resumo

	`ufInit()`	`ufFind()`	`ufUnion()`
`QuickFindUF`	$\Theta(n)$	$\Theta(1)$	$O(n)$
`QuickUnionUF`	$\Theta(n)$	$O(n)$	$O(n)$
`WeightedQuickUnionUF`	$\Theta(n)$	$O(\lg n)$	$O(\lg n)$

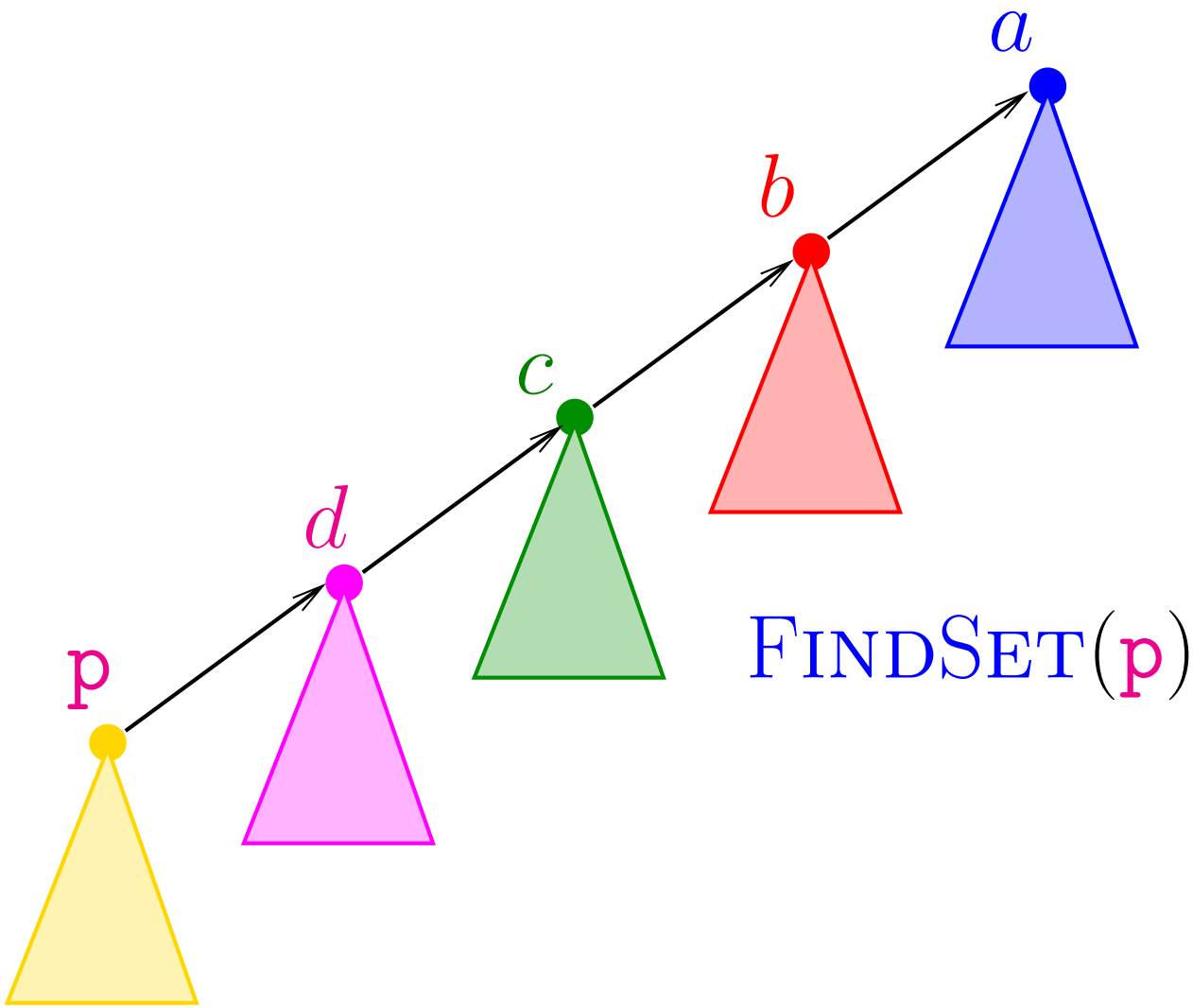
Fonte: Pinterest

AULA 5

Path compression

Ideia:

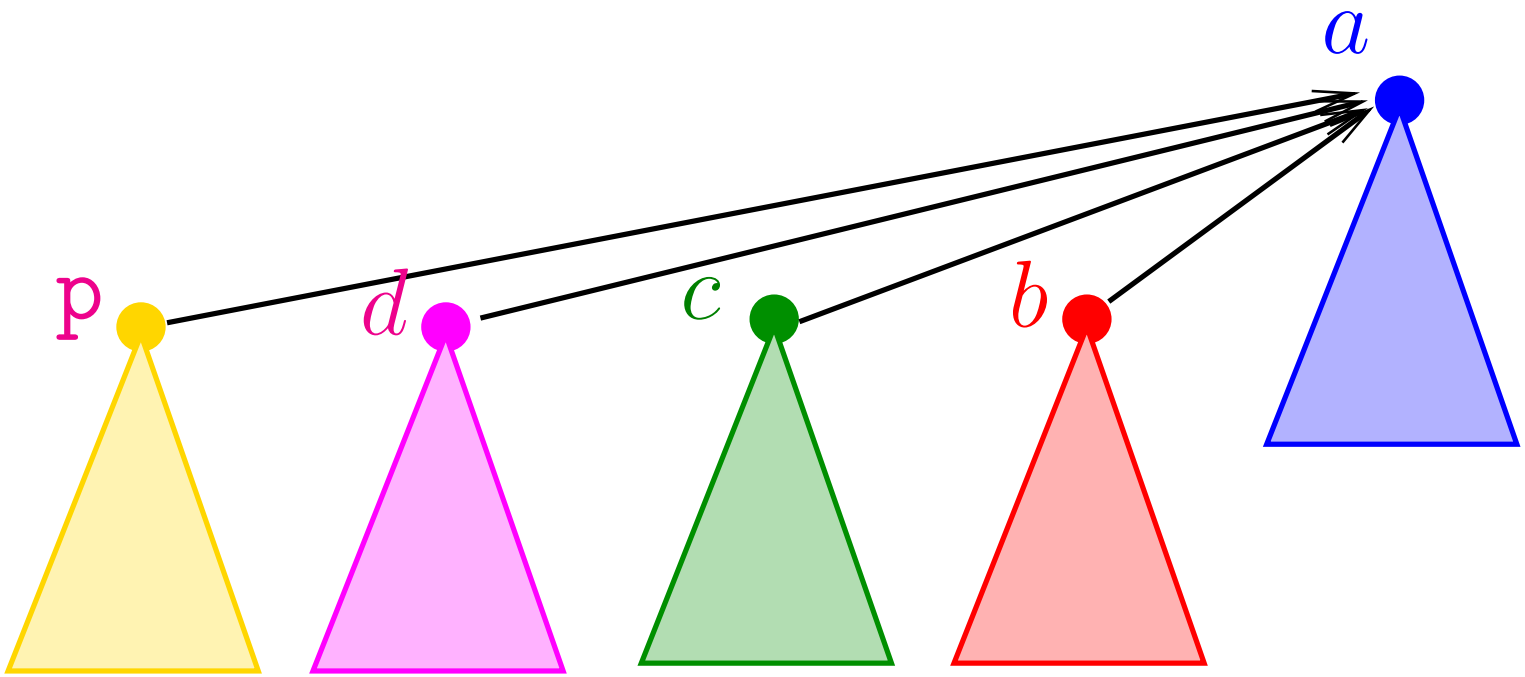
encurtar os caminhos durante cada `ufFind()`.



Path compression

Ideia:

encurtar os caminhos durante cada `ufFind()`.



FindSet(p)

path compression

```
FindSet (p)     ▷ com "path compression"
1    se p ≠ pai[p]
2        então pai[p] ← FindSet (pai[p])
3    devolva pai[p]
```

path compression

```
FindSet (p)    ▷ com "path compression"
1   se p ≠ pai[p]
2       então pai[p] ← FindSet (pai[p])
3   devolva pai[p]
```

```
int ufFind(int p) {
   if (p != pai[p])
      pai[p] = ufFind(pai[p]);
   return pai[p];
}
```

Função log-estrela

$\lg^* n$ é o menor k tal que

$$\underbrace{\lg \lg \ldots \lg}_{k} n \leq 1$$

n	$\lg^* n$
1	0
2	1
3	2
4	2
5	3
$\vdots$	$\vdots$
15	3
16	3
$\vdots$	$\vdots$
65535	4
65536	4
$\vdots$	$\vdots$
$\underbrace{10000000000000000\cdots000000000000}_{80}$	5
$\vdots$	$\vdots$

Função ‘torre’

$$t(i) := \begin{cases} 1 & \text{se } i = 0 \\ 2^{t(i-1)} & \text{se } i = 1, 2, 3, \ldots \end{cases}$$

i	$t(i)$
0	1
1	$2^1 = 2$
2	$2^{2^2} = 16$
3	$2^{2^{2^2}} = 16$
4	$2^{2^{2^{2^2}}} = 2^{16} = 65536$
5	$2^{2^{2^{2^{2^2}}}} > \underbrace{10000000000000000 \cdots 000000000000}_{80}$
$\vdots$	$\vdots$

Consumo de tempo

ufInit(n)	$\Theta(n)$	
ufFind(p)	$O(\lg^* n)$	**amortizado!**
ufUnion(p, q)	$O(\lg^* n)$	**amortizado!**

M M M U F U U F U F F F U F

(the first three M's: n; the whole sequence: m)

Custo total da sequência:

$$\Theta(n) + m\,O(\lg^* n) + n\,O(\lg^* n) = O(m \lg^* n)$$

Conclusões

Se conjuntos disjuntos são representados através de disjoint-set forest com *union by rank* e *path compression*, então uma sequência de `ufInit(n)` e `m` operações `ufUnion()` e `ufFind()` consome tempo $\mathrm{O}(\mathtt{m}\,\lg^*\mathtt{n})$.

Experimentos

```
% time client < tinyUF.txt
```
2 components
0.0003seg

```
% time client < mediumUF.txt
```
3 components
0.004seg

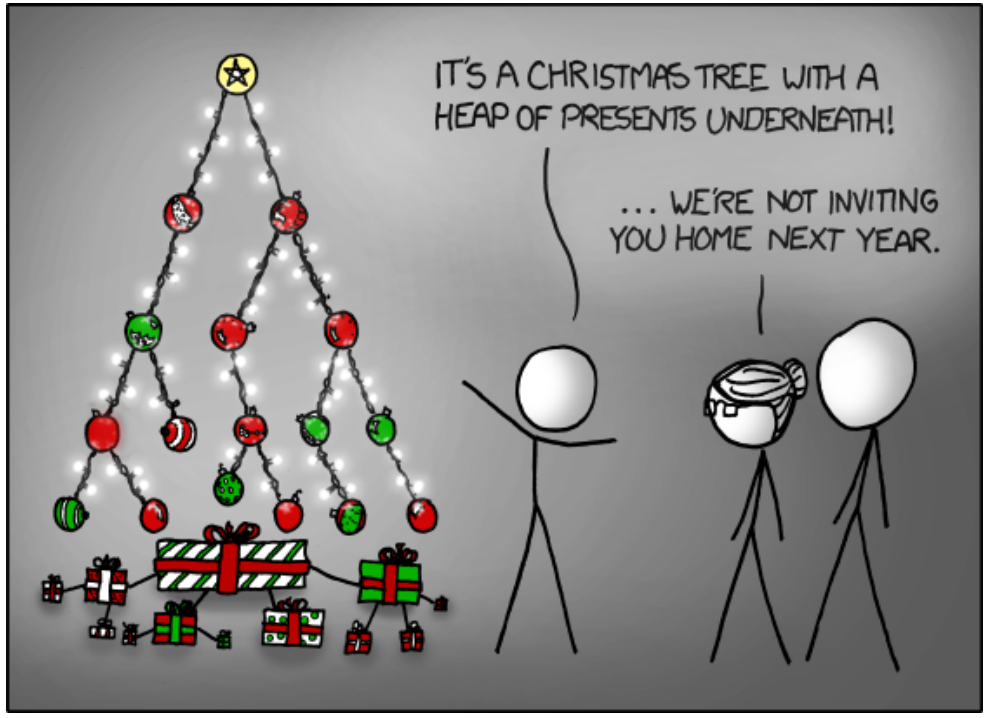
```
% time client < largeUF.txt
```
6 components
3.522seg

Parece que na prática weighted quick-union
e weighted quick-union com path-compression
não são muito diferentes.

Resumo

	`ufInit()`	`ufFind()`	`ufUnion()`
`QuickFindUF`	$\Theta(n)$	$\Theta(1)$	$O(n)$
`QuickUnionUF`	$\Theta(n)$	$O(n)$	$O(n)$
`WeightedQuickUnionUF`	$\Theta(n)$	$O(\lg n)$	$O(\lg n)$
`PathCompressionUF`	$\Theta(n)$	$O(\lg^* n)$	$O(\lg^* n)$

Árvores em vetores e heaps

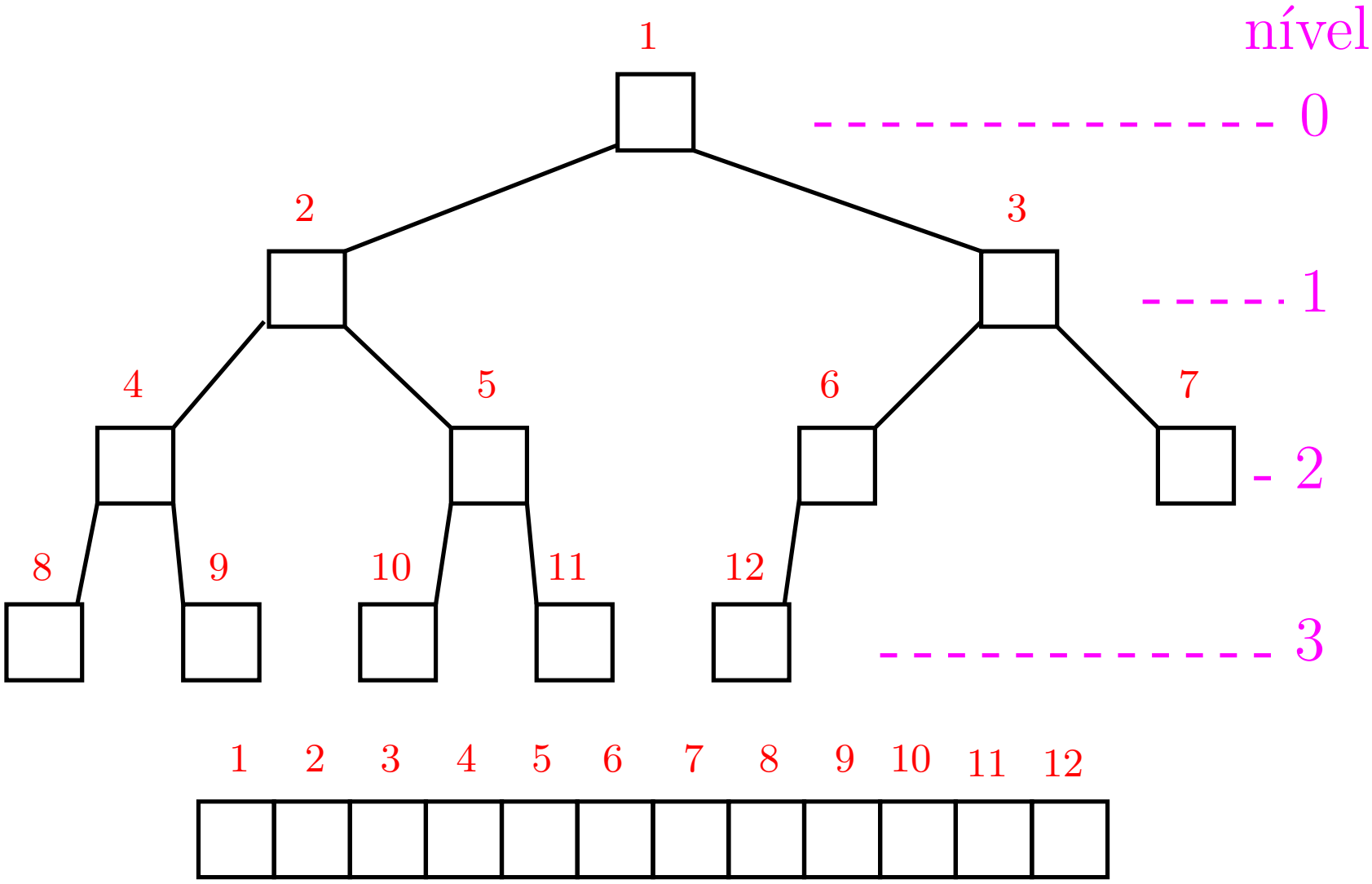


Fonte: http://xkcd.com/835/

PF 10

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

Representação de árvores em vetores



Pais e filhos

$\mathtt{a}[1\,..\,\mathtt{m}]$ 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 $2\,i \leq \mathtt{m}$.

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

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, $2\,i > \mathtt{m}$.

Todo nó i é raiz da subárvore formada por

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

Níveis

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

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

Níveis

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

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

O nó i pertence ao nível ???.

Níveis

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

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

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

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

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

Prova: Se p é o nível do nó i, então

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

Logo, $\mathrm{p} = \lfloor \lg i \rfloor$.

Níveis

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

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

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

Prova: Se p é o nível do nó i, então

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

Logo, $\mathrm{p} = \lfloor \lg i \rfloor$.

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

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 i pertence ao nível $\lfloor \lg i \rfloor$.

Prova: Se $\mathtt{p}$ é o nível do nó i, então

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

Logo, $\mathtt{p} = \lfloor \lg i \rfloor$.

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

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

onde $\texttt{filho}(i)$ 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 sequência da forma

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

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

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

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

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

Heaps

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

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

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

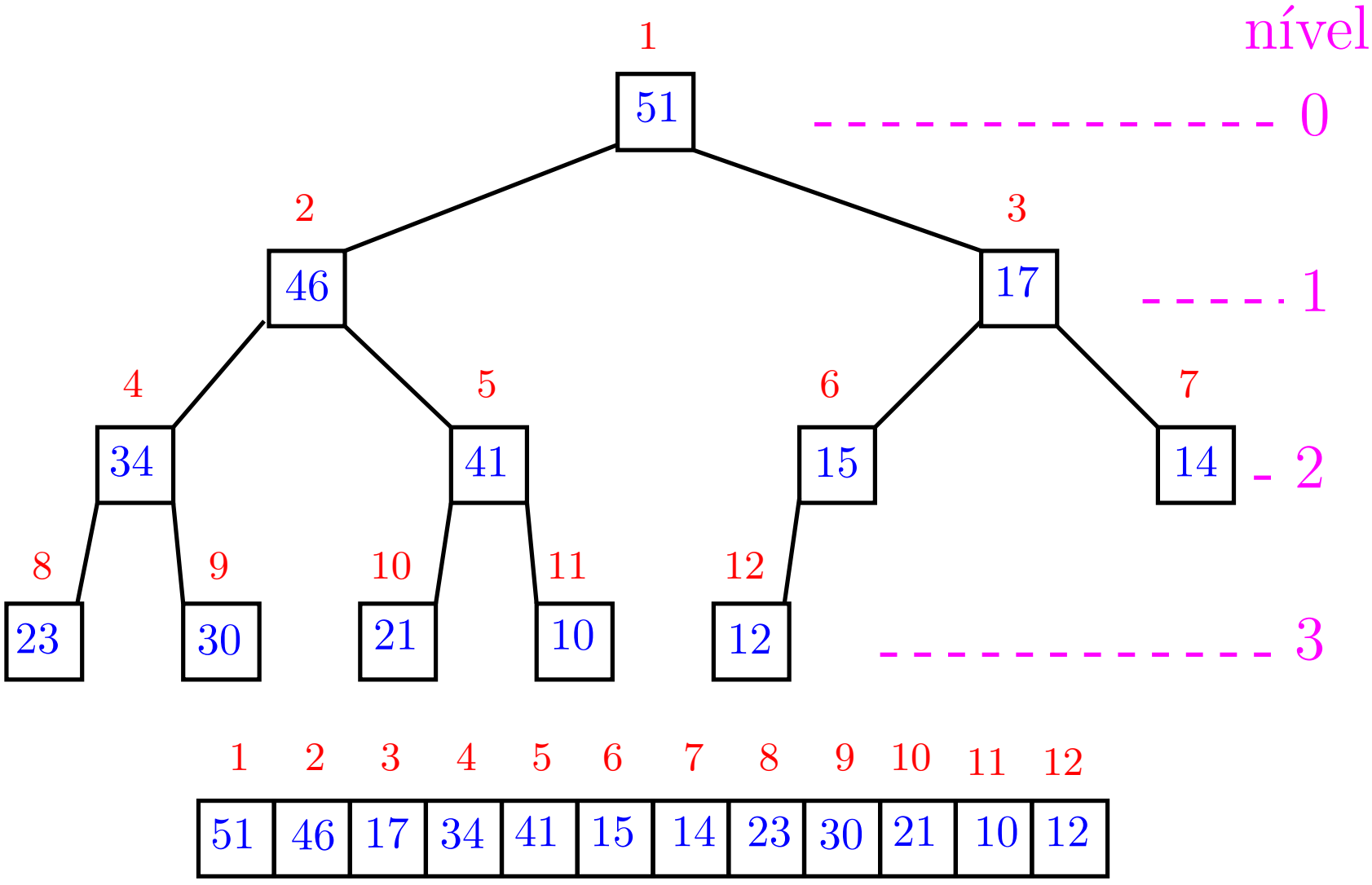
De forma mais geral, $\mathtt{a}[\mathtt{j}\,..\,\mathtt{m}]$ é um **max-heap** se

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

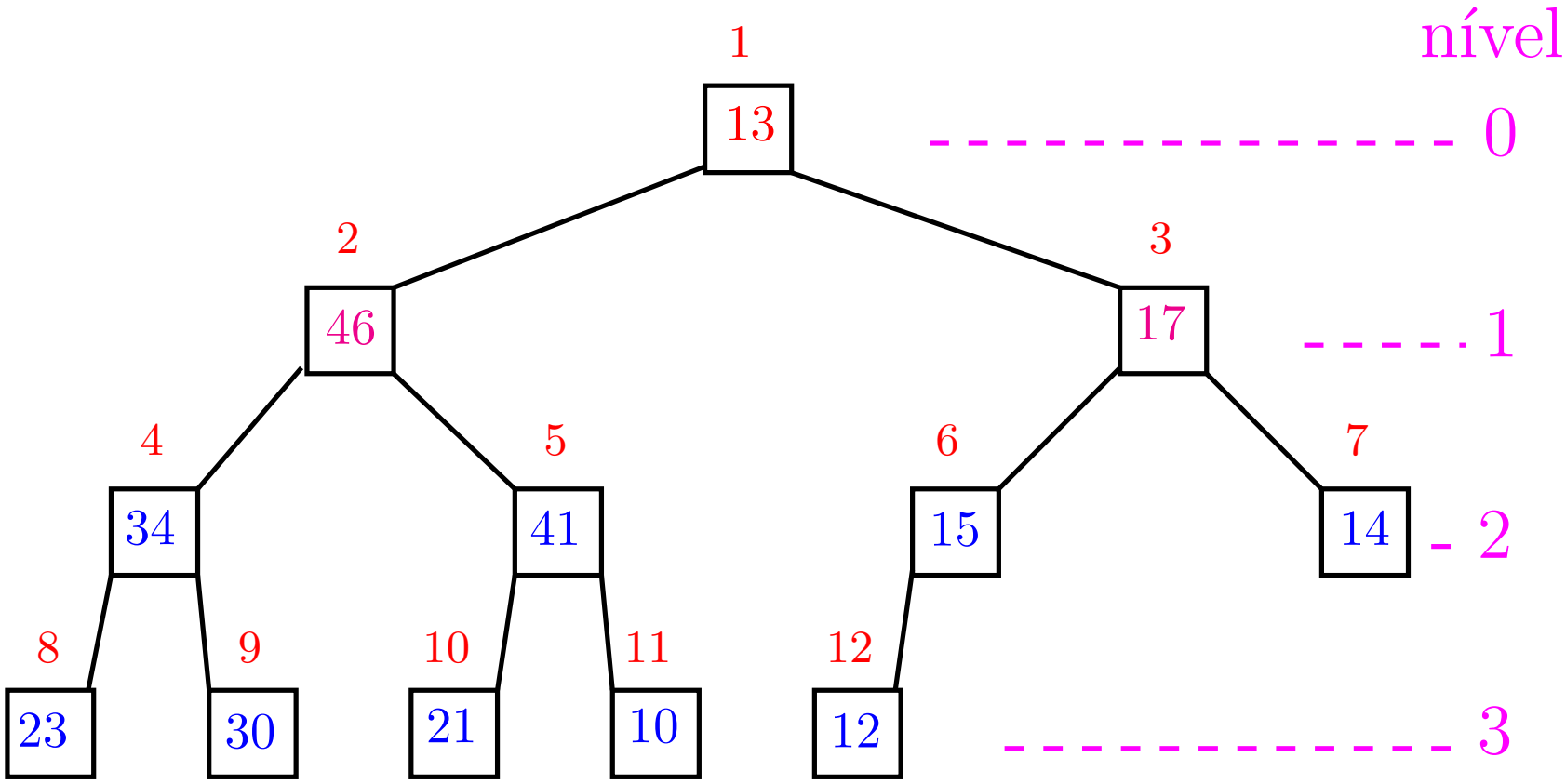
para todo $i = 2\mathtt{j}, 2\mathtt{j}+1, 4\mathtt{j}, \ldots, 4\mathtt{j}+3,$
$8\mathtt{j}, \ldots, 8\mathtt{j}+7, \ldots$

Neste caso também diremos que
a subárvore com raiz $\mathtt{j}$ é um **max-heap**.

max-heap

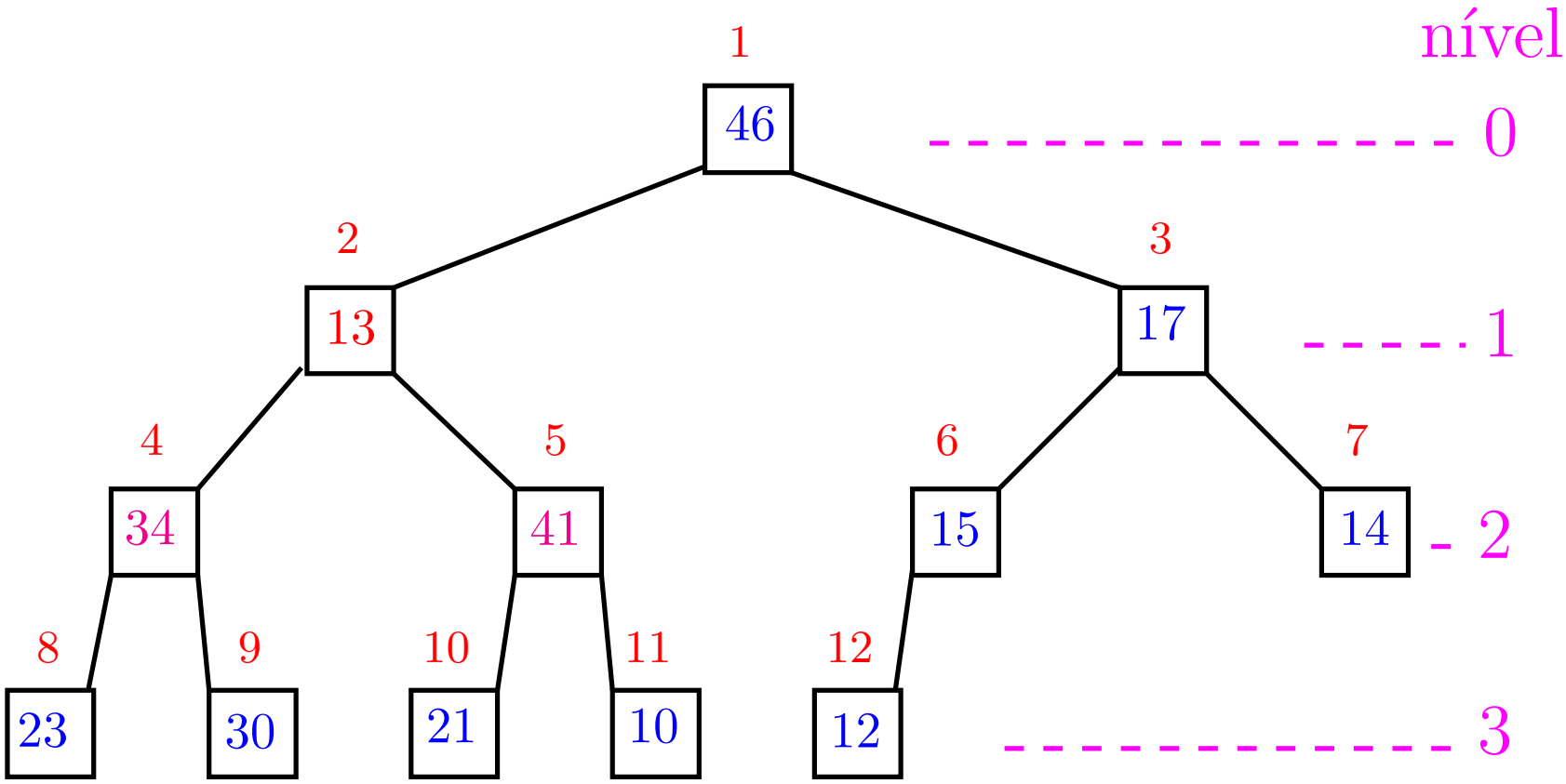

nível
1
51
0
2
46
3
17
1
4
34
5
41
6
15
7
14
2
8
23
9
30
10
21
11
10
12
12
3
1 2 3 4 5 6 7 8 9 10 11 12
51 46 17 34 41 15 14 23 30 21 10 12

Função básica de manipulação de max-heap



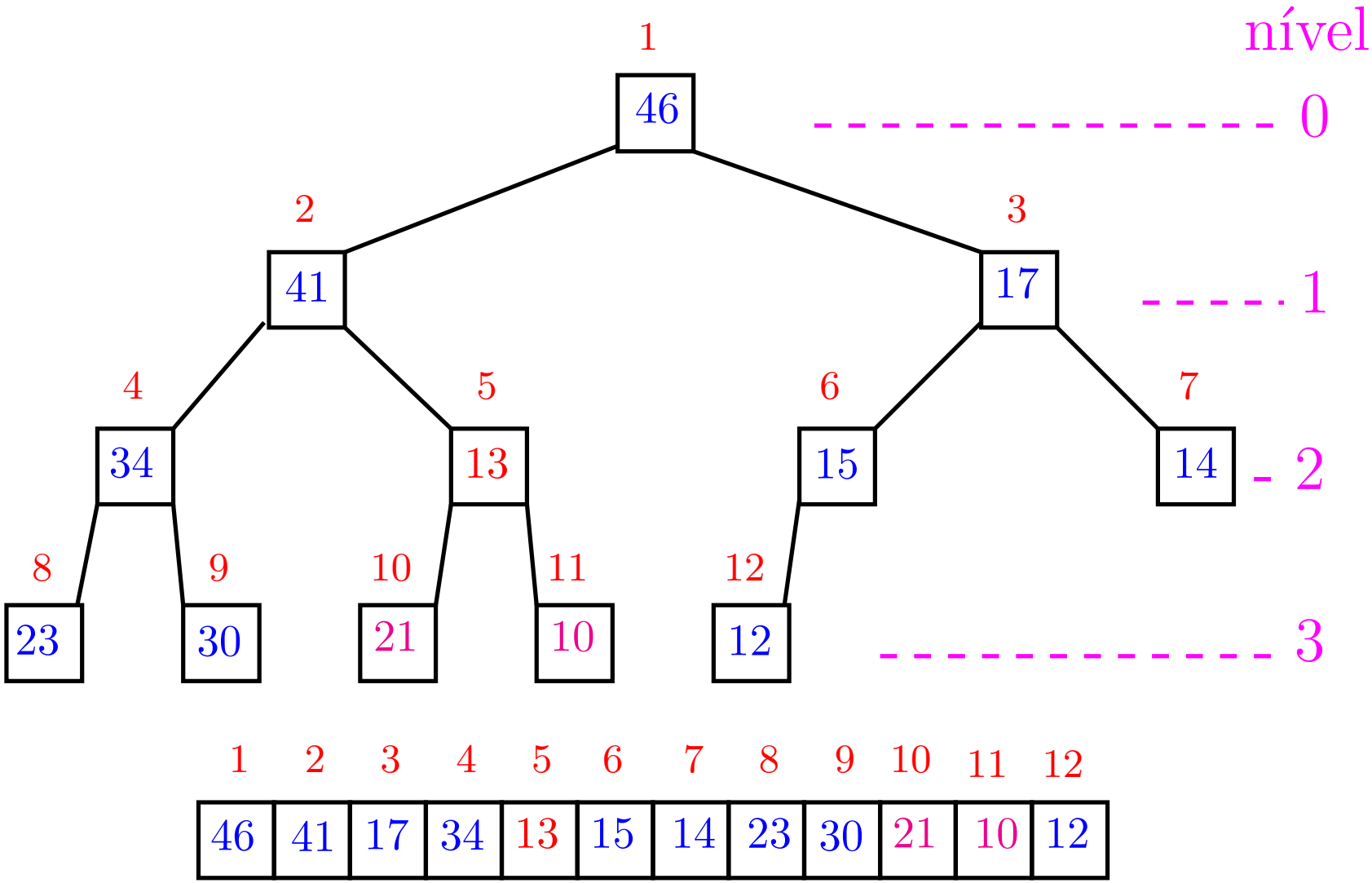
1	2	3	4	5	6	7	8	9	10	11	12
13	46	17	34	41	15	14	23	30	21	10	12

Função básica de manipulação de max-heap

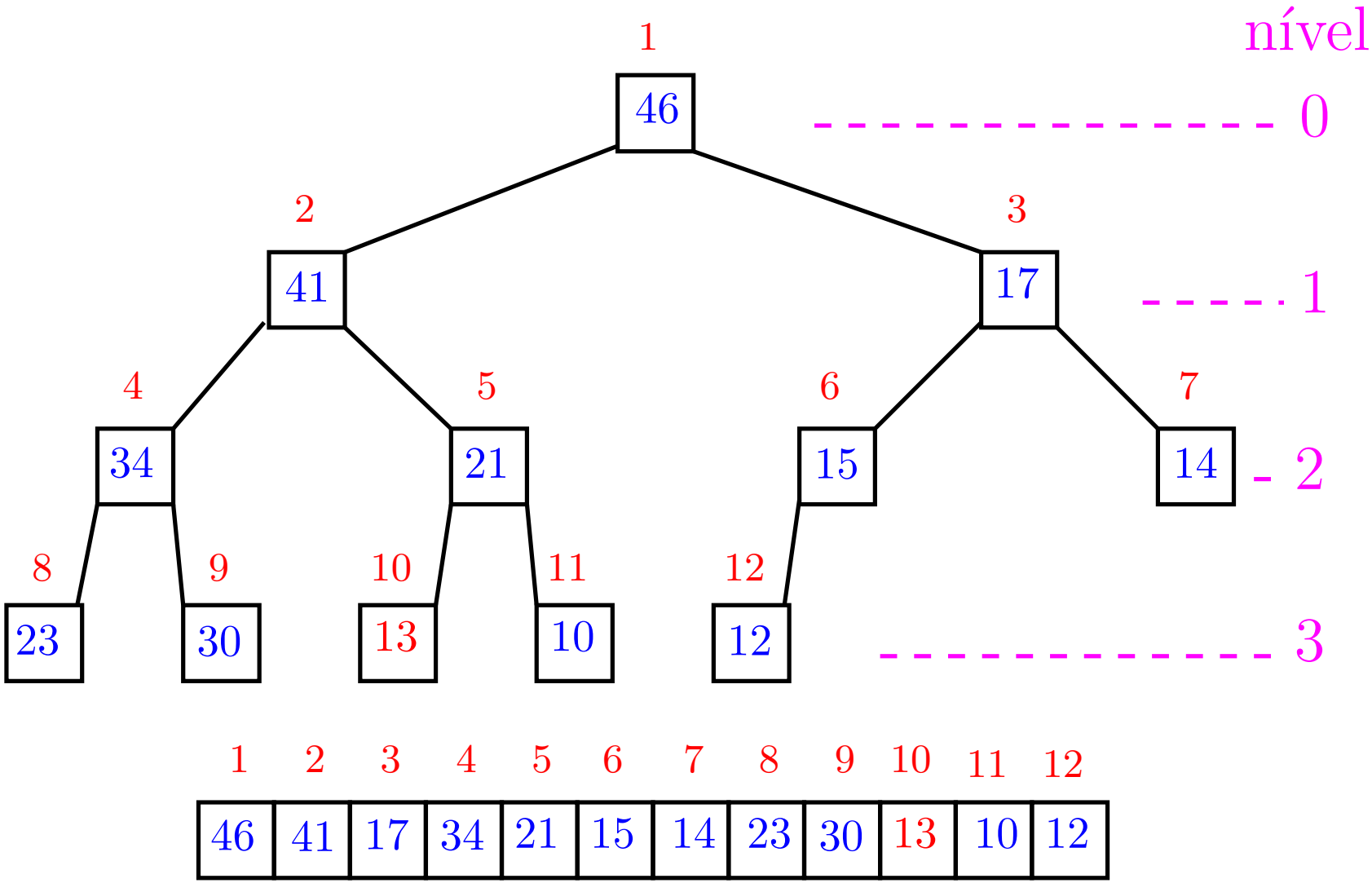


1	2	3	4	5	6	7	8	9	10	11	12
46	13	17	34	41	15	14	23	30	21	10	12

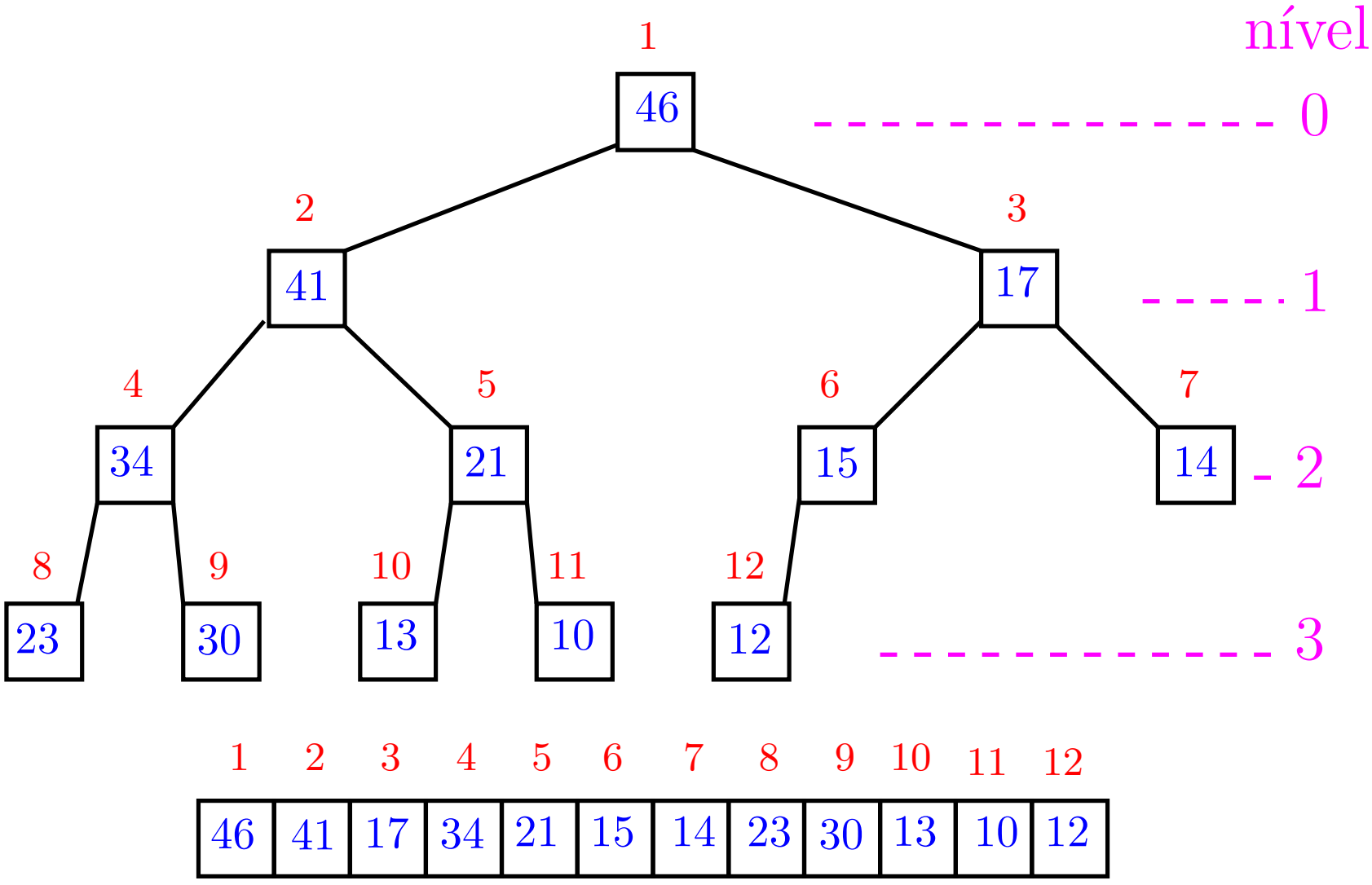
Função básica de manipulação de `max-heap`



Função básica de manipulação de max-heap



Função básica de manipulação de `max-heap`



Função `sink`

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

Função sink

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

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

Função sink

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

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

Função sink

Implementação um pouco melhor pois em vez de trocas faz apenas deslocamentos (linha 5).

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

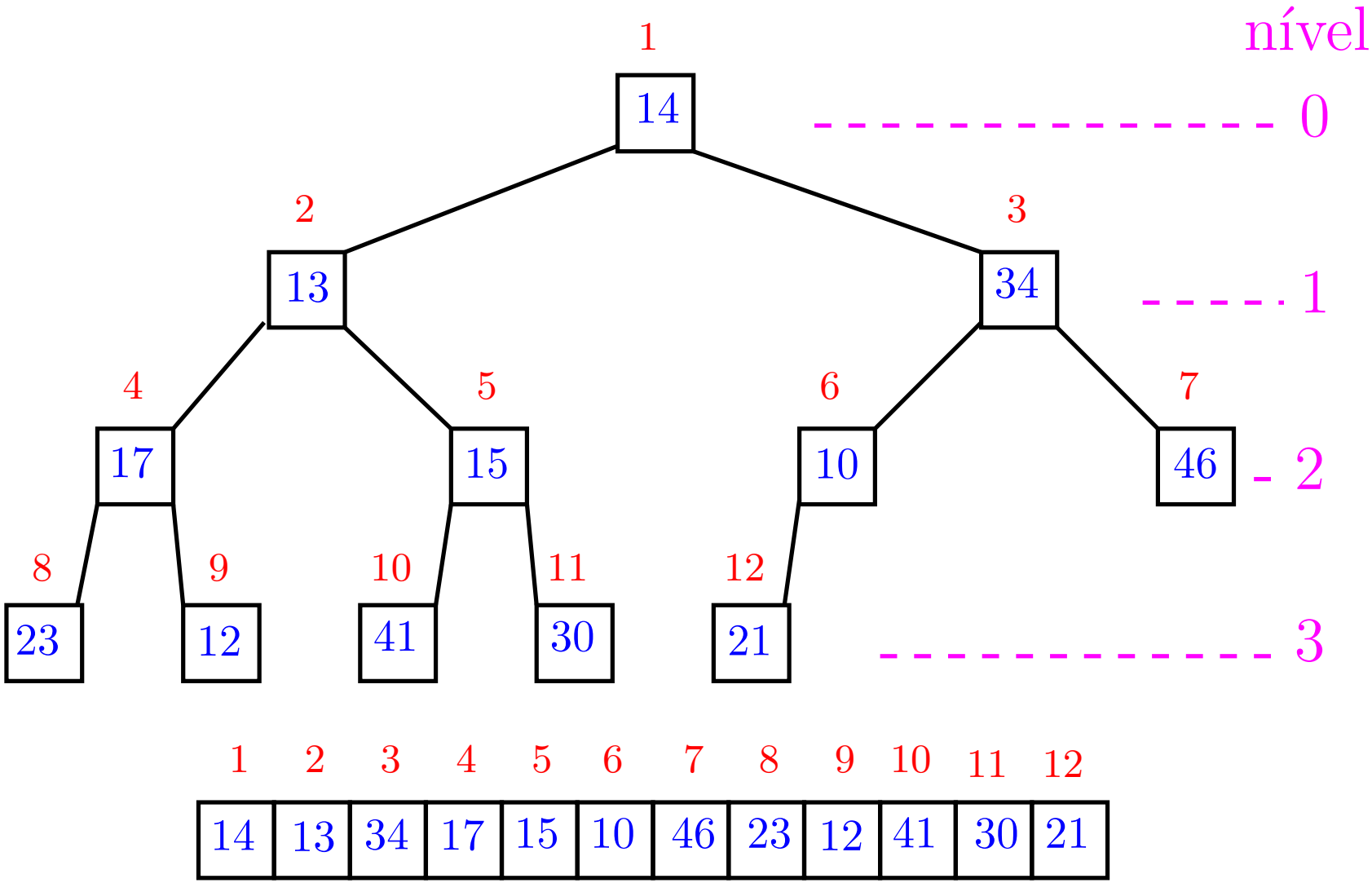
O consumo de tempo da função `sink` é proporcional a $\lg \mathtt{m}$.

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

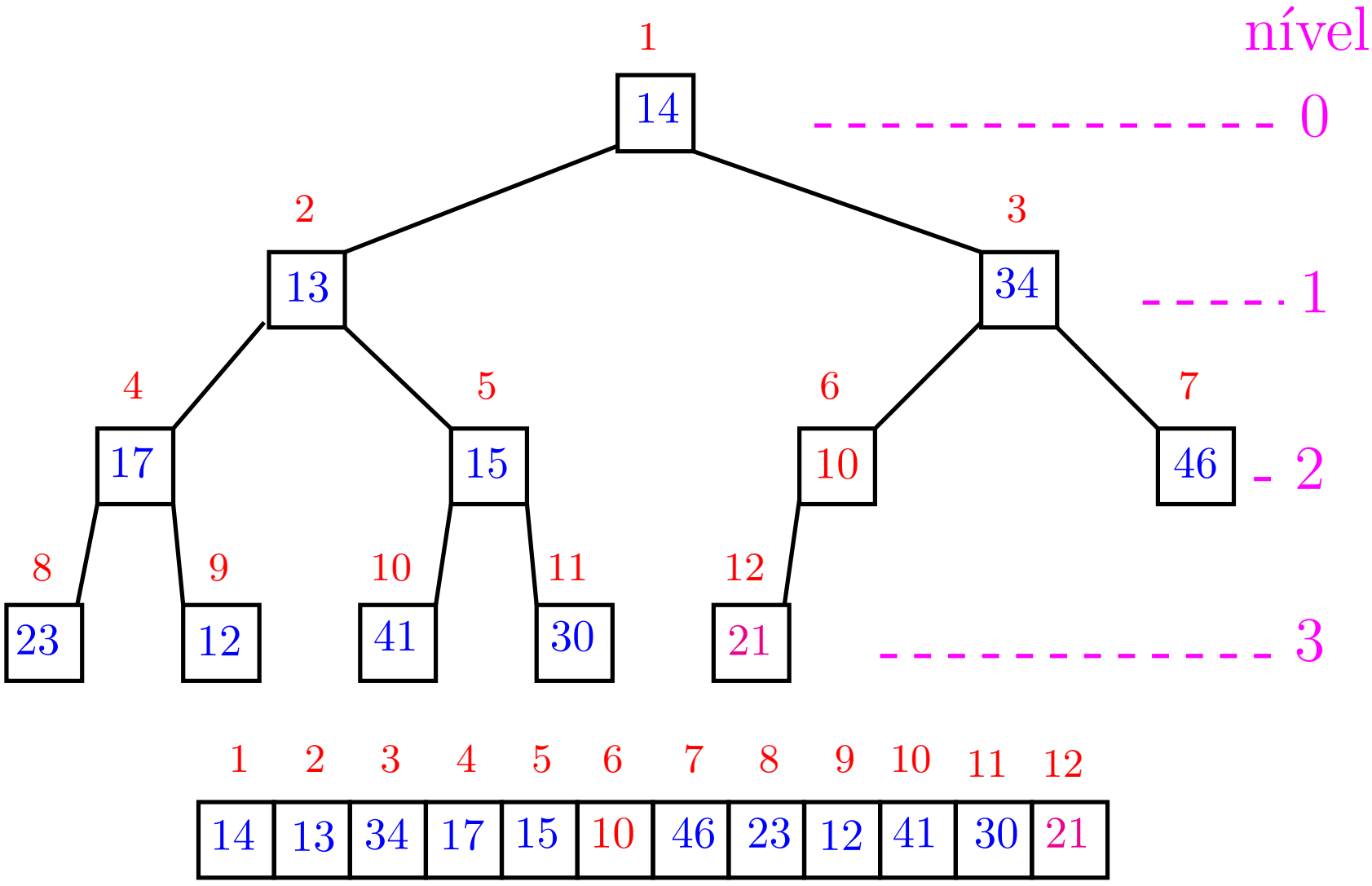
Verdade seja dita ...

O consumo de tempo da função `sink` é proporcional a $\mathrm{O}(\lg(\mathtt{m}/\mathtt{p}))$.

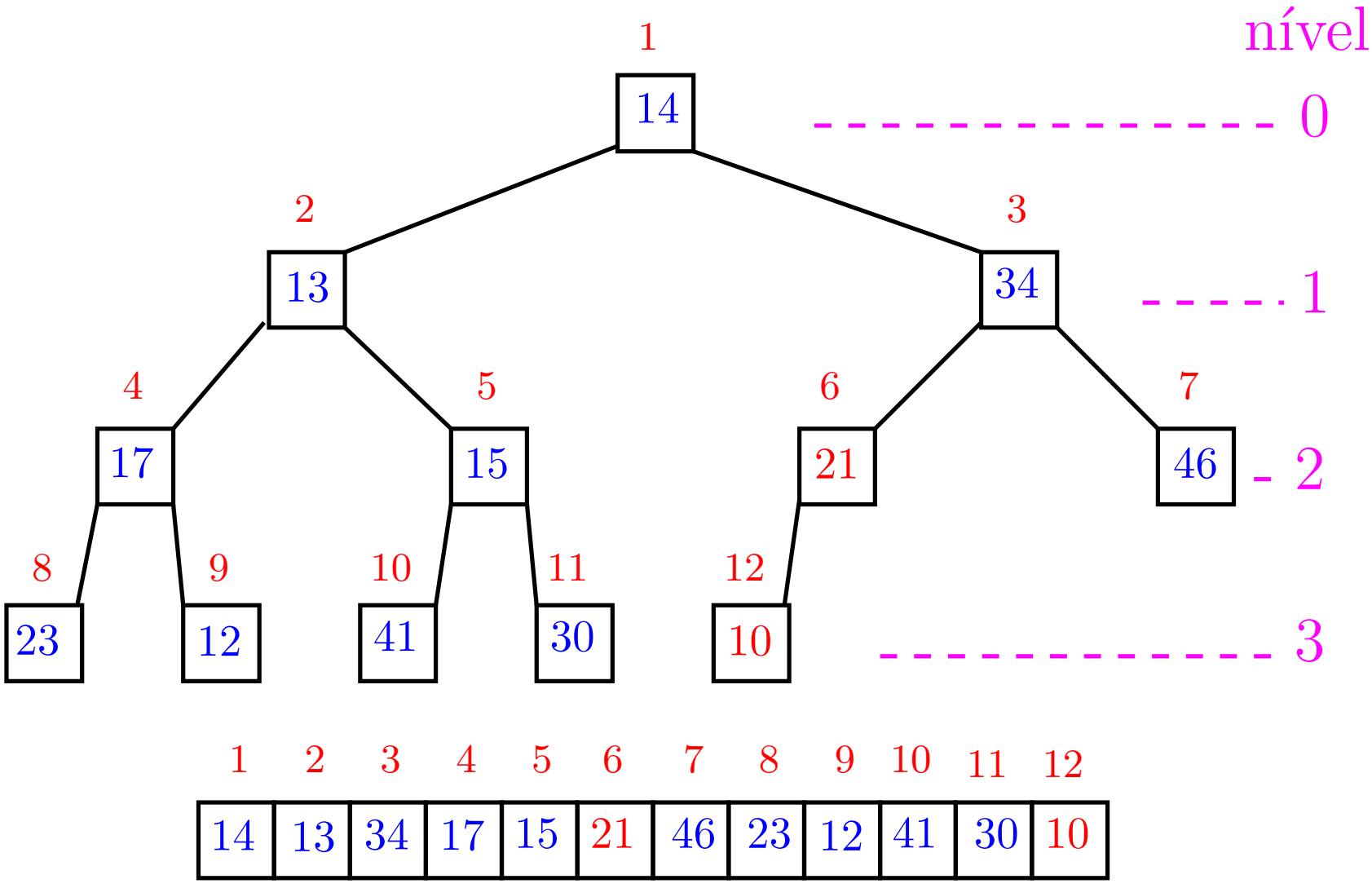
Construção de um `max-heap`



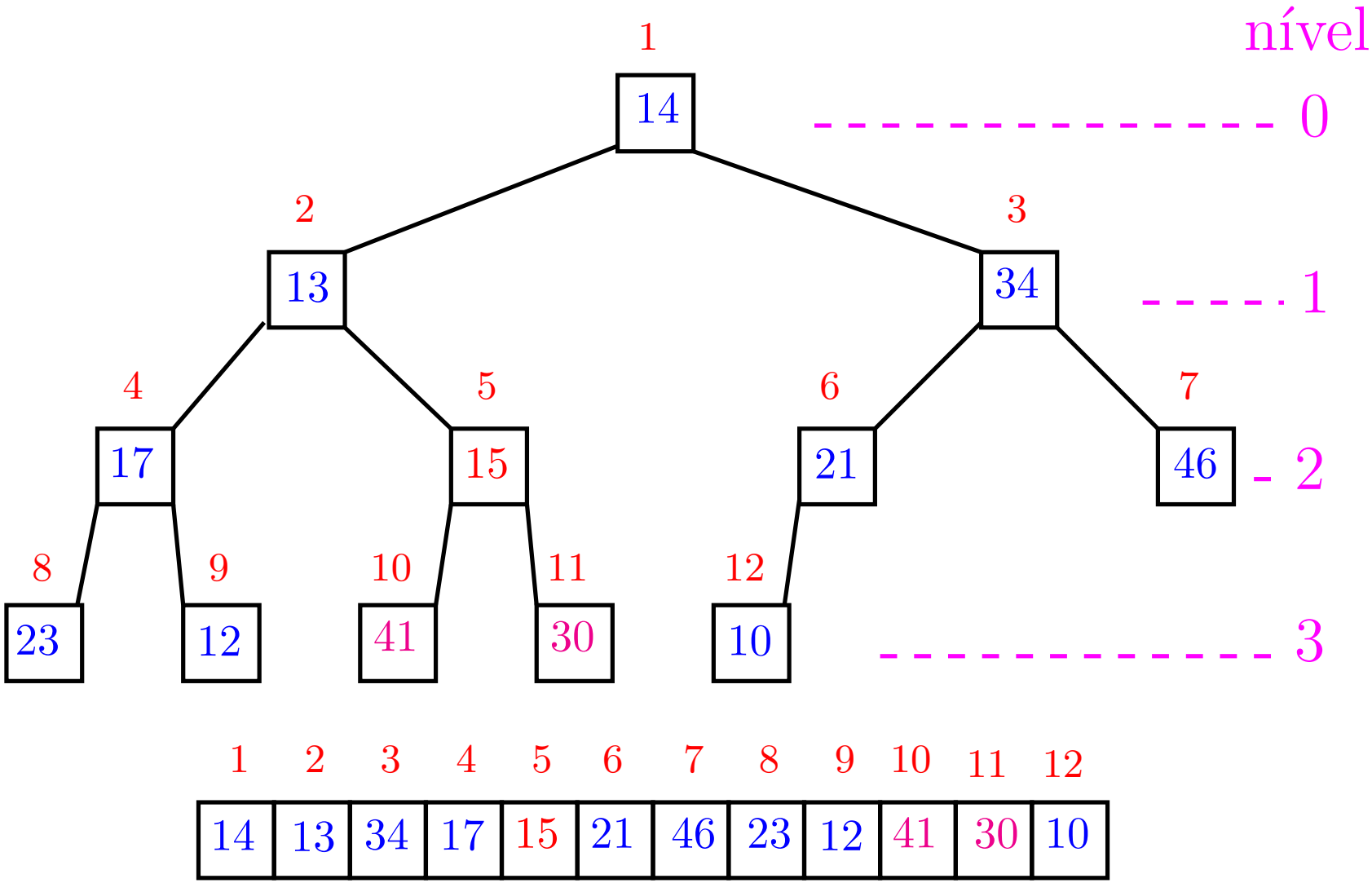
Construção de um `max-heap`



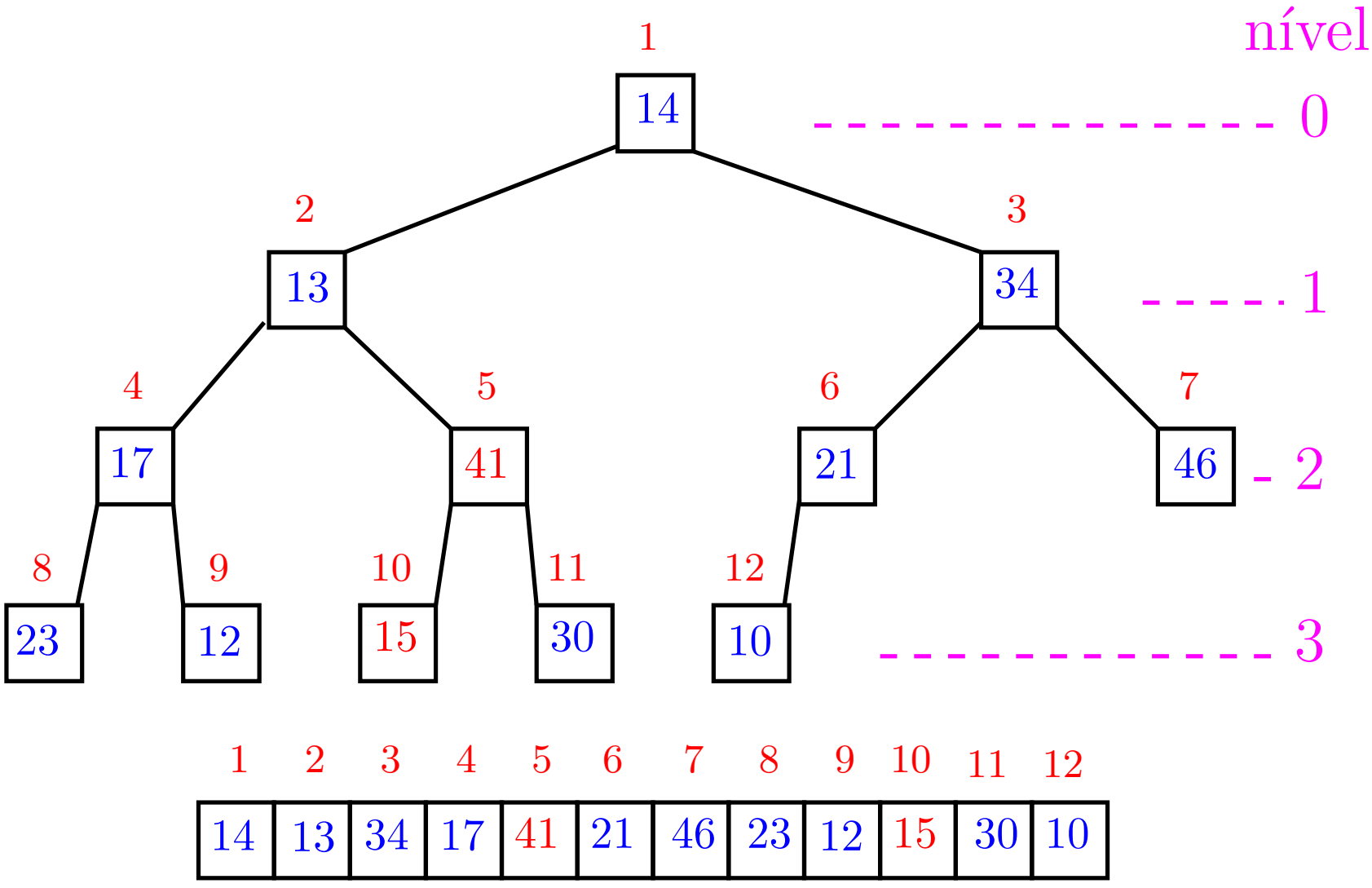
Construção de um `max-heap`



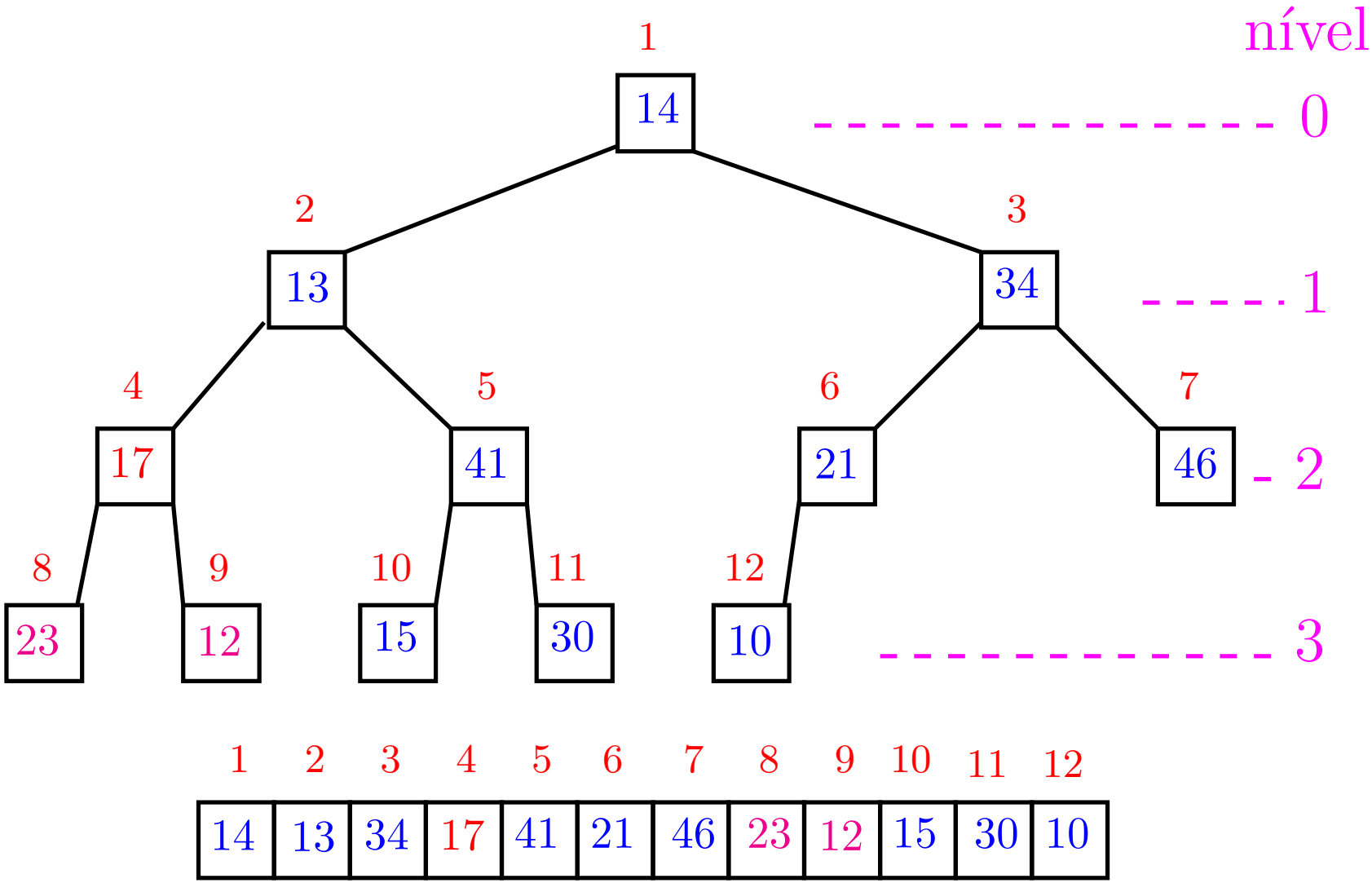
Construção de um `max-heap`



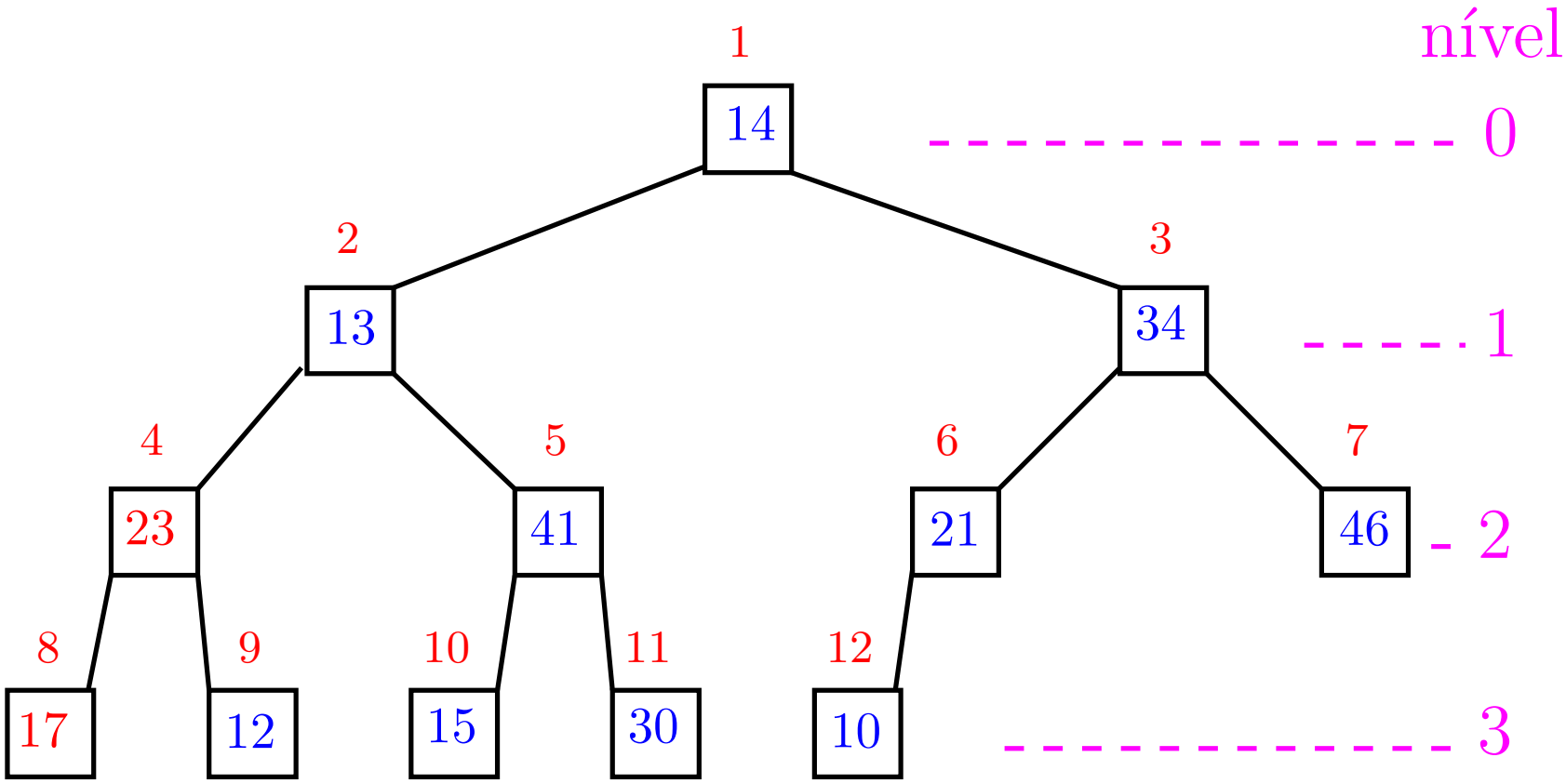
Construção de um `max-heap`



Construção de um `max-heap`

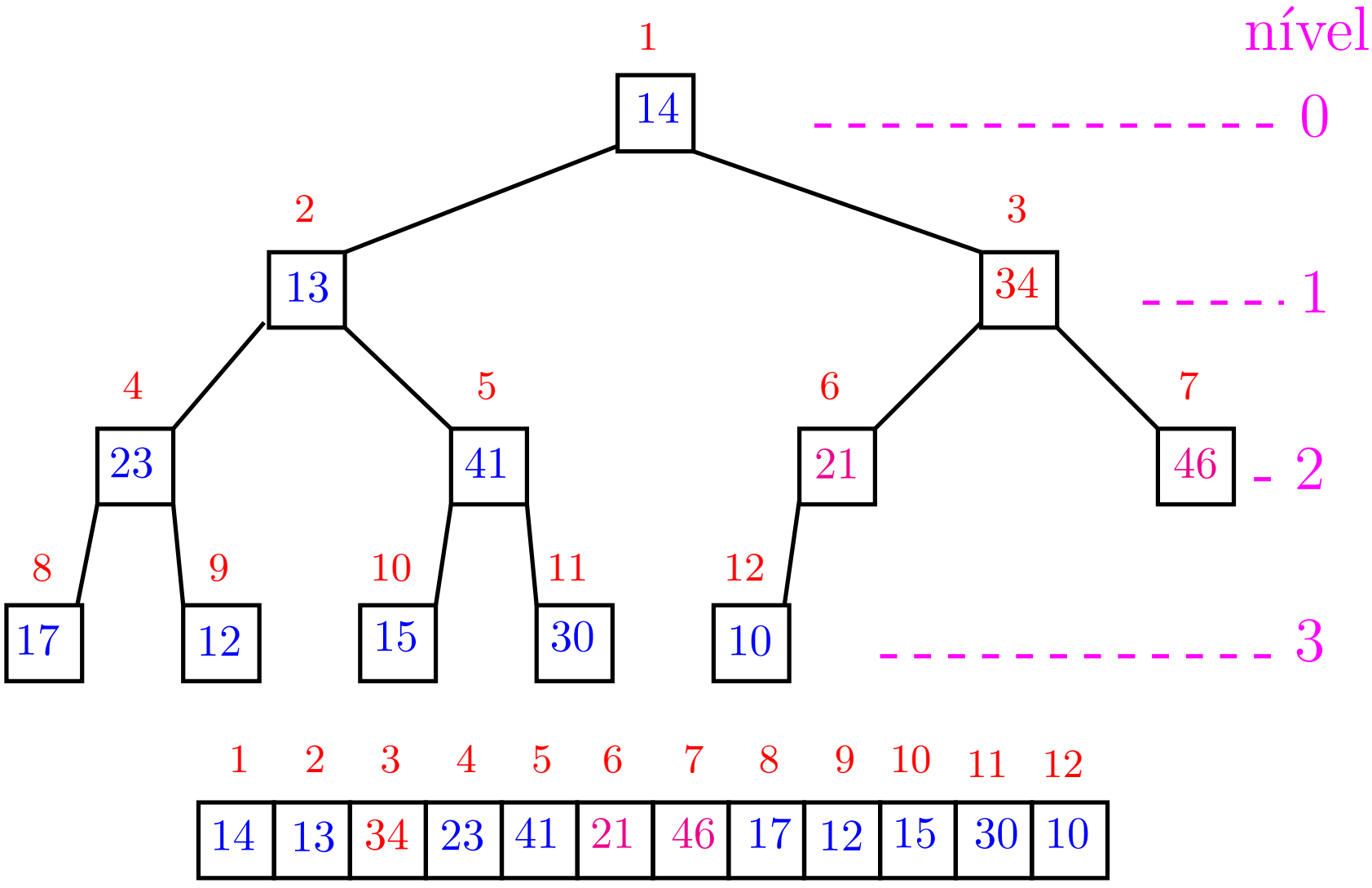


Construção de um `max-heap`

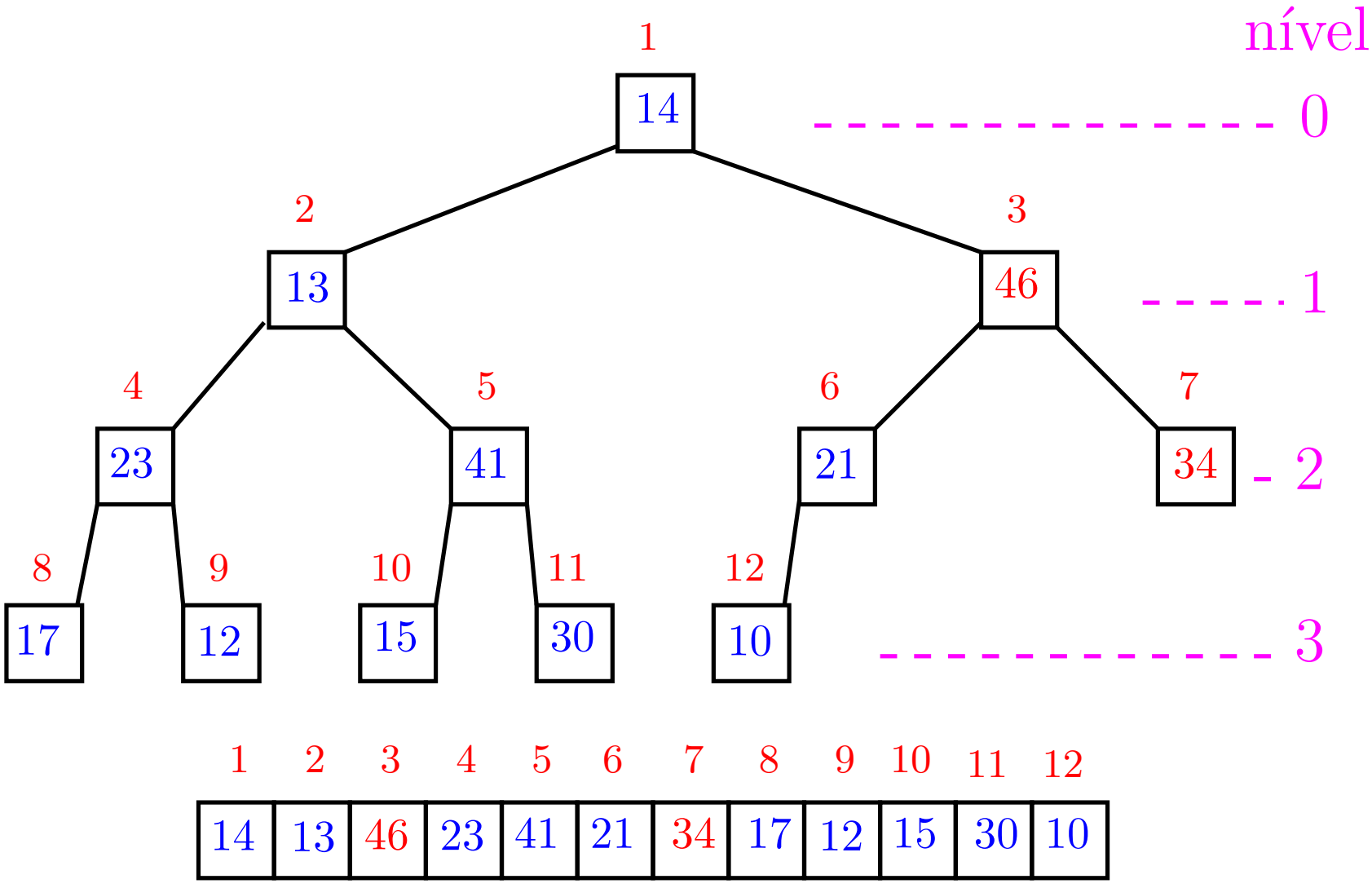


1	2	3	4	5	6	7	8	9	10	11	12
14	13	34	23	41	21	46	17	12	15	30	10

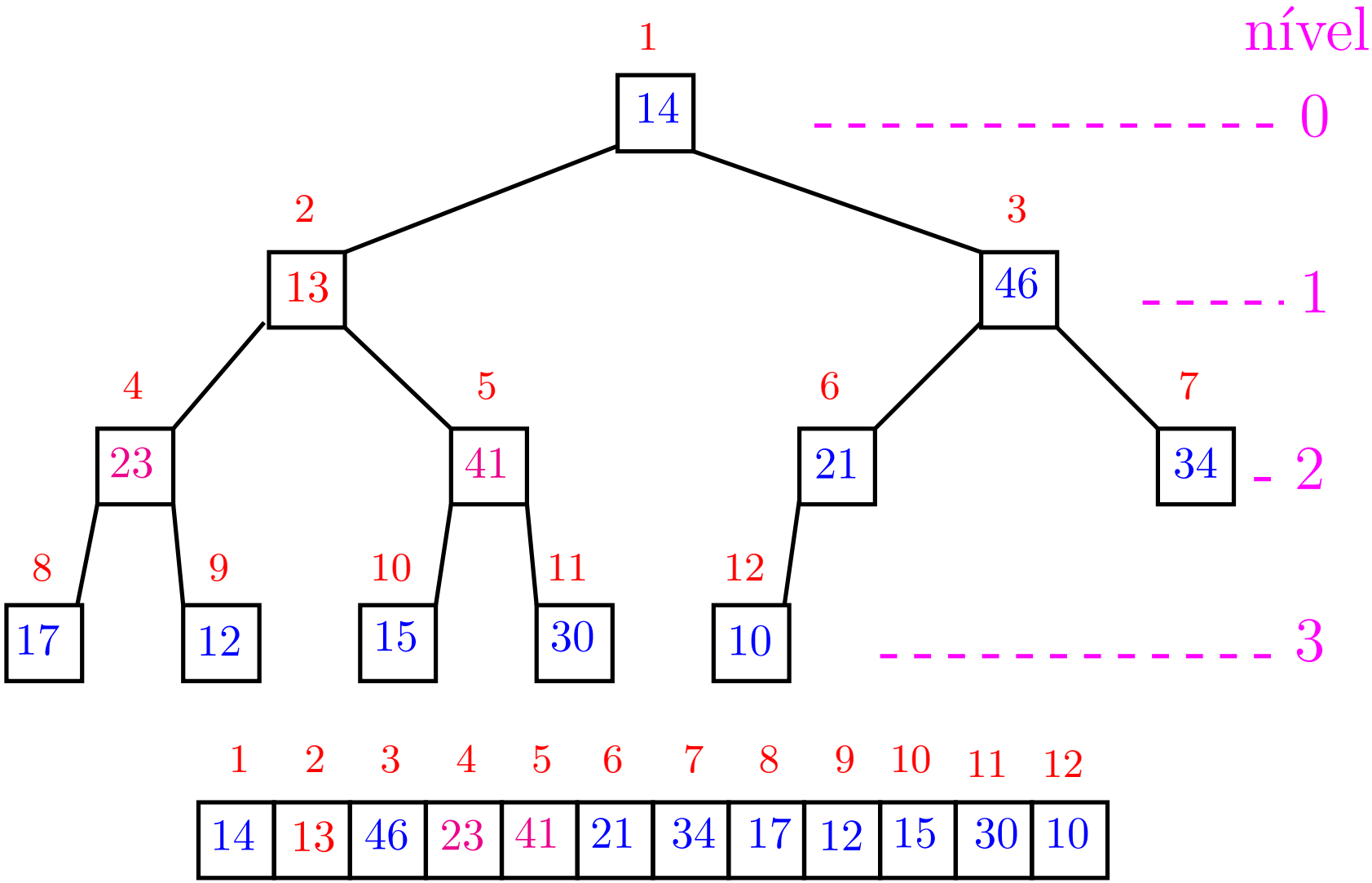
Construção de um `max-heap`



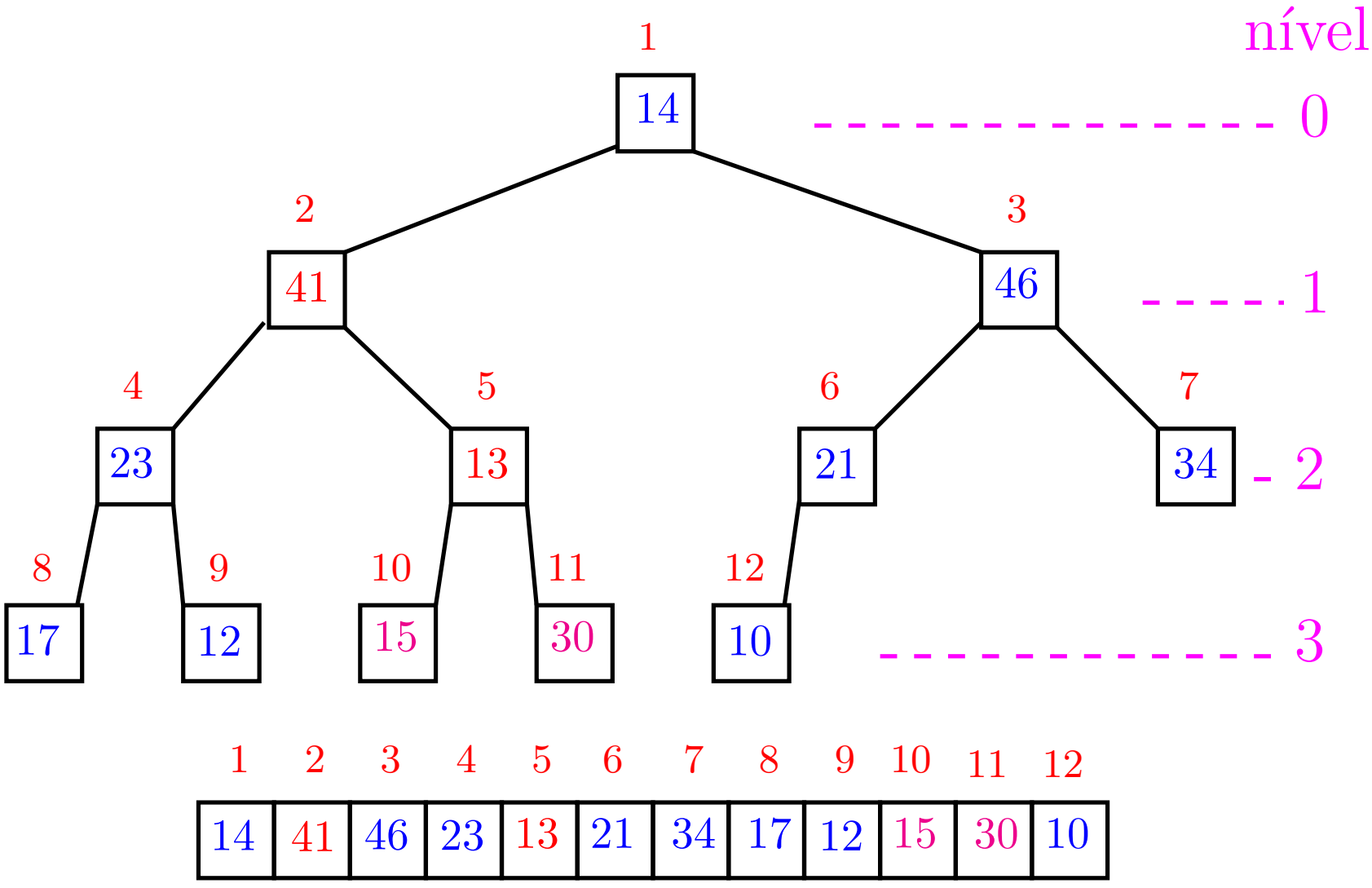
Construção de um `max-heap`



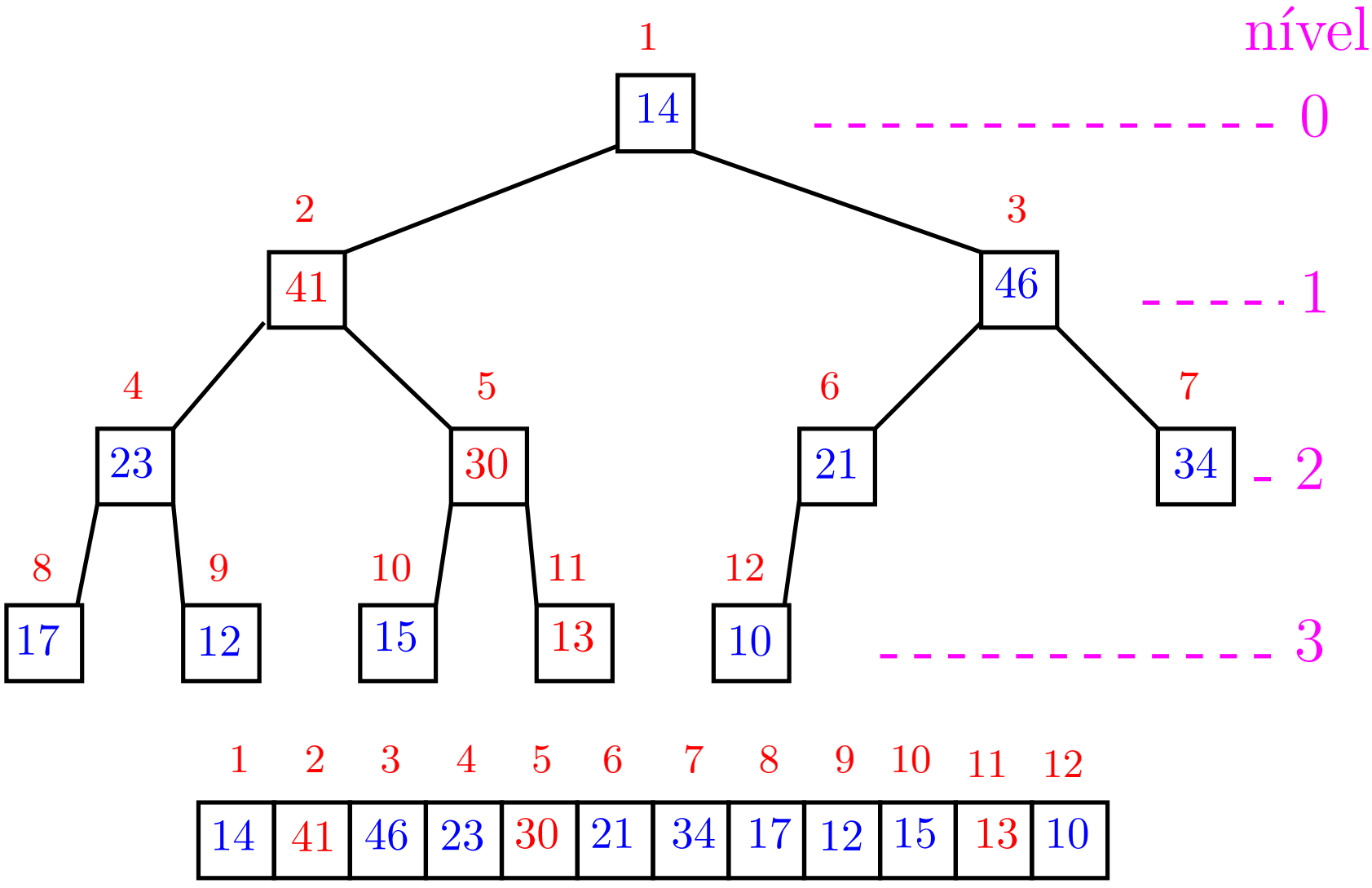
Construção de um max-heap



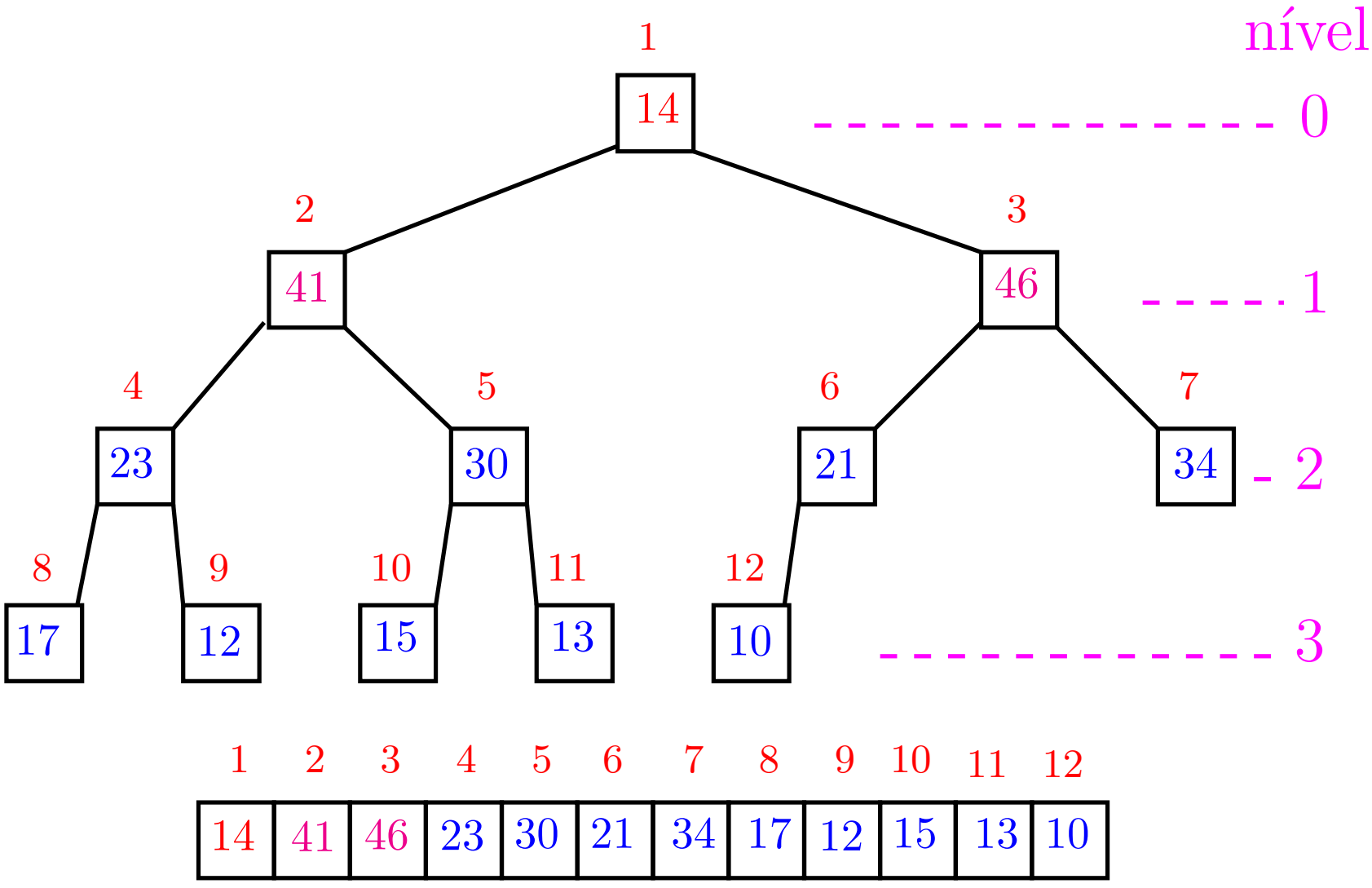
Construção de um max-heap



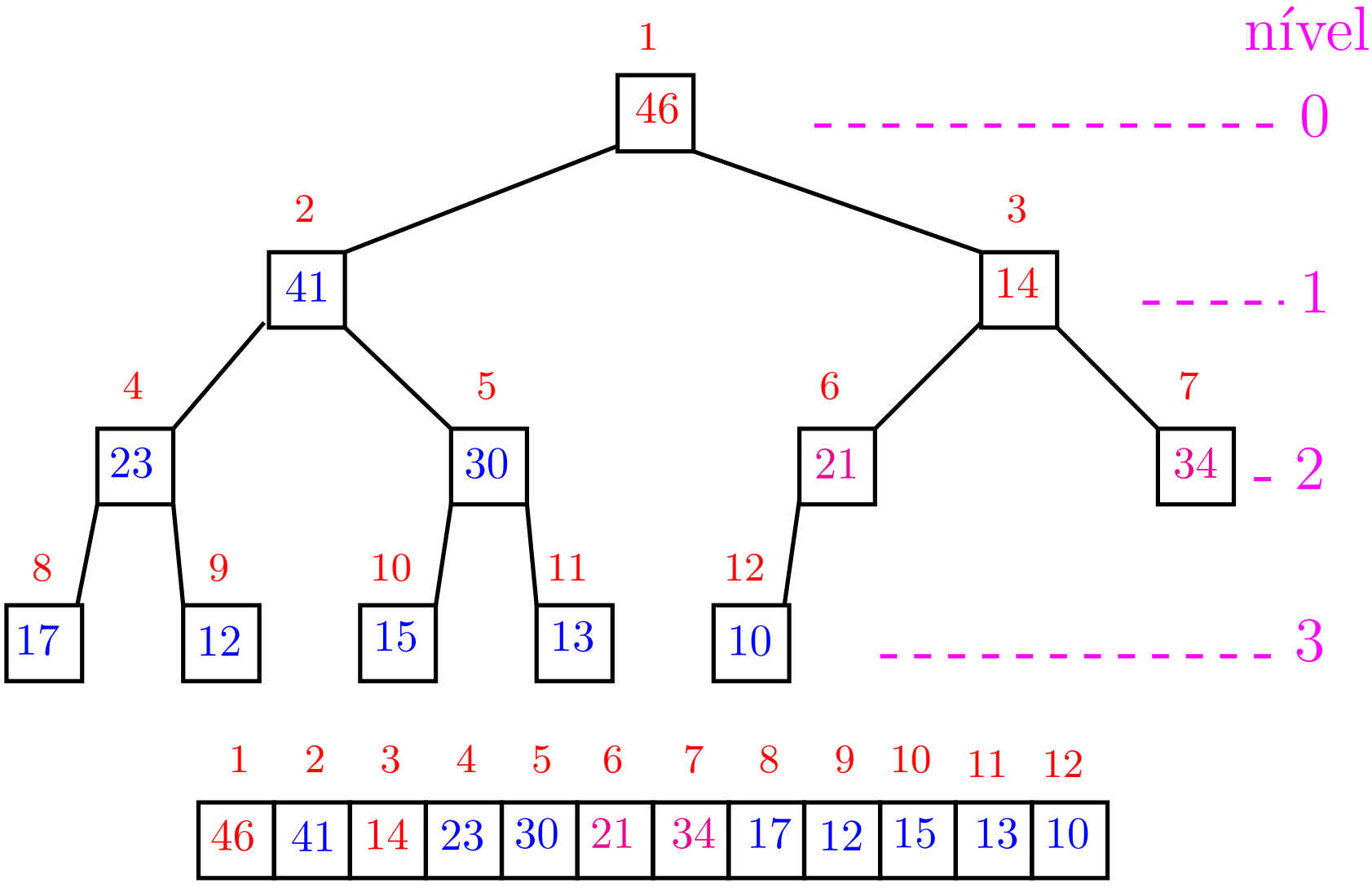
Construção de um `max-heap`



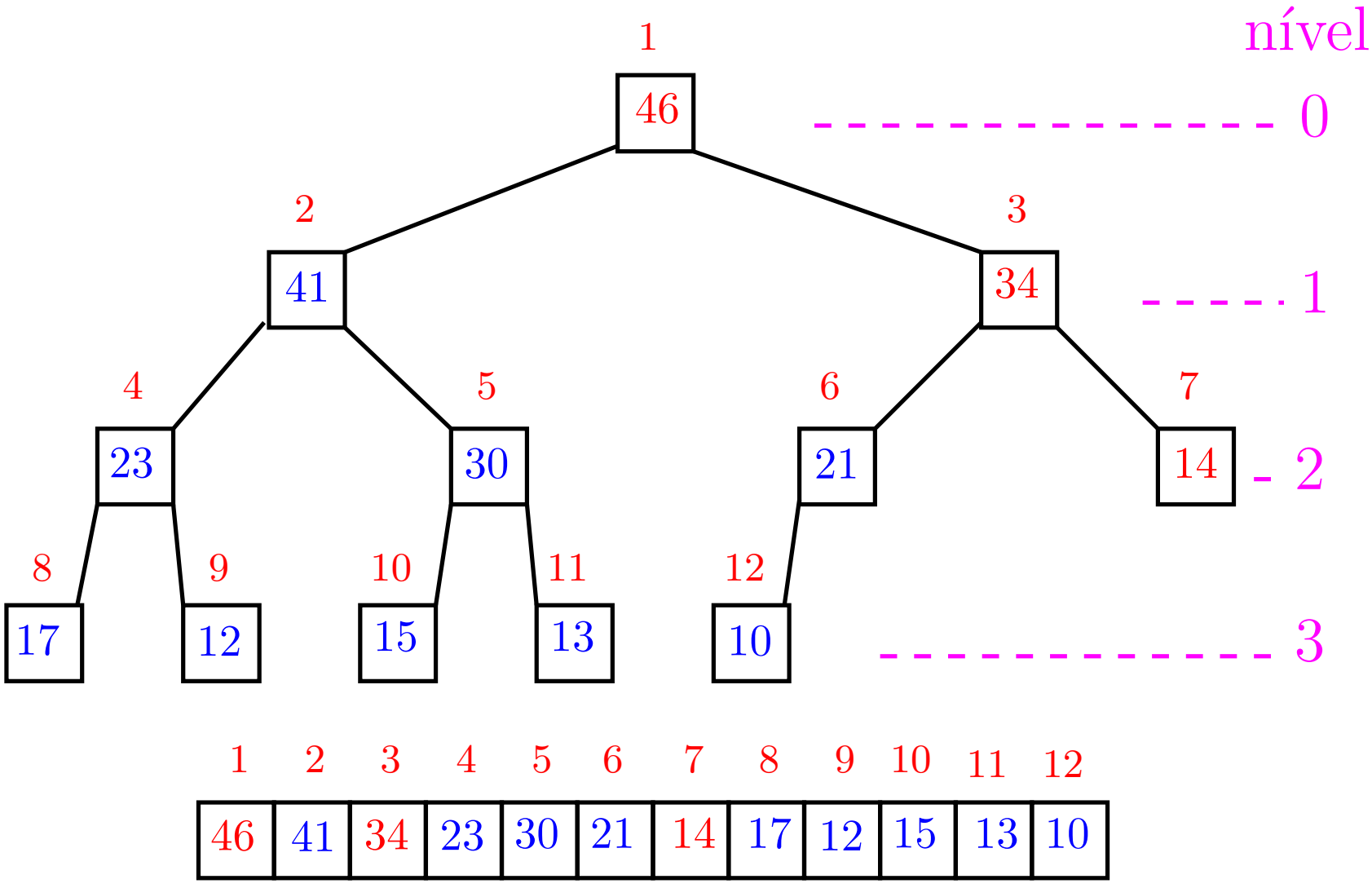
Construção de um `max-heap`



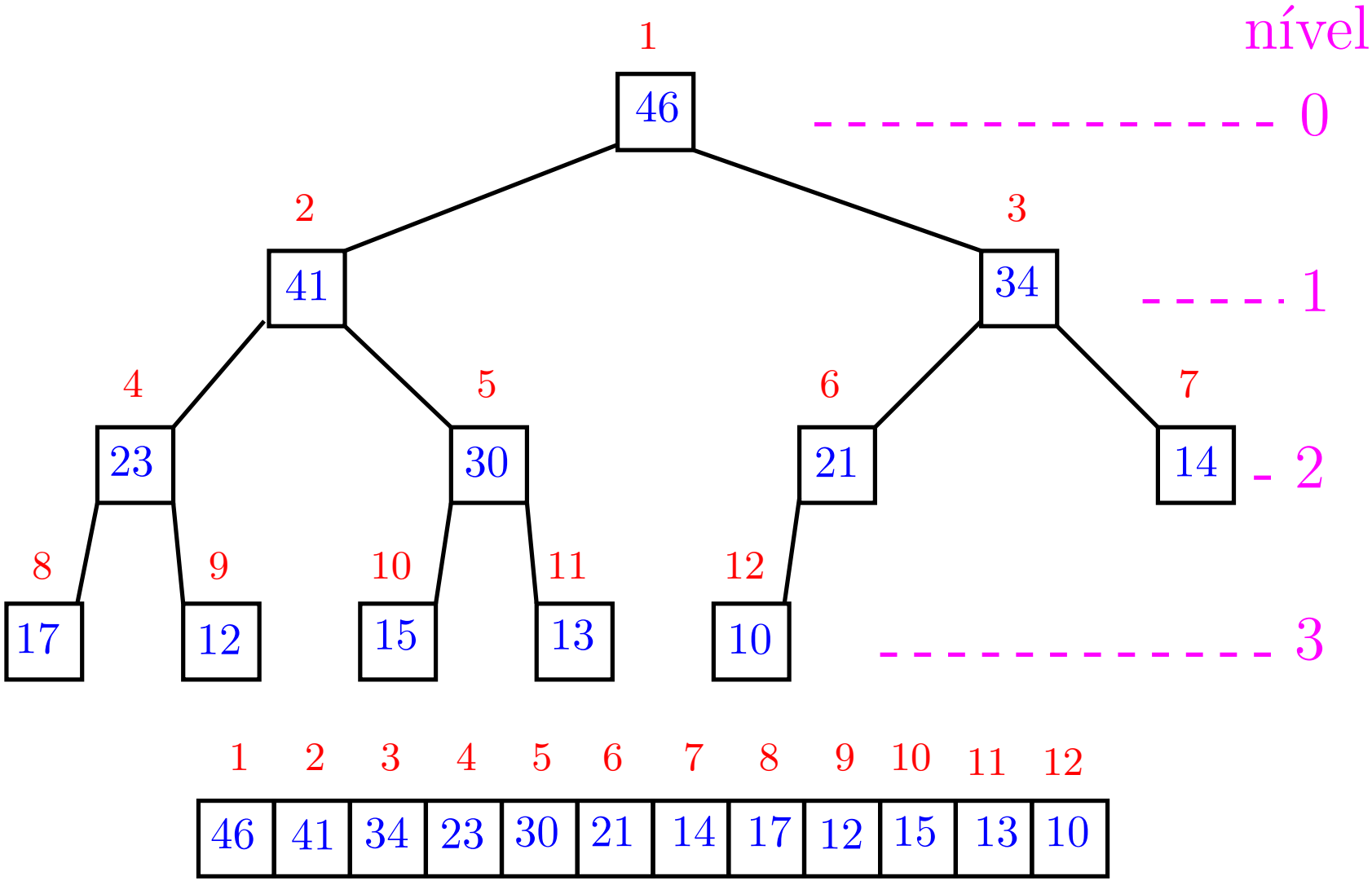
Construção de um `max-heap`



Construção de um max-heap



Construção de um `max-heap`



Construção de um max-heap

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

```
1    for (int p = n/2; /*A*/ p >= 1; p--)
2       sink(p, n, a);
```

Relação invariante:

(i0) em /*A*/ vale que
p+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 $O(n)$.

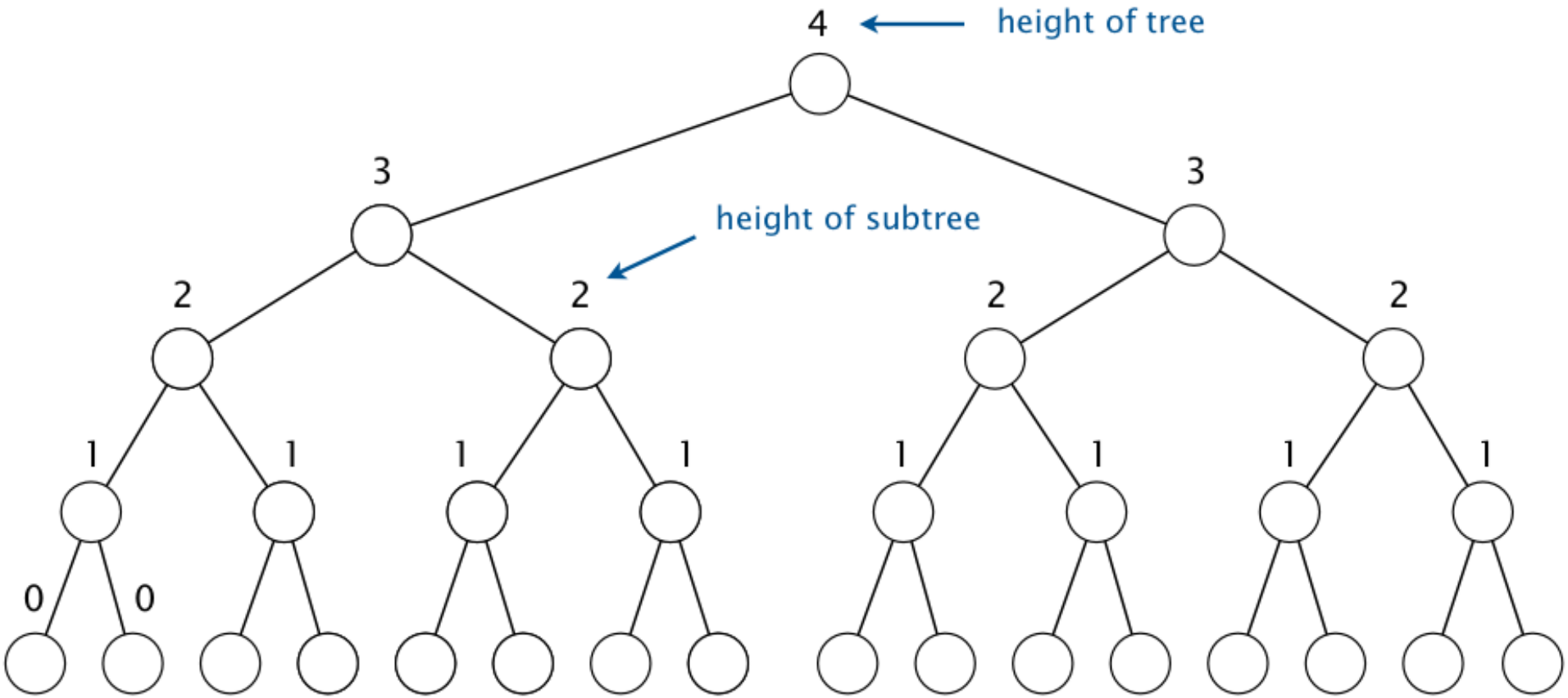
Conclusão

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

Verdade seja dita . . .

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

Altura das subárvores



Fonte: algs4

Algumas séries

Para todo número real $\mathtt{x}$ com $|\mathtt{x}| < 1$, temos que $\sum_{i=0}^{\infty} \mathtt{x}^i = \frac{1}{1-\mathtt{x}}$.

Algumas séries

Para todo número real $\mathtt{x}$ com $|\mathtt{x}| < 1$,
temos que $\sum_{i=0}^{\infty} \mathtt{x}^i = \frac{1}{1-\mathtt{x}}$.

Para todo número real $\mathtt{x}$ com $|\mathtt{x}| < 1$, temos que

$$\sum_{i=1}^{\infty} i\,\mathtt{x}^i = \frac{\mathtt{x}}{(1-\mathtt{x})^2}$$

Algumas séries

Para todo número real $\mathtt{x}$ com $|\mathtt{x}| < 1$, temos que $\sum_{i=0}^{\infty} \mathtt{x}^i = \frac{1}{1-\mathtt{x}}$.

Para todo número real $\mathtt{x}$ com $|\mathtt{x}| < 1$, temos que

$$\sum_{i=1}^{\infty} i\,\mathtt{x}^i = \frac{\mathtt{x}}{(1-\mathtt{x})^2}$$

Prova:

$$\begin{aligned}\sum_{i=1}^{\infty} i\,\mathtt{x}^i &= \sum_{i=1}^{\infty} x^i + \sum_{i=2}^{\infty} \mathtt{x}^i + \cdots + \sum_{i=k}^{\infty} \mathtt{x}^i + \cdots \\ &= \frac{\mathtt{x}}{1-\mathtt{x}} + \frac{\mathtt{x}^2}{1-\mathtt{x}} + \cdots + \frac{\mathtt{x}^{\mathtt{k}}}{1-\mathtt{x}} + \cdots \\ &= \frac{\mathtt{x}}{1-\mathtt{x}}(\mathtt{x}^0 + \mathtt{x}^1 + \mathtt{x}^2 + \cdots + \mathtt{x}^{\mathtt{k}} + \cdots) = \frac{\mathtt{x}}{(1-\mathtt{x})^2}.\end{aligned}$$