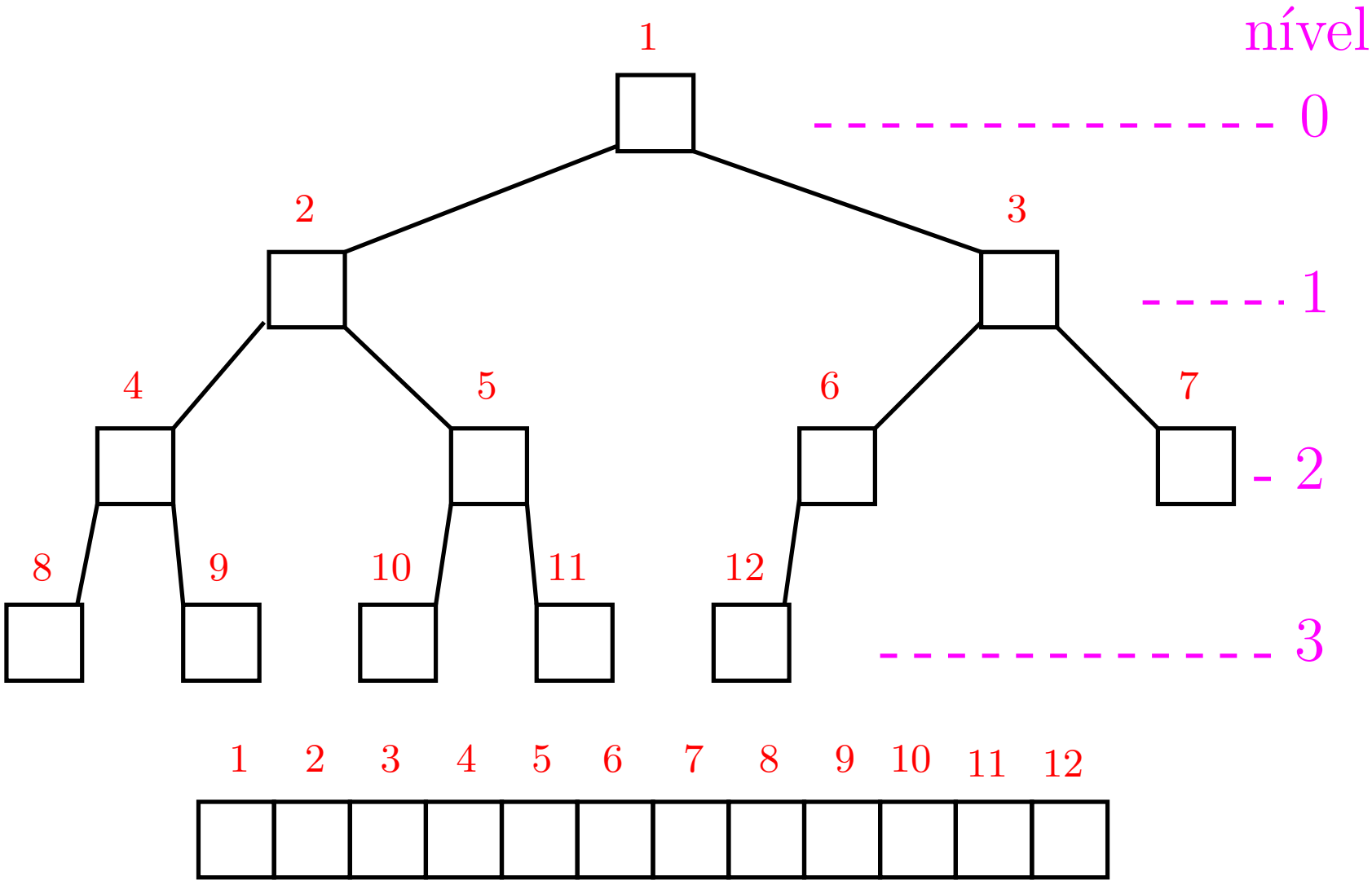
Fonte: ash.atozviews.com

Compacto dos melhores momentos

AULA 4

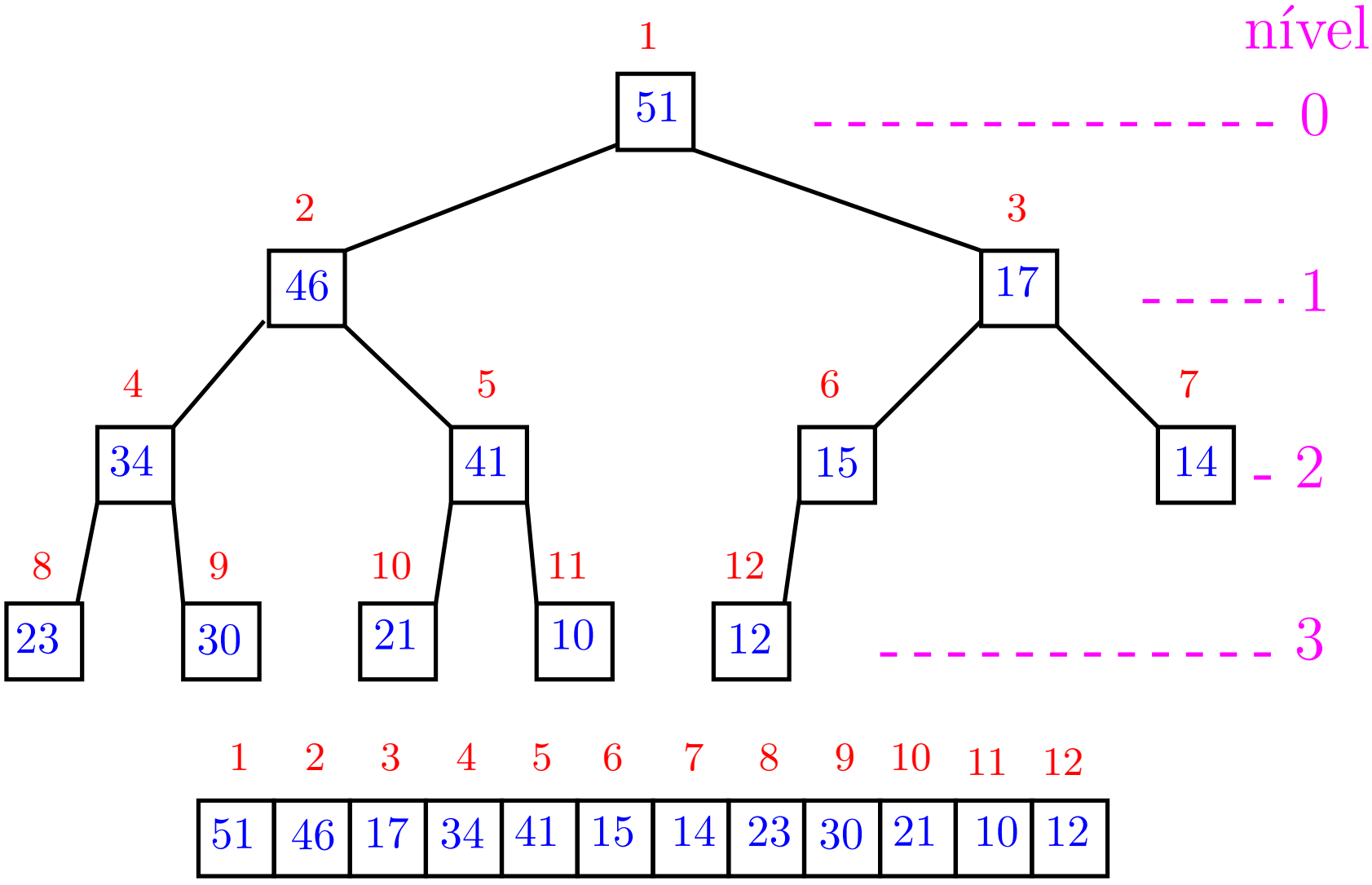
Representação de árvores em vetores



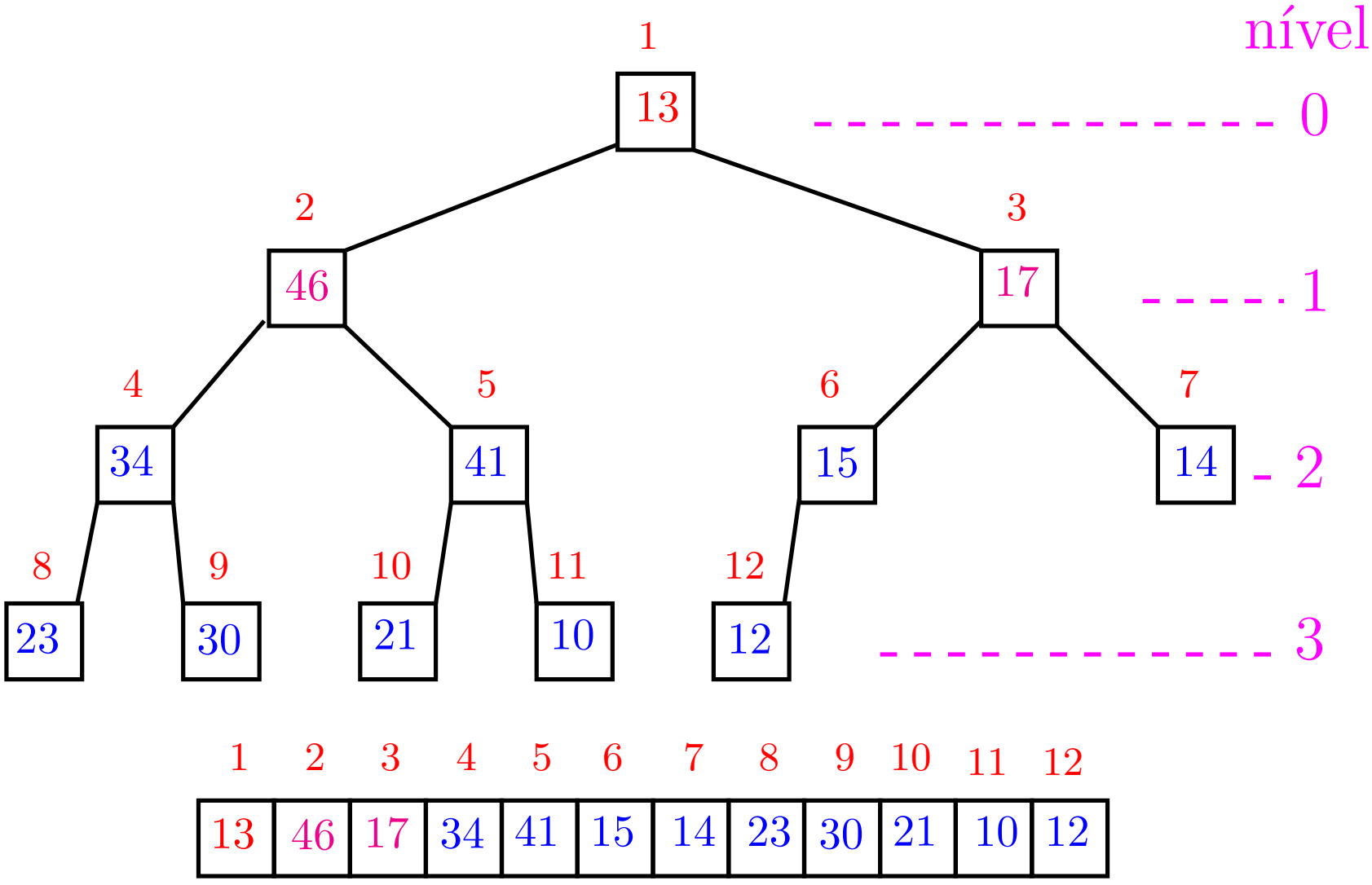
Resumão da estrutura

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:	$\approx \lg i$
altura da raiz:	$\approx \lg m$
altura da árvore:	$\approx \lg m$
altura de i:	$\approx \lg(m/i)$ $(\ldots)$
altura de uma folha:	0

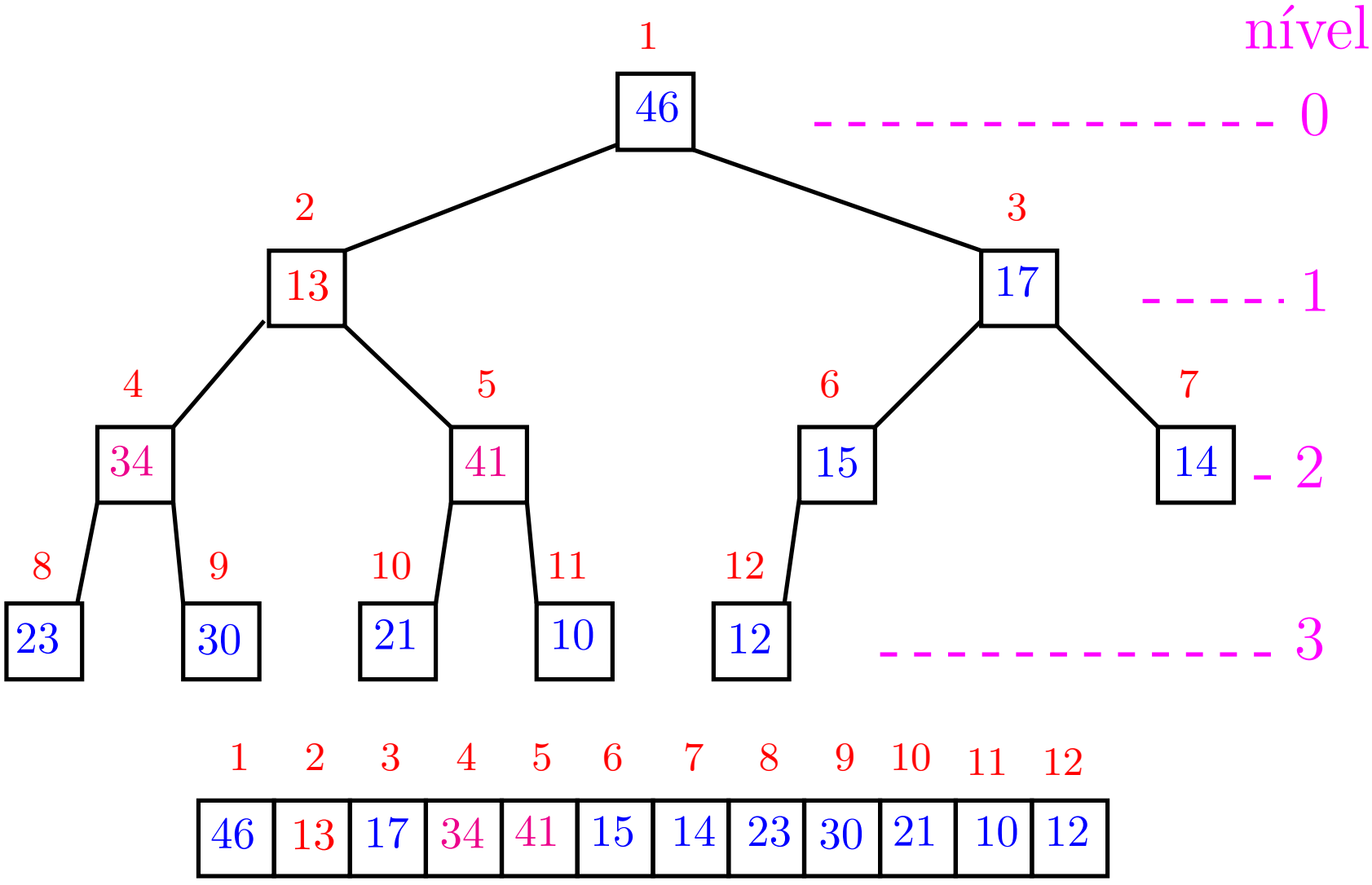
max-heap


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

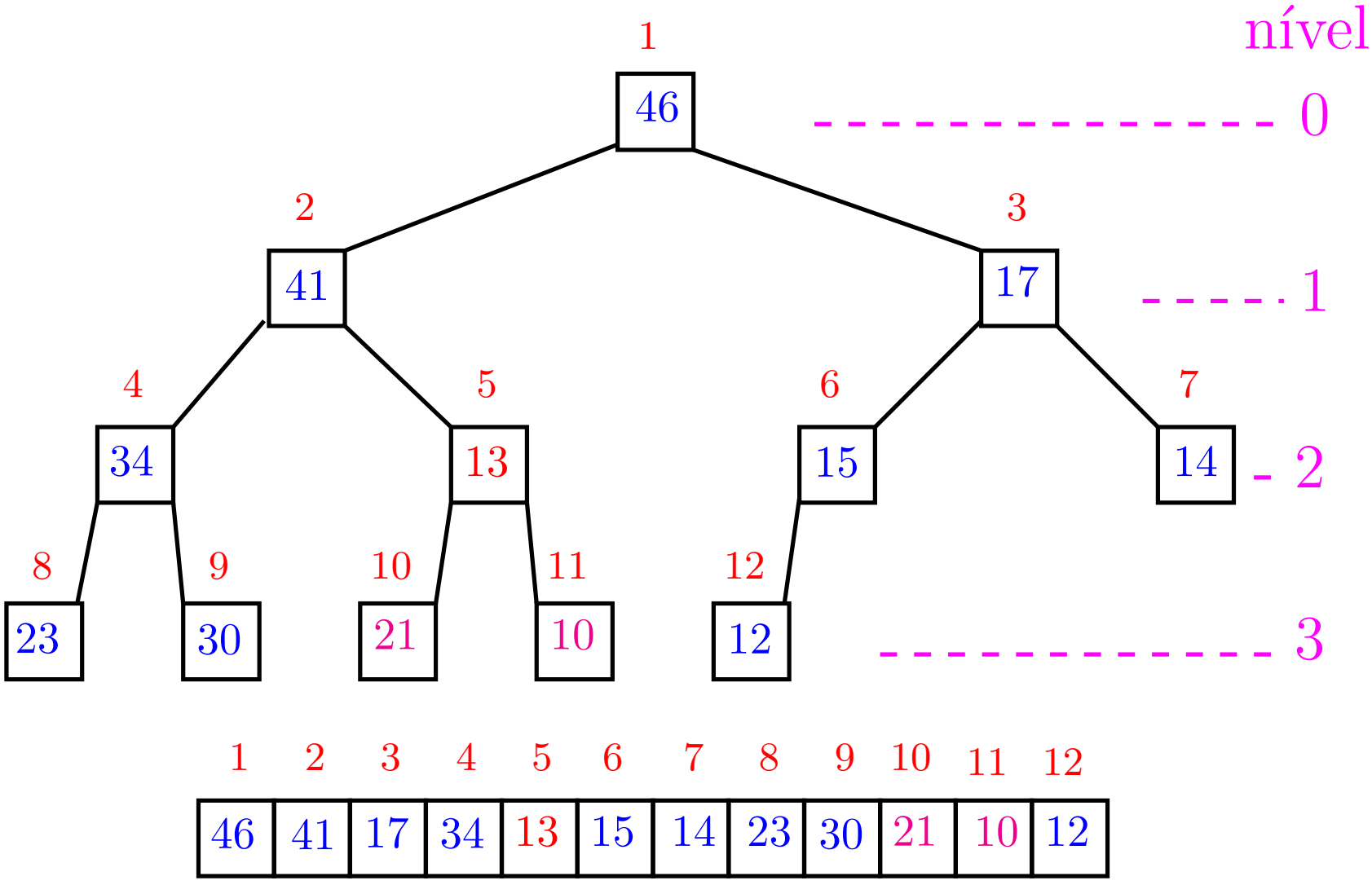
Função básica: `sink()`



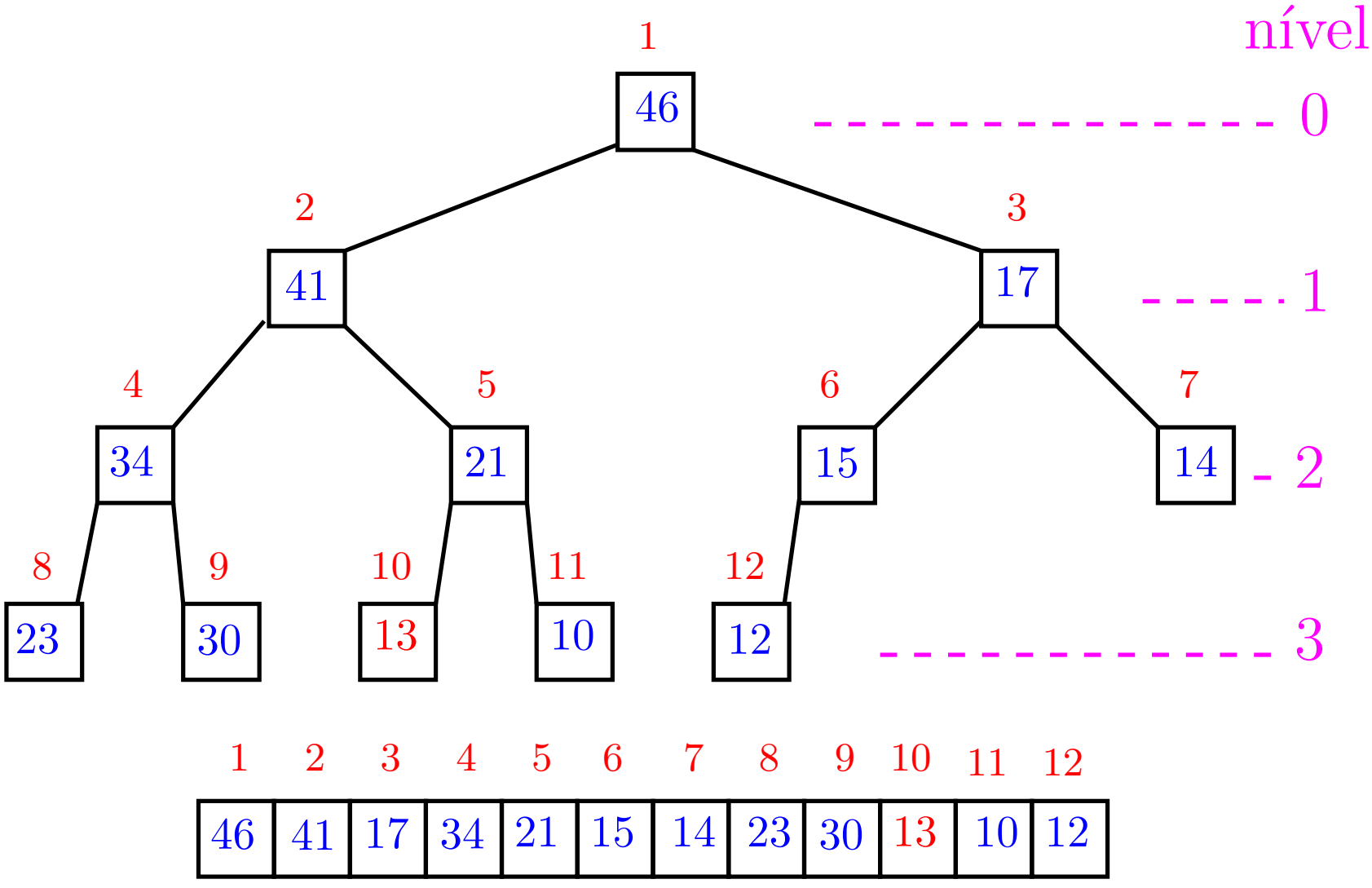
Função básica: `sink()`



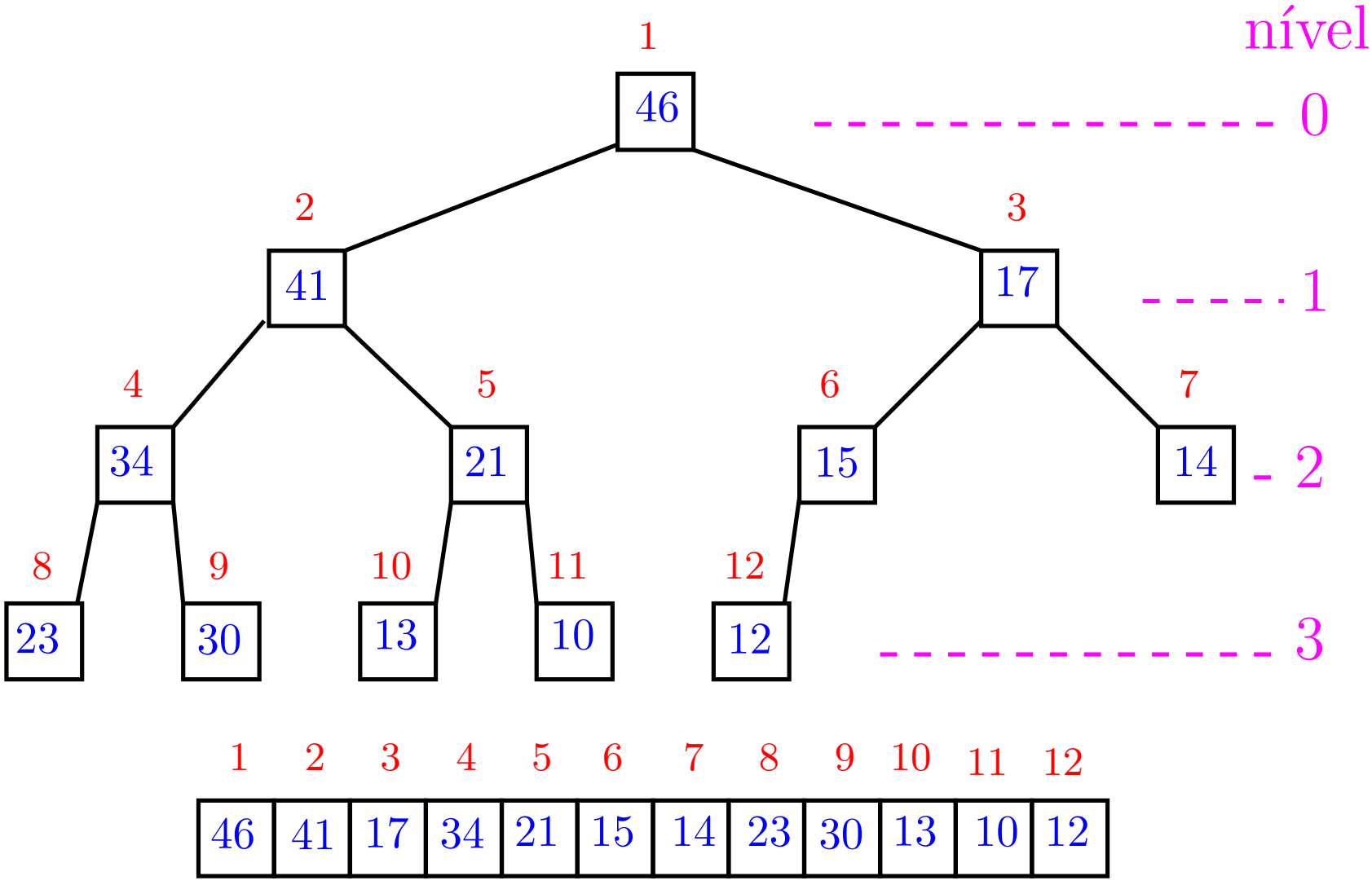
Função básica: `sink()`



Função básica: `sink()`



Função básica: `sink()`



Função sink

Implementação faz apenas deslocamentos (linha 5).

```
private static
void sink (int p, int m, Comparable[] v){
1   int f = 2*p; Object x = v[p];
2   while (f <= m) {
3       if (f<m && less(a[f],a[f+1])) f++;
4       if (!less(x, v[f])) break;
5       v[p] = v[f];
6       p = f; f = 2*p; // sink
    }
7   v[p] = x;
 }
```

Consumo de tempo

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 . . . (. . .)

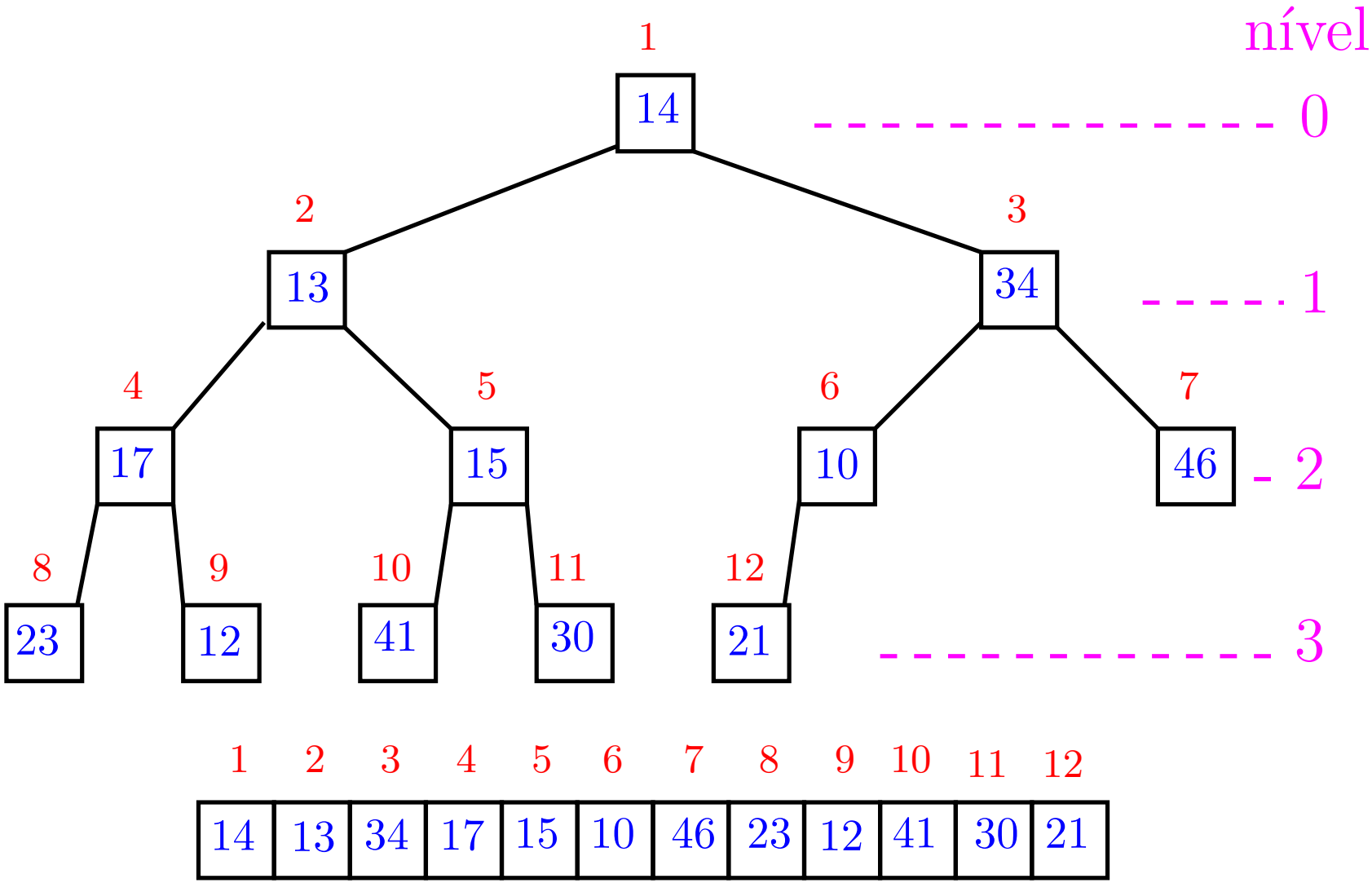
O consumo de tempo da função `sink()` é proporcional a $\mathrm{O}(\lg \mathtt{m}/\mathtt{i})$.

AULA 5

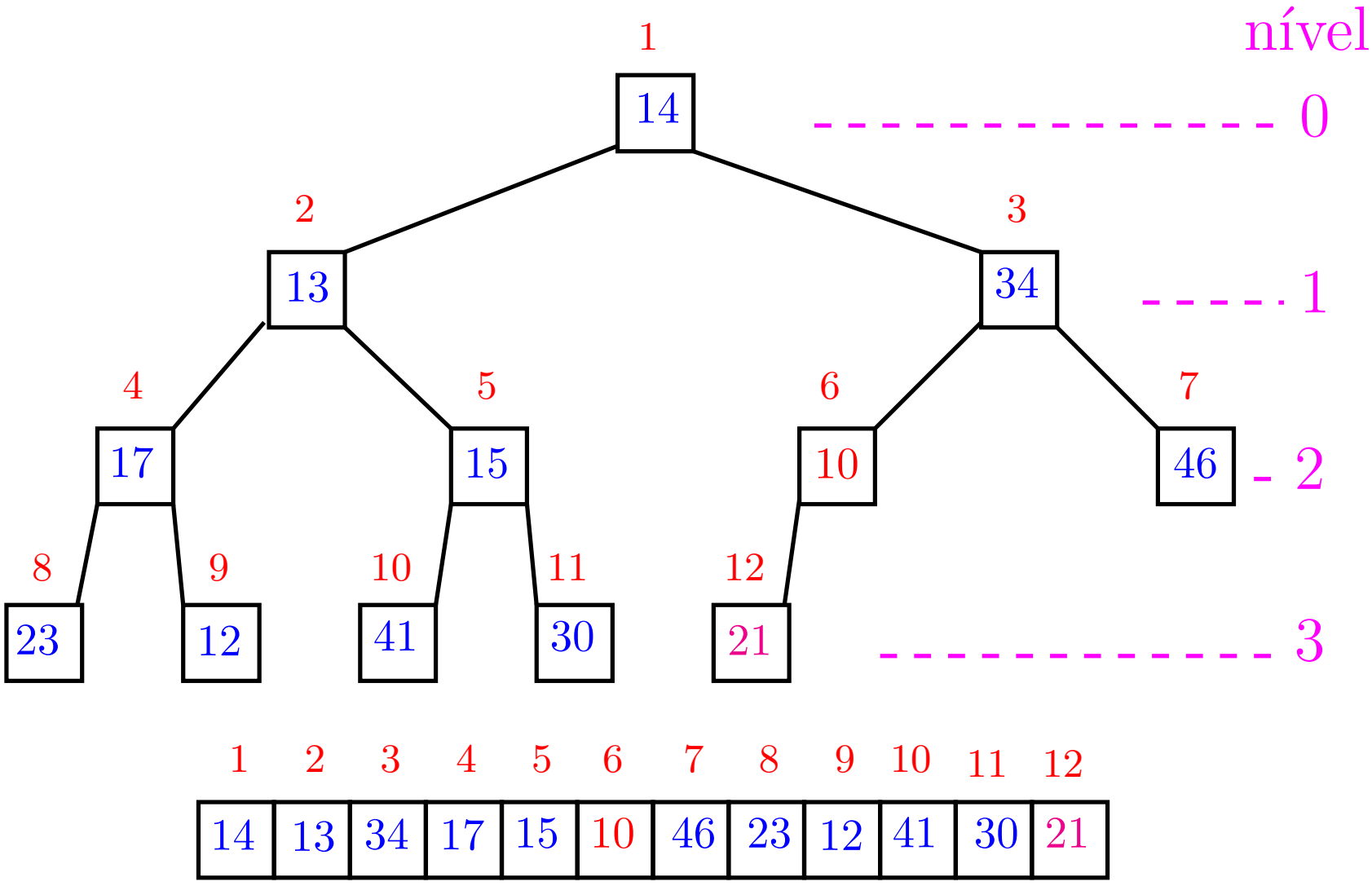
Construção de um `max-heap`

Fonte: EPBOT

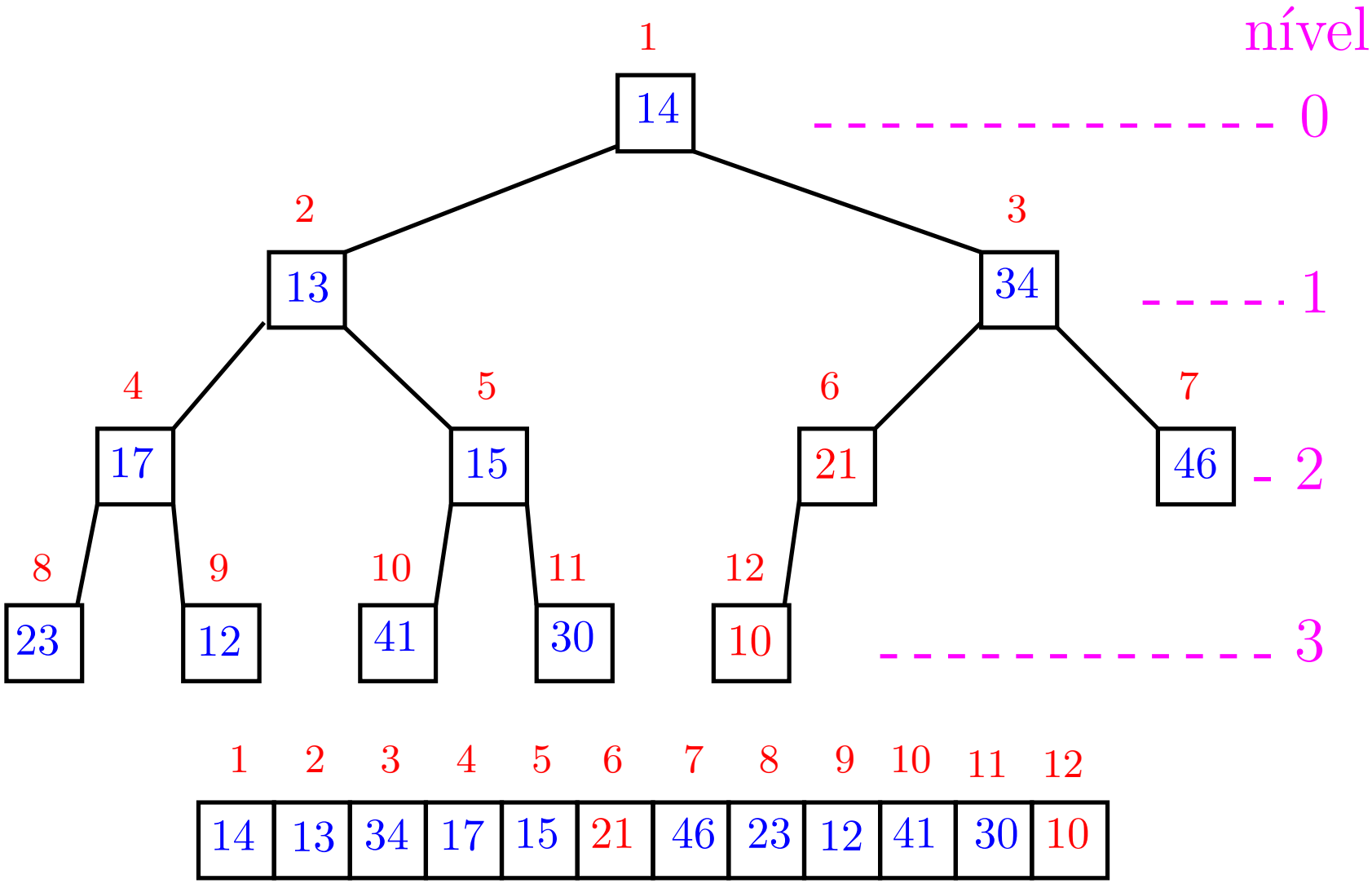
Construção de um `max-heap`



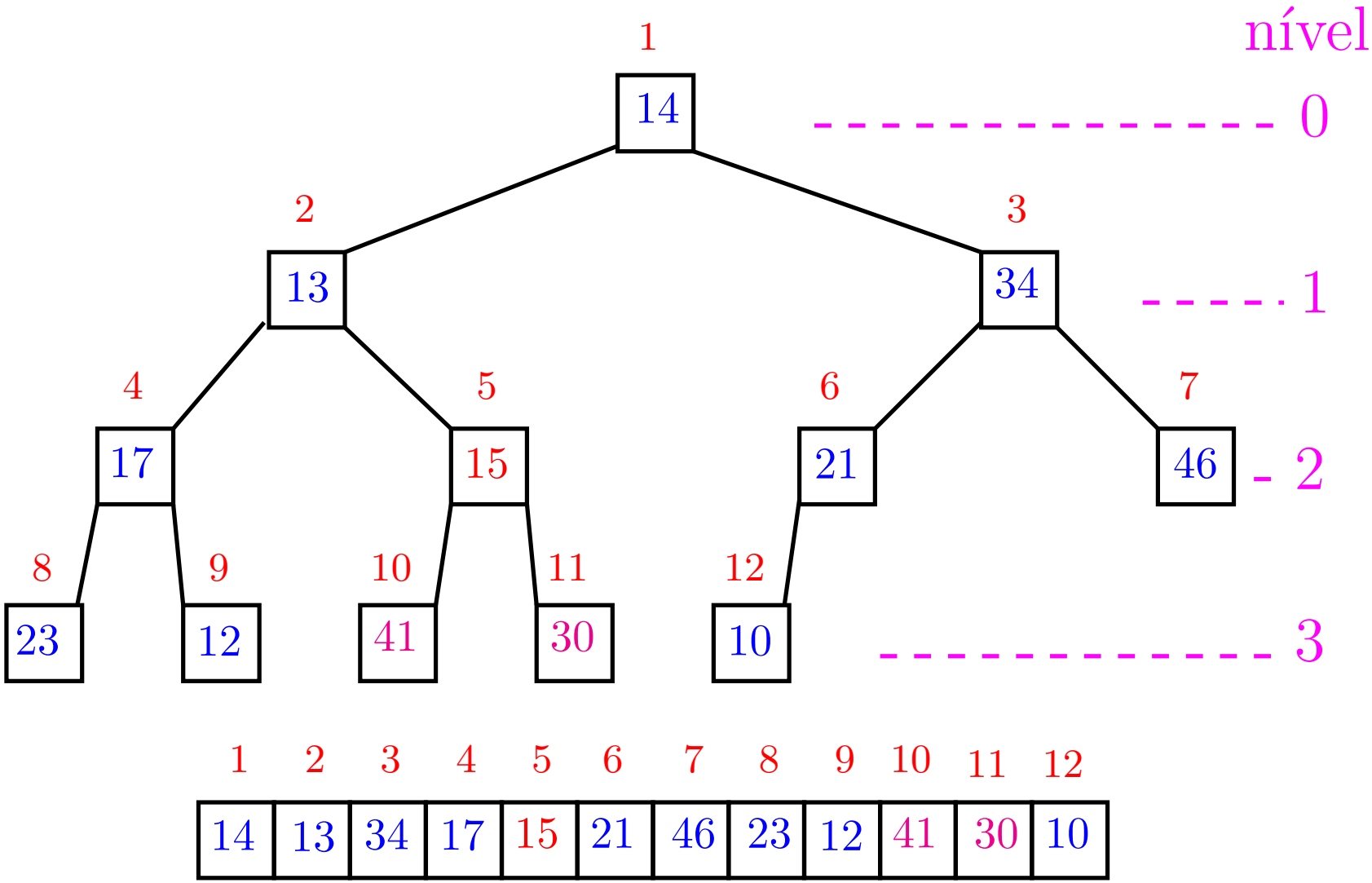
Construção de um `max-heap`



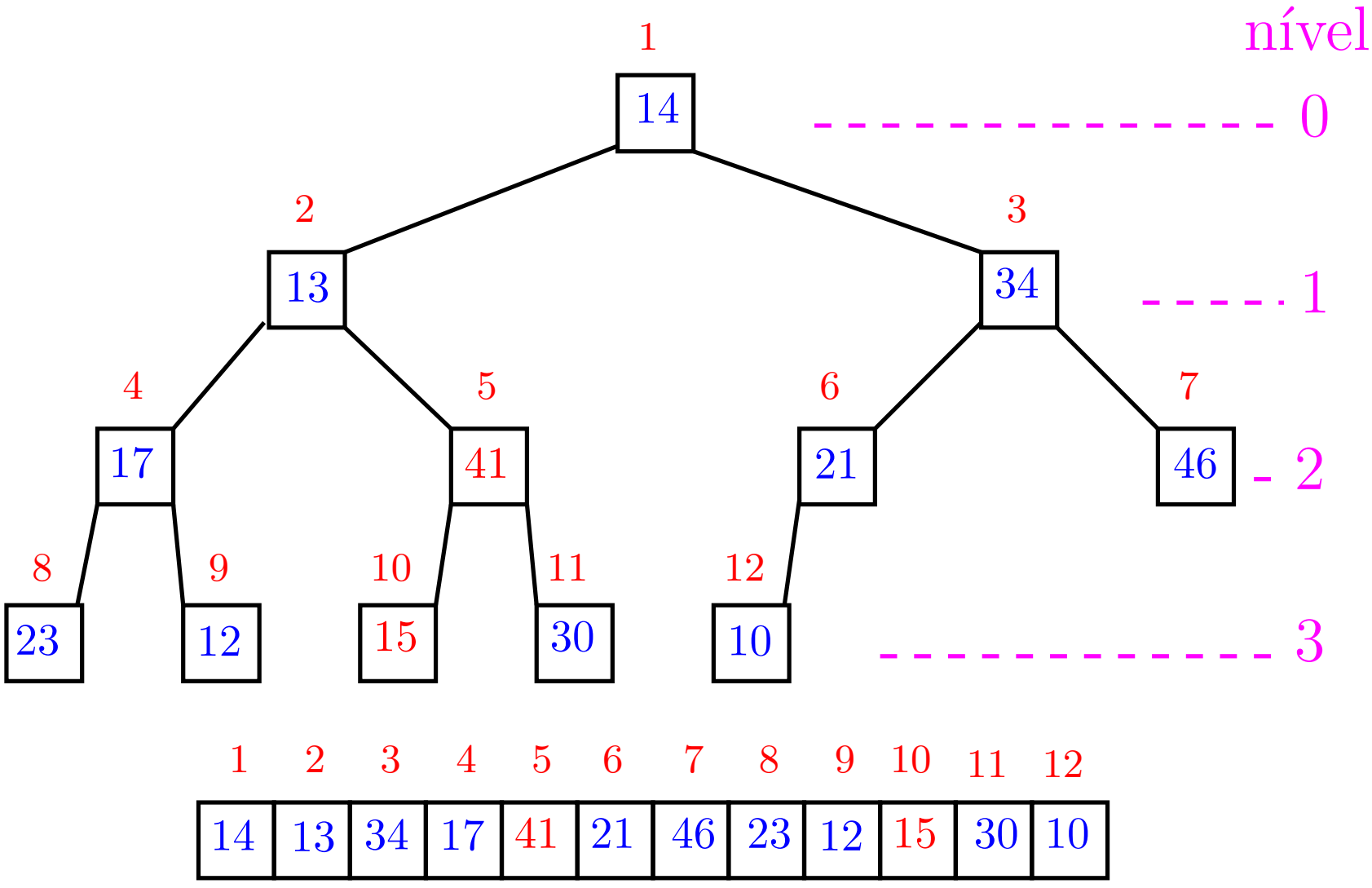
Construção de um `max-heap`



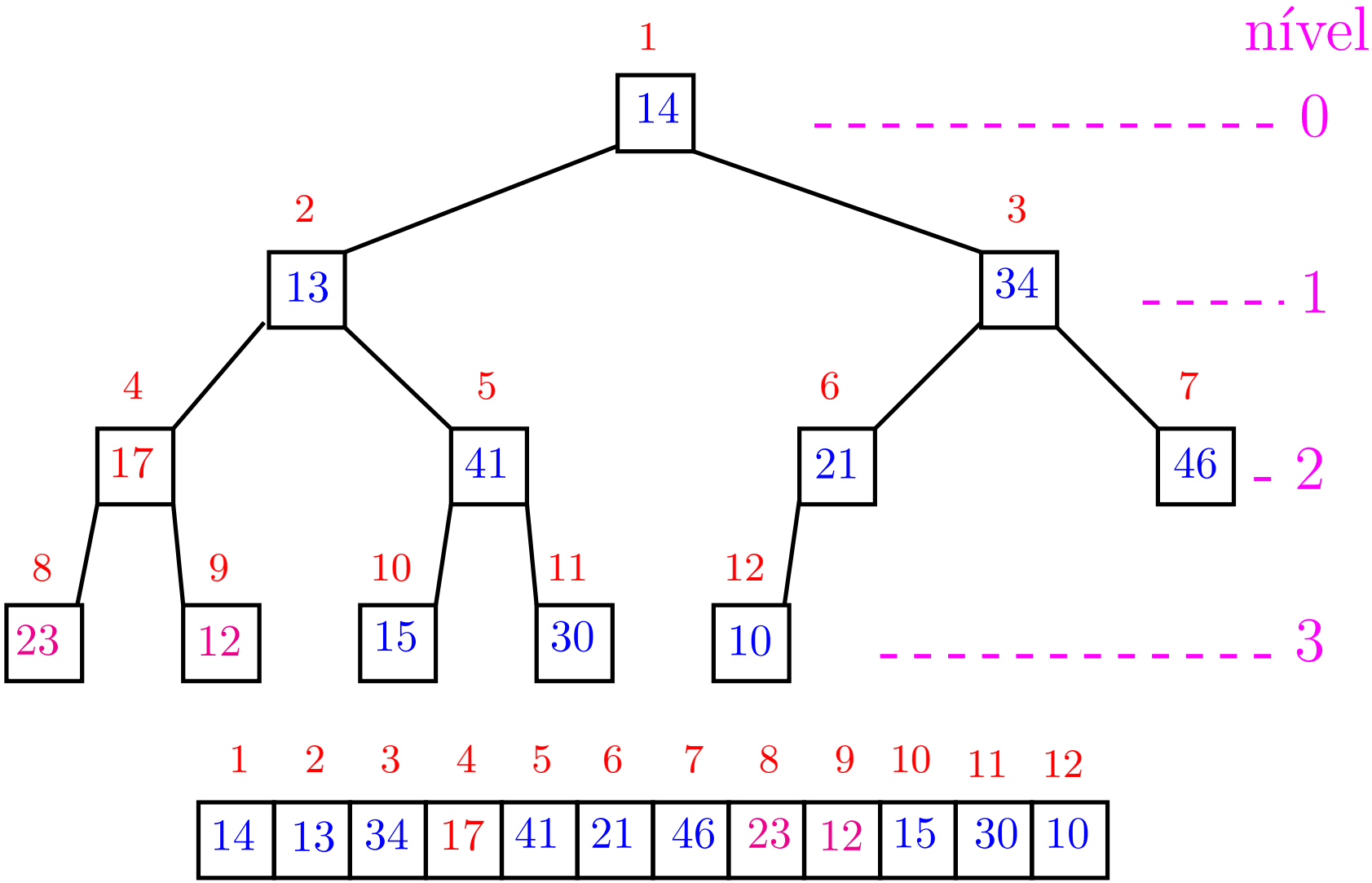
Construção de um `max-heap`



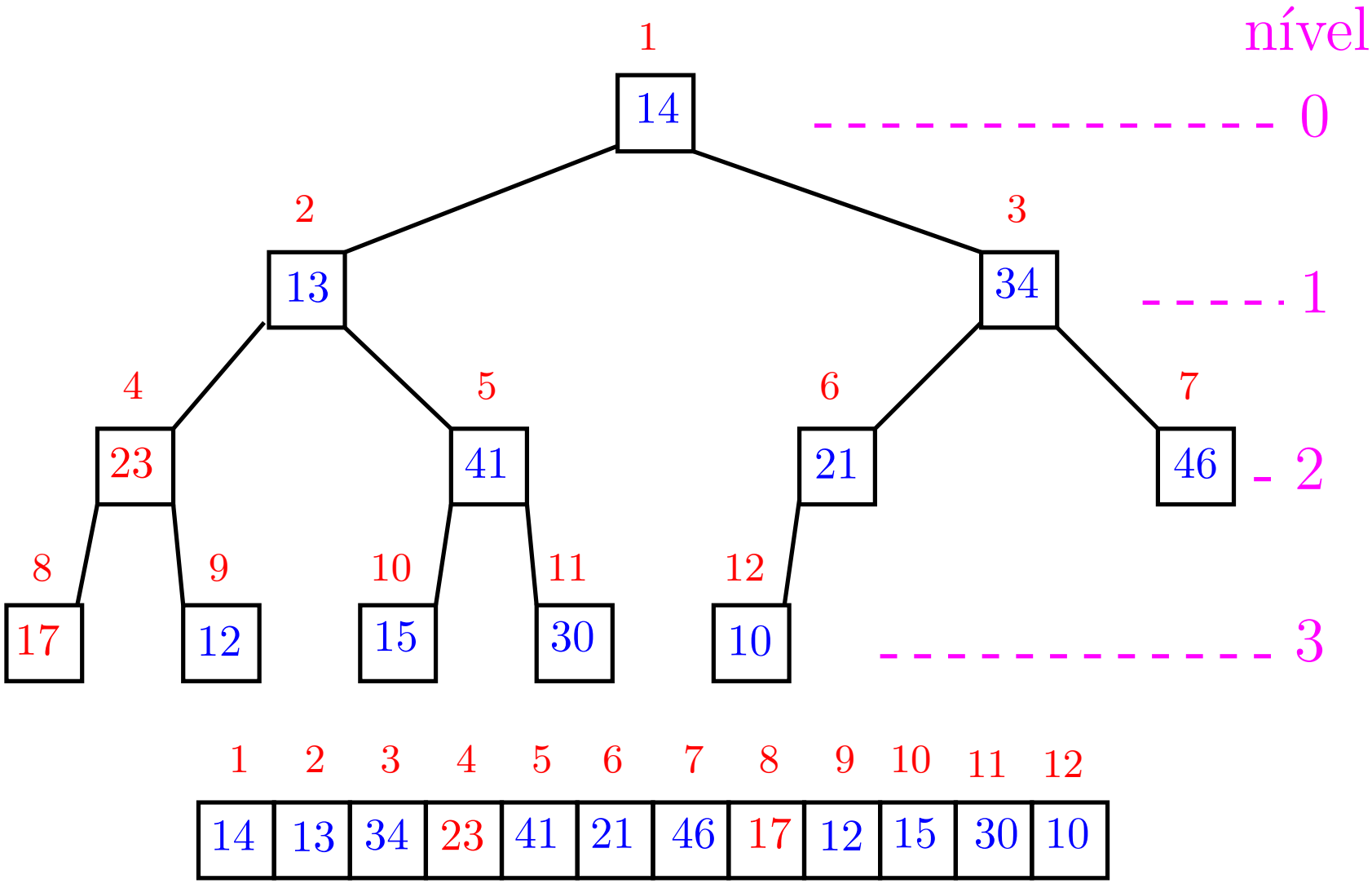
Construção de um `max-heap`



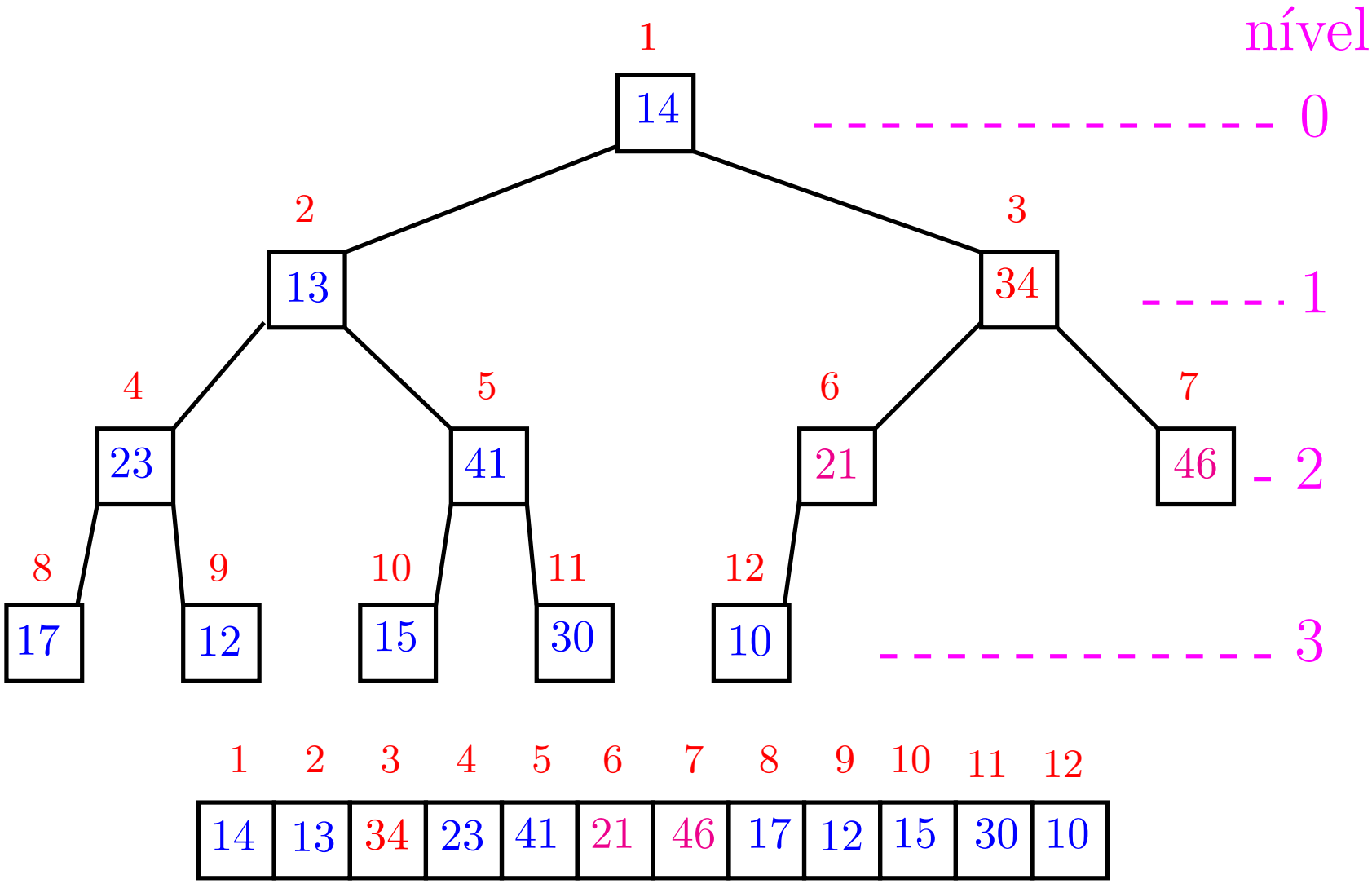
Construção de um `max-heap`



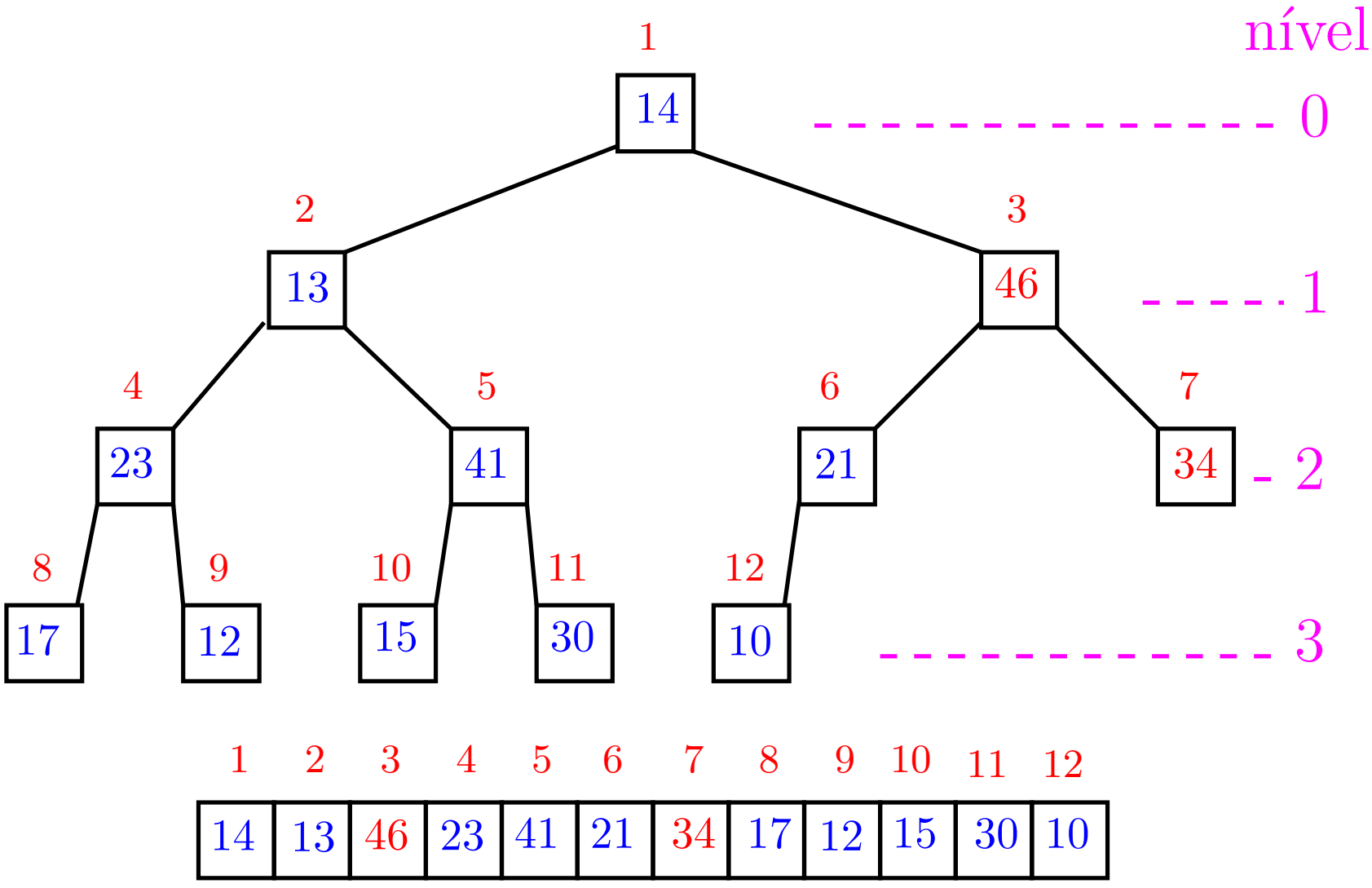
Construção de um `max-heap`



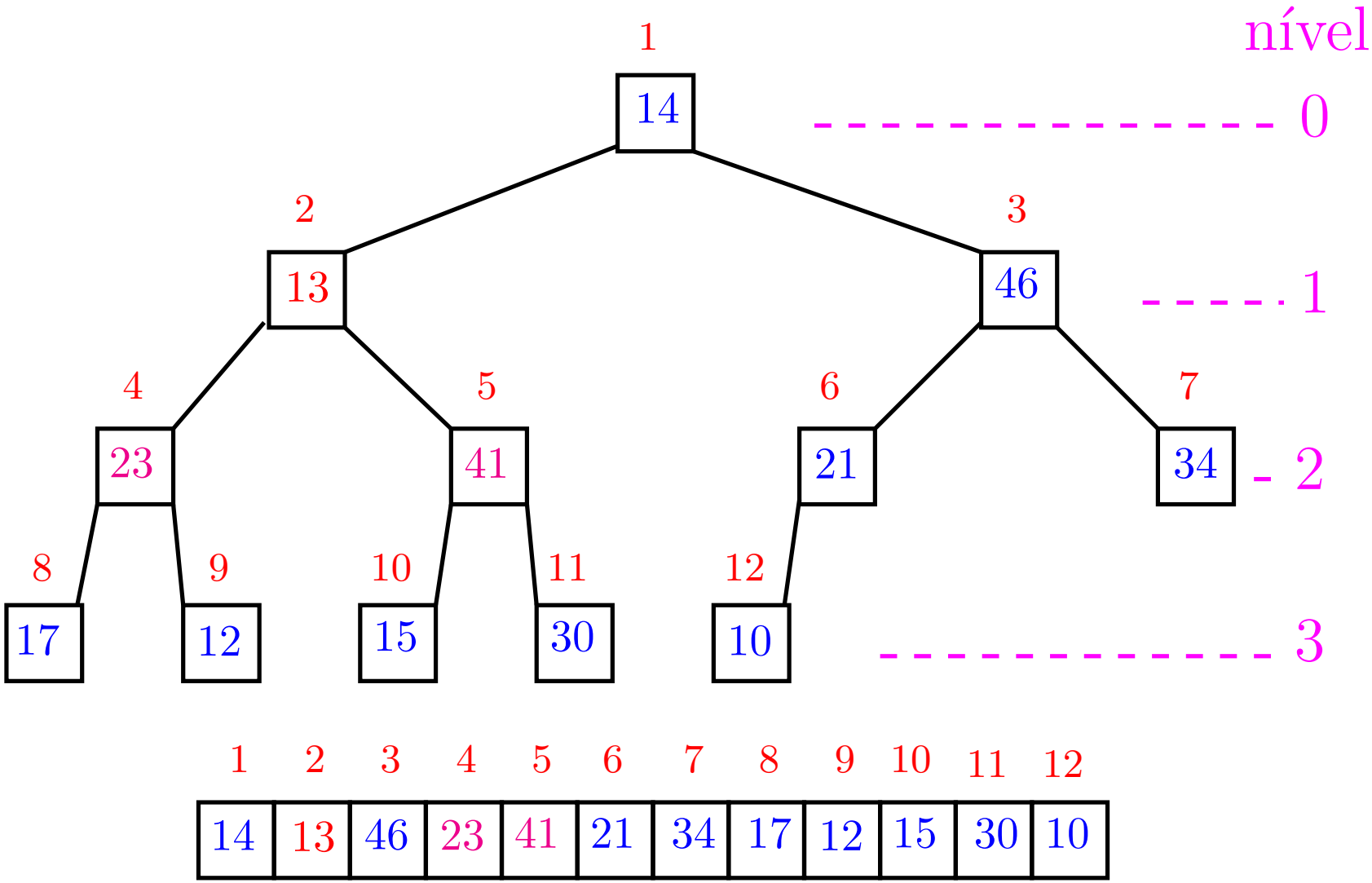
Construção de um `max-heap`



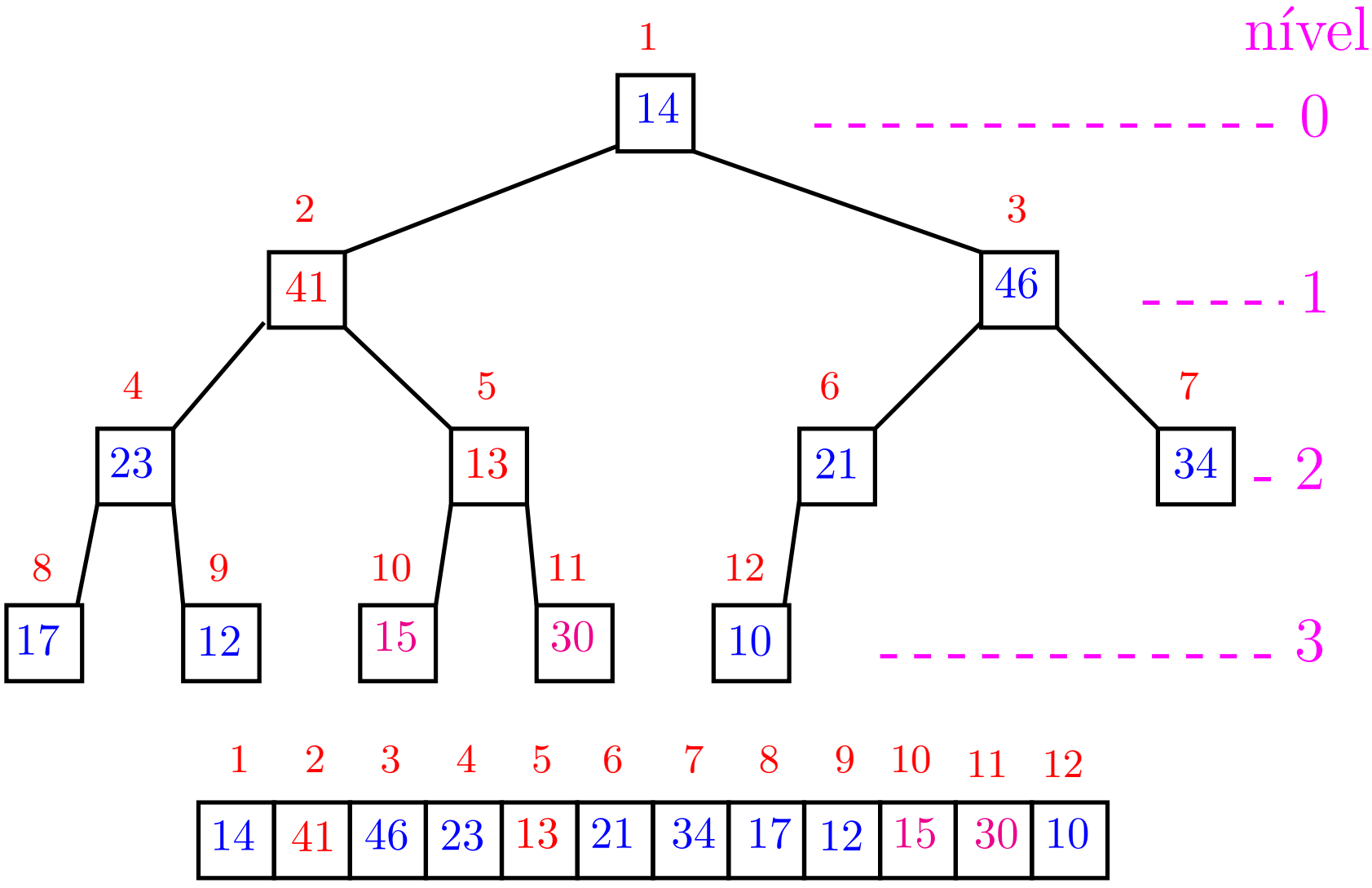
Construção de um `max-heap`



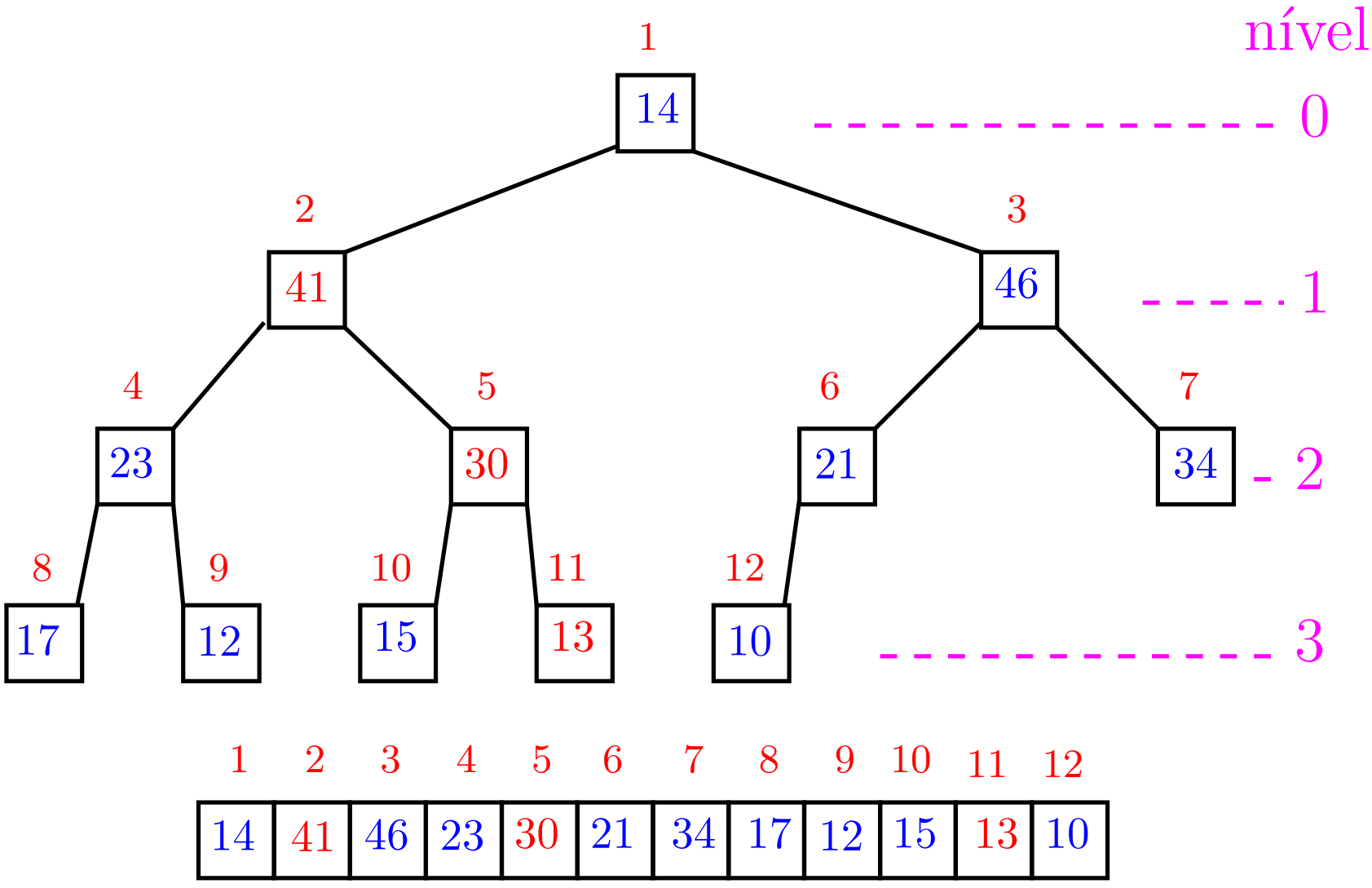
Construção de um max-heap



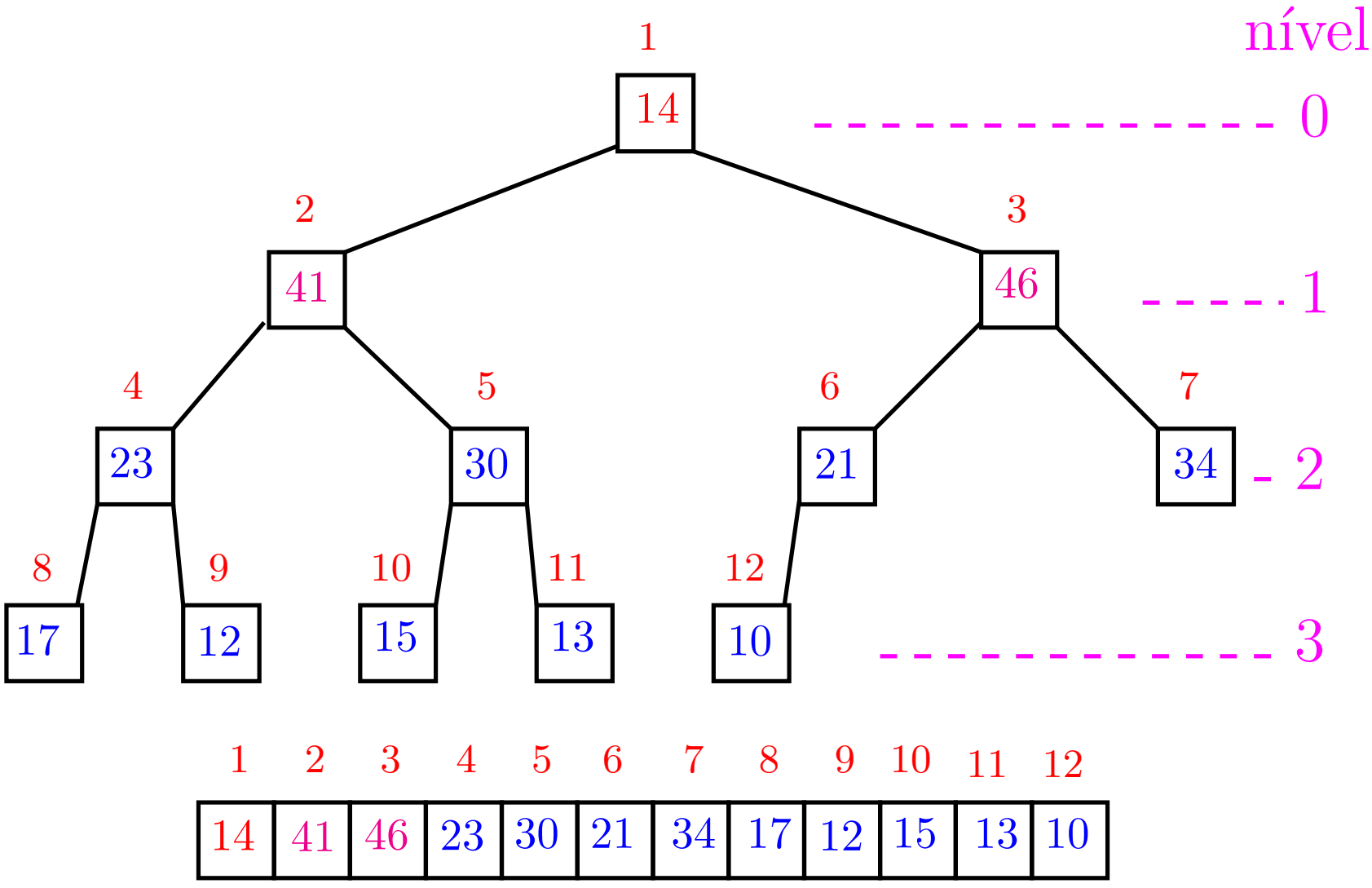
Construção de um max-heap



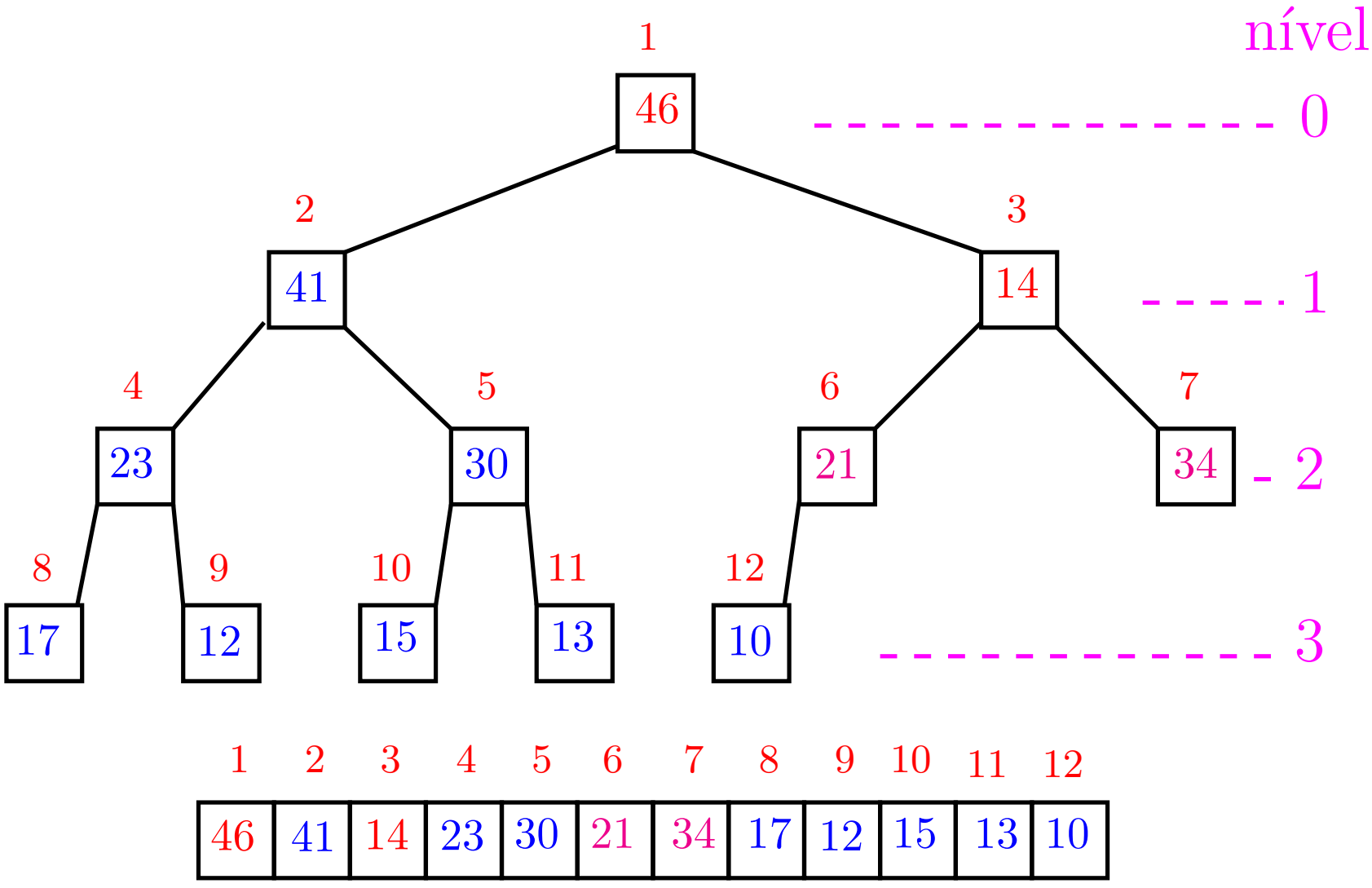
Construção de um `max-heap`



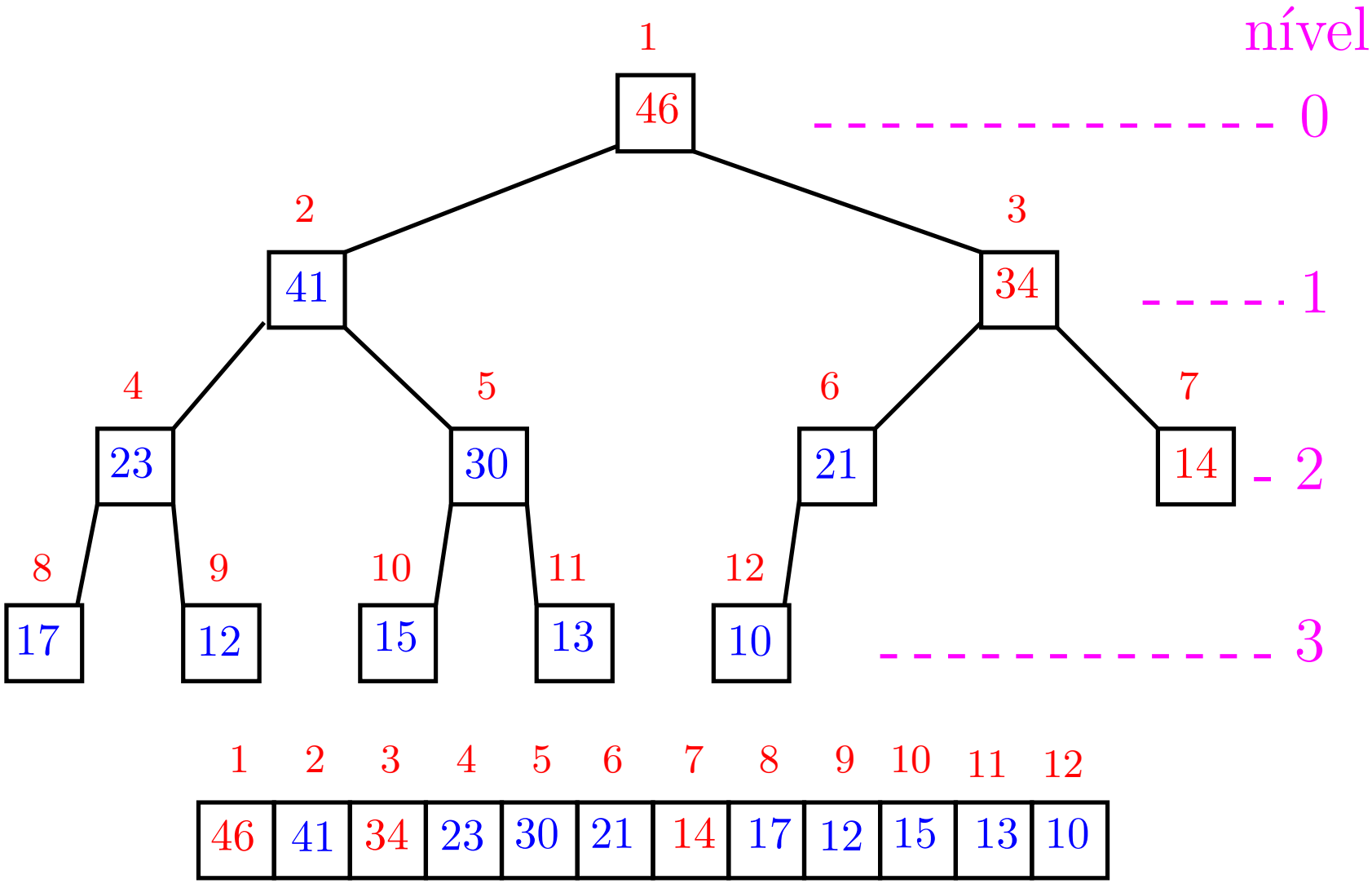
Construção de um `max-heap`



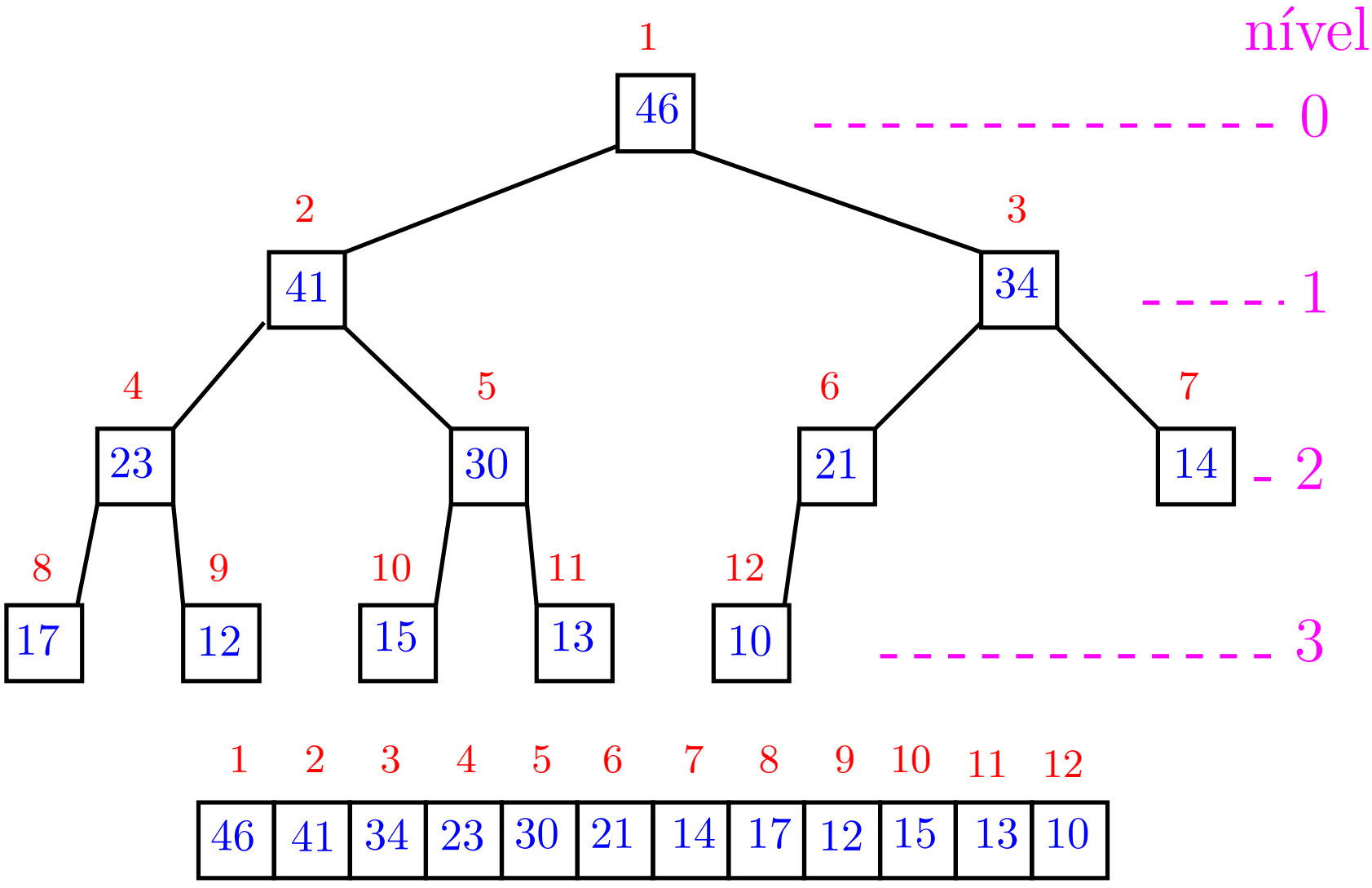
Construção de um `max-heap`



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

Relação invariante:

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

Consumo de tempo

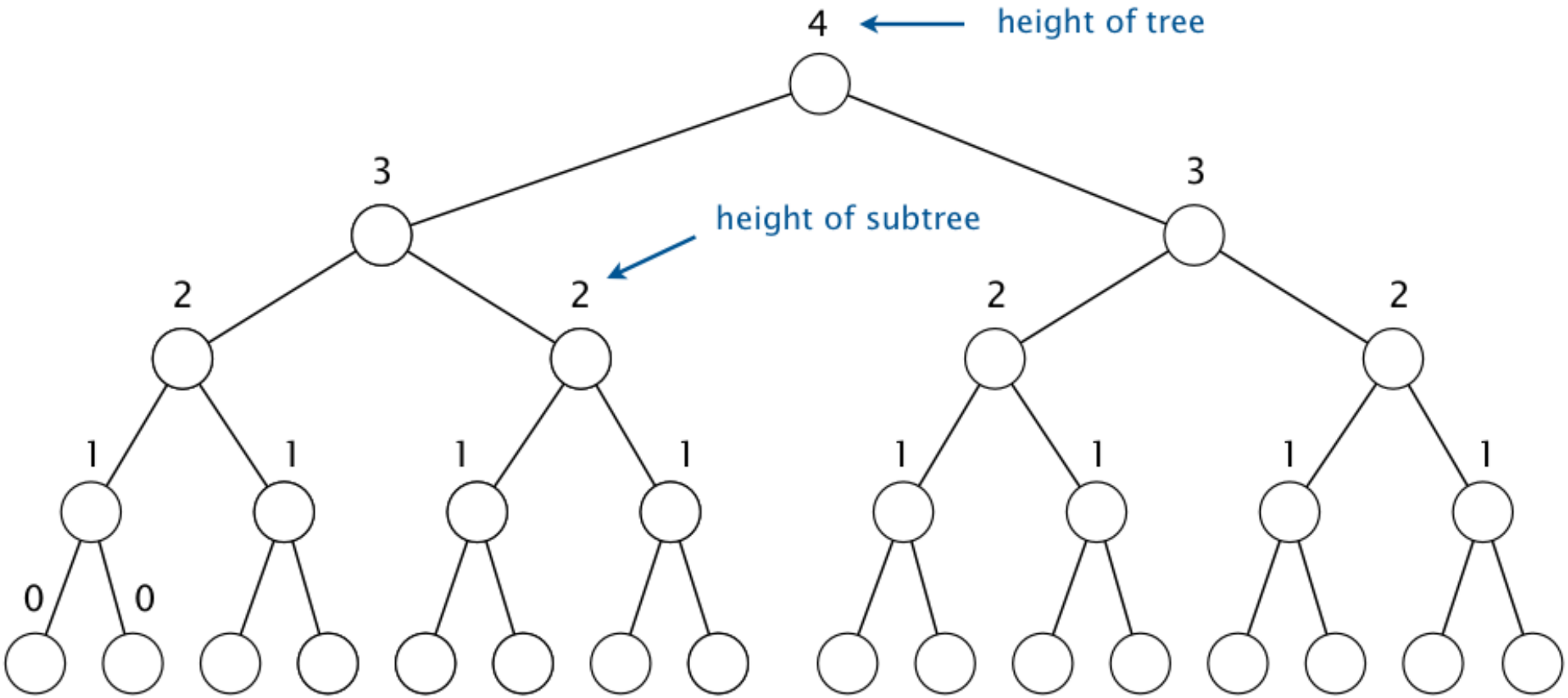
Análise grosseira: consumo de tempo é

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

Verdade seja dita . . . (. . .)

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

Altura das subárvores



Fonte: algs4

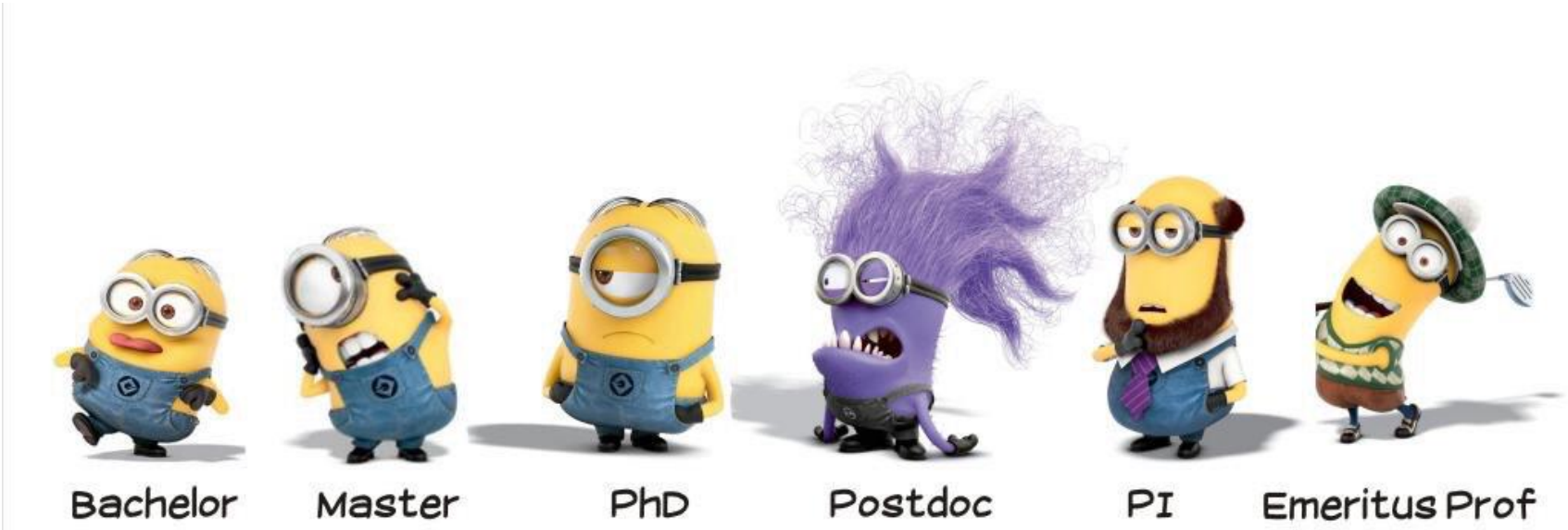
Consumo de tempo

Suponha que a altura do max-heap seja h.
Existem no máximo 2^{h-k} nós de altura k.
Número de deslocamentos feitos durante a construção do heap é não superior a

$$\begin{aligned} h + 2(h-1) + 2^2(h-2) + \cdots + 2^h(0) &= 2^{h+1} - h - 2 \\ &= m - (h+1) \\ &< m. \end{aligned}$$

A segunda igualdade é devida ao fato de que uma árvore binária completa de altura h tem $2^{h+1} - 1$ nós.

Ordenação: algoritmo Heapsort



Fonte: tag:benkler

PF 10

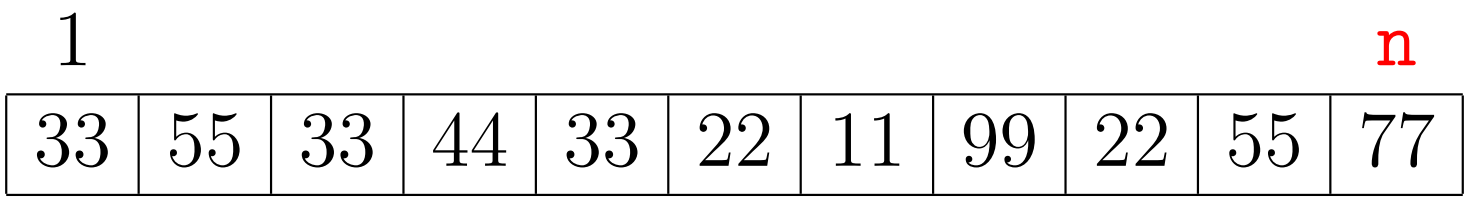
http://www.ime.usp.br/~pf/algoritmos/aulas/hpsrt.html

Ordenação

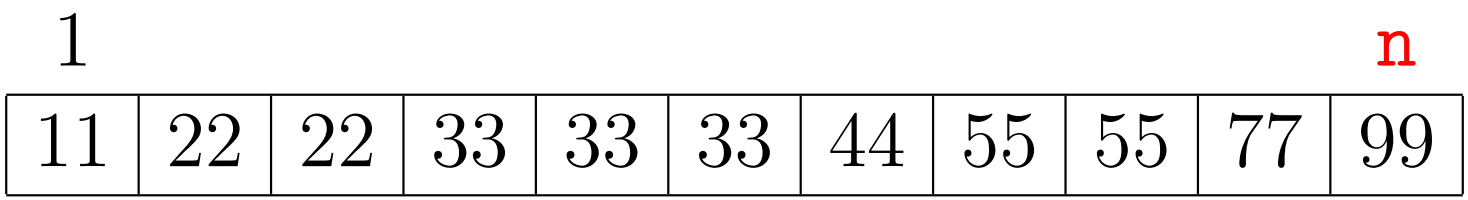
$\mathtt{v}[1\,..\,\mathtt{n}]$ é crescente se $\mathtt{v}[1] \leq \cdots \leq \mathtt{v}[\mathtt{n}]$.

Problema: Rearranjar um vetor $\mathtt{v}[1\,..\,\mathtt{n}]$ de modo que ele fique crescente.

Entra:



Sai:



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:

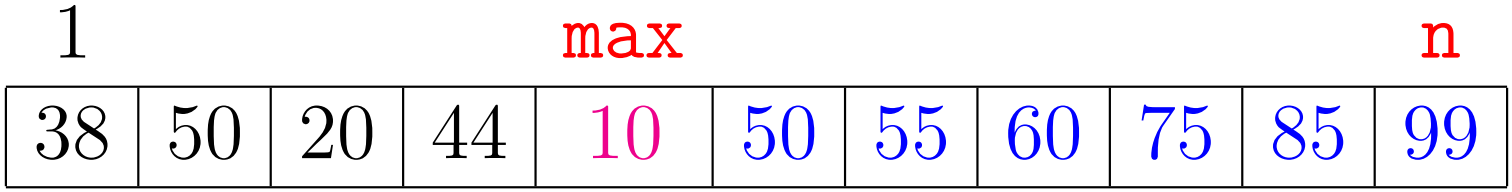
1										n
33	55	33	44	33	22	11	99	22	55	77

Sai:

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

Ordenação por seleção

$\texttt{i} = 5$



Ordenação por seleção

$\texttt{i} = 5$

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

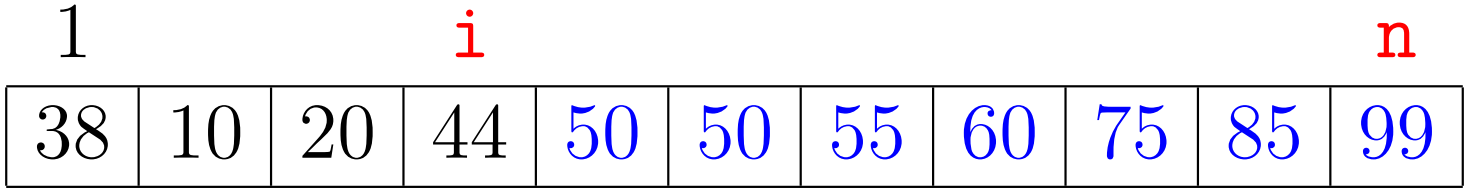
j max n

38	50	20	44	10	50	55	60	75	85	99

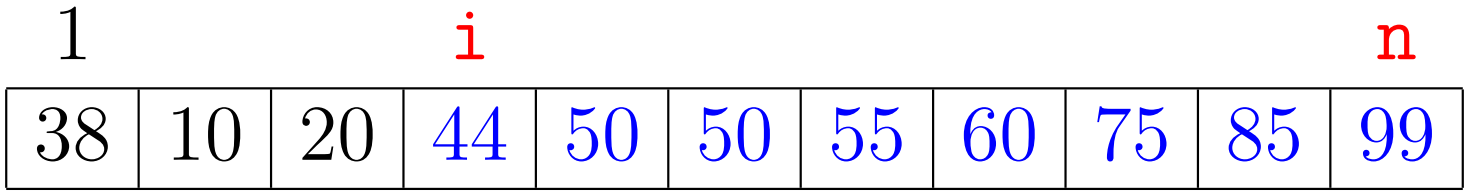
1 max n

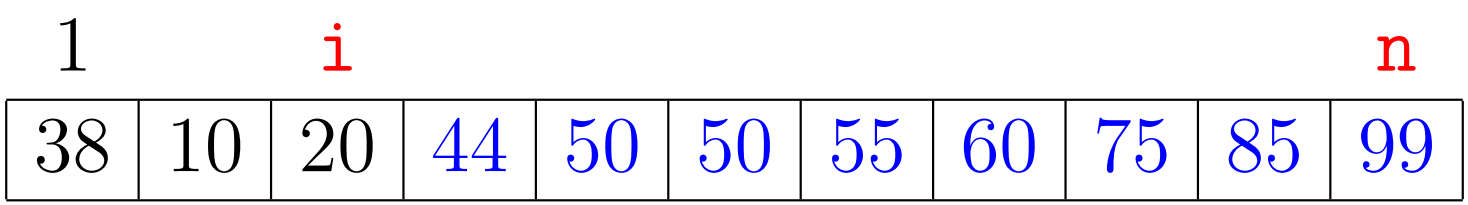
38	50	20	44	10	50	55	60	75	85	99

Ordenação por seleção

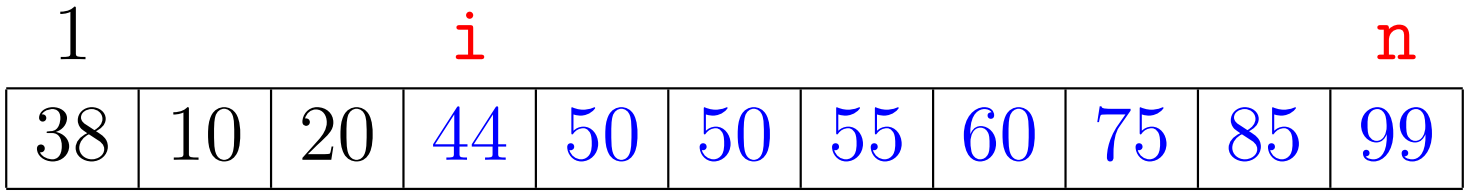


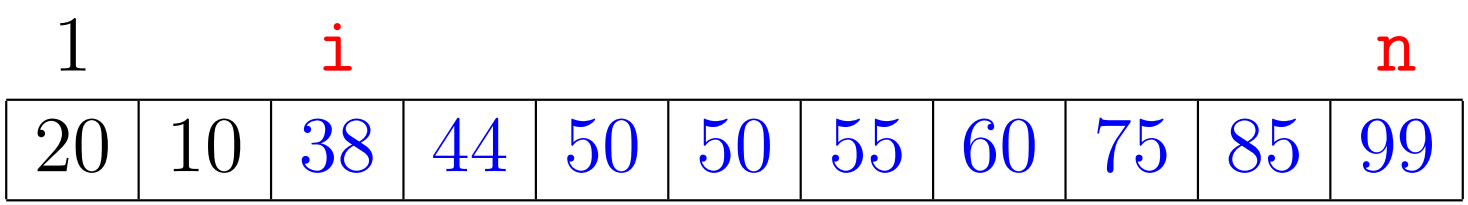
Ordenação por seleção

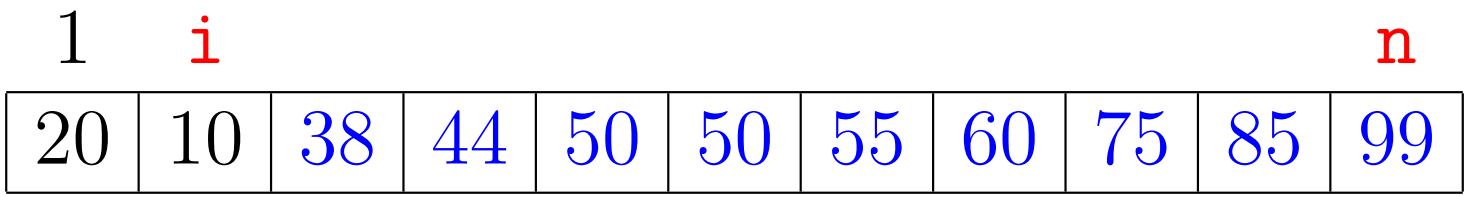




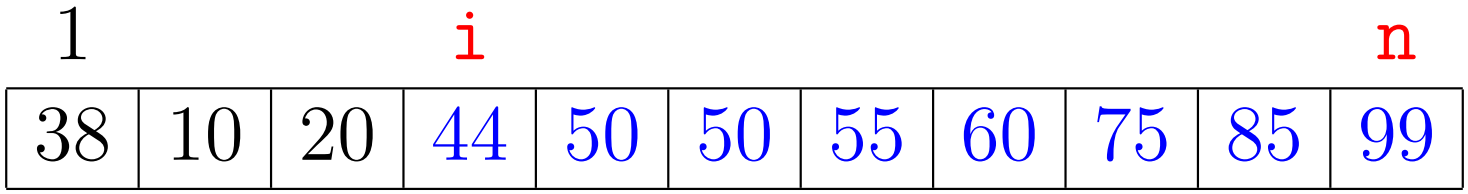
Ordenação por seleção

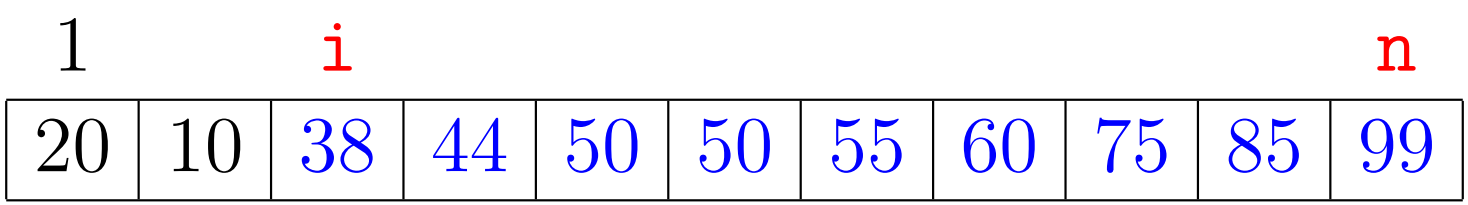


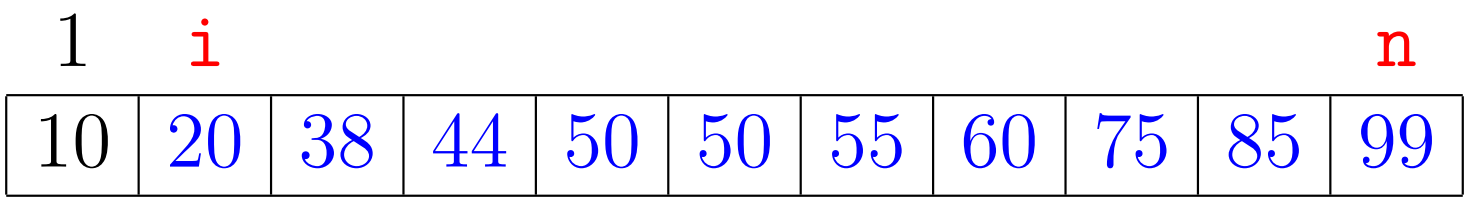


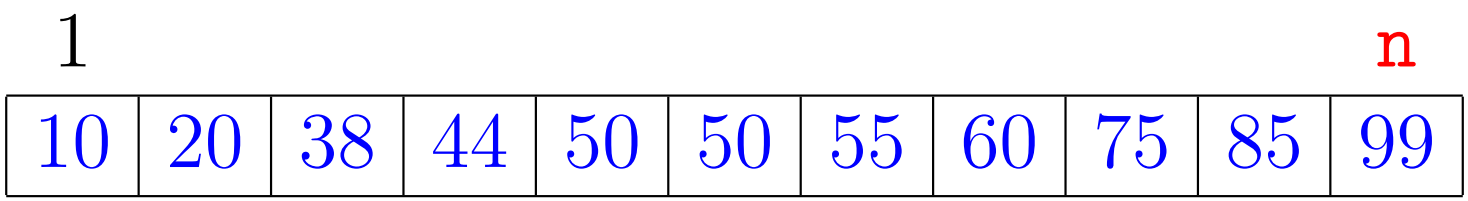


Ordenação por seleção









Função selecao

Algoritmo rearranja $v[0\,..\,n-1]$ em ordem crescente

```
public static
void selecao (int n, Comparable[] v)
{
    int i, j, max; Object x;
1   for (i = n-1; /*B*/ i > 0; i--) {
2       max = i;
3       for (j = i-1; j >= 0; j--)
4           if (!less(v[j],v[max]))
5               max = j;
6       x=v[i]; v[i]=v[max]; v[max]=x;
    }
}
```

Função selecao

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

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

Função selecao

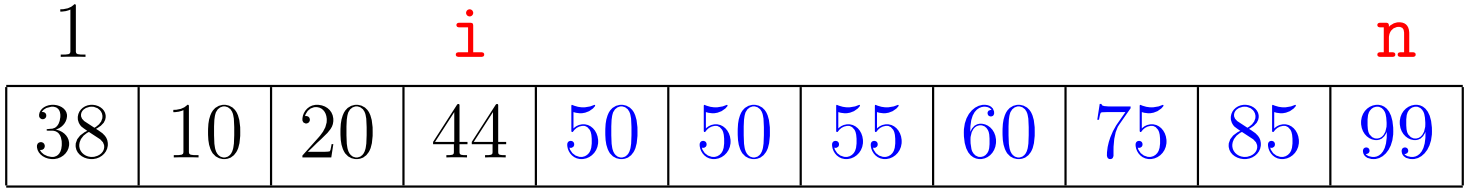
```
private static
boolean less(Comparable x, Comparable y)
{
   return x.compareTo(y) < 0
}
```

Função `selecao`

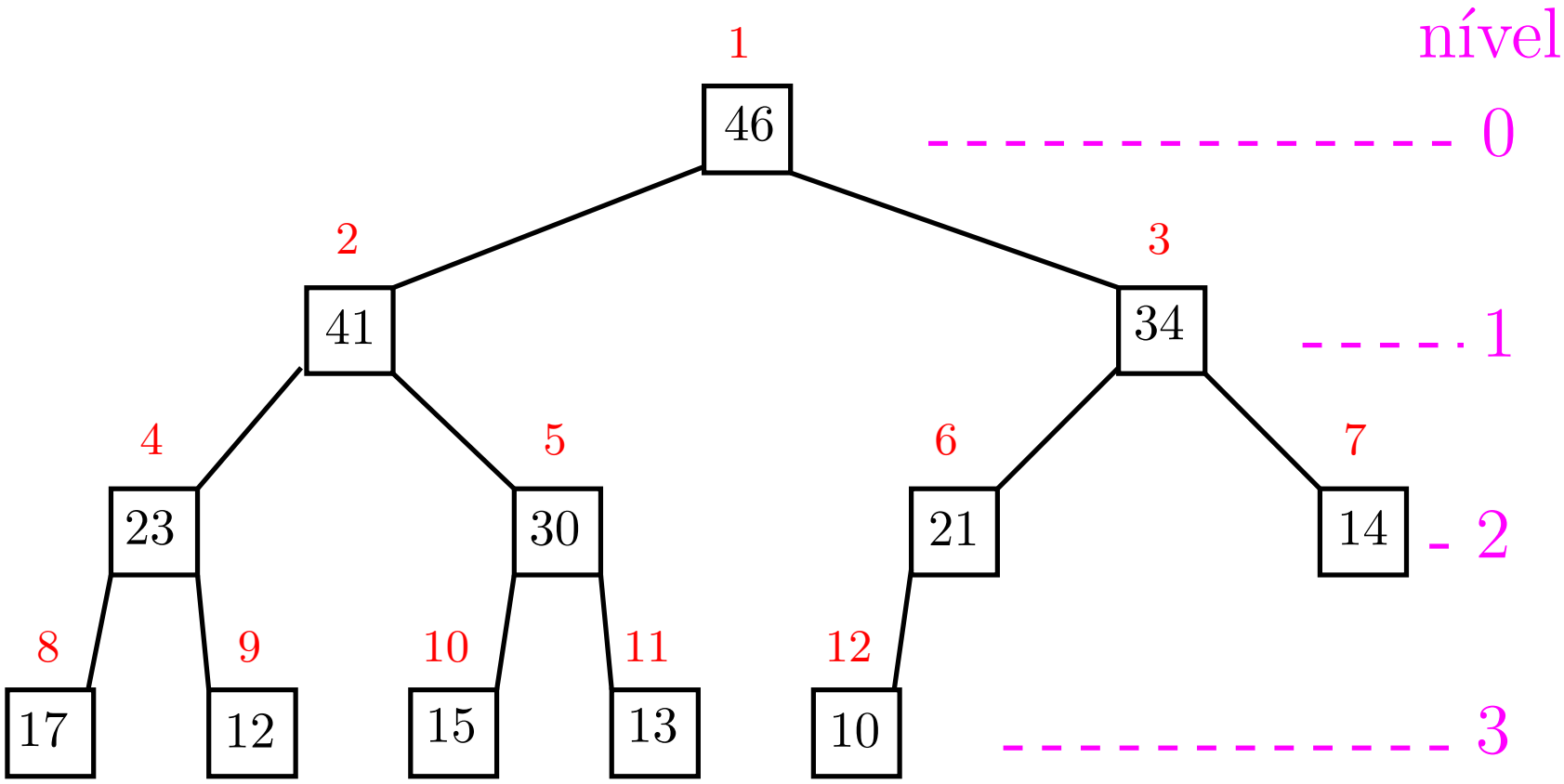
Relações invariantes: Em /*B*/ vale que:

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

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

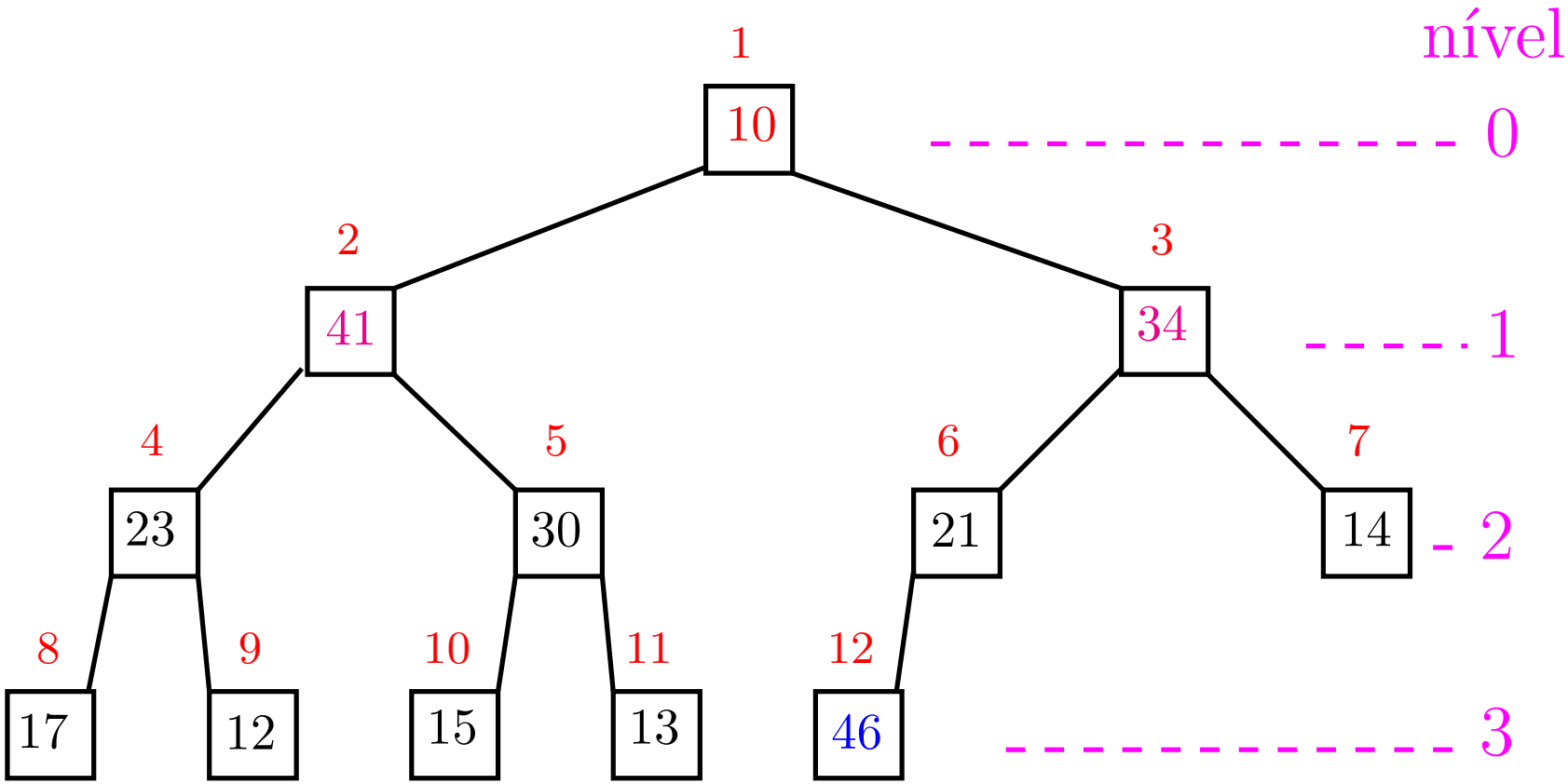


Heapsort



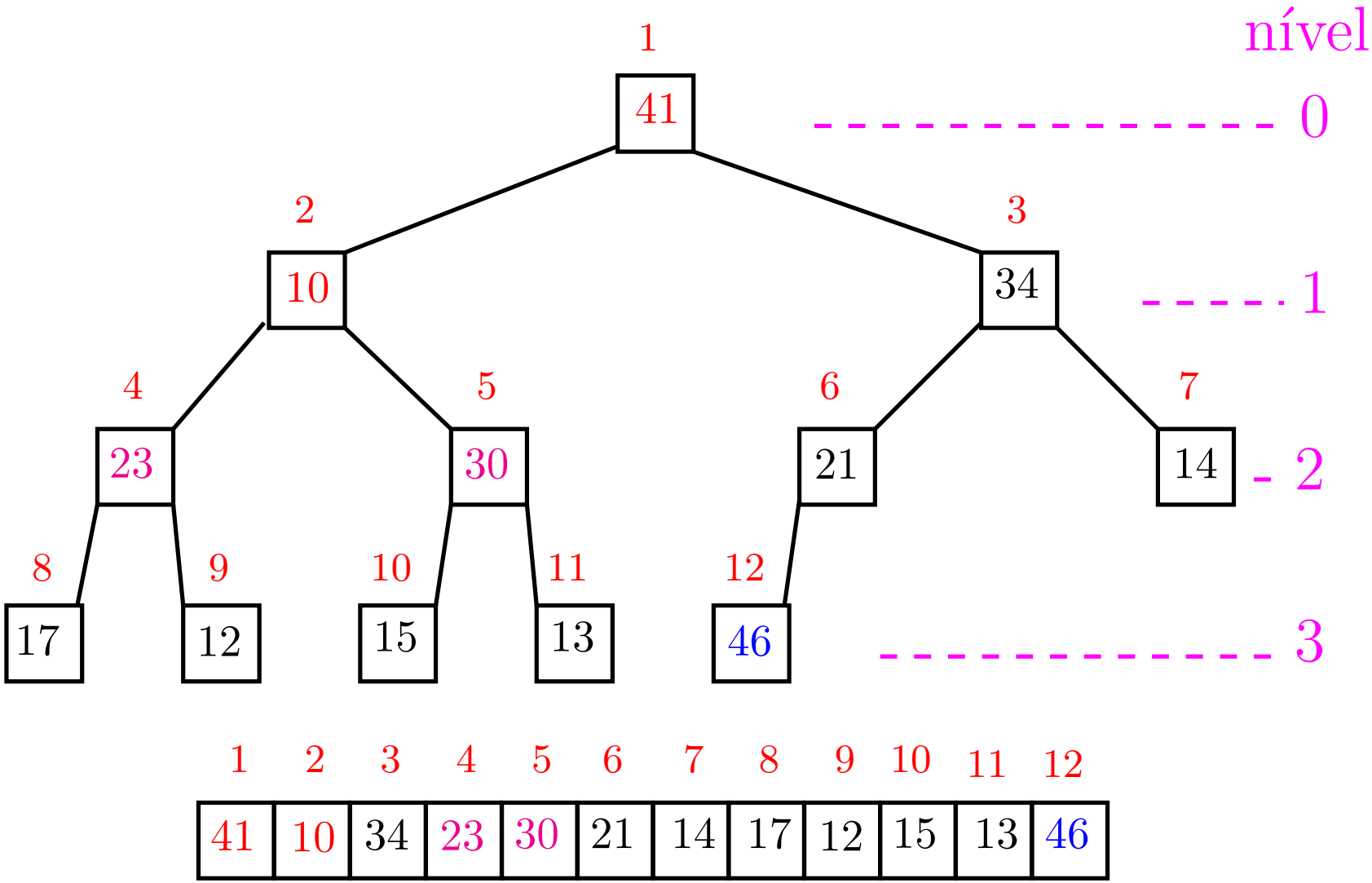
1	2	3	4	5	6	7	8	9	10	11	12
46	41	34	23	30	21	14	17	12	15	13	10

Heapsort

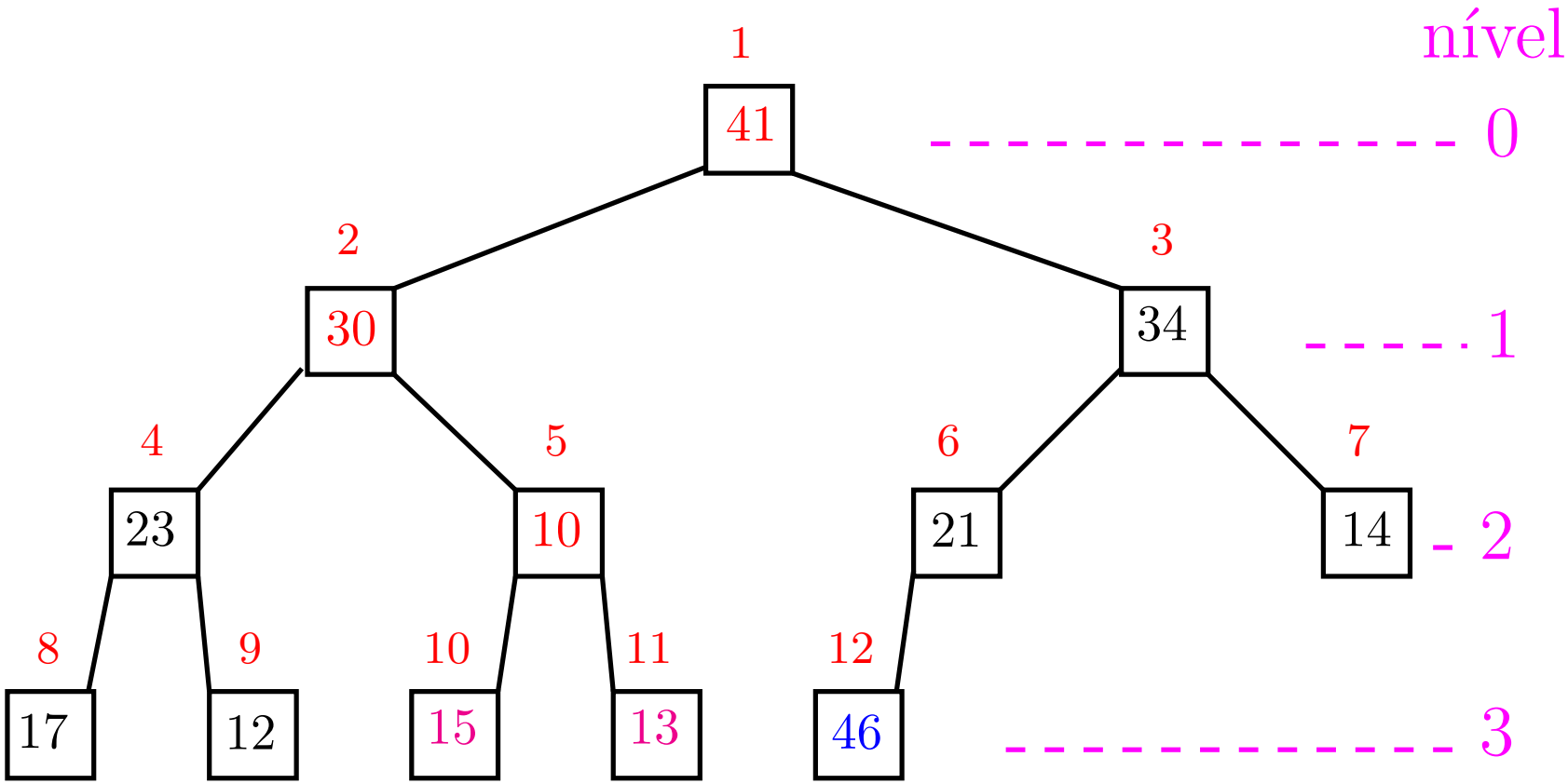


1	2	3	4	5	6	7	8	9	10	11	12
10	41	34	23	30	21	14	17	12	15	13	46

Heapsort

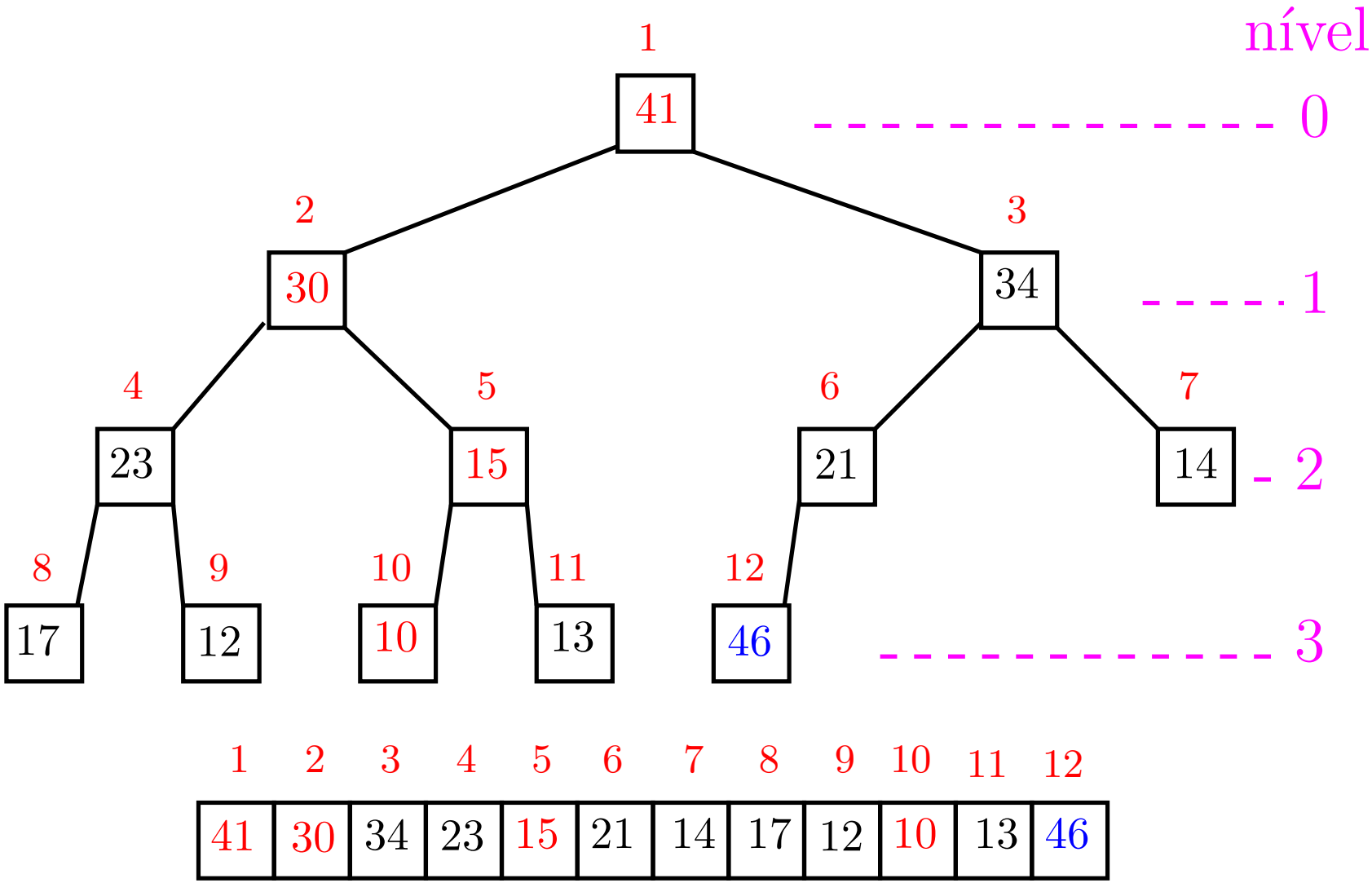

nível
1
41
0
2
10
3
34
1
4
23
5
30
6
21
7
14
2
8
17
9
12
10
15
11
13
12
46
3
1 2 3 4 5 6 7 8 9 10 11 12
41 10 34 23 30 21 14 17 12 15 13 46

Heapsort

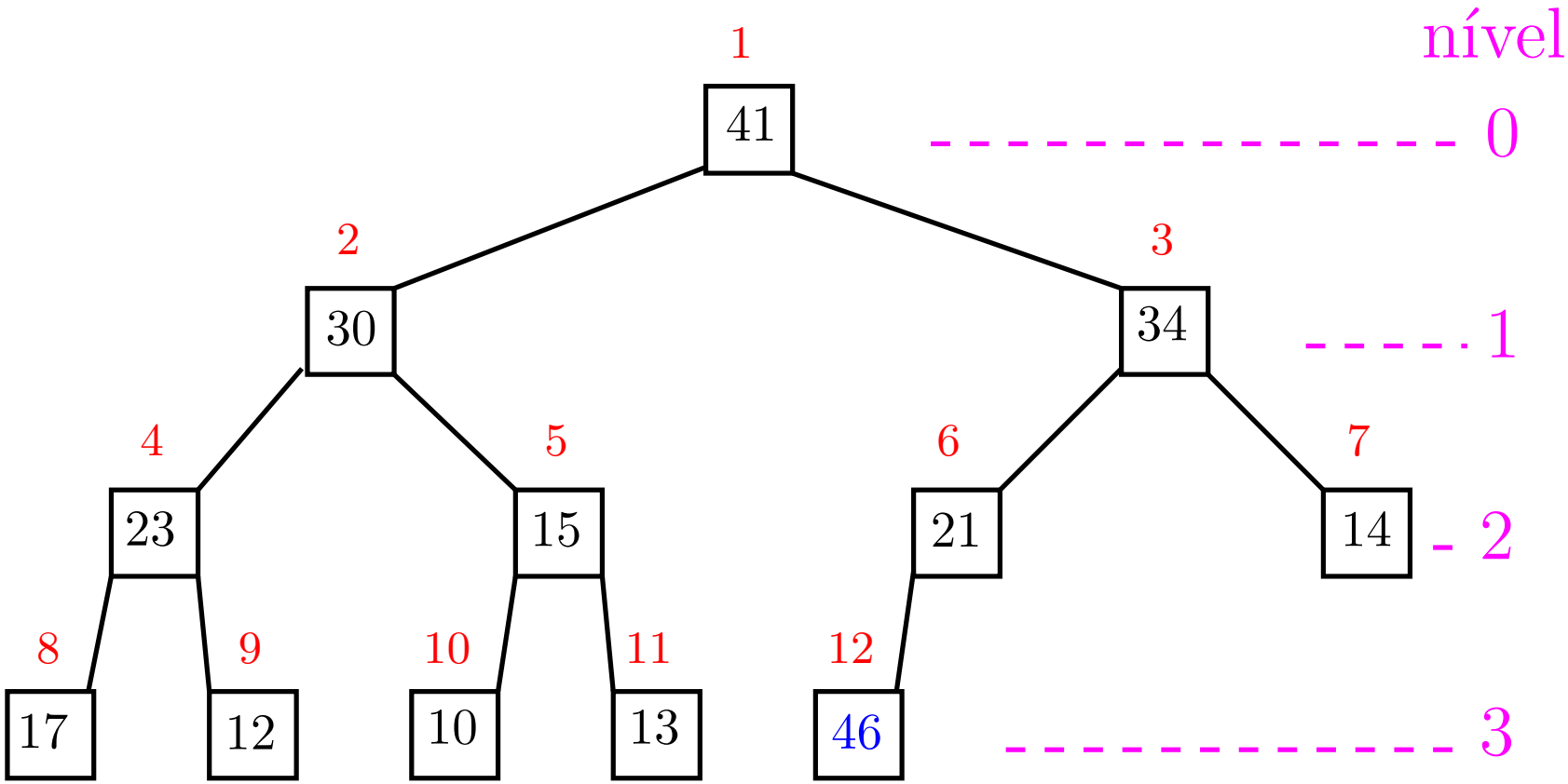


1	2	3	4	5	6	7	8	9	10	11	12
41	30	34	23	10	21	14	17	12	15	13	46

Heapsort

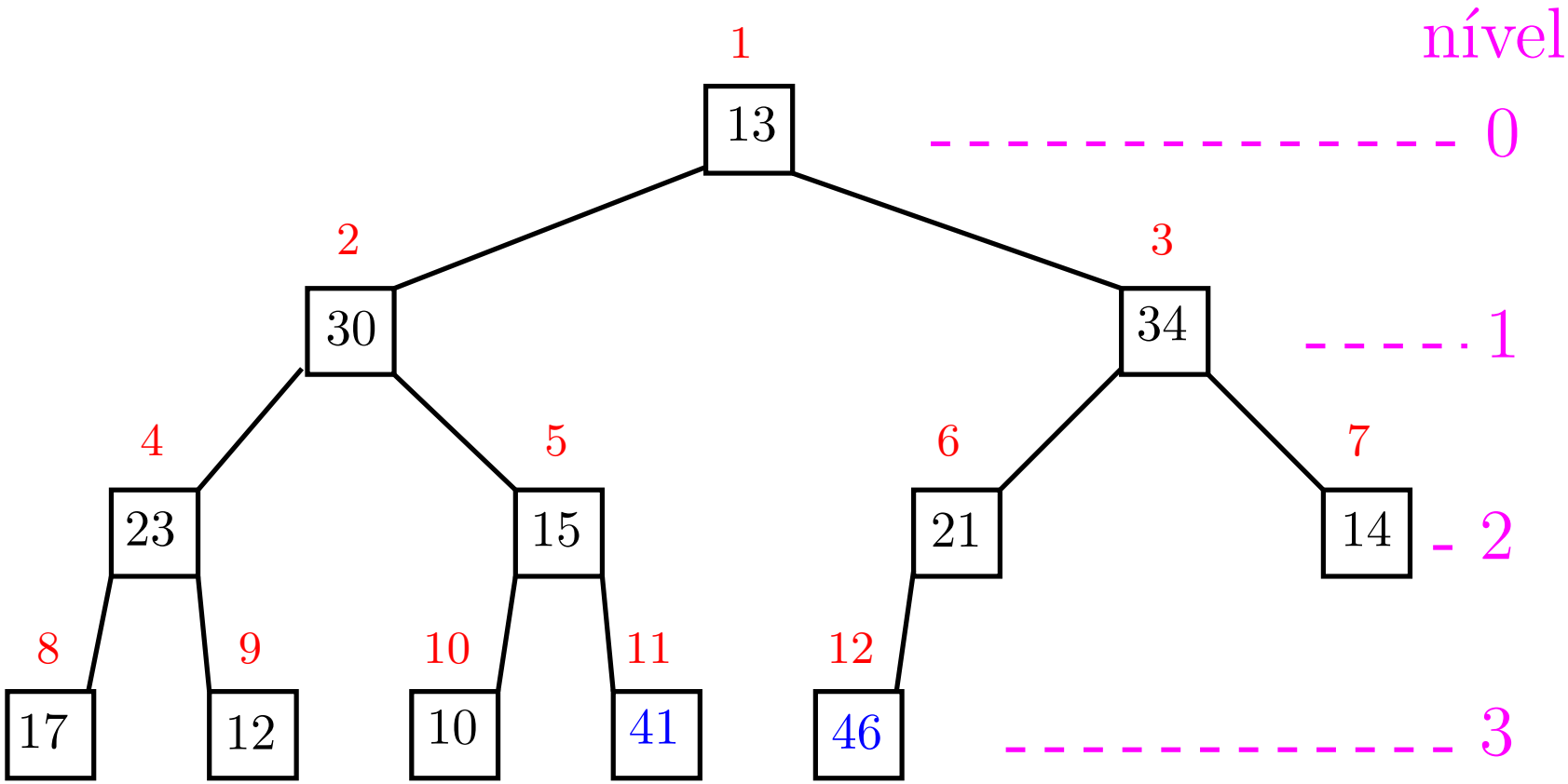

nível
1
41
0
2
30
3
34
1
4
23
5
15
6
21
7
14
2
8
17
9
12
10
10
11
13
12
46
3
1 2 3 4 5 6 7 8 9 10 11 12
41 30 34 23 15 21 14 17 12 10 13 46

Heapsort



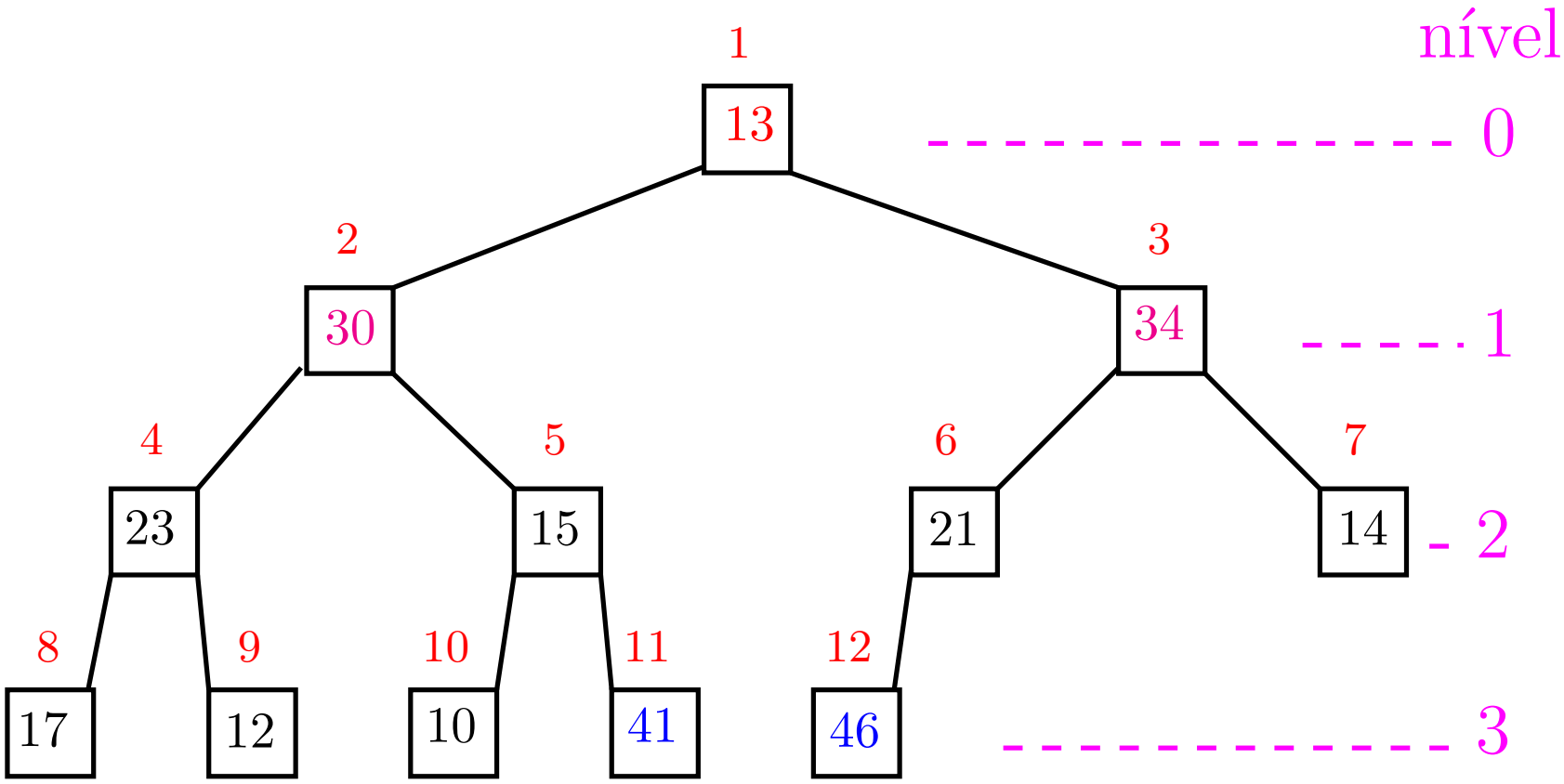
1	2	3	4	5	6	7	8	9	10	11	12
41	30	34	23	15	21	14	17	12	10	13	46

Heapsort



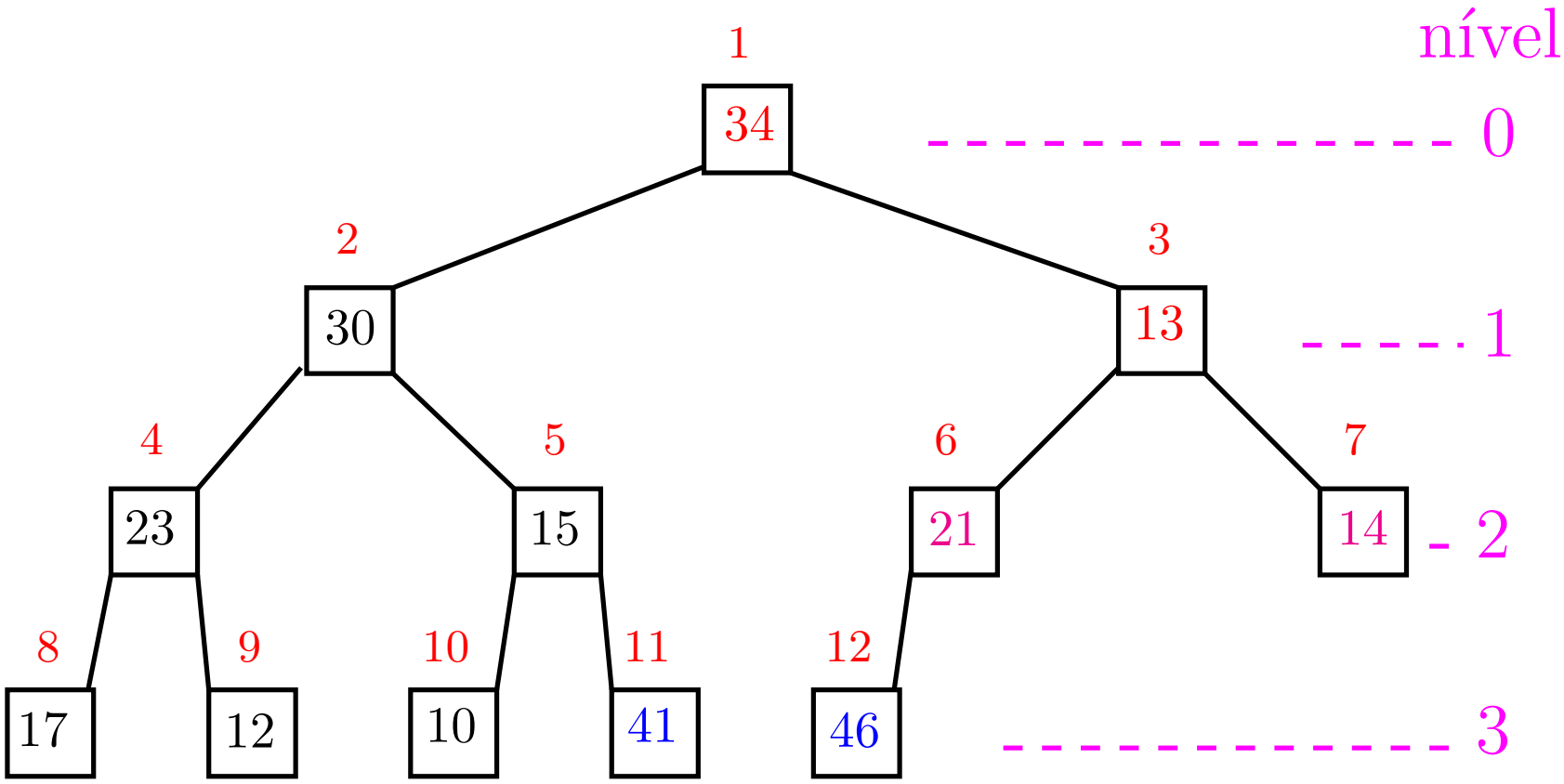
1	2	3	4	5	6	7	8	9	10	11	12
13	30	34	23	15	21	14	17	12	10	41	46

Heapsort



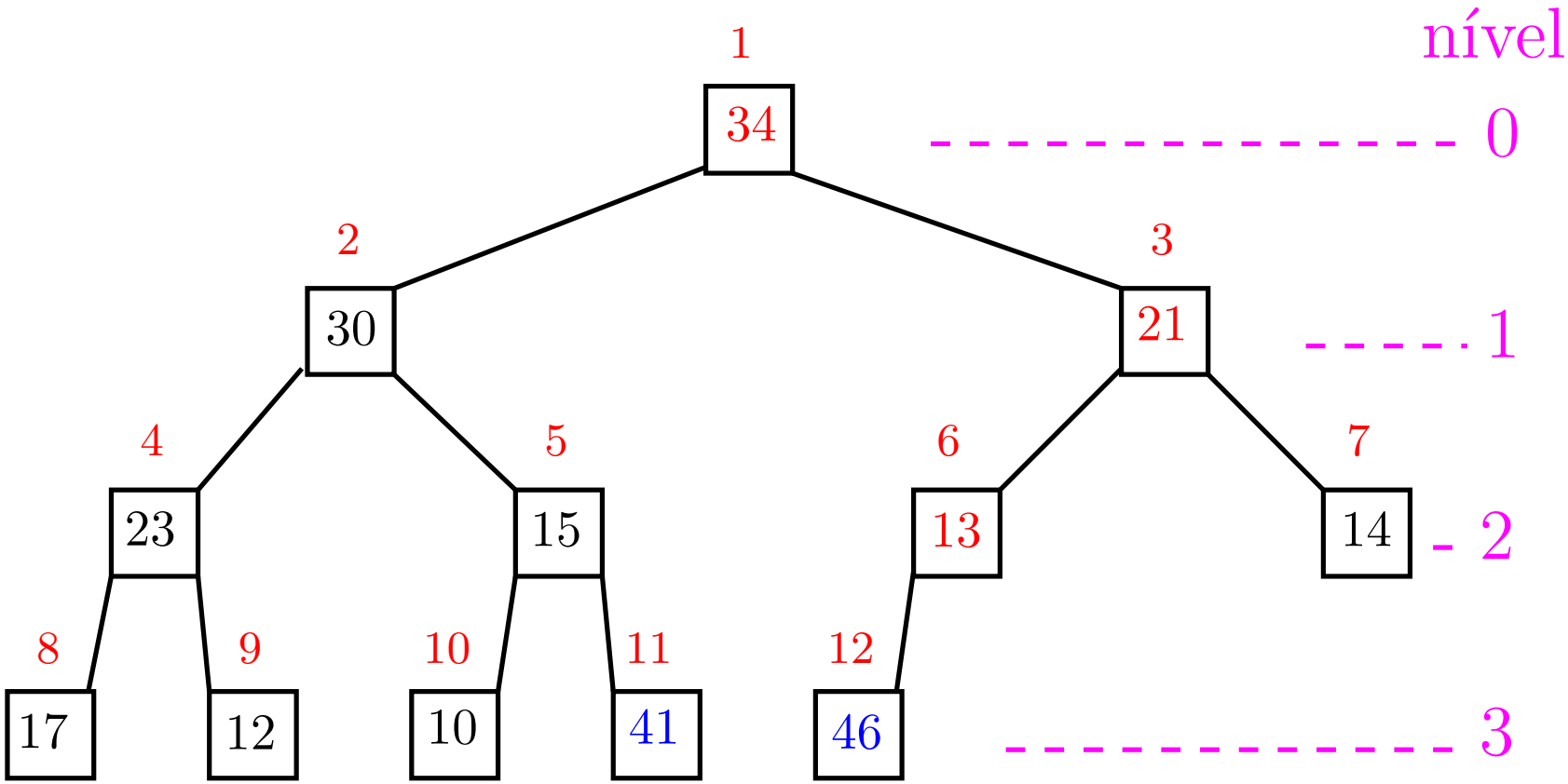
1	2	3	4	5	6	7	8	9	10	11	12
13	30	34	23	15	21	14	17	12	10	41	46

Heapsort



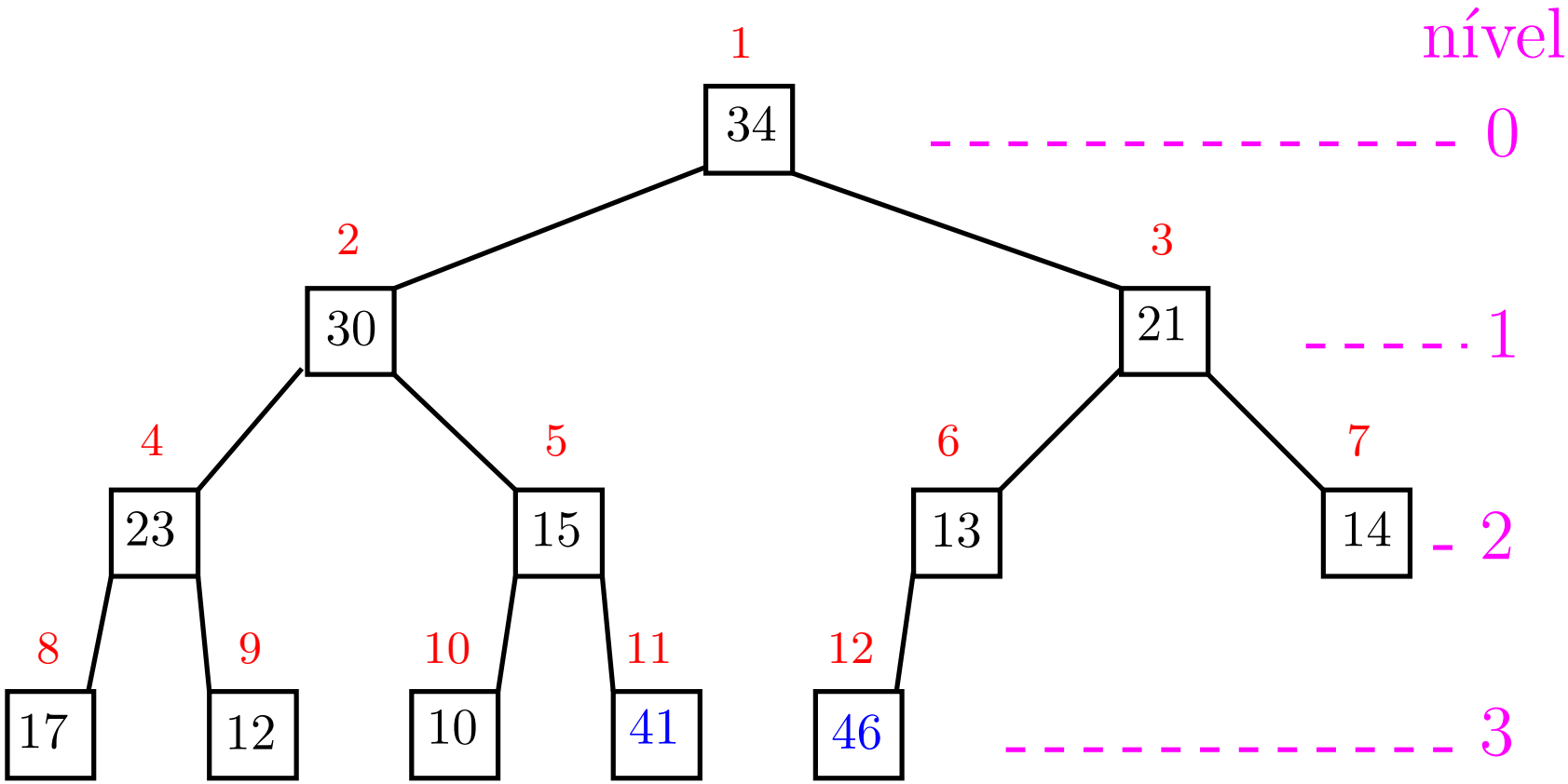
1	2	3	4	5	6	7	8	9	10	11	12
34	30	13	23	15	21	14	17	12	10	41	46

Heapsort



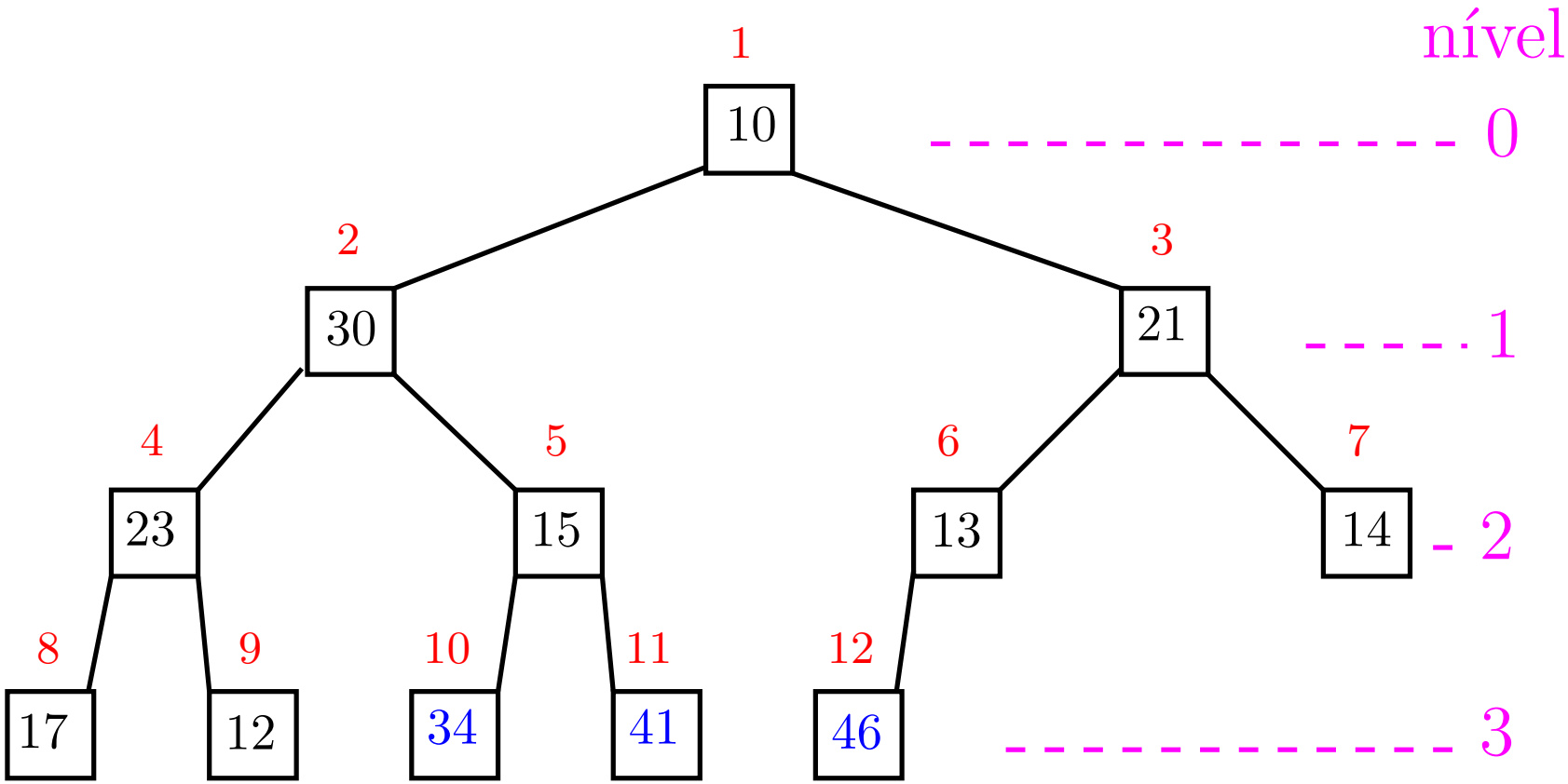
1	2	3	4	5	6	7	8	9	10	11	12
34	30	21	23	15	13	14	17	12	10	41	46

Heapsort



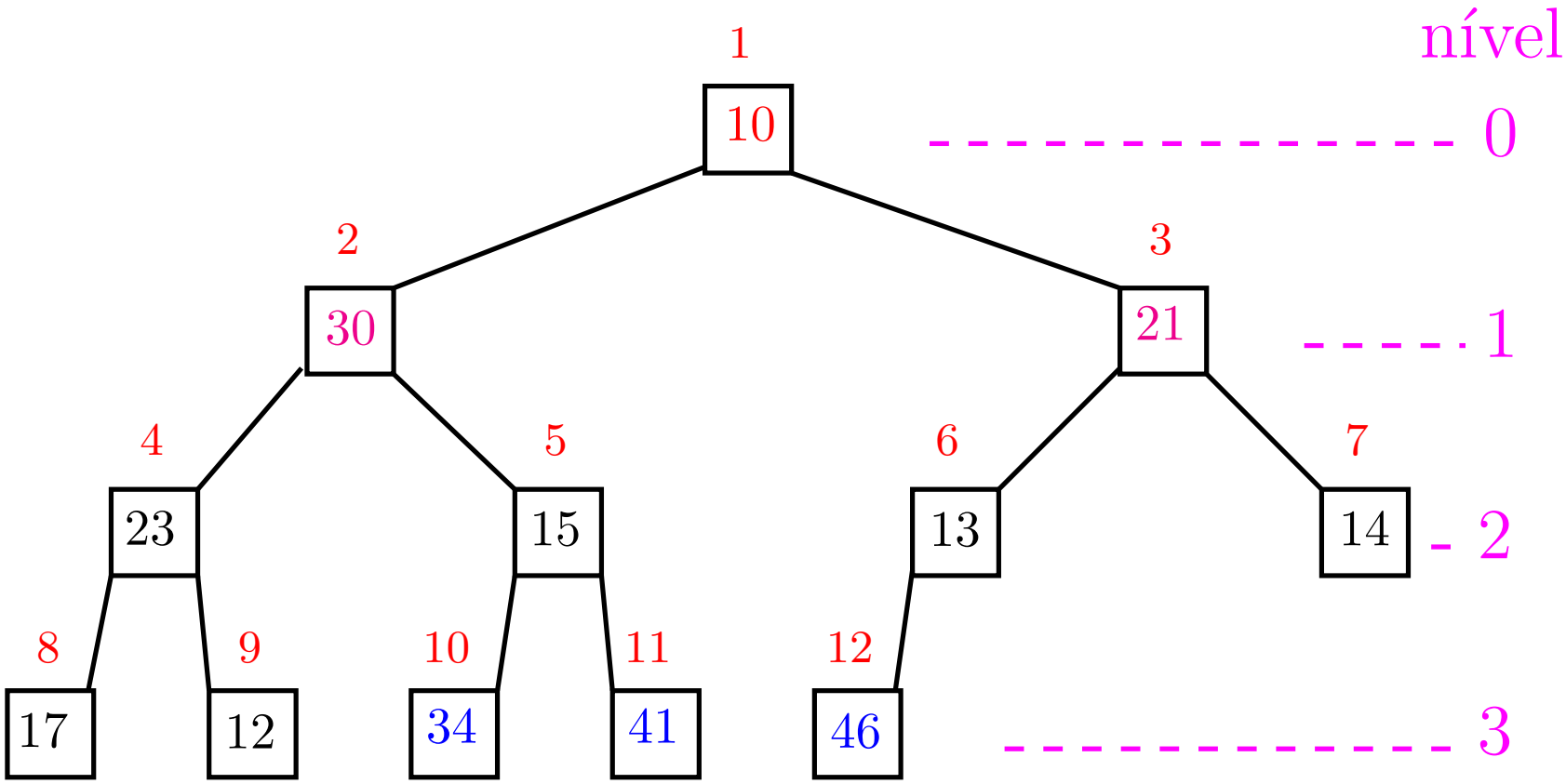
1	2	3	4	5	6	7	8	9	10	11	12
34	30	21	23	15	13	14	17	12	10	41	46

Heapsort



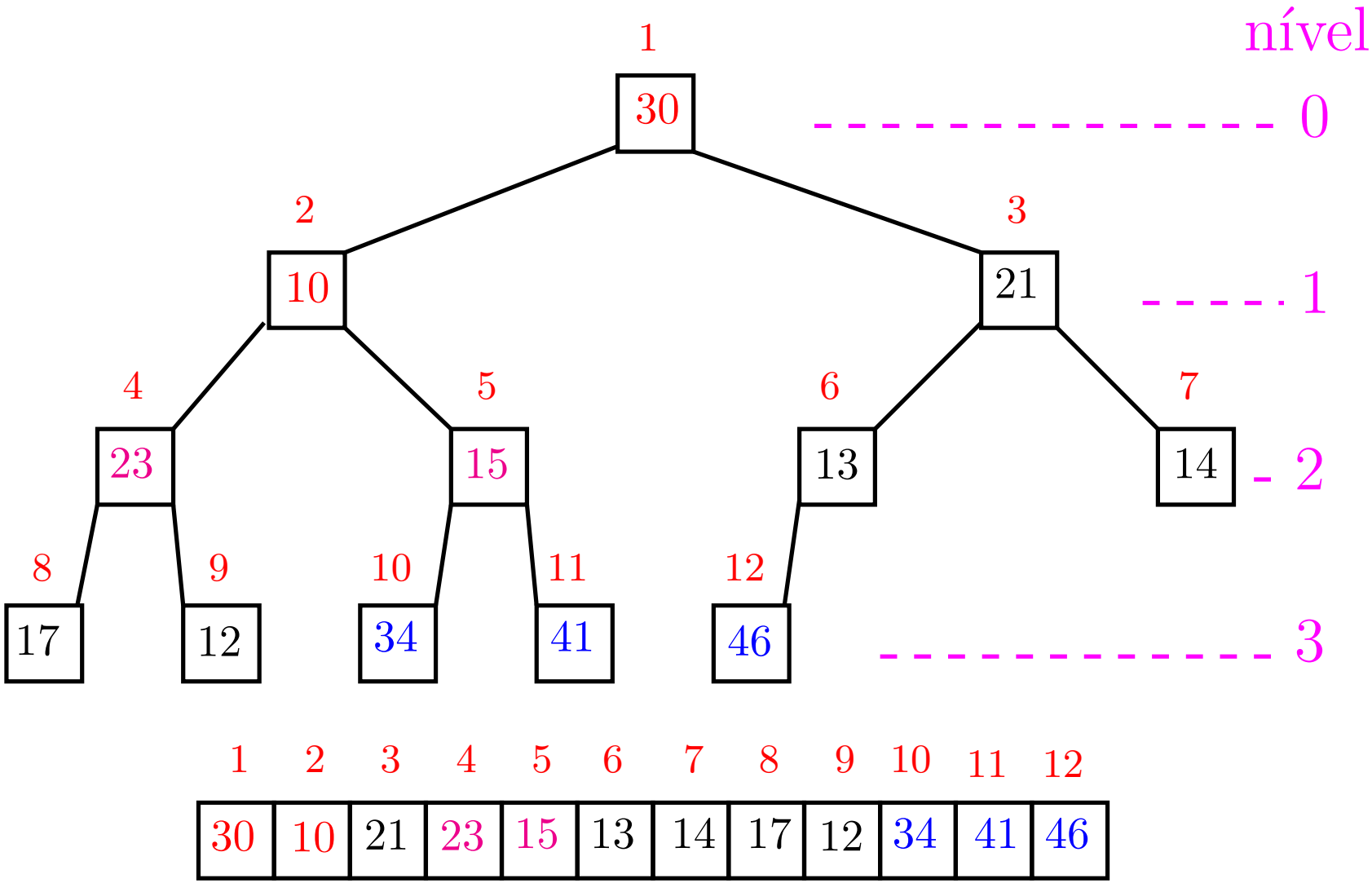
1	2	3	4	5	6	7	8	9	10	11	12
10	30	21	23	15	13	14	17	12	34	41	46

Heapsort

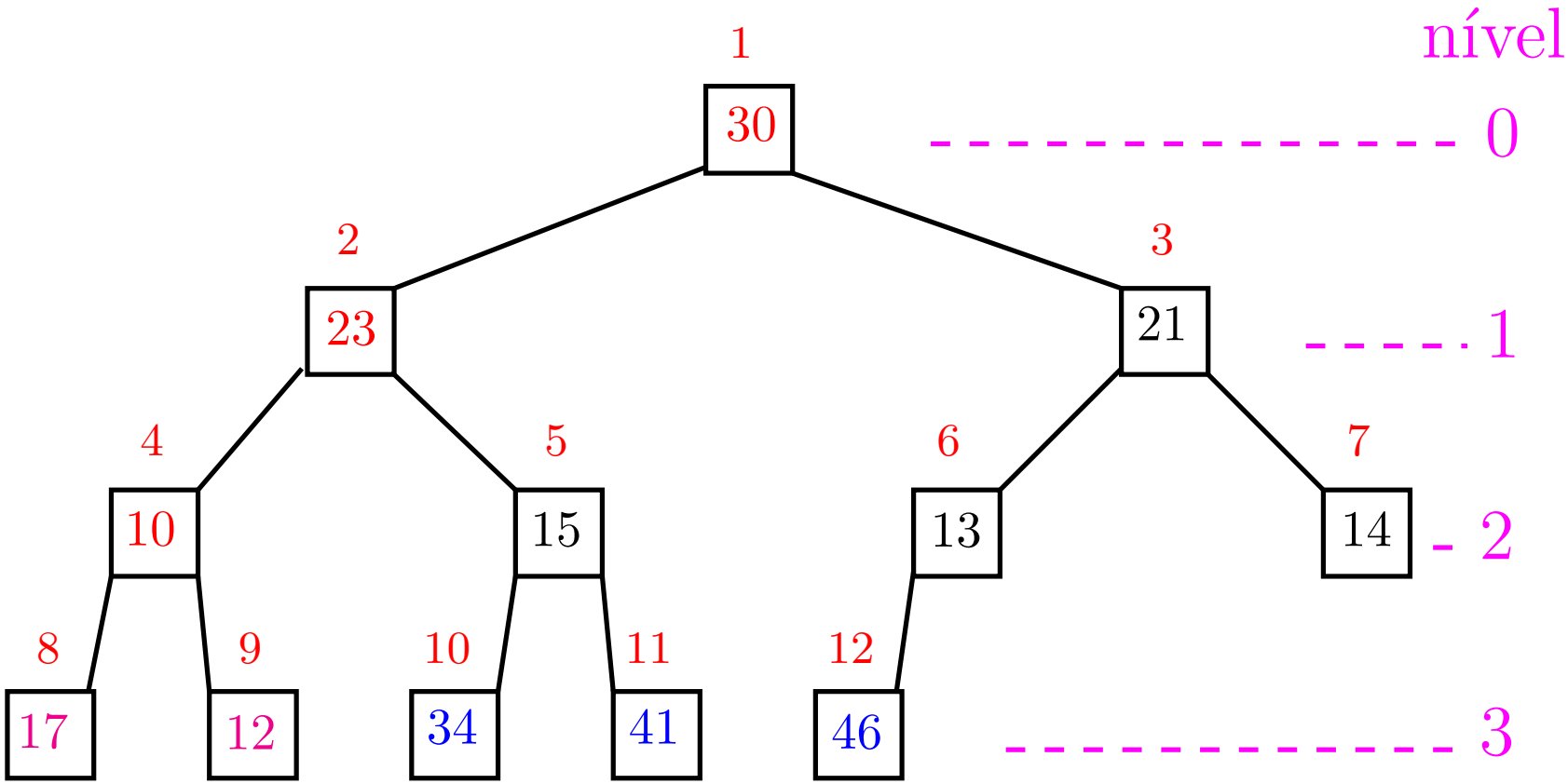


1	2	3	4	5	6	7	8	9	10	11	12
10	30	21	23	15	13	14	17	12	34	41	46

Heapsort

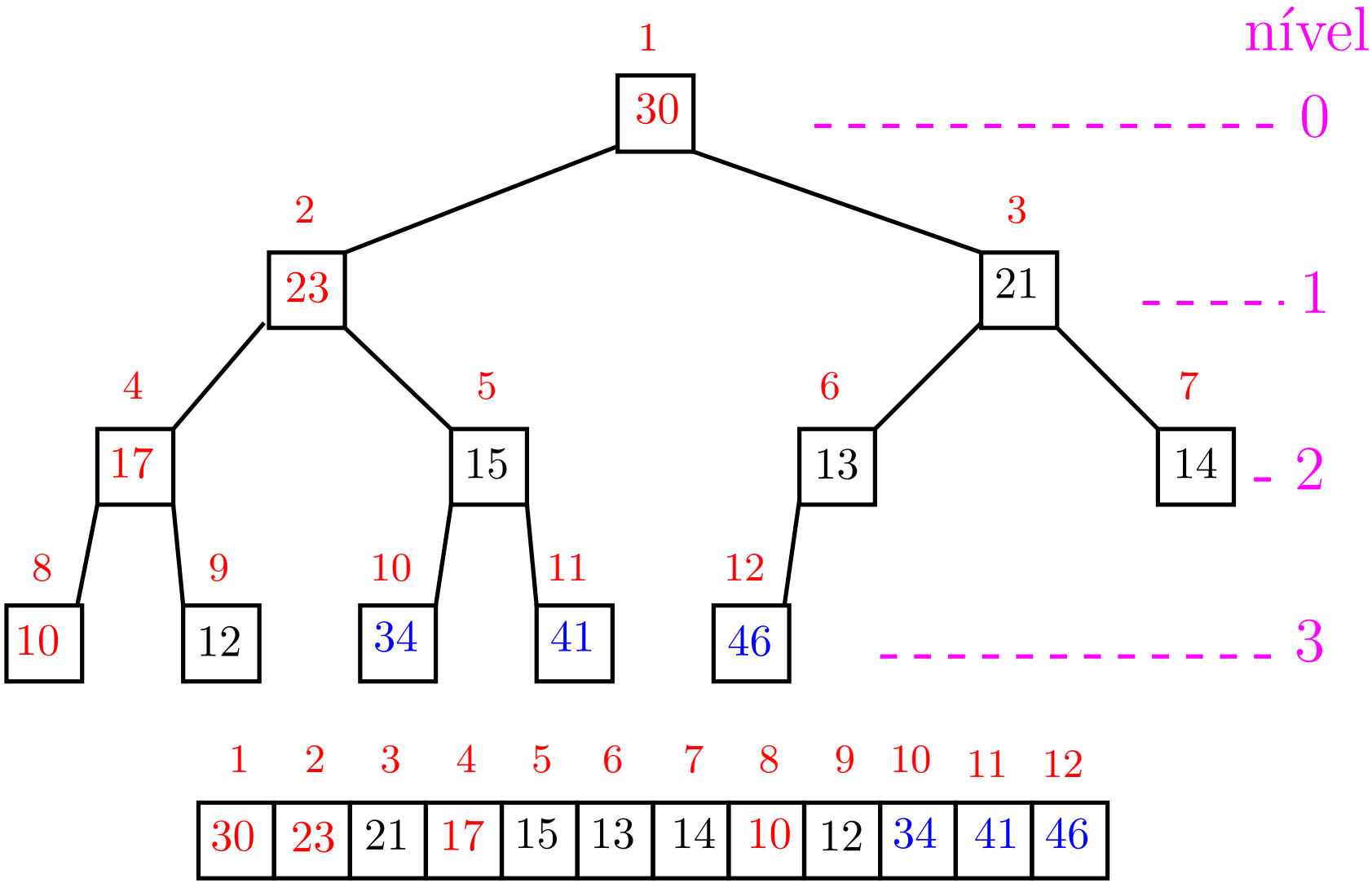

nível
1
30
0
2
10
3
21
1
4
23
5
15
6
13
7
14
2
8
17
9
12
10
34
11
41
12
46
3
1 2 3 4 5 6 7 8 9 10 11 12
30 10 21 23 15 13 14 17 12 34 41 46

Heapsort

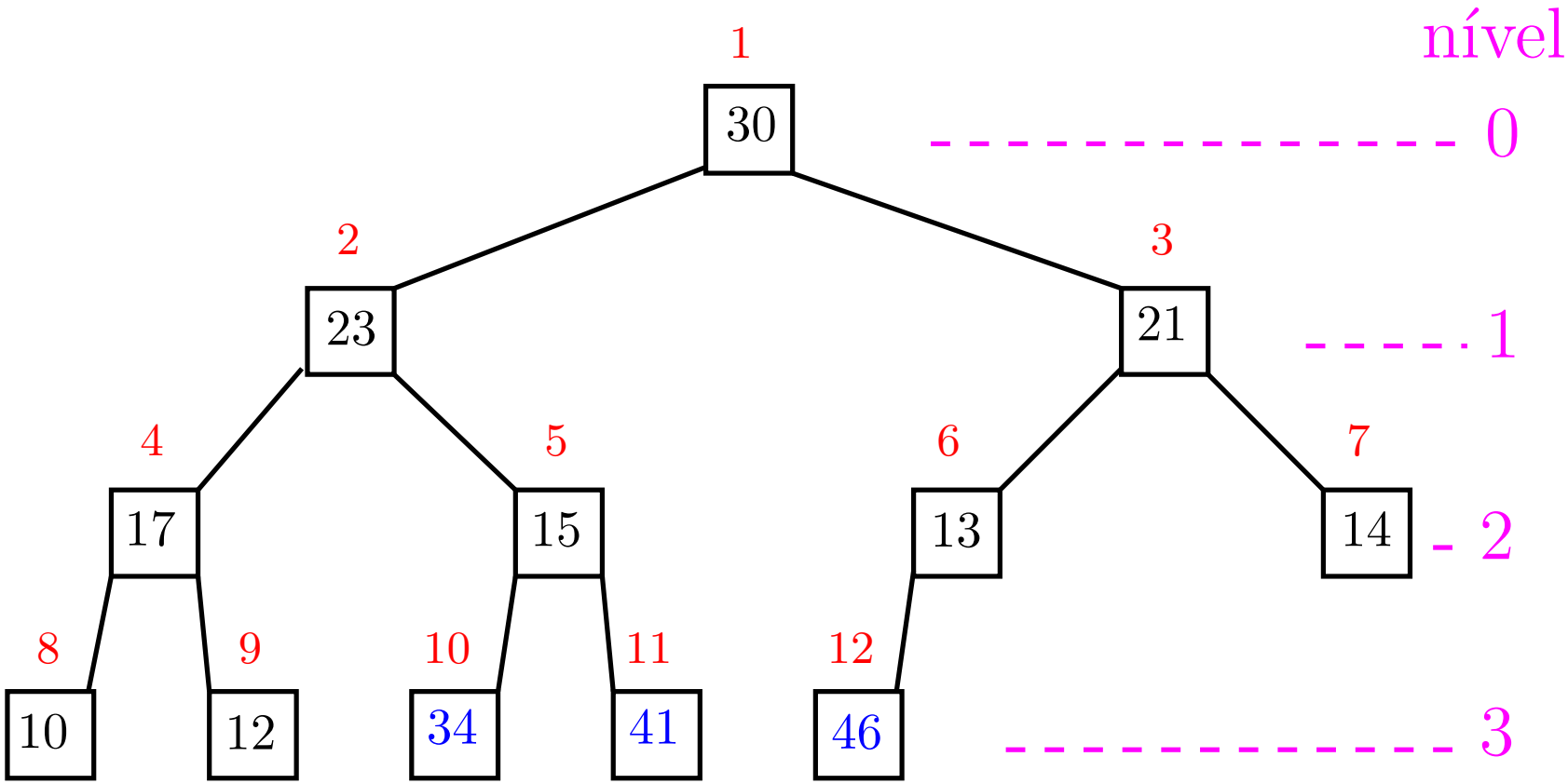


1	2	3	4	5	6	7	8	9	10	11	12
30	23	21	10	15	13	14	17	12	34	41	46

Heapsort

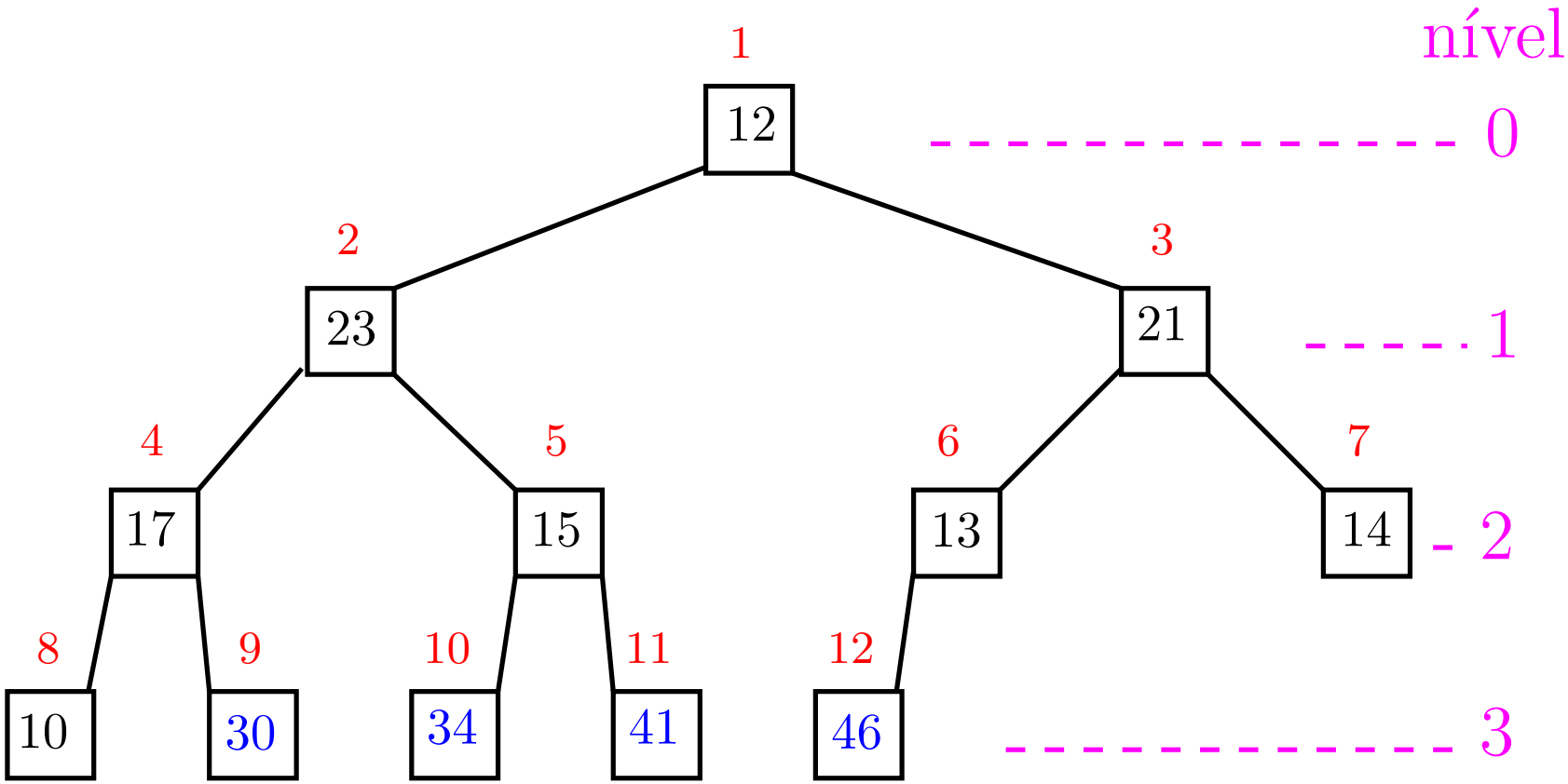

nível
1
30
0
2
23
3
21
1
4
17
5
15
6
13
7
14
2
8
10
9
12
10
34
11
41
12
46
3
1 2 3 4 5 6 7 8 9 10 11 12
30 23 21 17 15 13 14 10 12 34 41 46

Heapsort



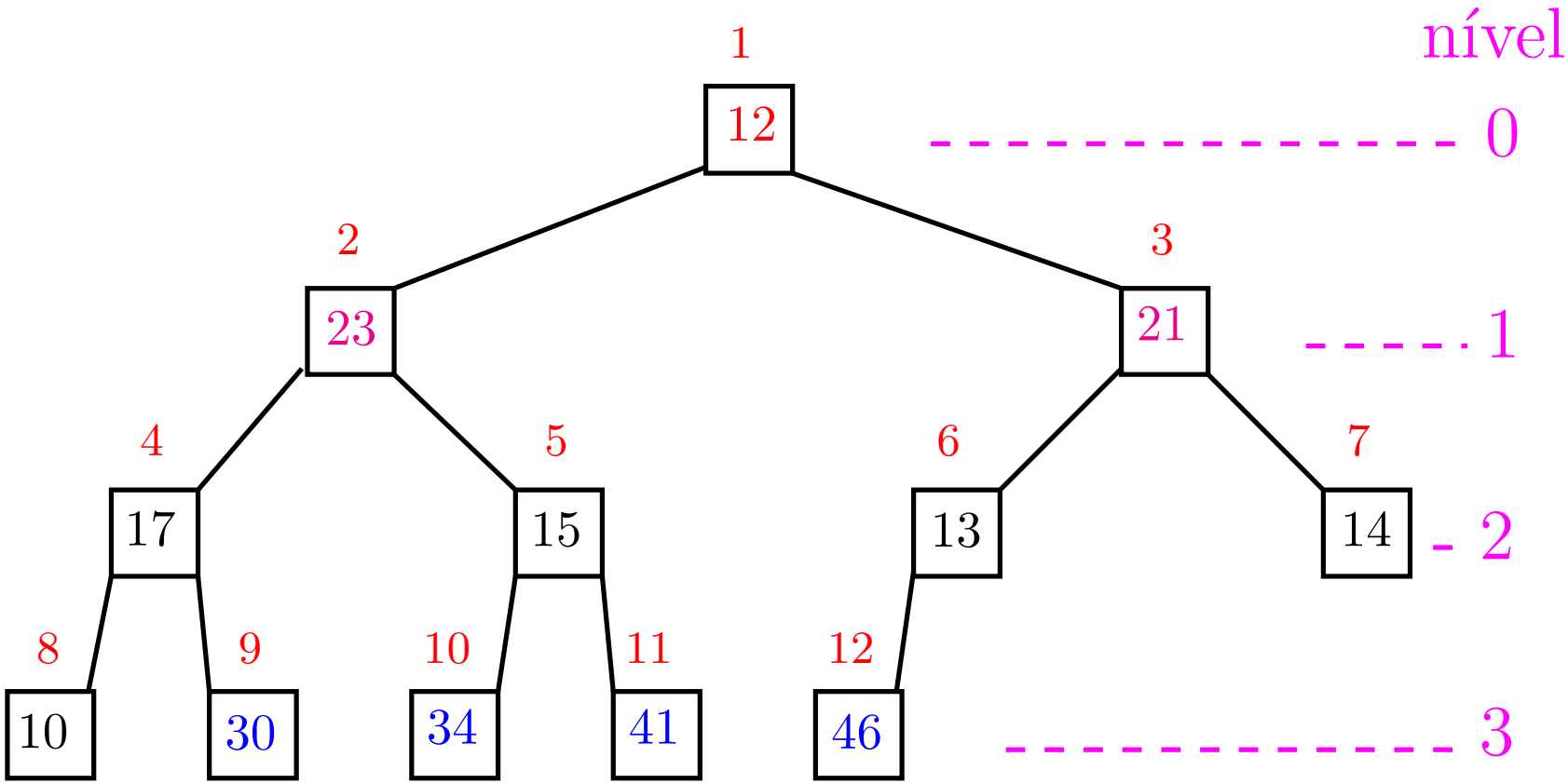
1	2	3	4	5	6	7	8	9	10	11	12
30	23	21	17	15	13	14	10	12	34	41	46

Heapsort



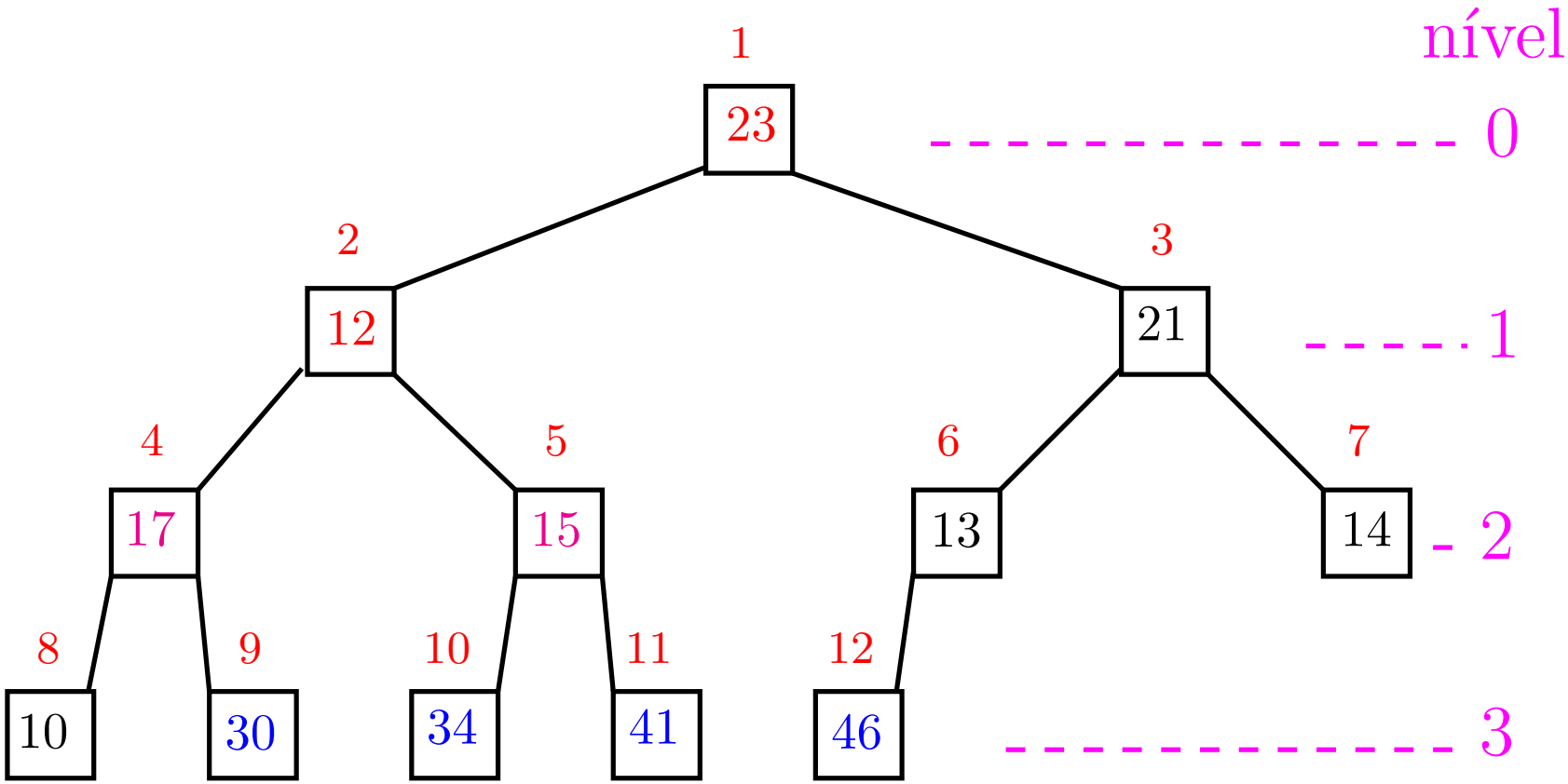
1	2	3	4	5	6	7	8	9	10	11	12
12	23	21	17	15	13	14	10	30	34	41	46

Heapsort



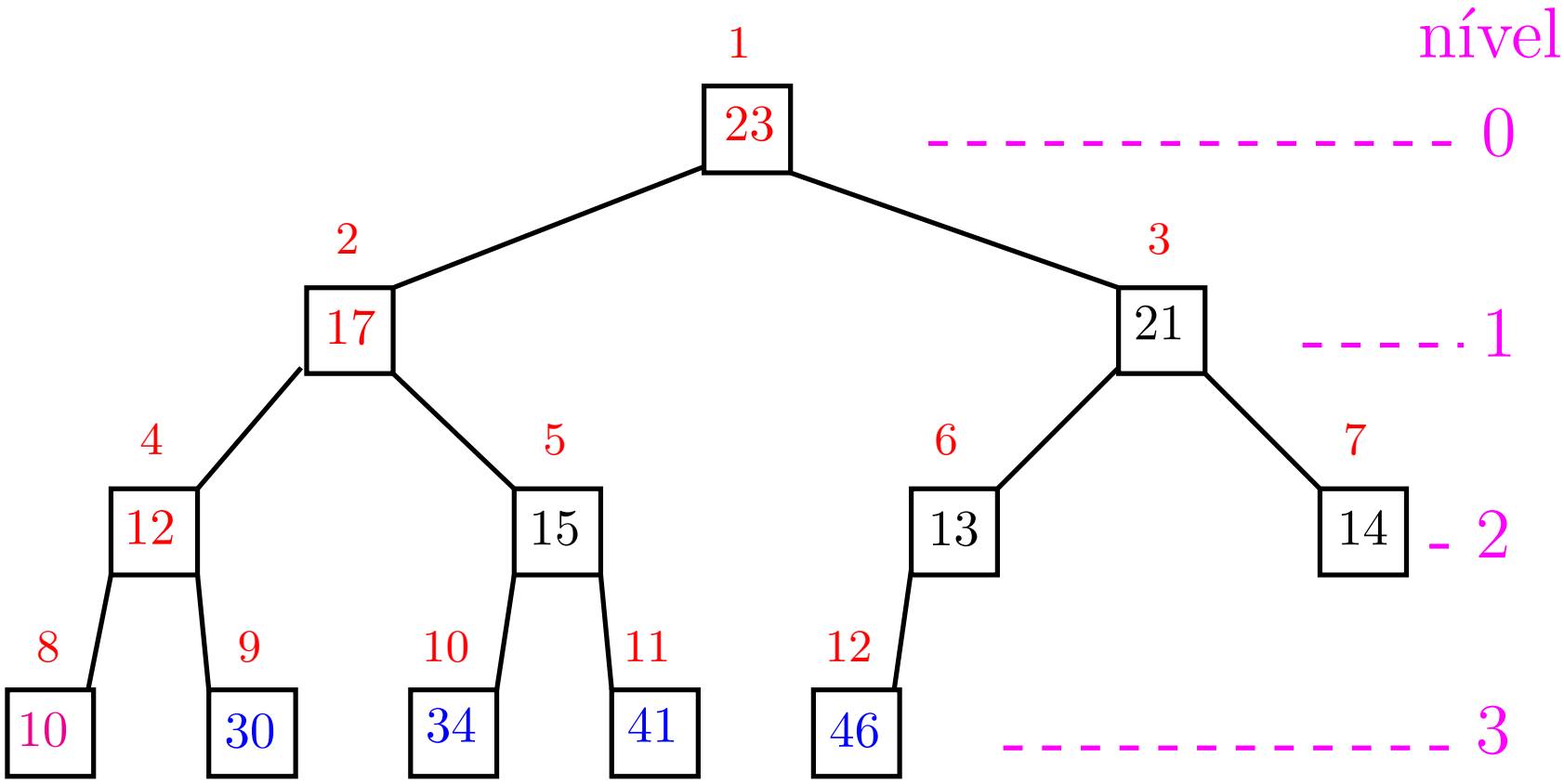
1	2	3	4	5	6	7	8	9	10	11	12
12	23	21	17	15	13	14	10	30	34	41	46

Heapsort



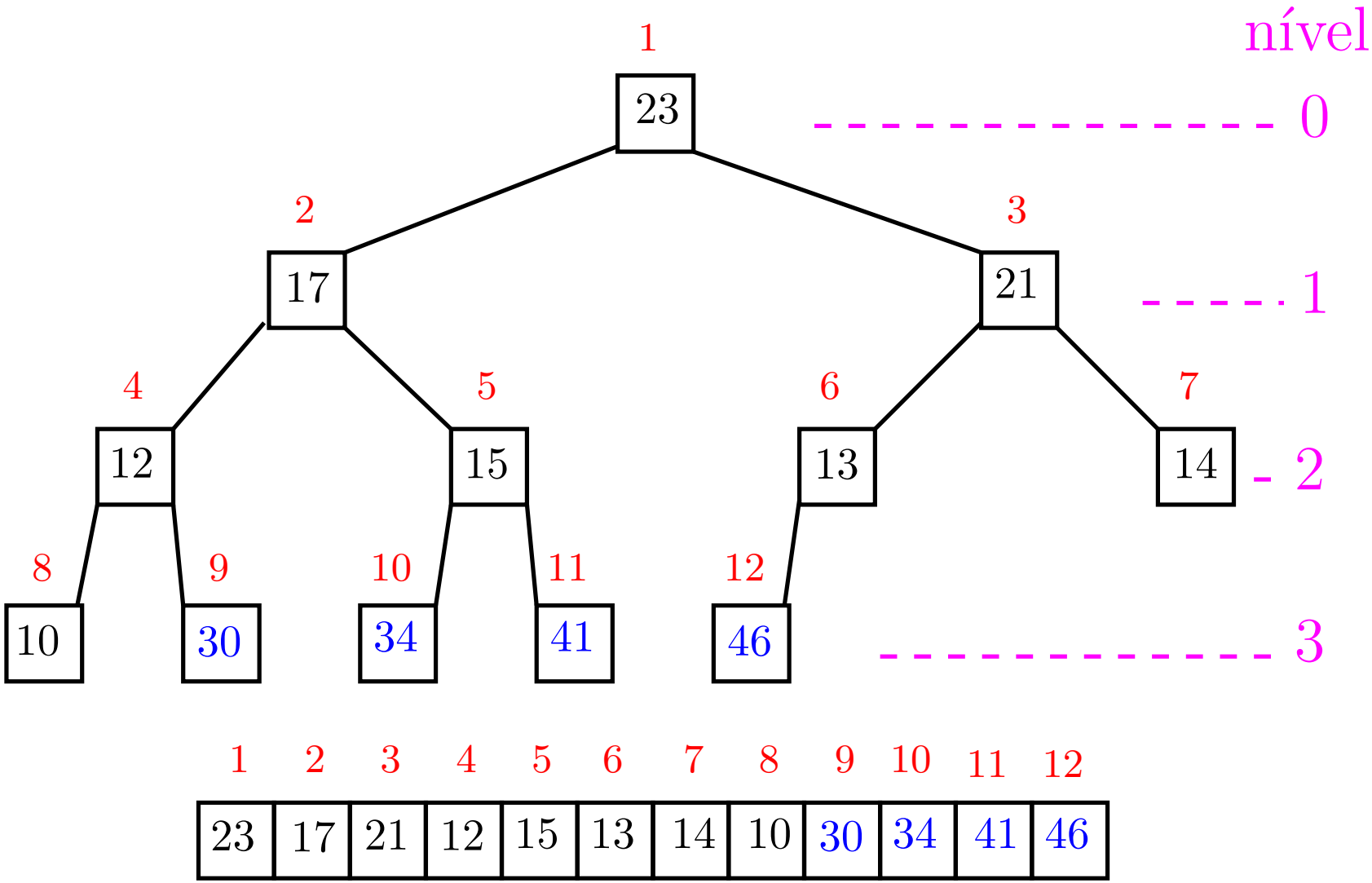
1	2	3	4	5	6	7	8	9	10	11	12
23	12	21	17	15	13	14	10	30	34	41	46

Heapsort

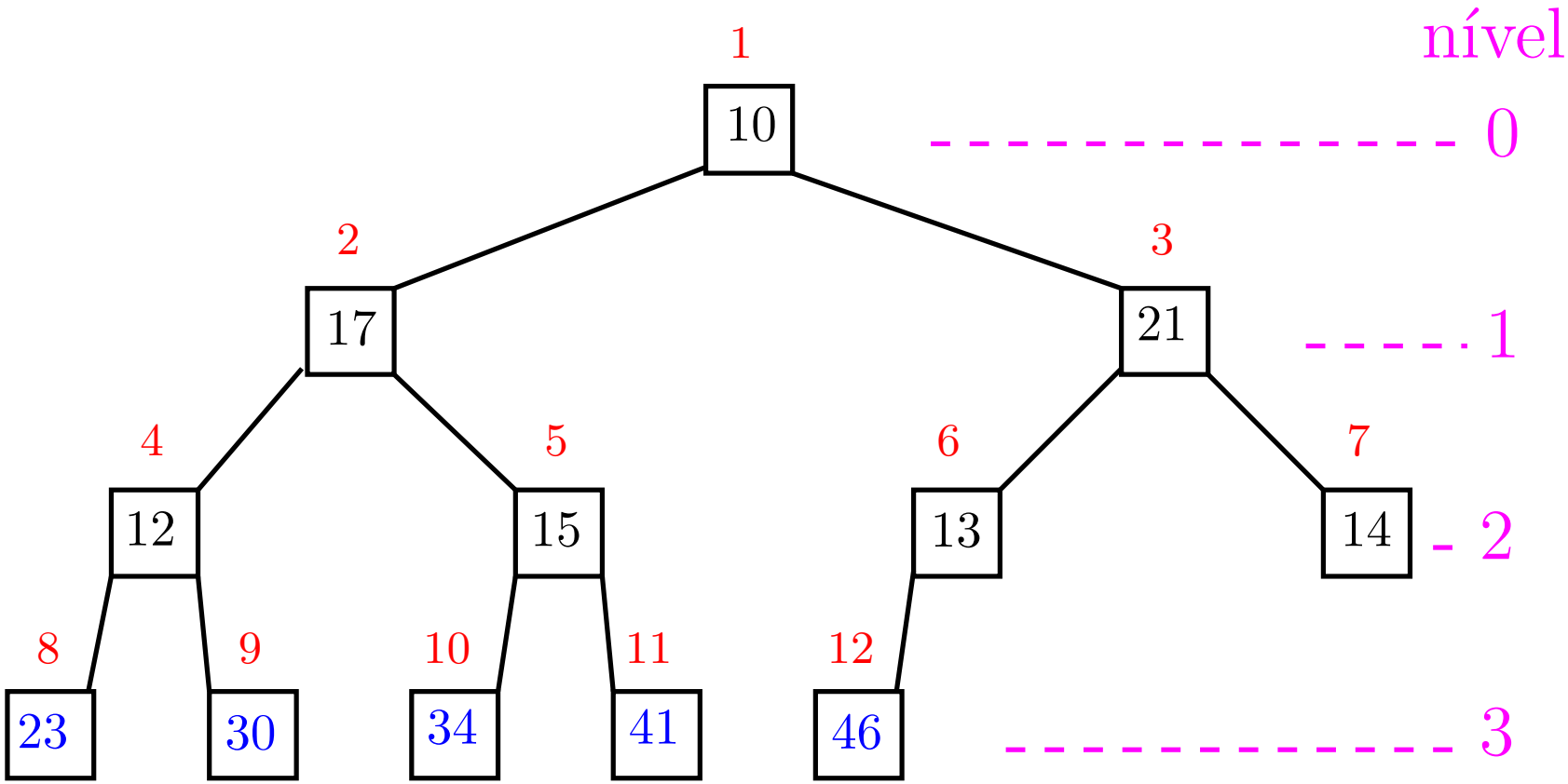


1	2	3	4	5	6	7	8	9	10	11	12
23	17	21	12	15	13	14	10	30	34	41	46

Heapsort

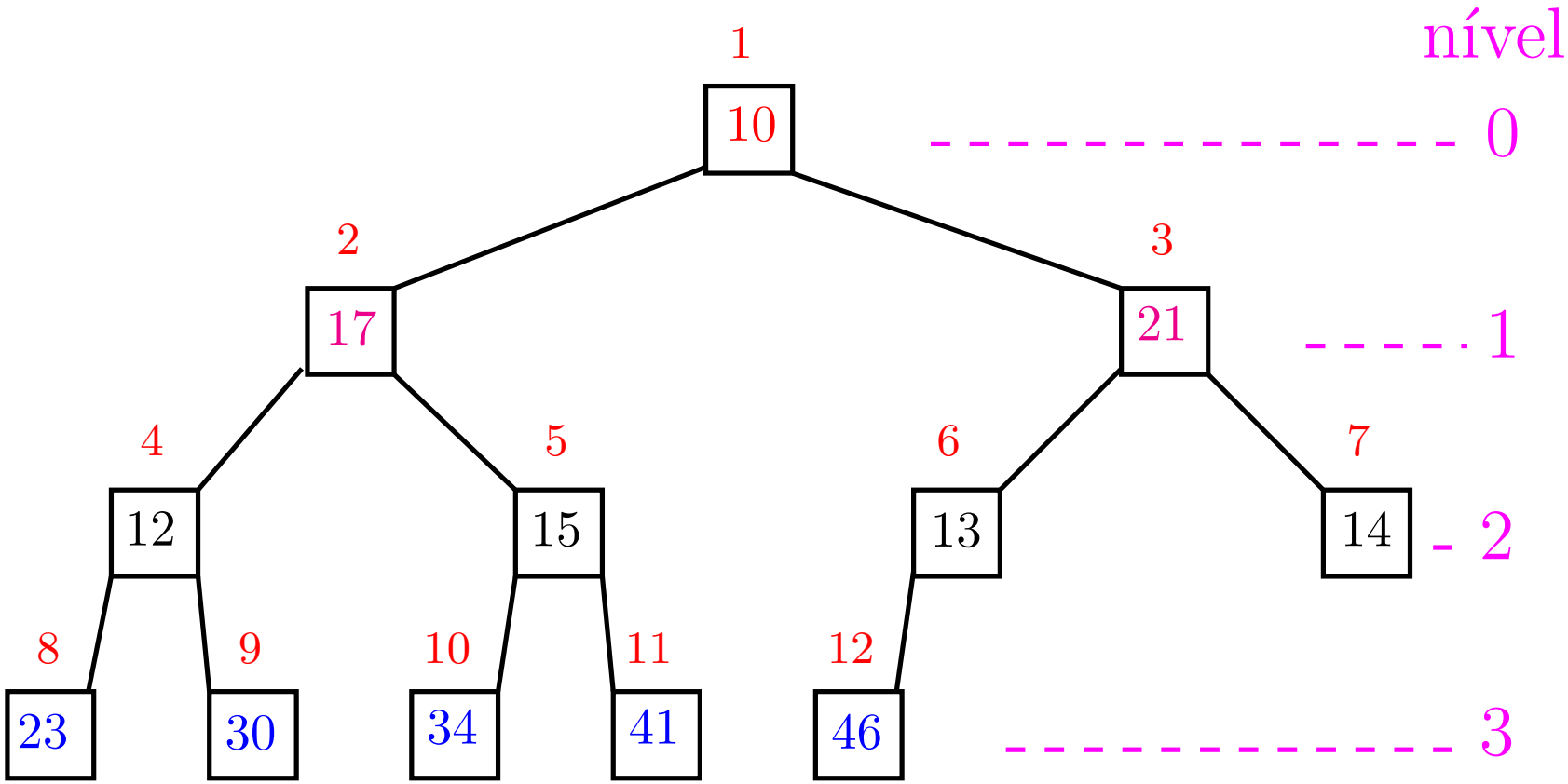

nível
1
23
0
2
17
3
21
1
4
12
5
15
6
13
7
14
2
8
10
9
30
10
34
11
41
12
46
3
1 2 3 4 5 6 7 8 9 10 11 12
23 17 21 12 15 13 14 10 30 34 41 46

Heapsort



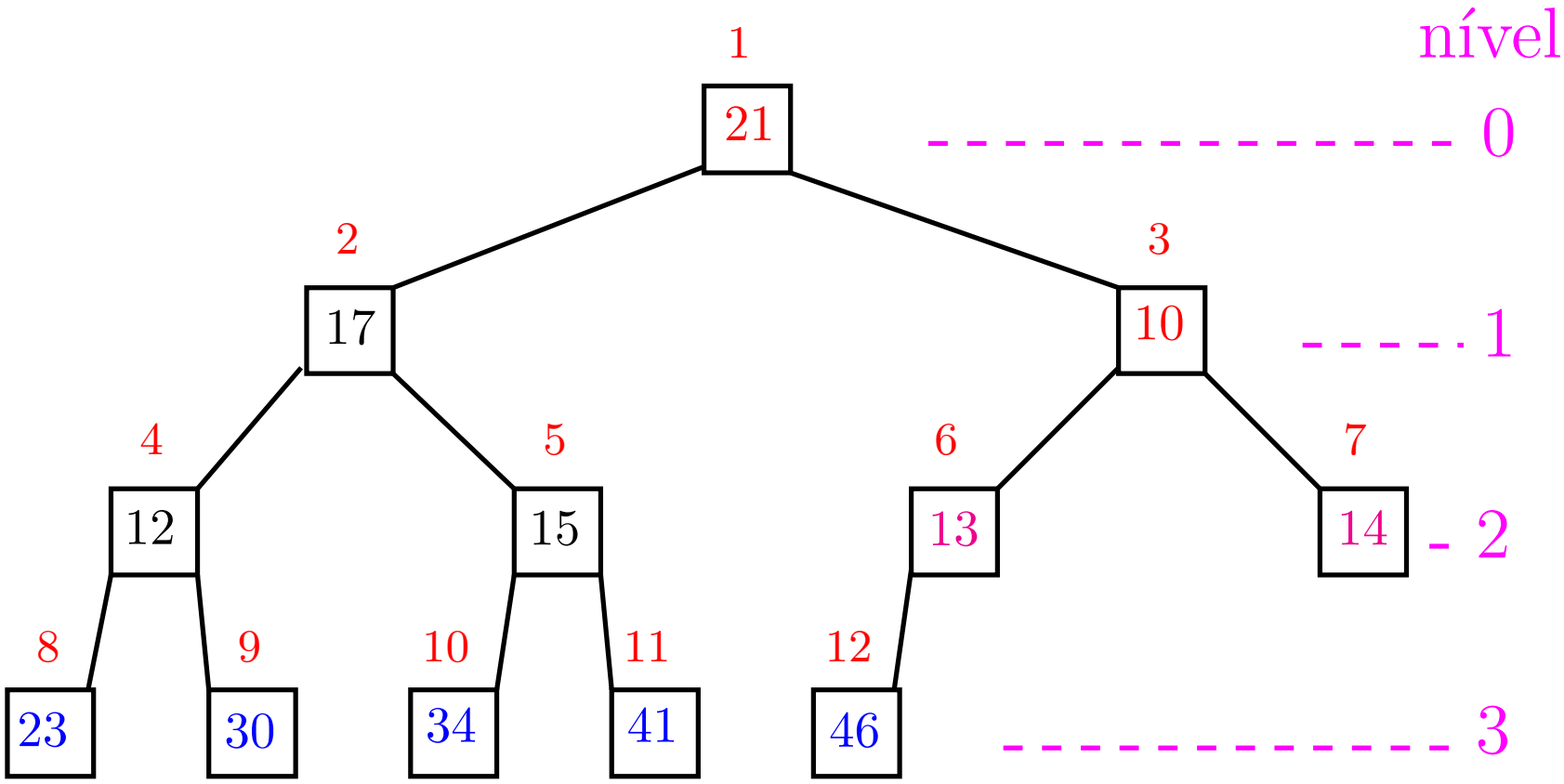
1	2	3	4	5	6	7	8	9	10	11	12
10	17	21	12	15	13	14	23	30	34	41	46

Heapsort



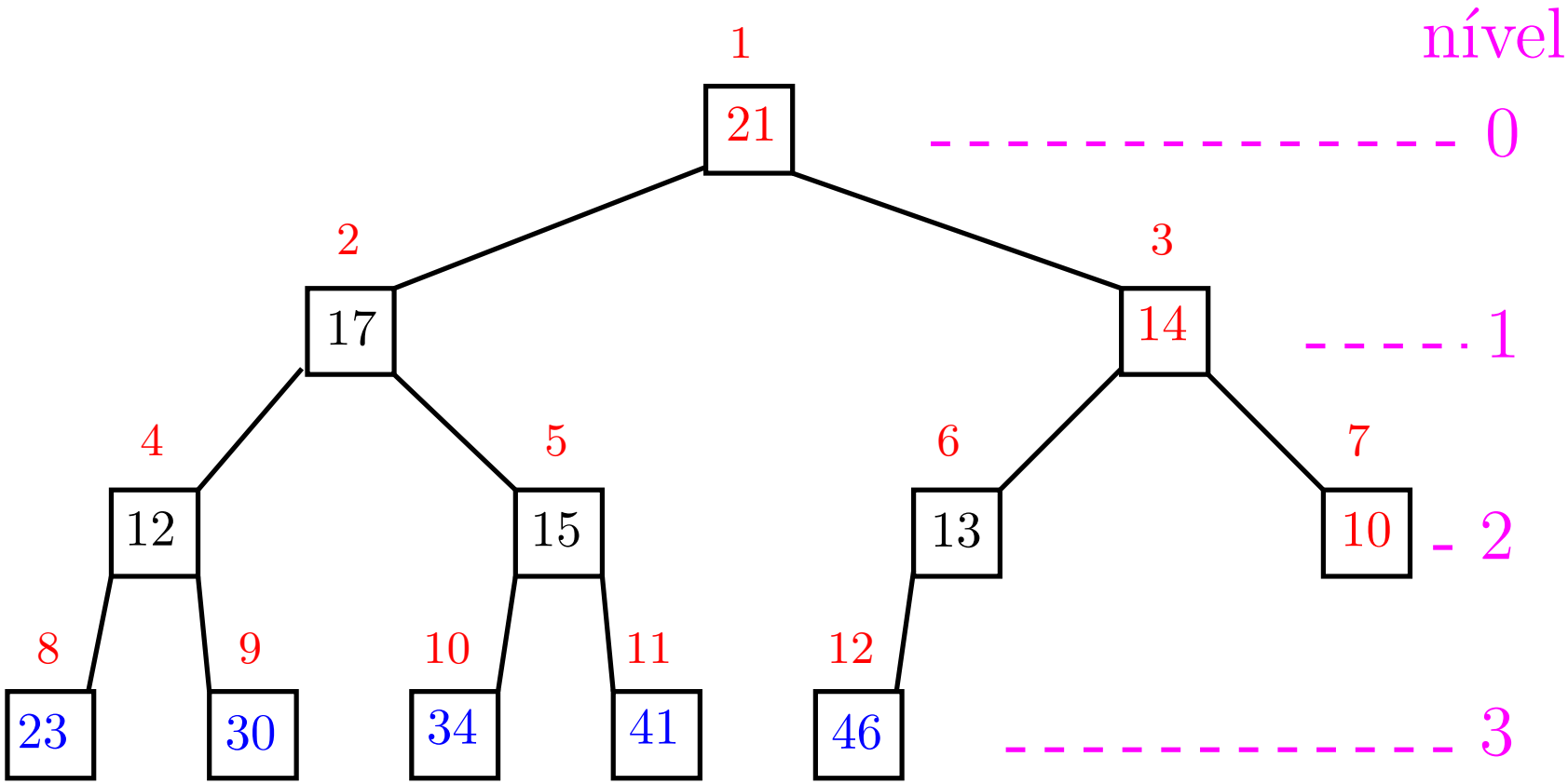
1	2	3	4	5	6	7	8	9	10	11	12
10	17	21	12	15	13	14	23	30	34	41	46

Heapsort



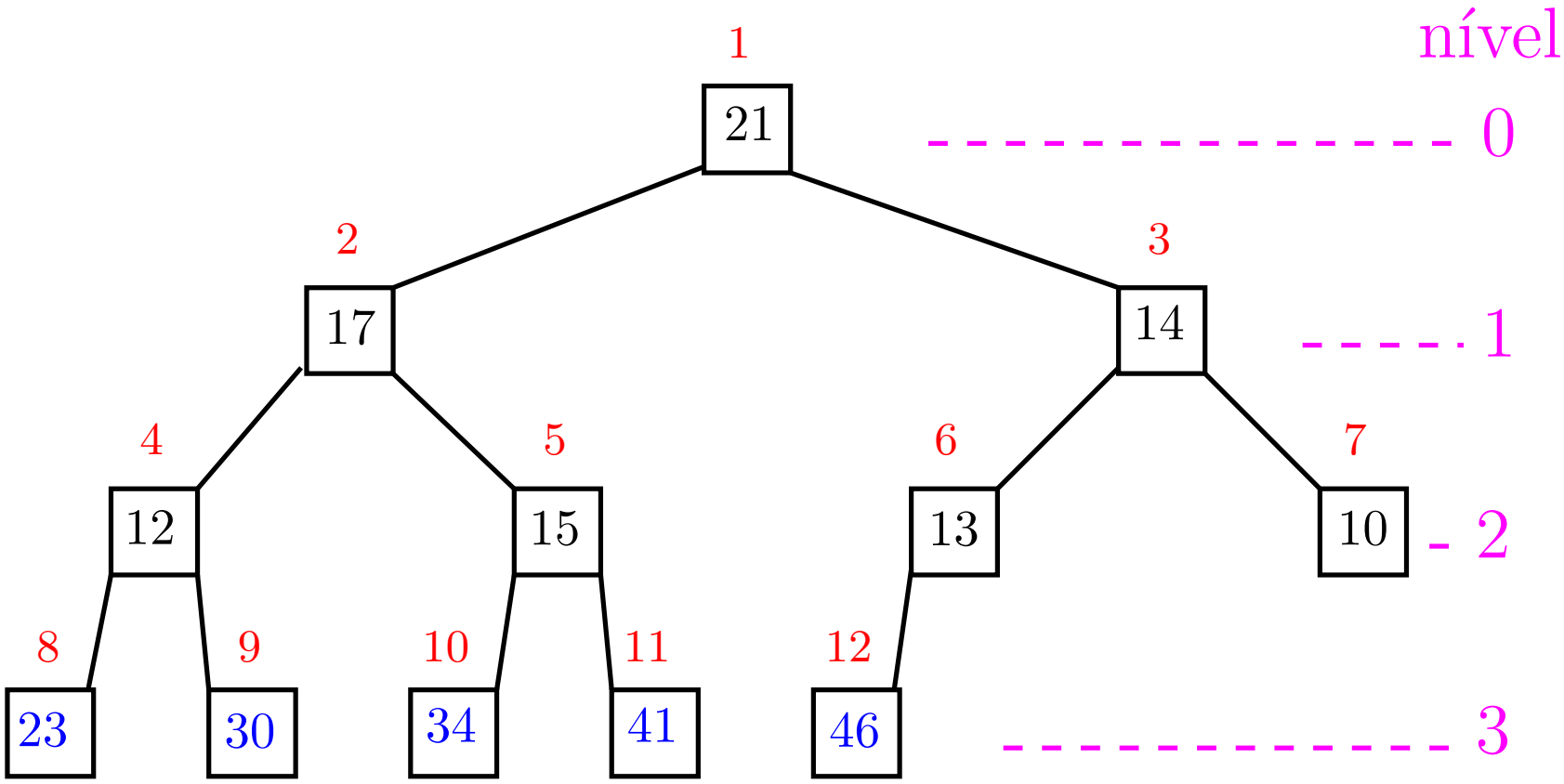
1	2	3	4	5	6	7	8	9	10	11	12
21	17	10	12	15	13	14	23	30	34	41	46

Heapsort



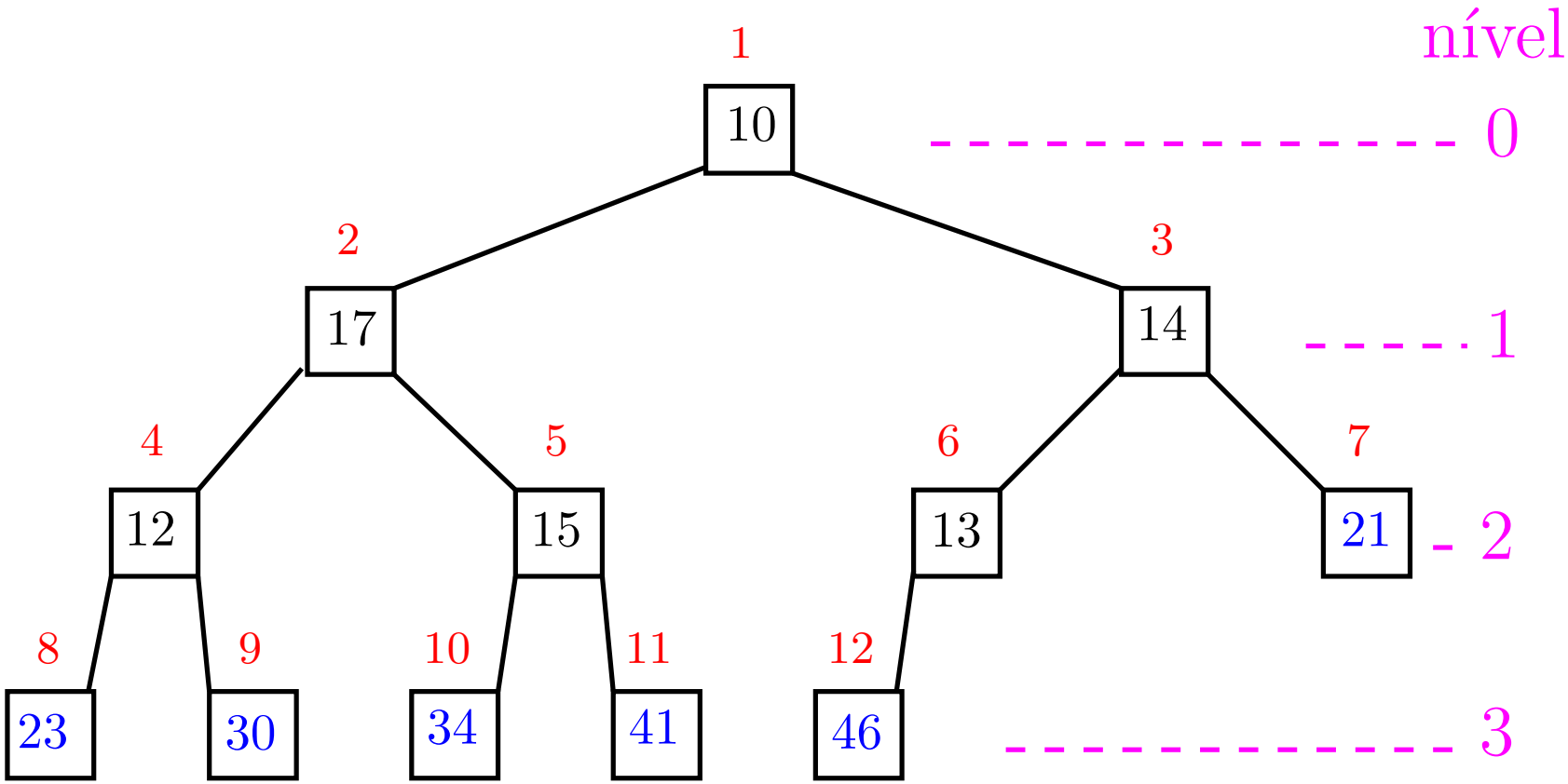
1	2	3	4	5	6	7	8	9	10	11	12
21	17	14	12	15	13	10	23	30	34	41	46

Heapsort



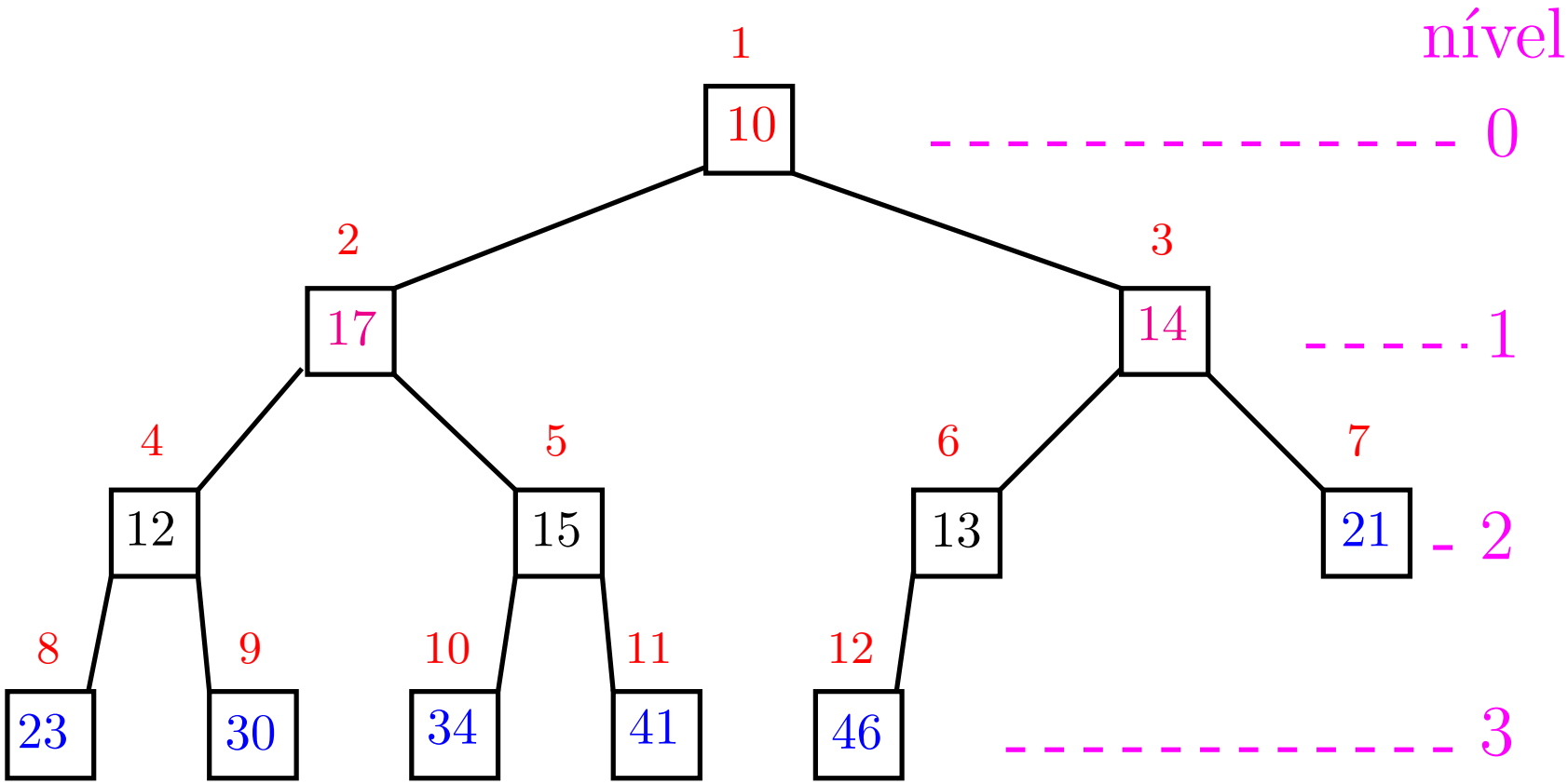
1	2	3	4	5	6	7	8	9	10	11	12
21	17	14	12	15	13	10	23	30	34	41	46

Heapsort



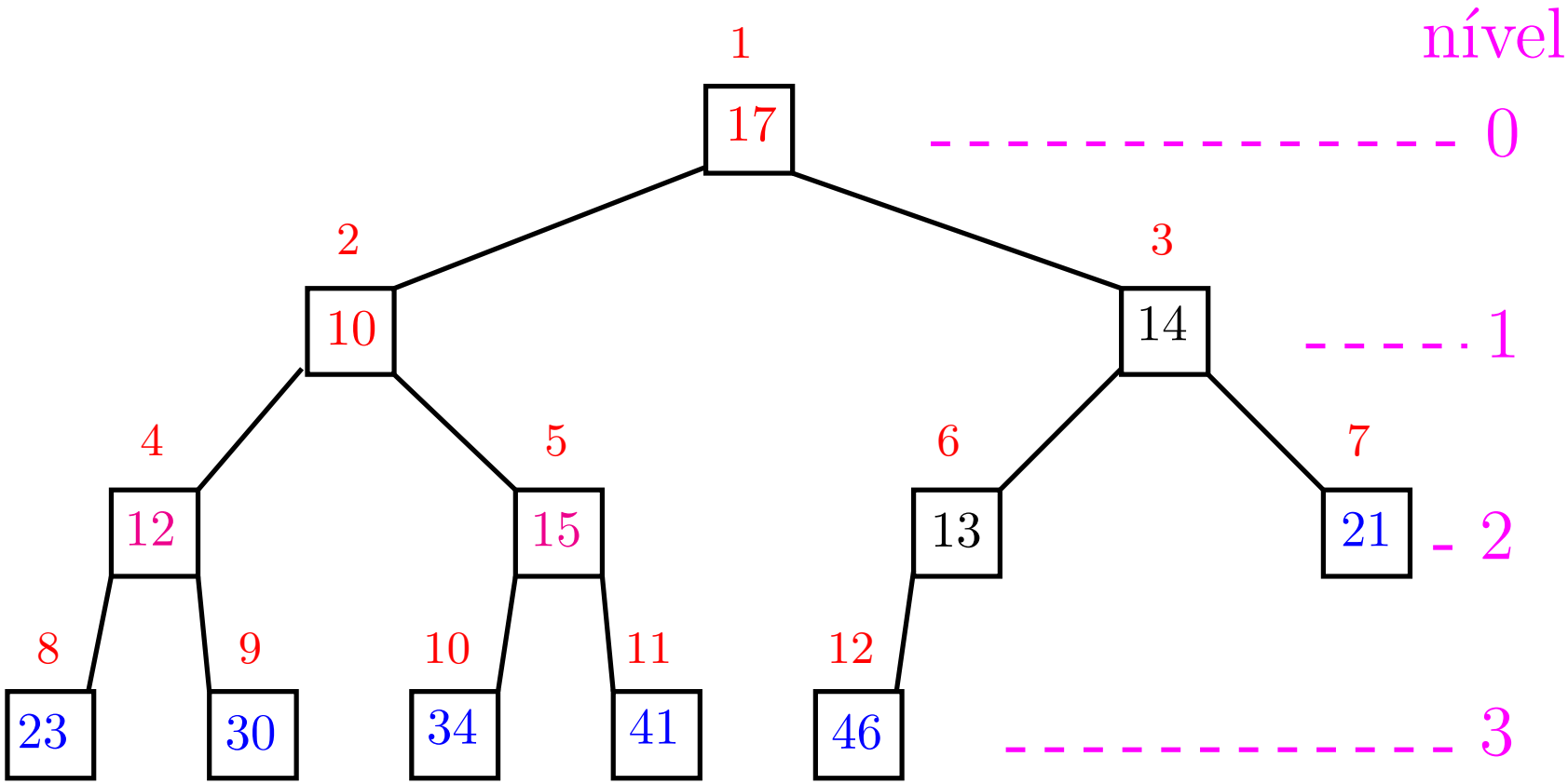
1	2	3	4	5	6	7	8	9	10	11	12
10	17	14	12	15	13	21	23	30	34	41	46

Heapsort



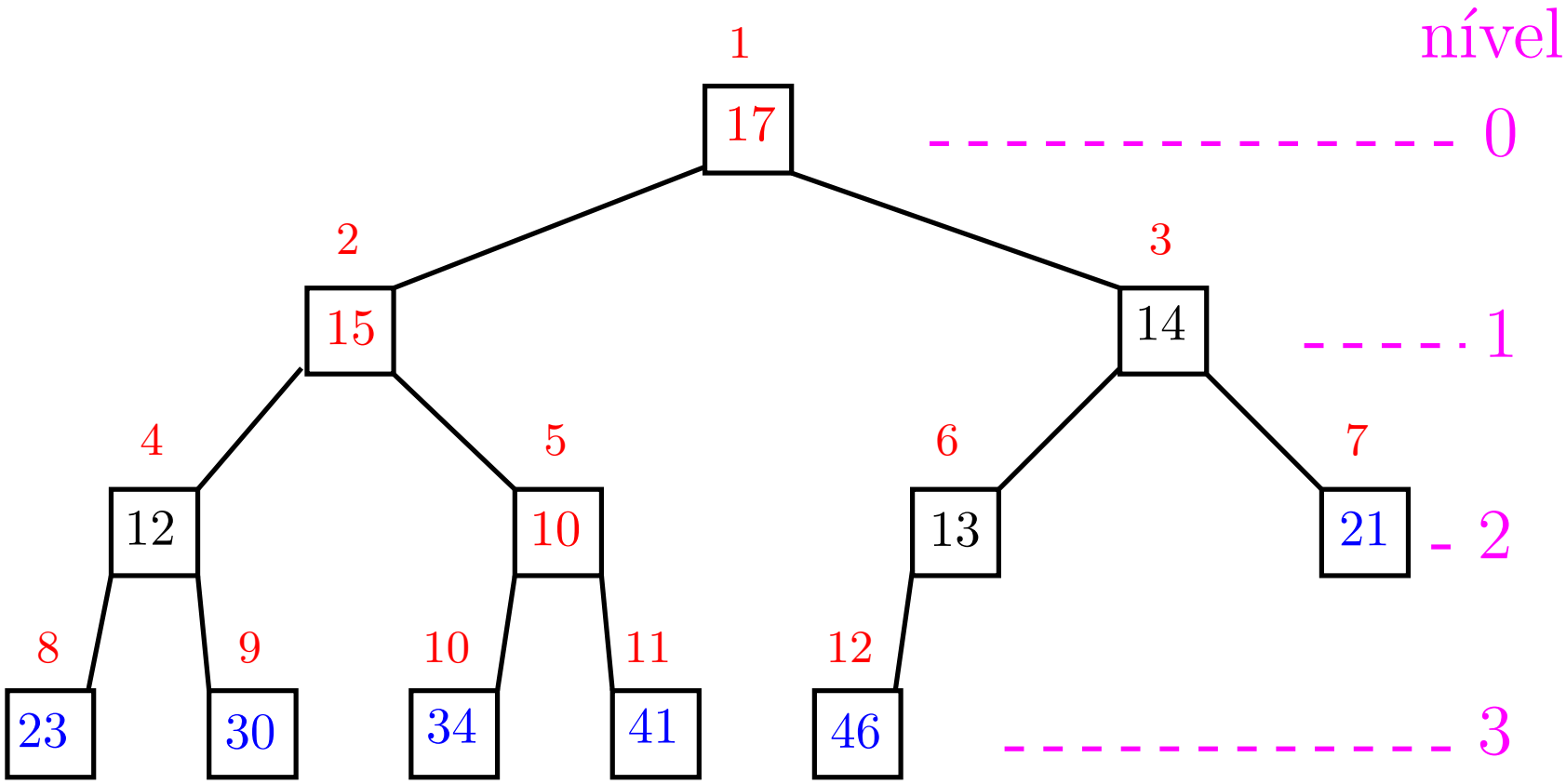
1	2	3	4	5	6	7	8	9	10	11	12
10	17	14	12	15	13	21	23	30	34	41	46

Heapsort



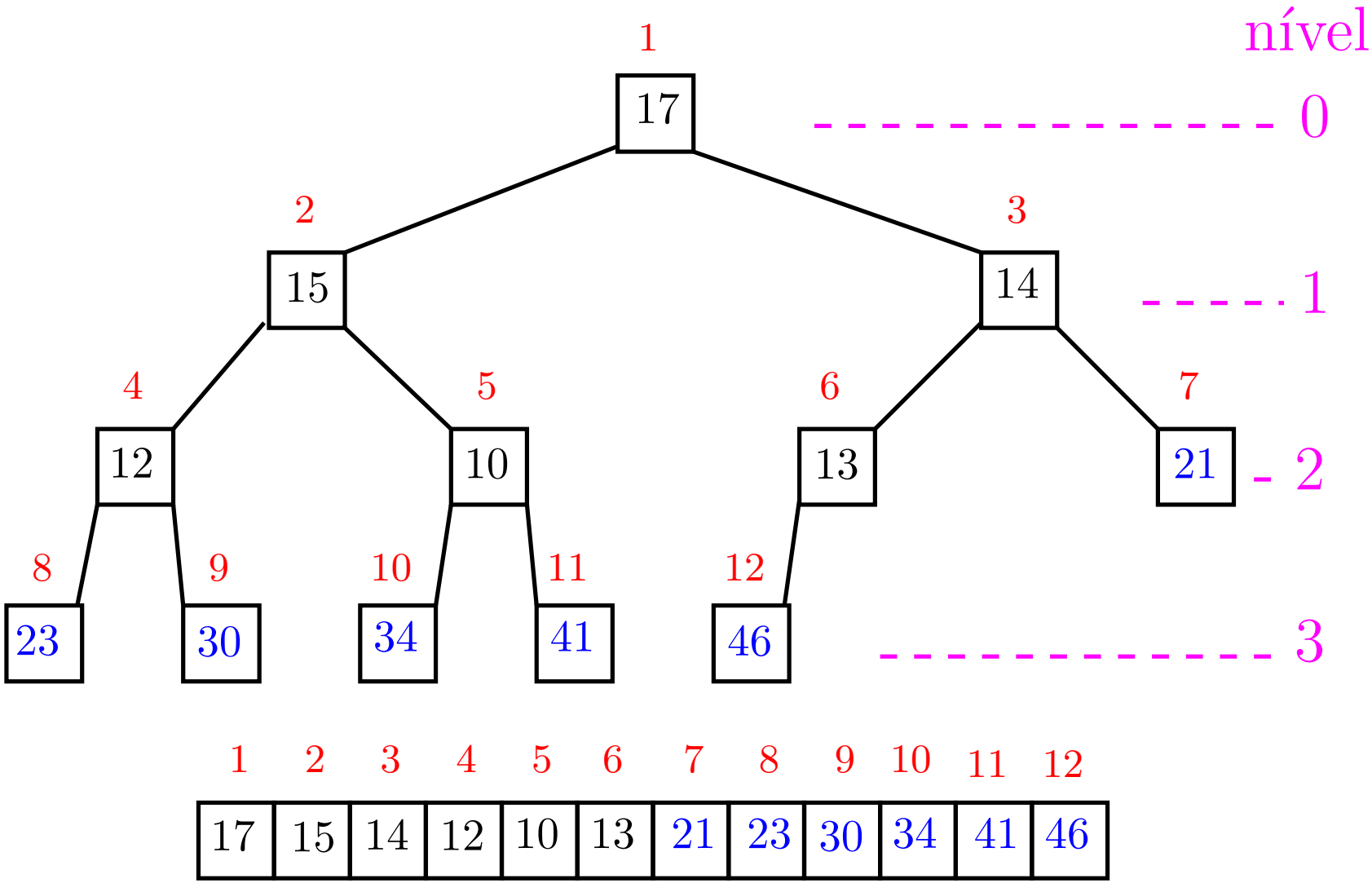
1	2	3	4	5	6	7	8	9	10	11	12
17	10	14	12	15	13	21	23	30	34	41	46

Heapsort

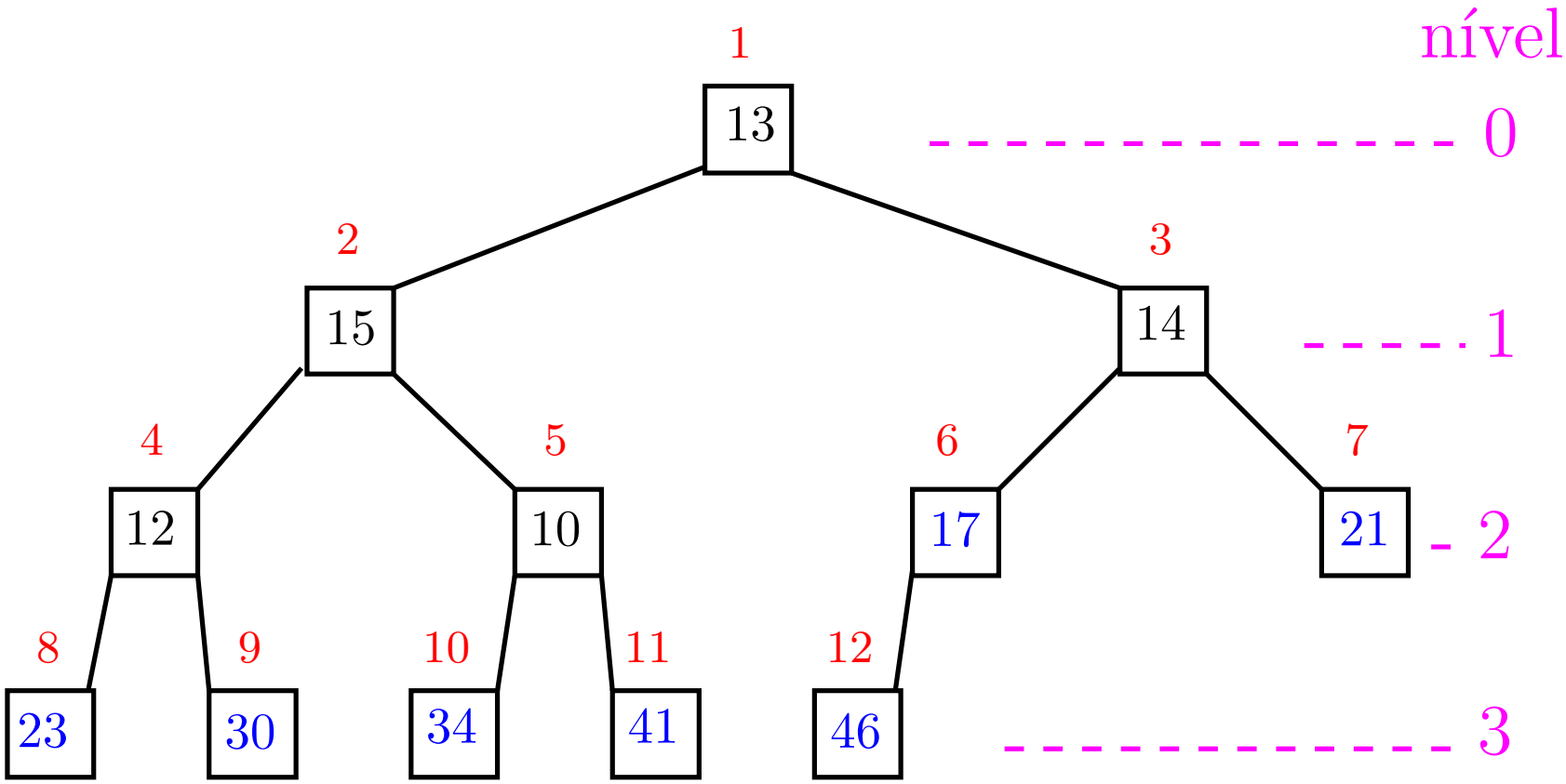


1	2	3	4	5	6	7	8	9	10	11	12
17	15	14	12	10	13	21	23	30	34	41	46

Heapsort

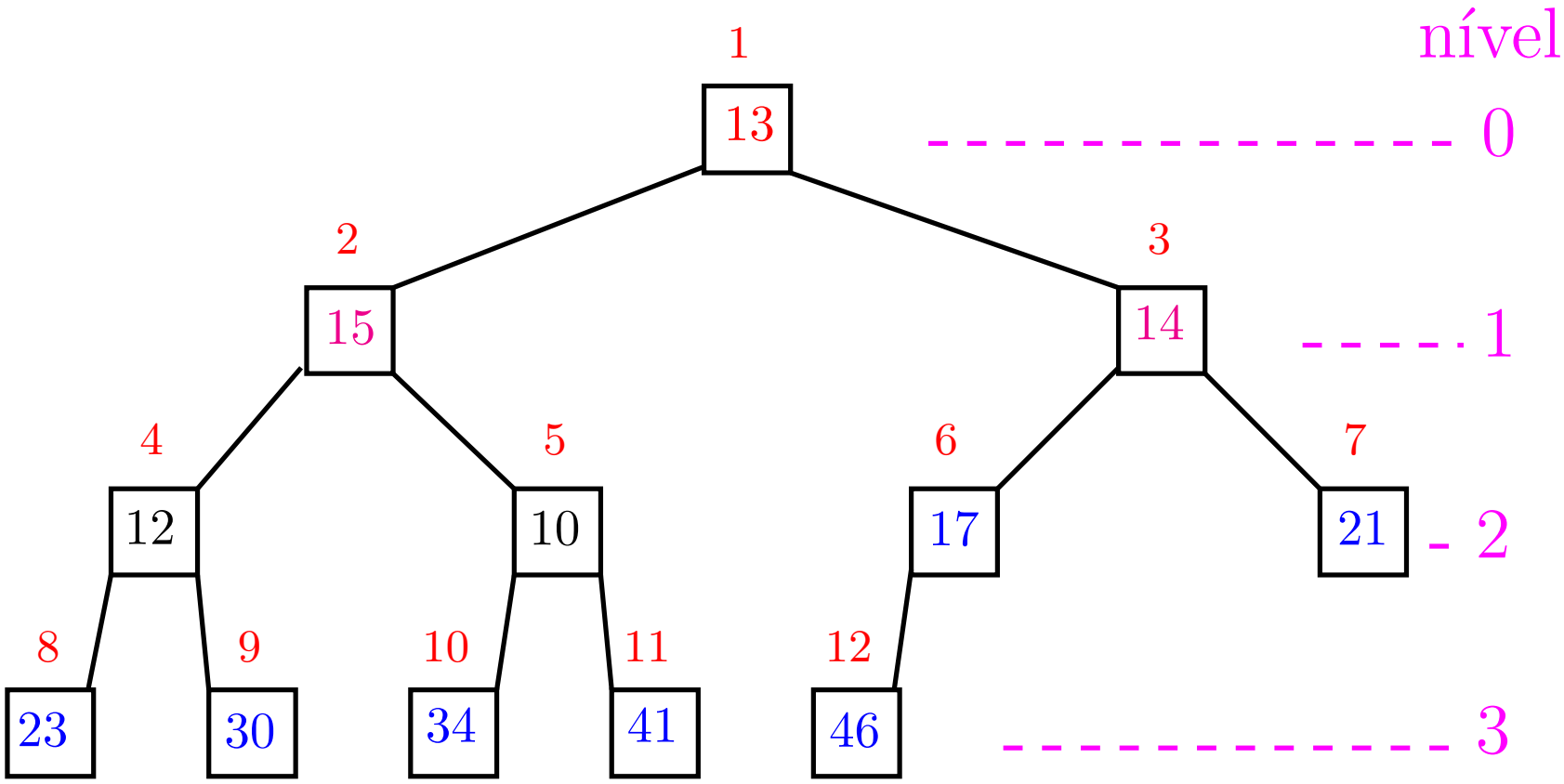

nível
1
17
0
2
15
3
14
1
4
12
5
10
6
13
7
21
2
8
23
9
30
10
34
11
41
12
46
3
1 2 3 4 5 6 7 8 9 10 11 12
17 15 14 12 10 13 21 23 30 34 41 46

Heapsort



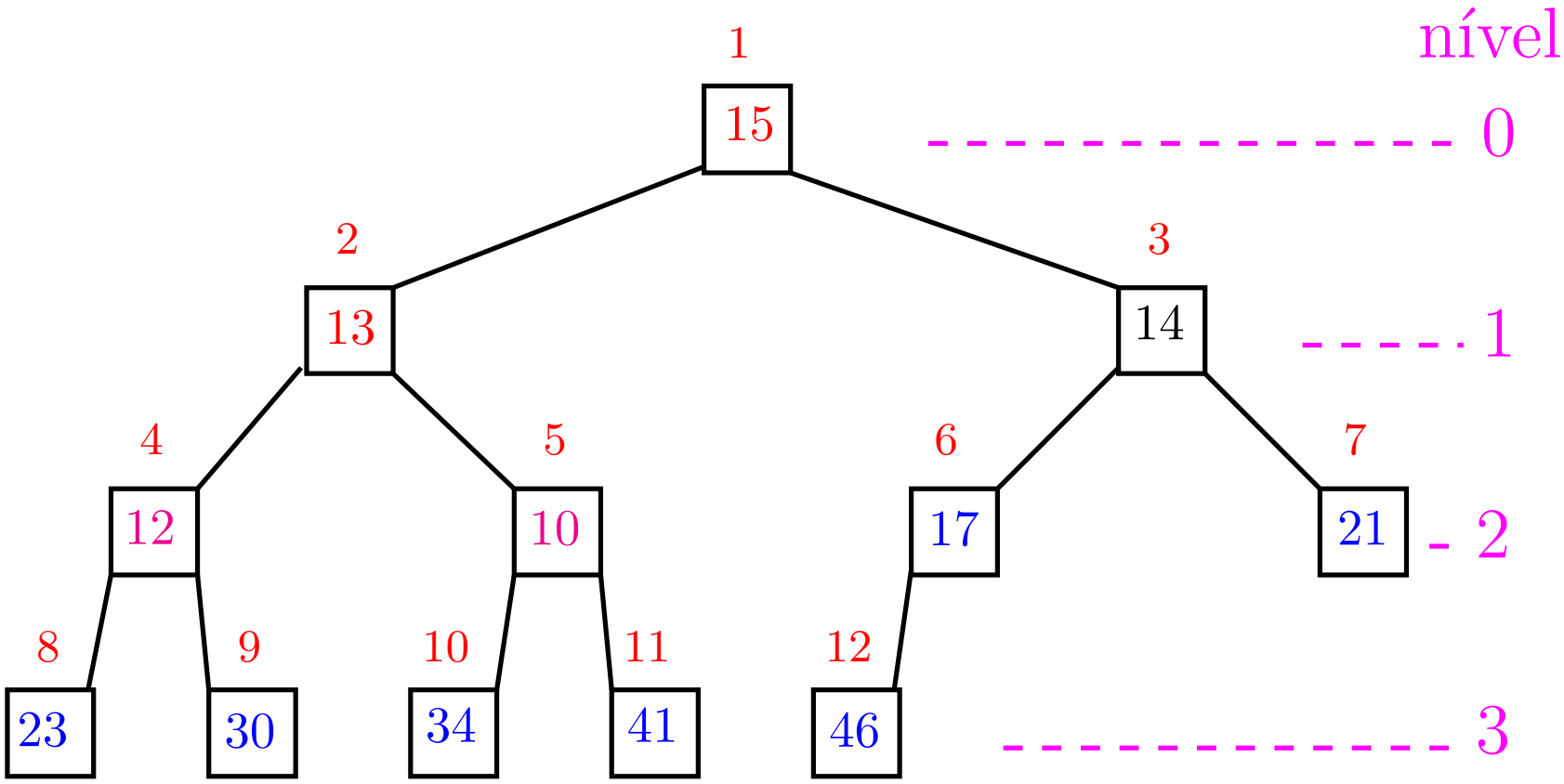
1	2	3	4	5	6	7	8	9	10	11	12
13	15	14	12	10	17	21	23	30	34	41	46

Heapsort



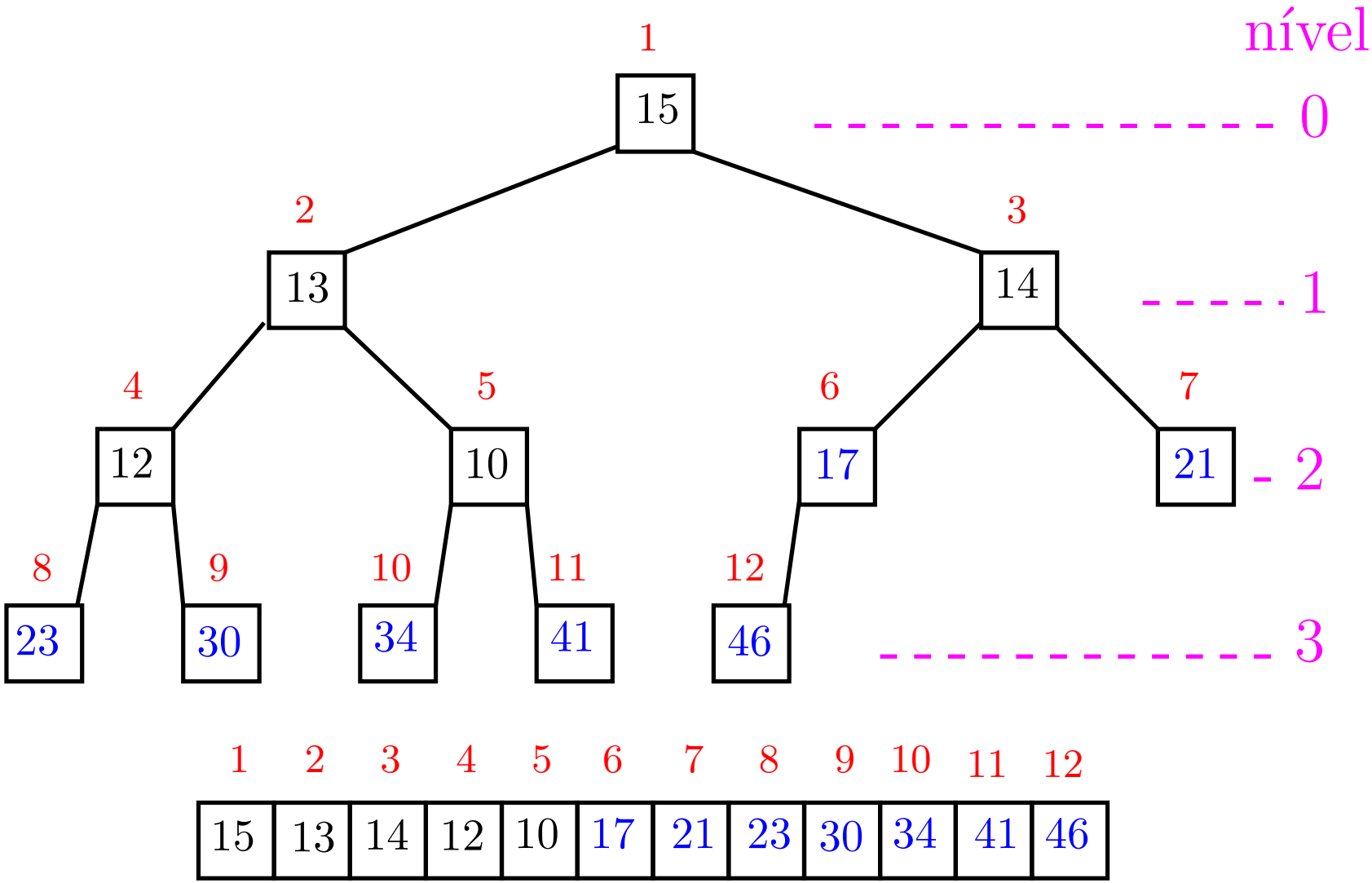
1	2	3	4	5	6	7	8	9	10	11	12
13	15	14	12	10	17	21	23	30	34	41	46

Heapsort

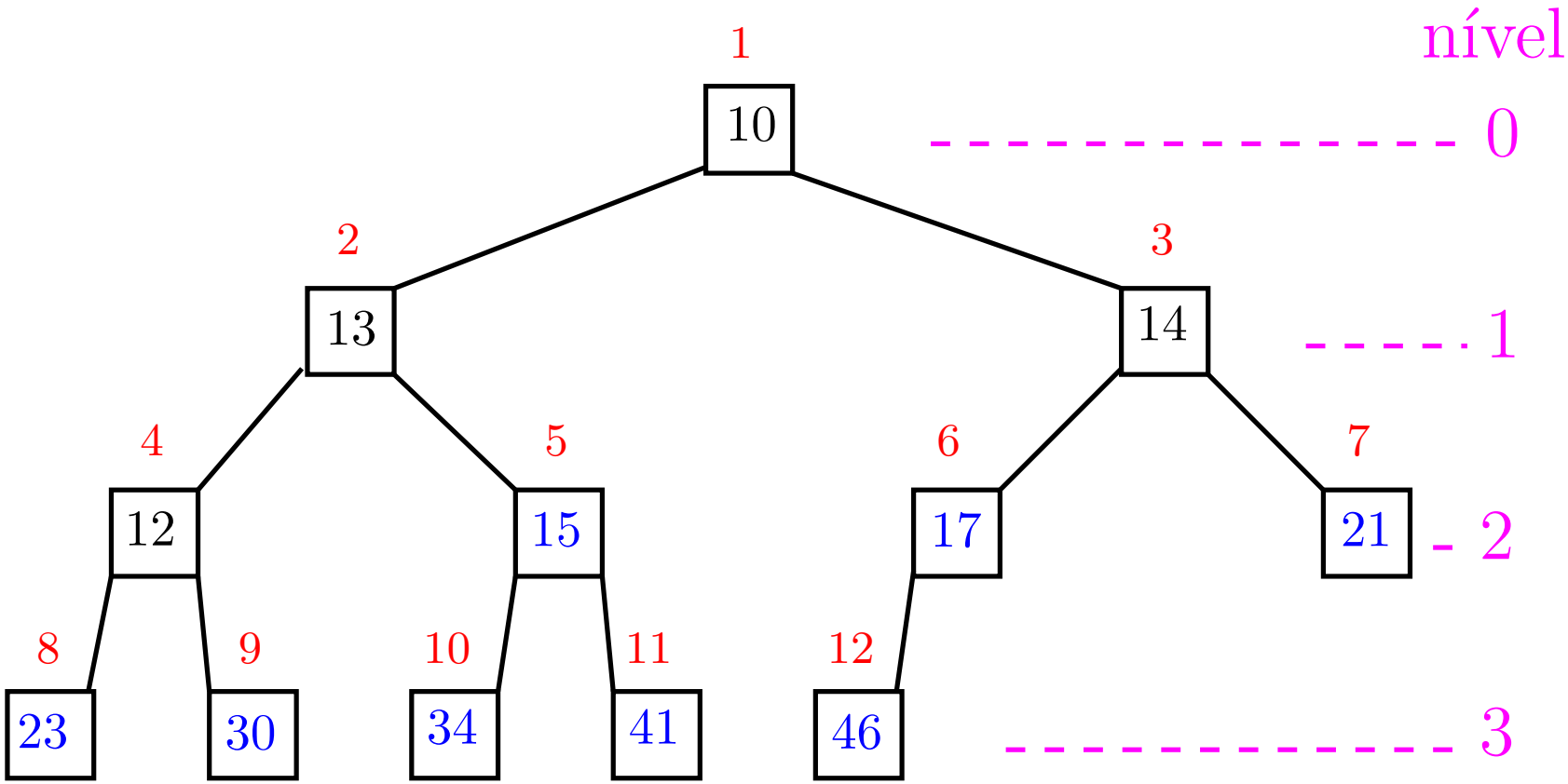


1	2	3	4	5	6	7	8	9	10	11	12
15	13	14	12	10	17	21	23	30	34	41	46

Heapsort

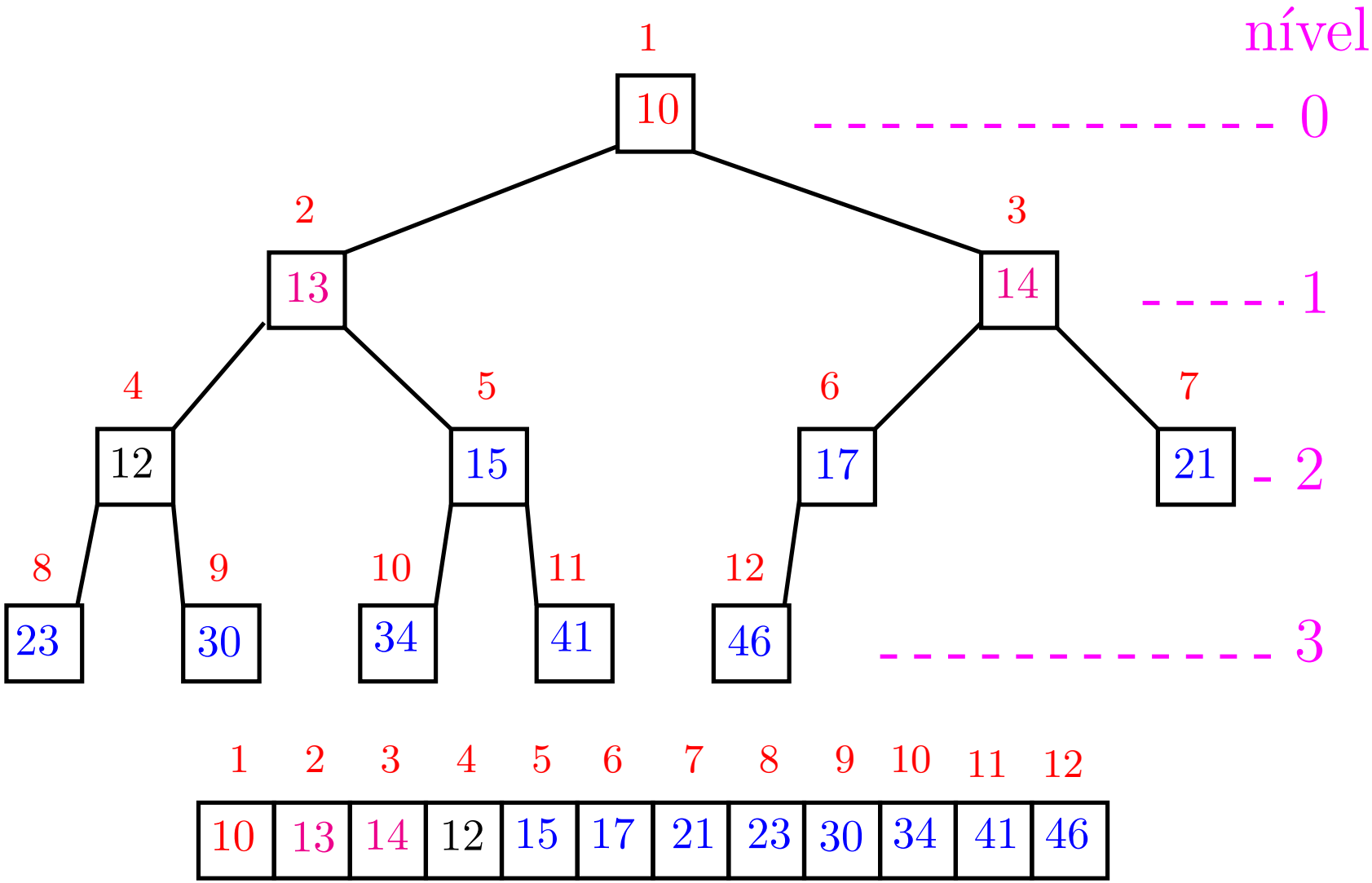

nível
1
15
0
2
3
13
14
1
4
5
6
7
12
10
17
21
2
8
9
10
11
12
23
30
34
41
46
3
1 2 3 4 5 6 7 8 9 10 11 12
15 13 14 12 10 17 21 23 30 34 41 46

Heapsort

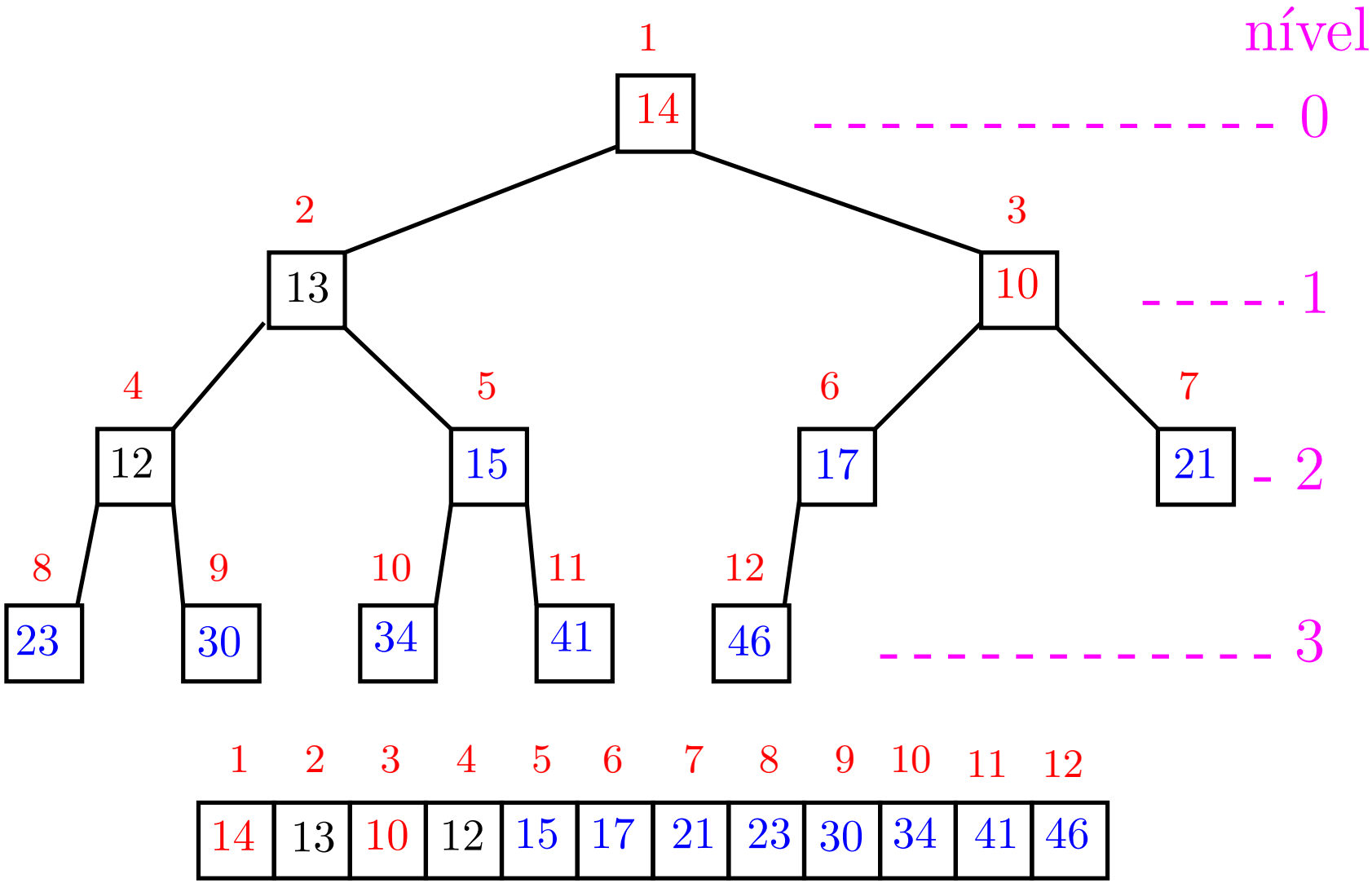


1	2	3	4	5	6	7	8	9	10	11	12
10	13	14	12	15	17	21	23	30	34	41	46

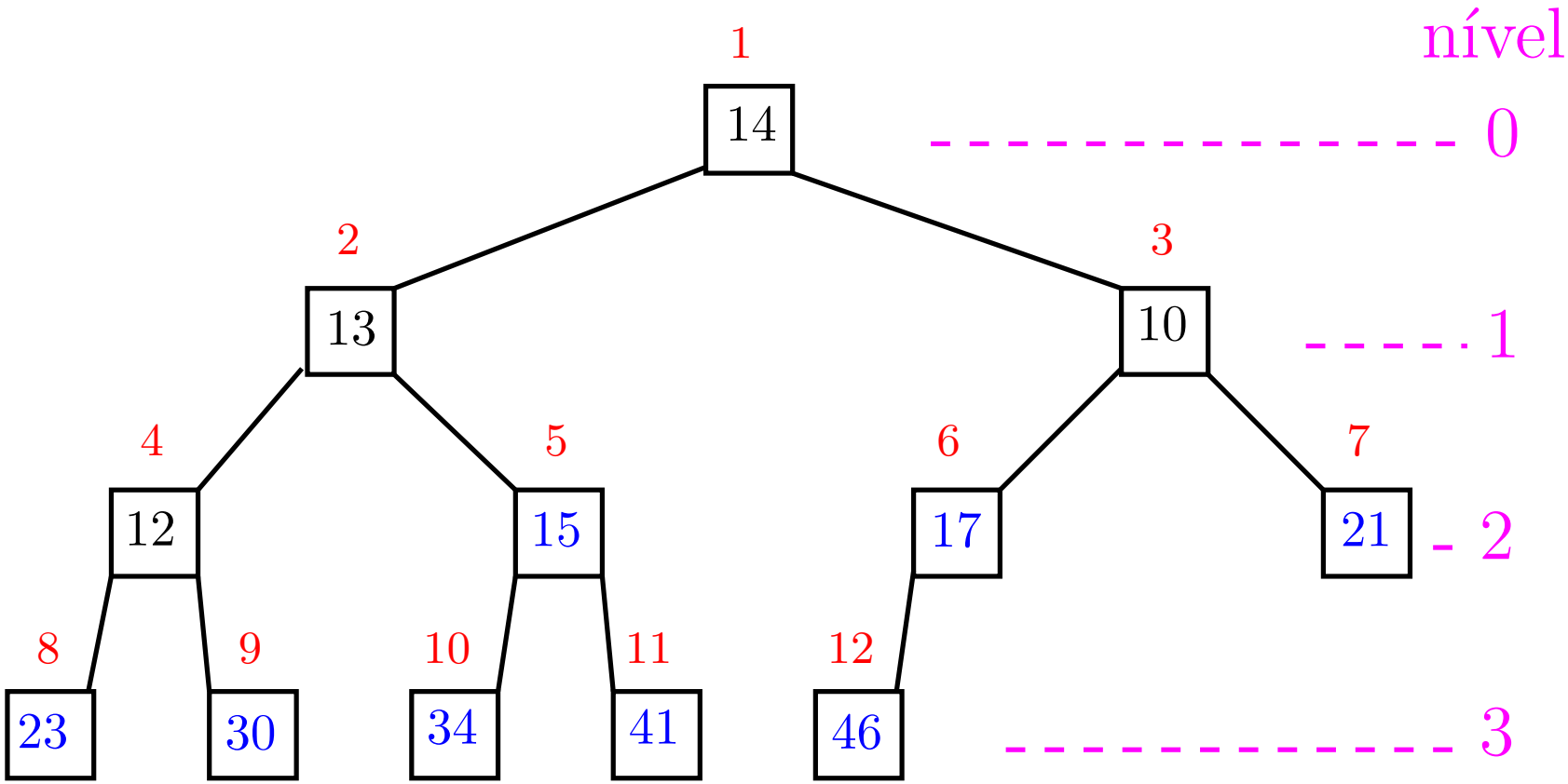
Heapsort


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

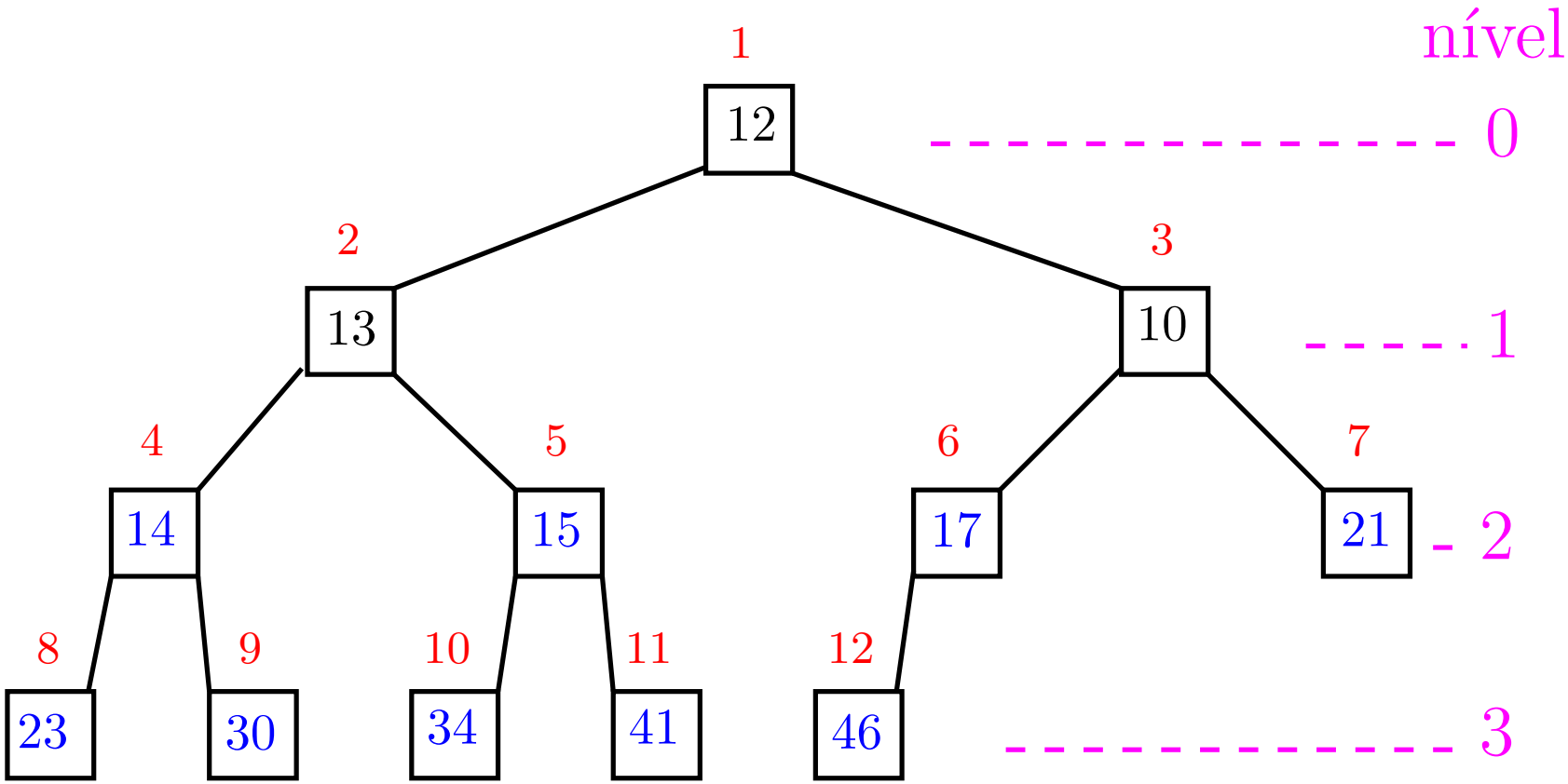

nível
1
14
0
2
13
3
10
1
4
12
5
15
6
17
7
21
2
8
23
9
30
10
34
11
41
12
46
3
1 2 3 4 5 6 7 8 9 10 11 12
14 13 10 12 15 17 21 23 30 34 41 46

Heapsort



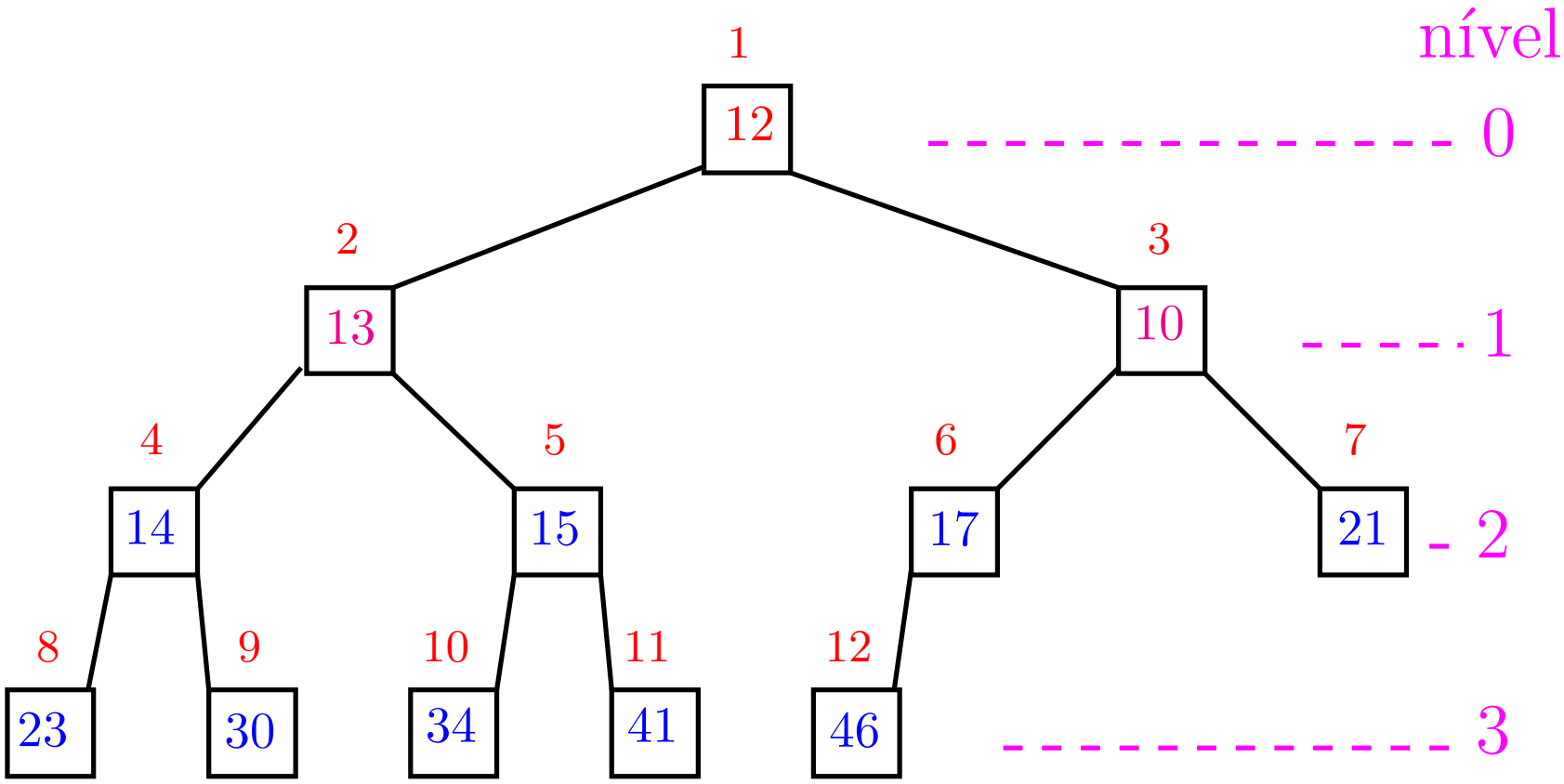
1	2	3	4	5	6	7	8	9	10	11	12
14	13	10	12	15	17	21	23	30	34	41	46

Heapsort



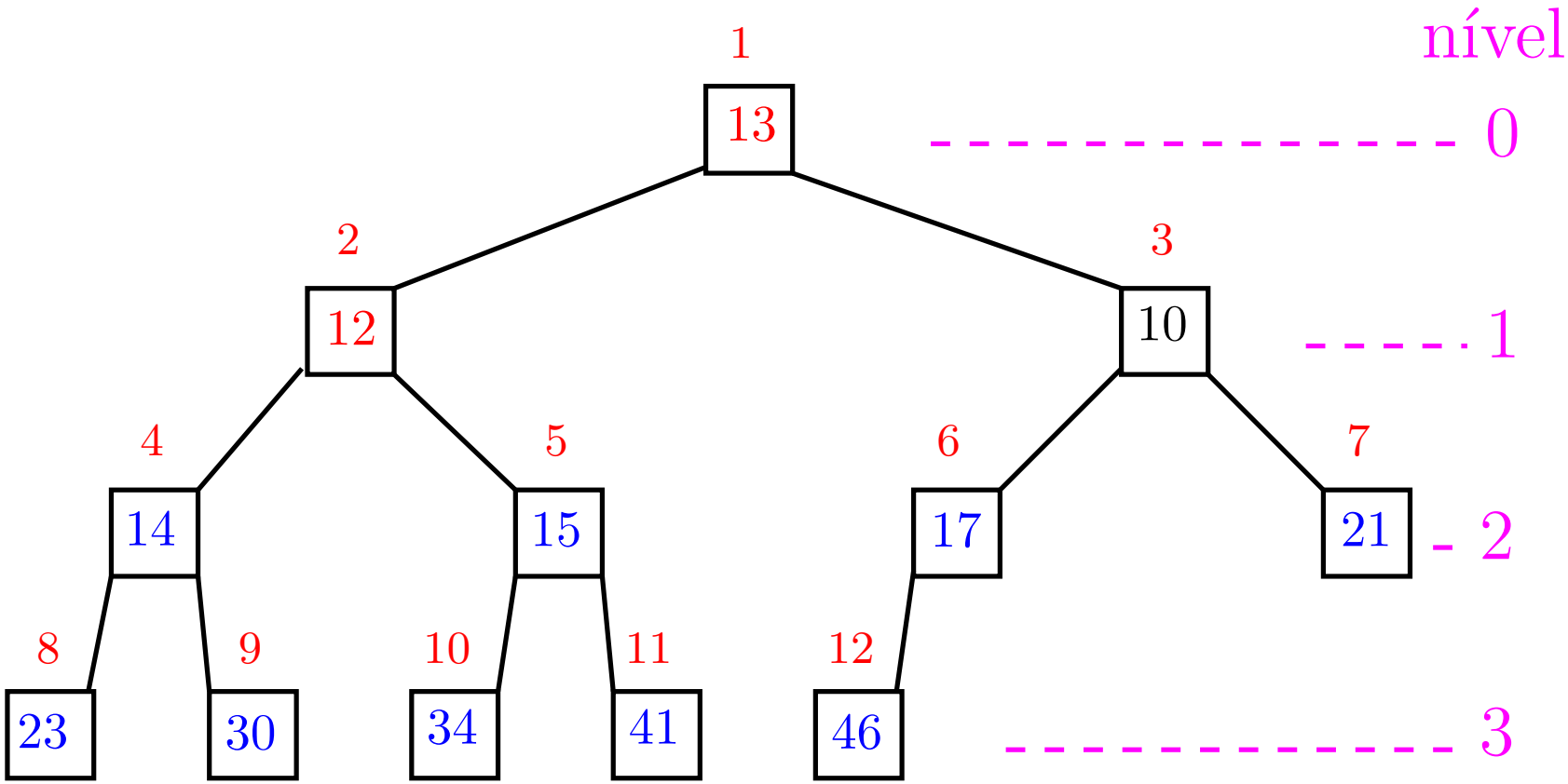
1	2	3	4	5	6	7	8	9	10	11	12
12	13	10	14	15	17	21	23	30	34	41	46

Heapsort



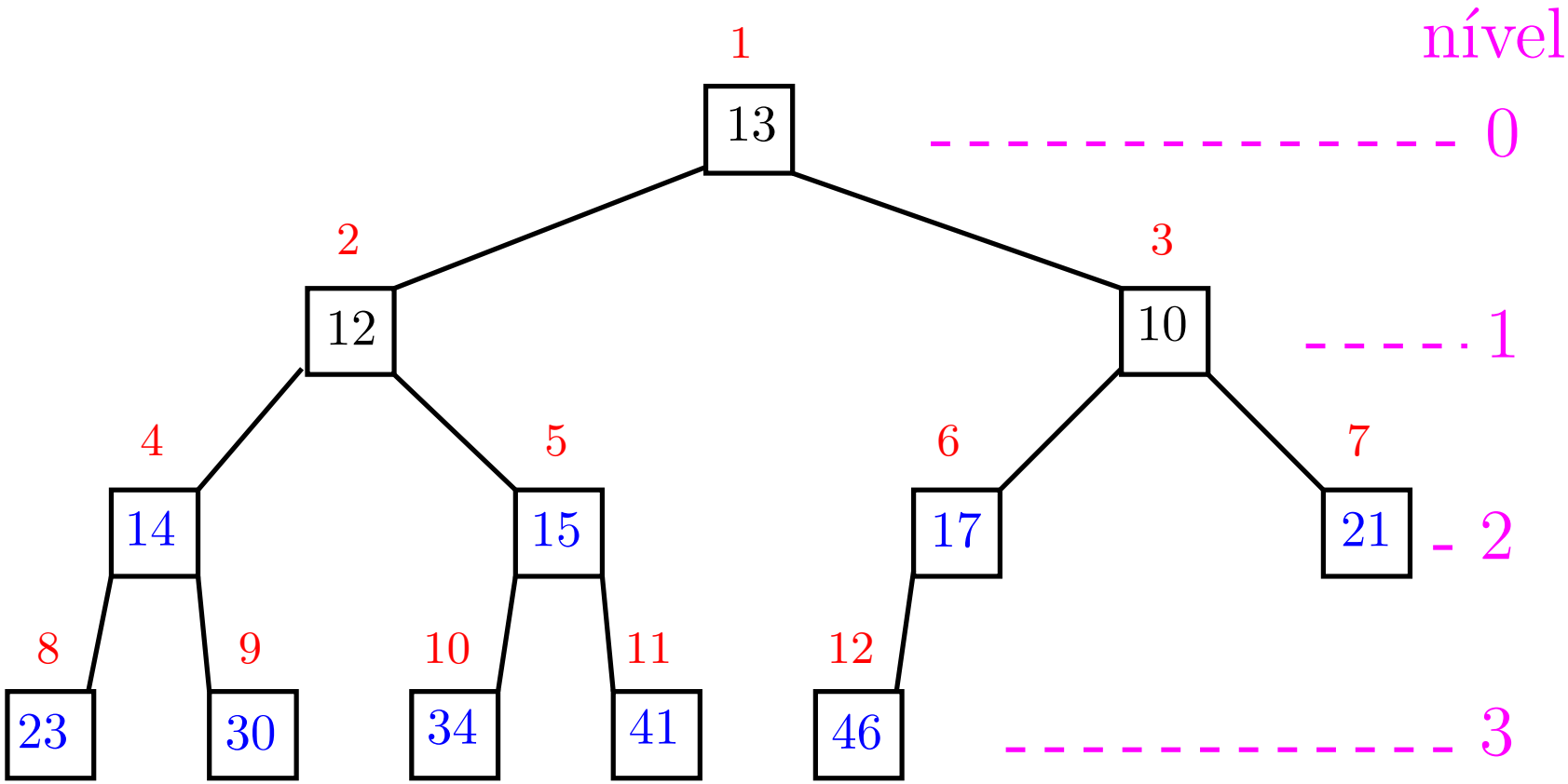
1	2	3	4	5	6	7	8	9	10	11	12
12	13	10	14	15	17	21	23	30	34	41	46

Heapsort



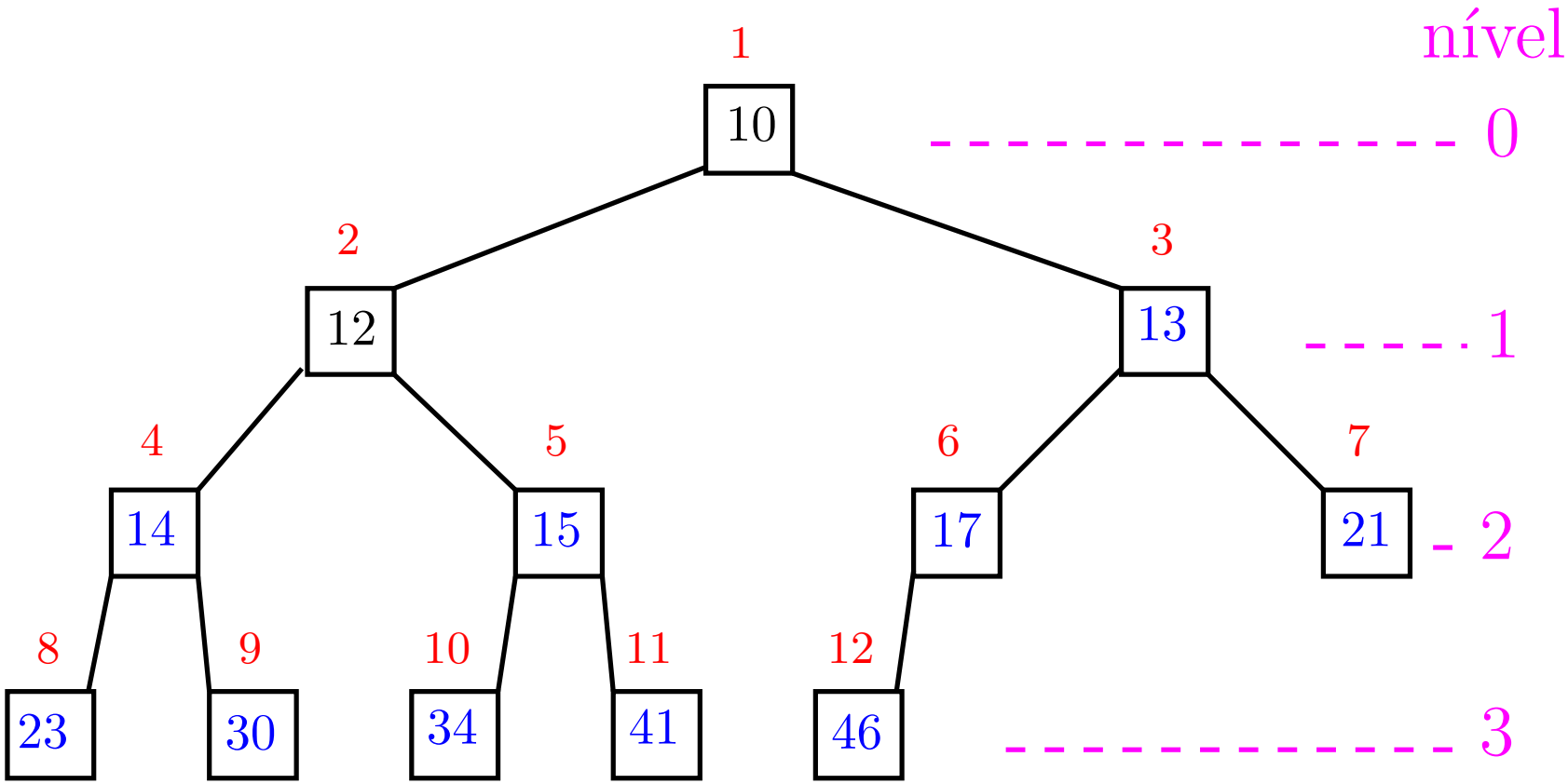
1	2	3	4	5	6	7	8	9	10	11	12
13	12	10	14	15	17	21	23	30	34	41	46

Heapsort



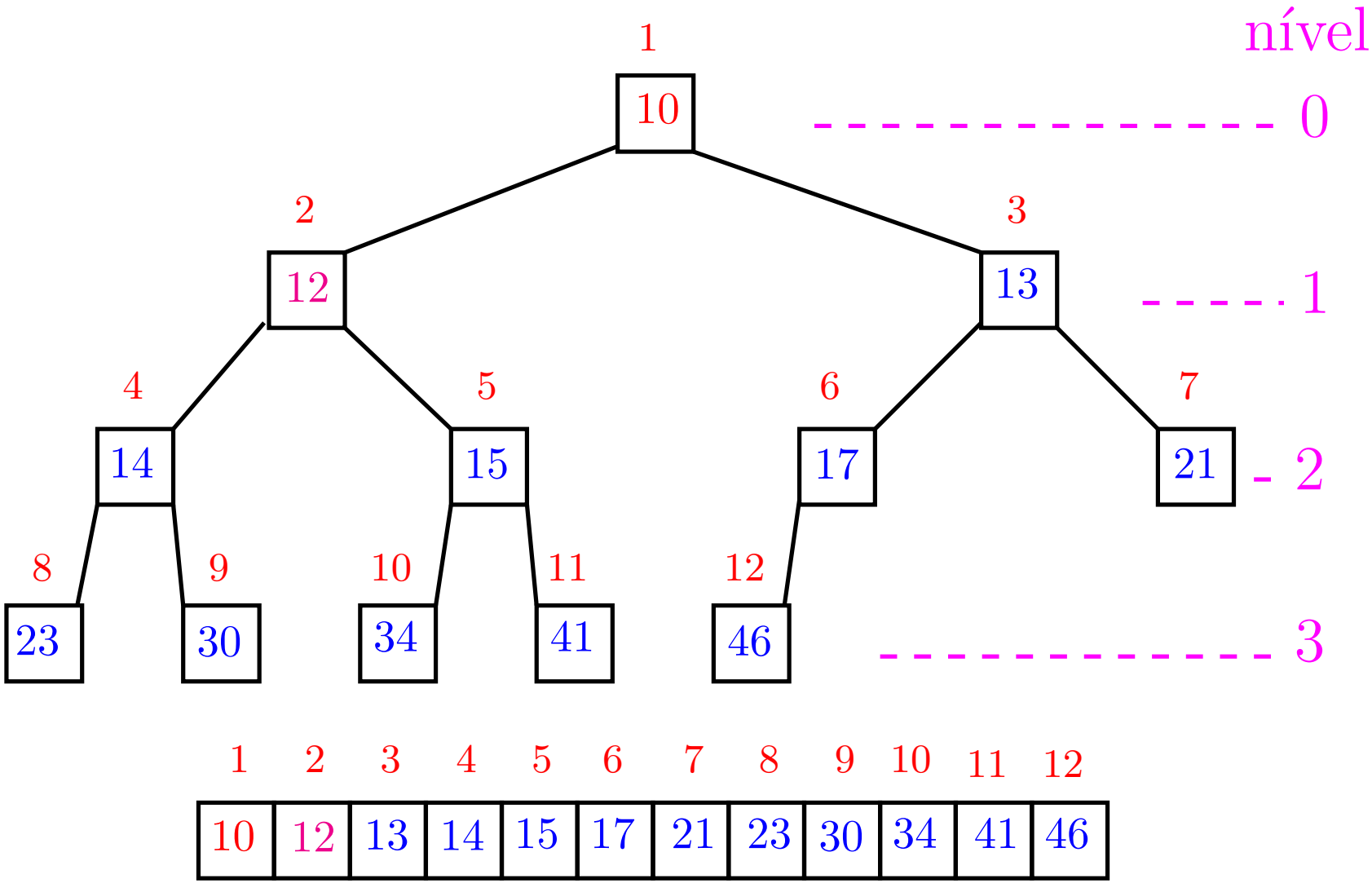
1	2	3	4	5	6	7	8	9	10	11	12
13	12	10	14	15	17	21	23	30	34	41	46

Heapsort

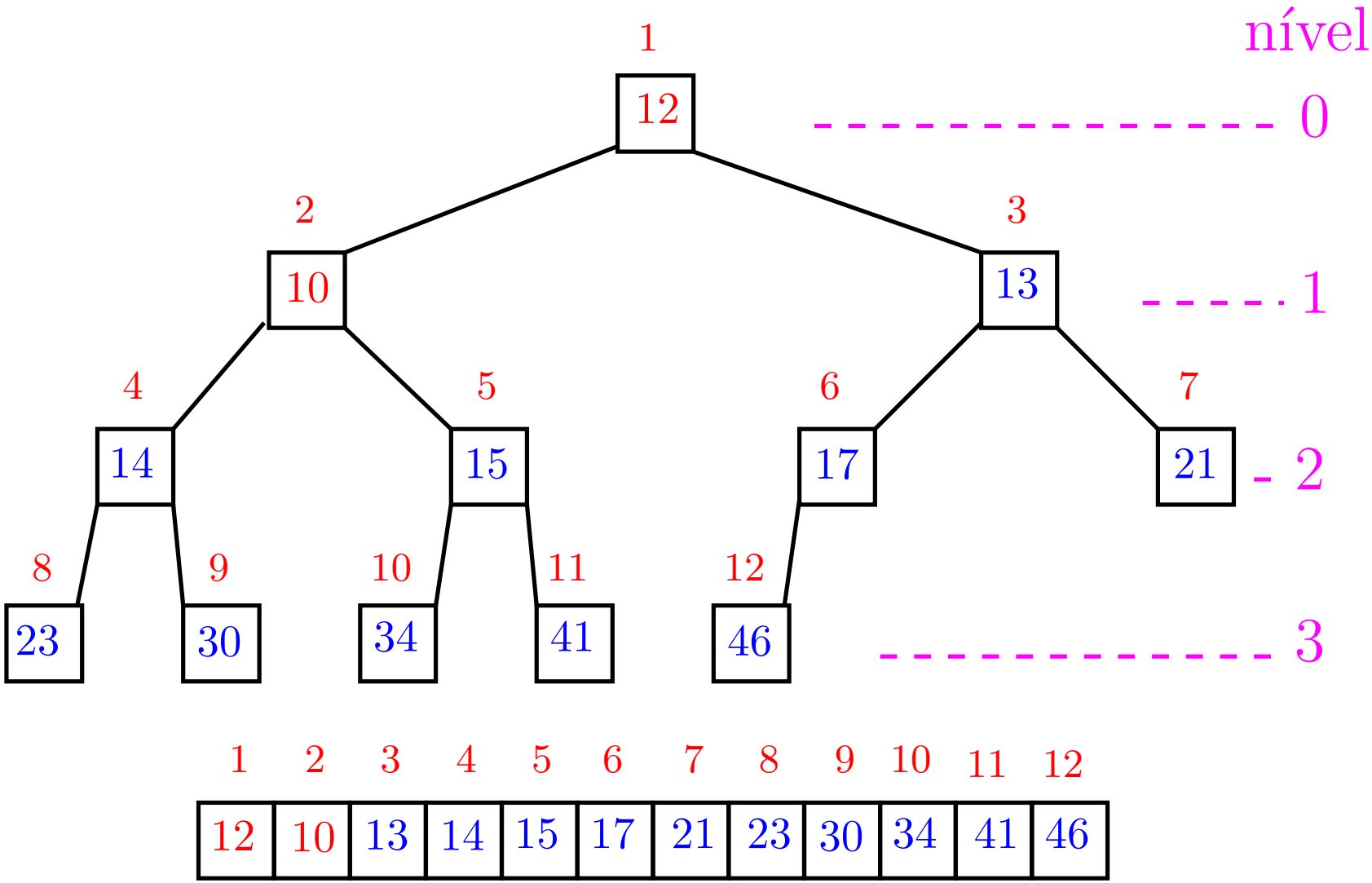


1	2	3	4	5	6	7	8	9	10	11	12
10	12	13	14	15	17	21	23	30	34	41	46

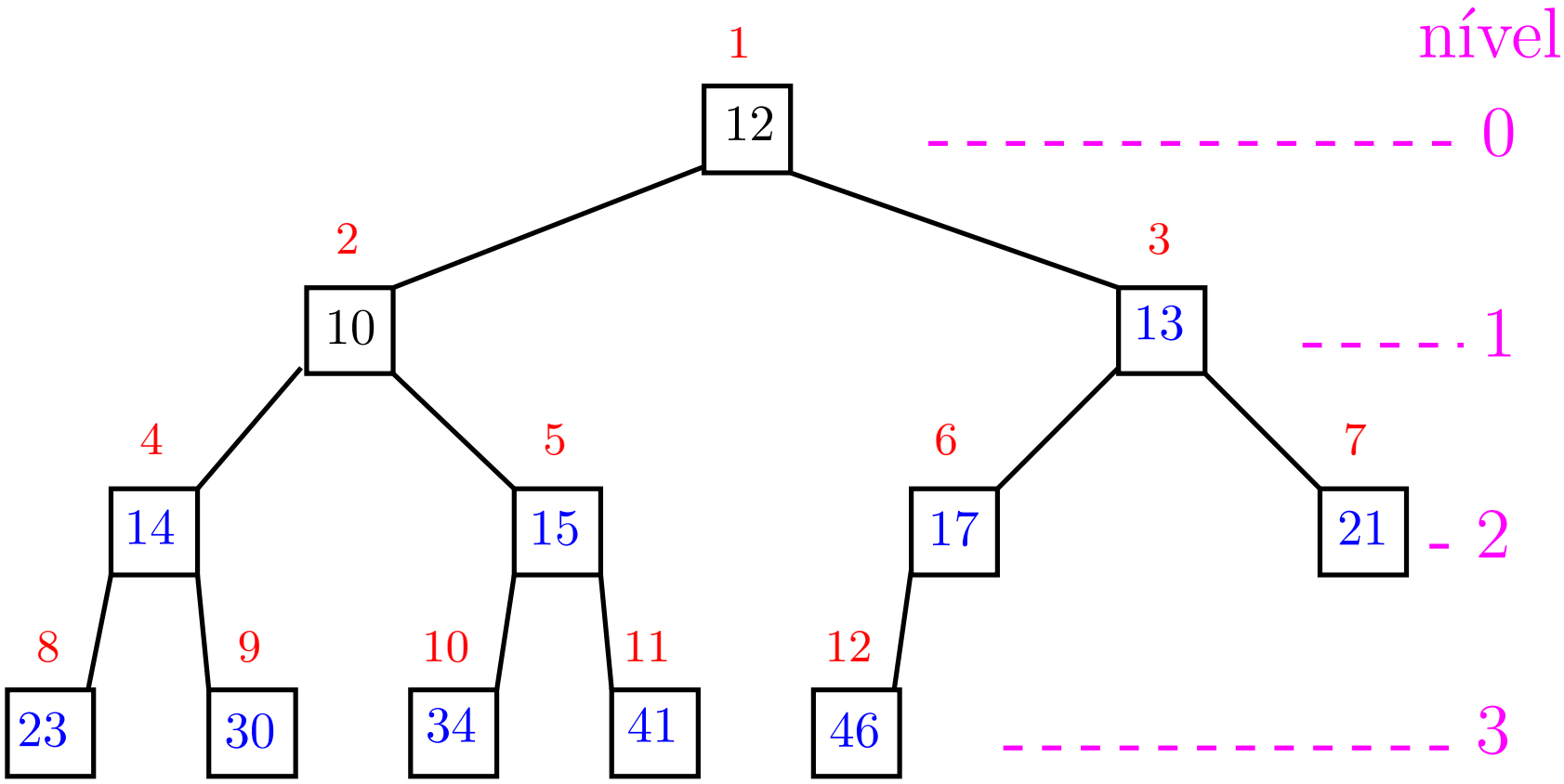
Heapsort


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

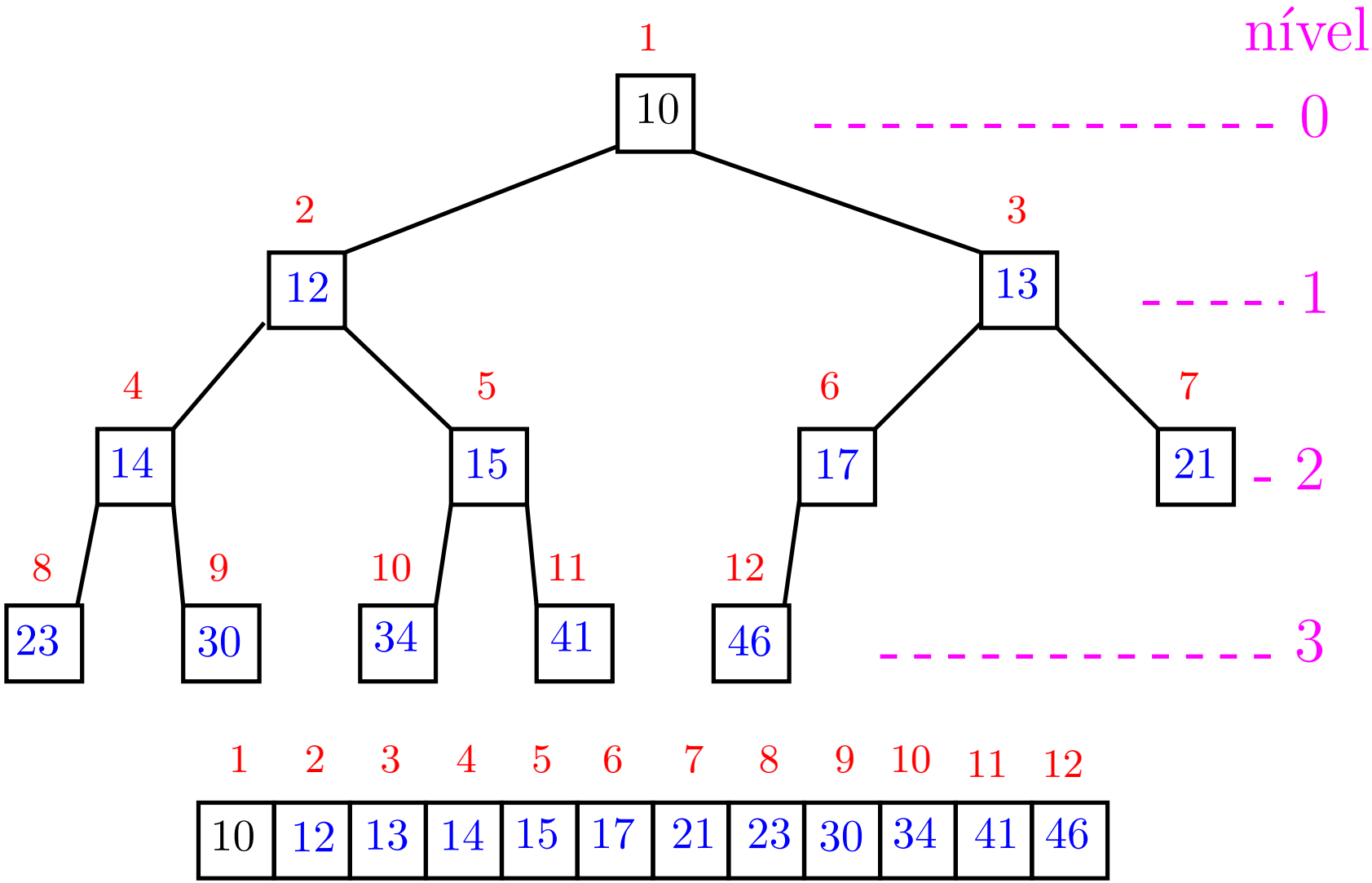

nível
1
12
0
2
10
3
13
1
4
14
5
15
6
17
7
21
2
8
23
9
30
10
34
11
41
12
46
3
1 2 3 4 5 6 7 8 9 10 11 12
12 10 13 14 15 17 21 23 30 34 41 46

Heapsort

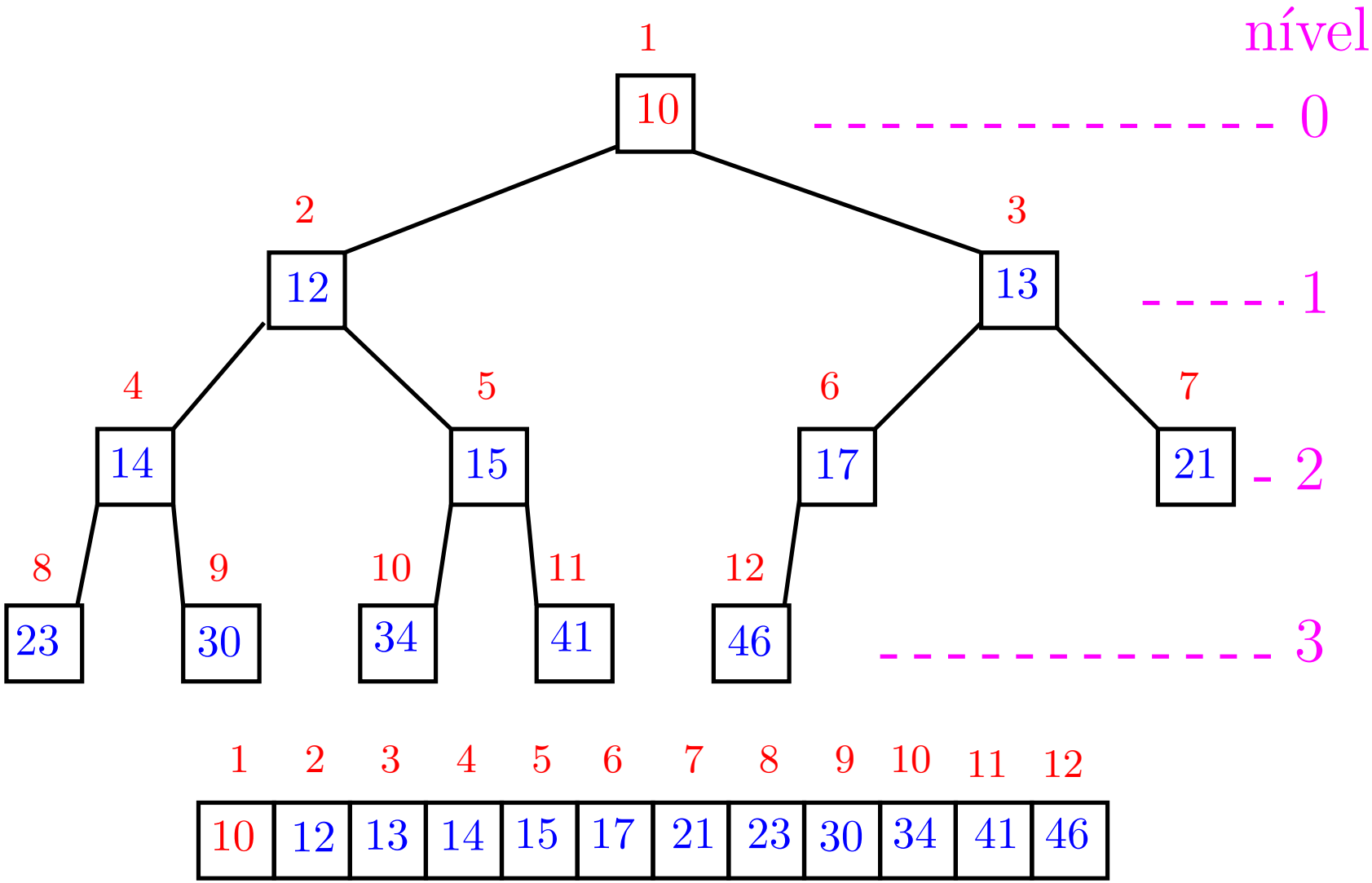


1	2	3	4	5	6	7	8	9	10	11	12
12	10	13	14	15	17	21	23	30	34	41	46

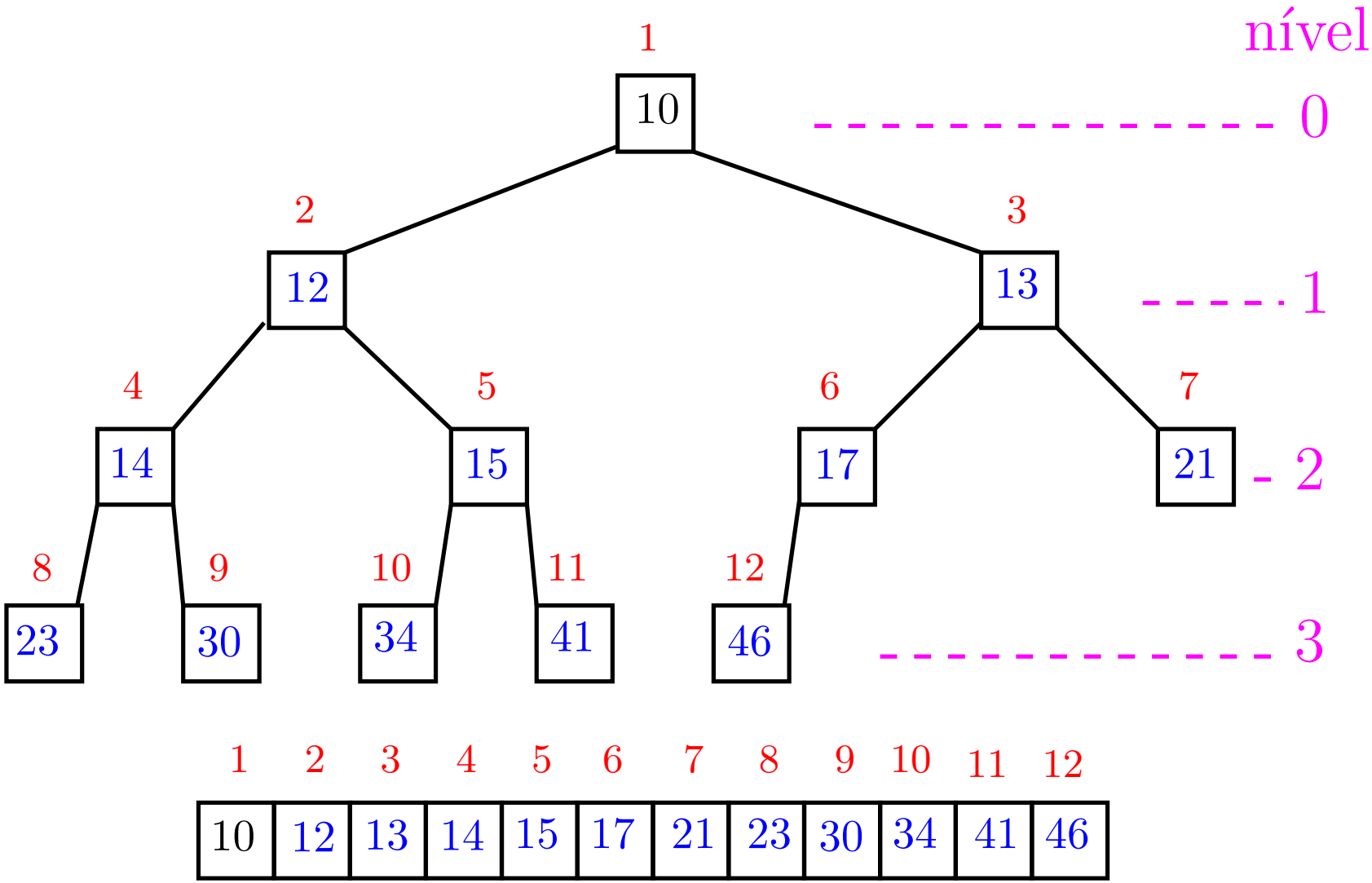
Heapsort


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

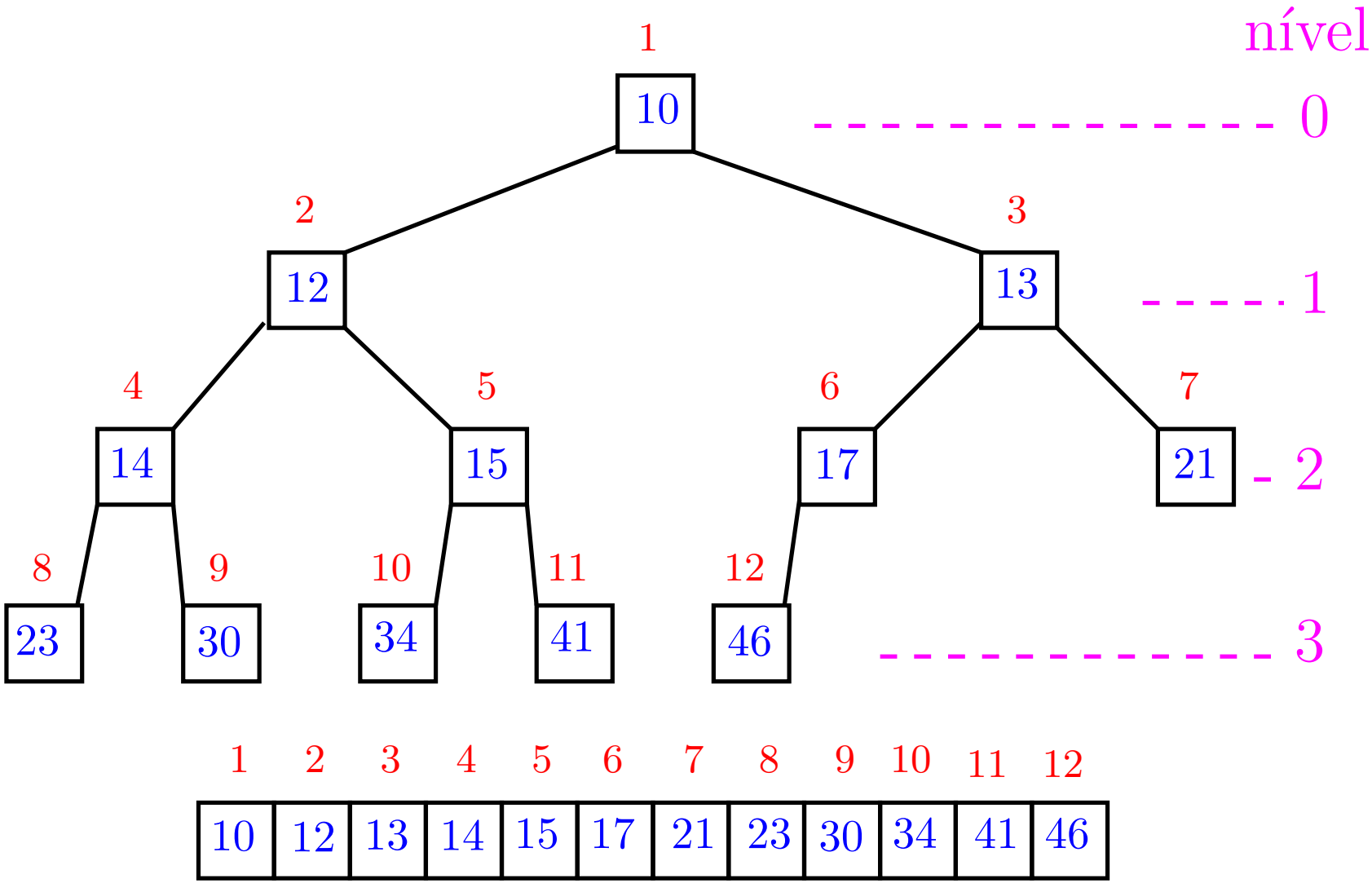
Heapsort


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

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

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

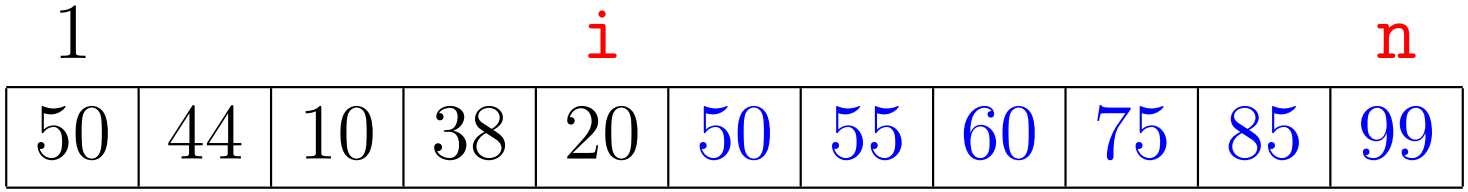
Função heapSort

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

(i0) v[i+1 .. n] é crescente;

(i1) v[1 .. i] ≤ 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 $n \lg n$.

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

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`	$O(n^2)$	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
`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/UCIqiLefbVHsOAXDAxQJH7X

Filas priorizadas

Fonte: `We love it`

Filas priorizadas, PF, Priority queues, S&W

Filas priorizadas

Uma **fila priorizada** (ou **fila com prioridades**) é um ADT (*abstract data type*) que generaliza tanto a fila quanto a pilha.

Uma fila priorizada decrescente ou **PQ de máximo** é um ADT que manipula um conjunto de itens por meio de duas operações fundamentais:

- inserção de um novo item no conjunto e
- remoção de um item máximo.

Isso significa que uma fila priorizada manipula itens comparáveis (Comparable).

API PQ-máximo

```
public class MaxPQ<Item extends
                   Comparable<Item>>
```

public class	MaxPQ	
	`MaxPQ(int cap)`	cria uma PQ
`void`	`insert(Item v)`	insere item v nesta F
`Item`	`max()`	devolve um máximo
`Item`	`delMax()`	remove e devolve
`boolean`	`isEmpty()`	PQ está vazia?
`int`	`size()`	número de itens

Cliente MinPQ: class TopM

No código a seguir, T é uma abreviatura para Transaction.

Transaction é uma das classes do algs4.

O programa retorna as M transações de maior valor.

As transações são lidas da entrada padrão e estão no arquivo transactions.txt.

Cliente MinPQ: class TopM

```
public static void main(String[] args) {
   int M = Integer.parseInt(args[0]);
   MinPQ<T> pq =new MinPQ<T>(M+1);
   while (StdIn.hasNextLine()) {
       pq.insert(new T(StdIn.readLine()));
       if (pq.size() > M)
           pq.delMin();
   }
   Stack<T> stack =new Stack<T>();
   while (!pq.isEmpty())
       stack.push(pq.delMin());
   for (T t :  stack)
       StdOut.println(t);
}
```

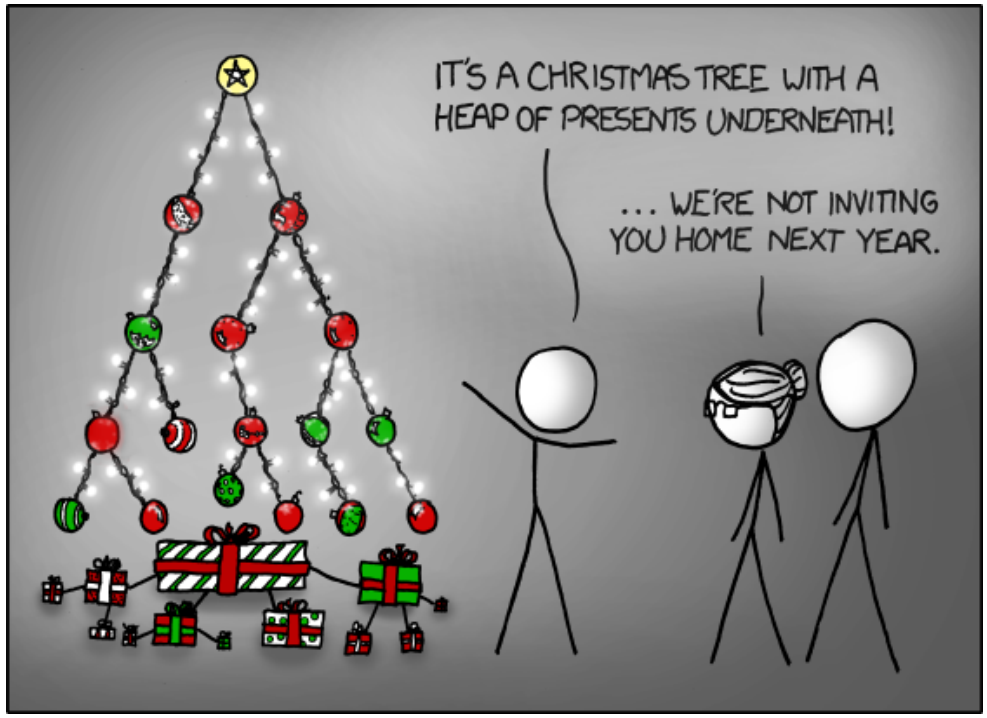
Implementações elementares

Podemos implementar a classe `MaxPQ` ou `MinPQ` com

- vetor de itens não-ordenados;
- vetor de itens ordenados;
- lista ligadas de itens ordenados ou não ordenados.

Em todas essas implementações o consumo de tempo pode ser proporcional ao número `n` de itens na fila.

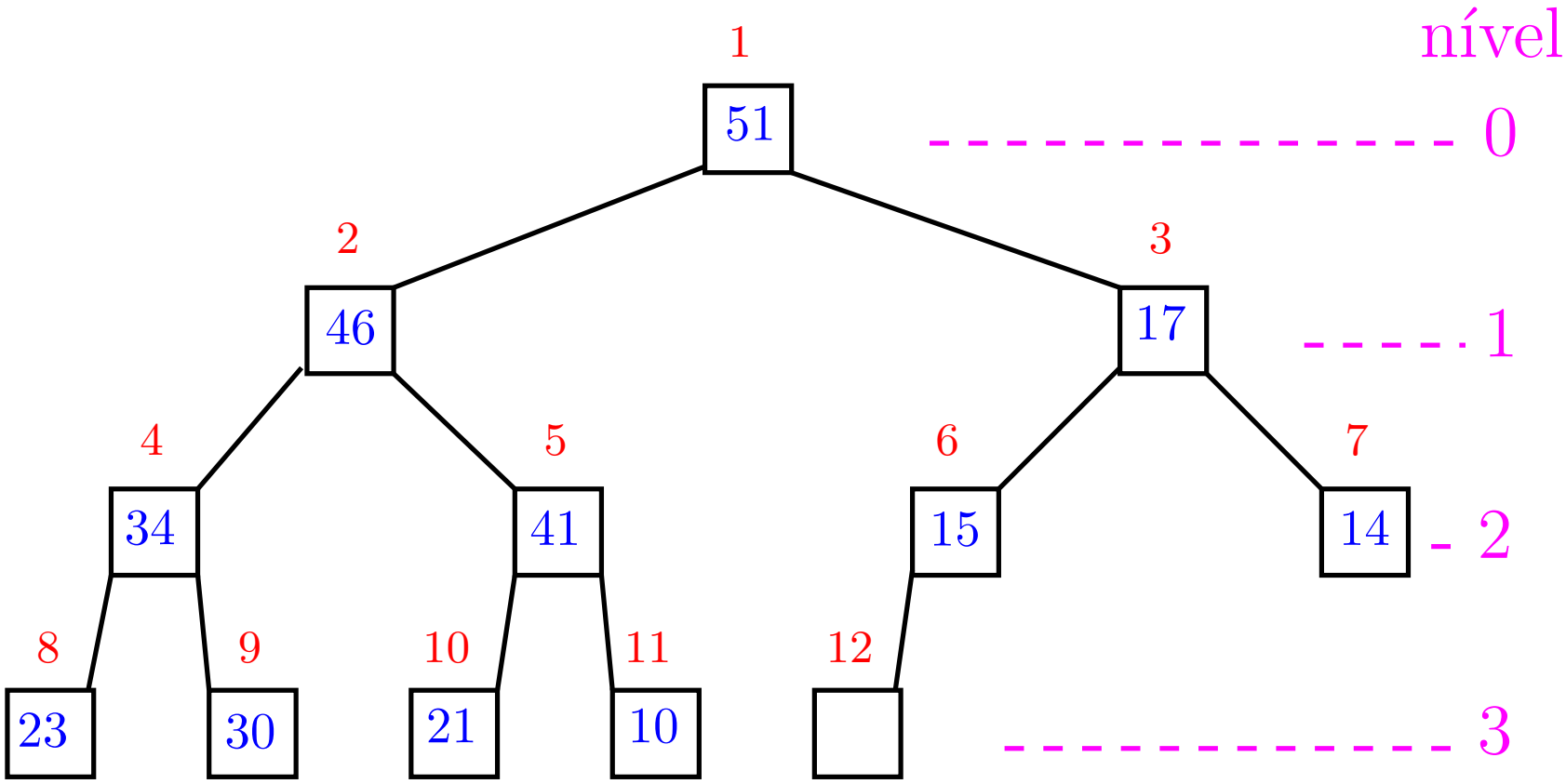
PQ de máximo



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

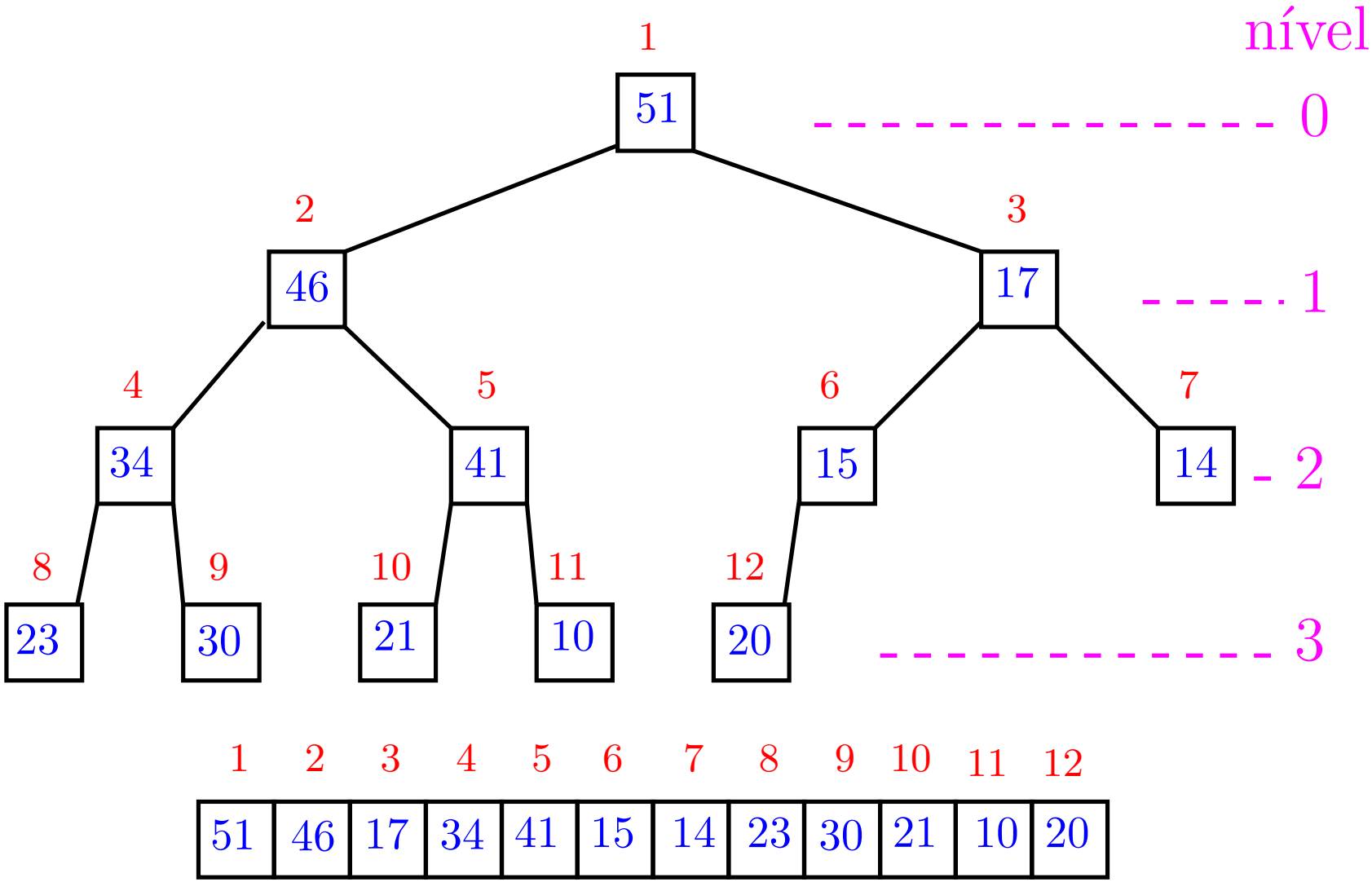
Filas priorizadas, PF, Priority queues, S&W

max-heap: insert()

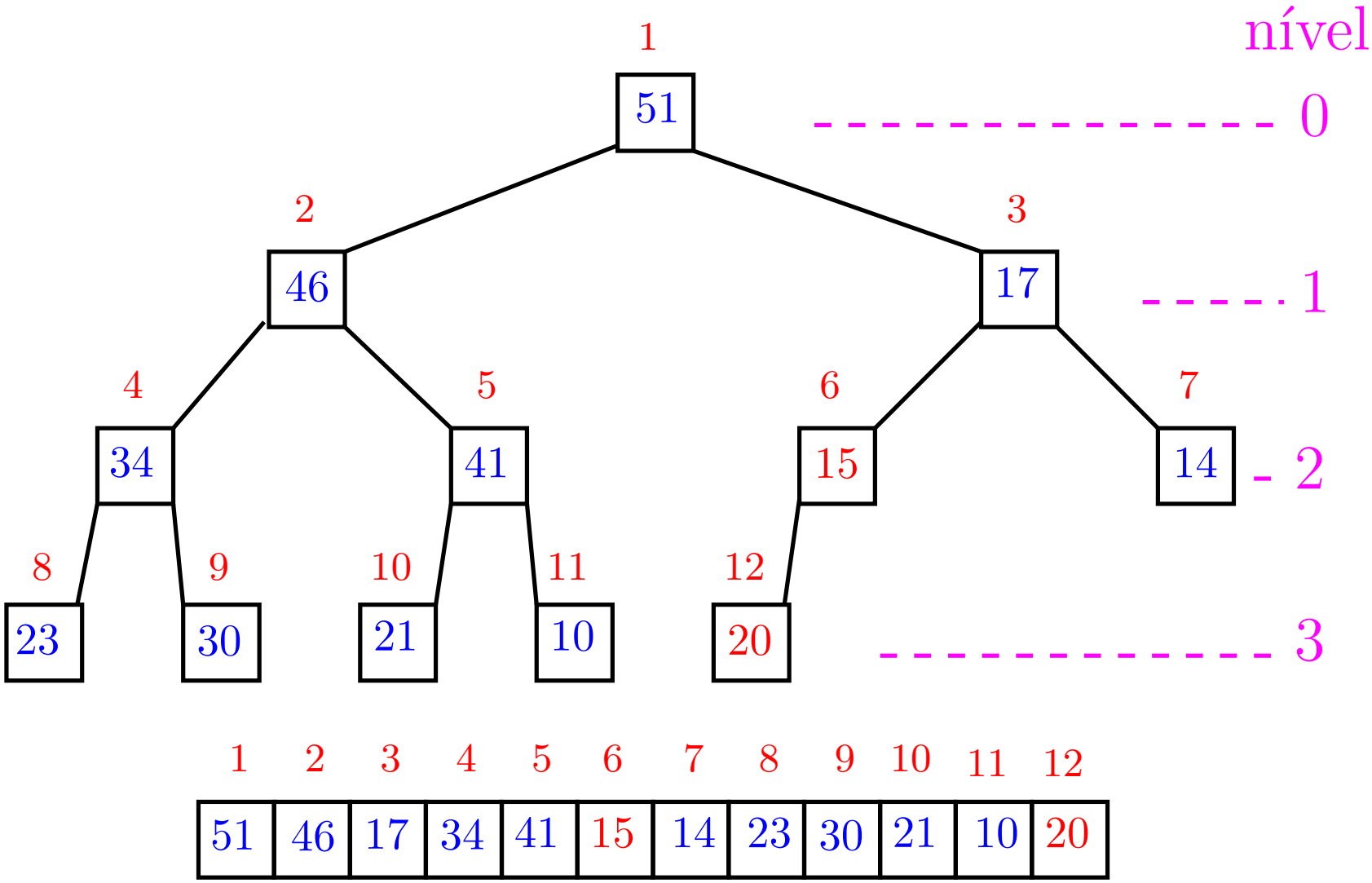


1	2	3	4	5	6	7	8	9	10	11	12
51	46	17	34	41	15	14	23	30	21	10	

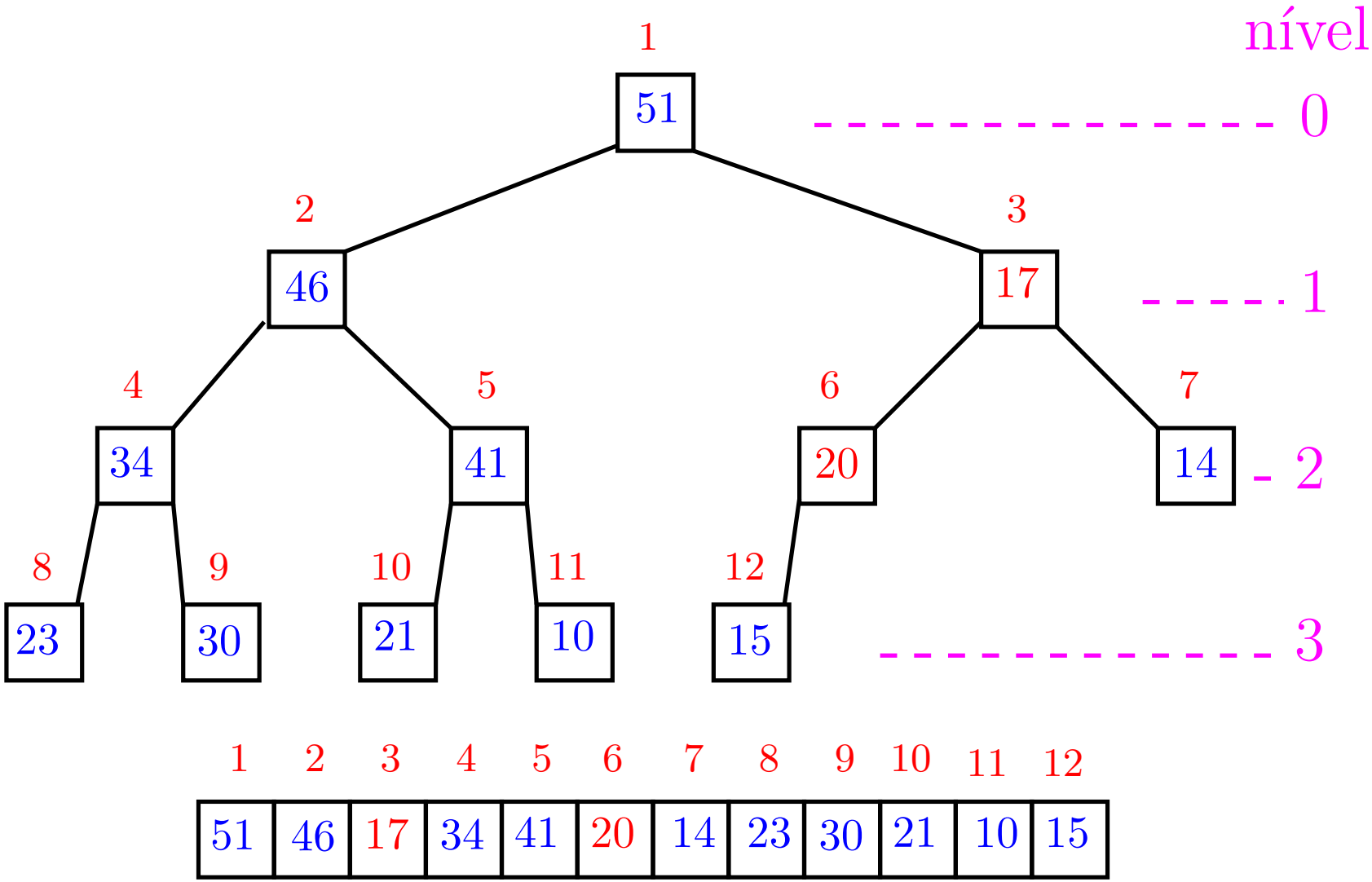
swim()


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
20
3
1 2 3 4 5 6 7 8 9 10 11 12
51 46 17 34 41 15 14 23 30 21 10 20

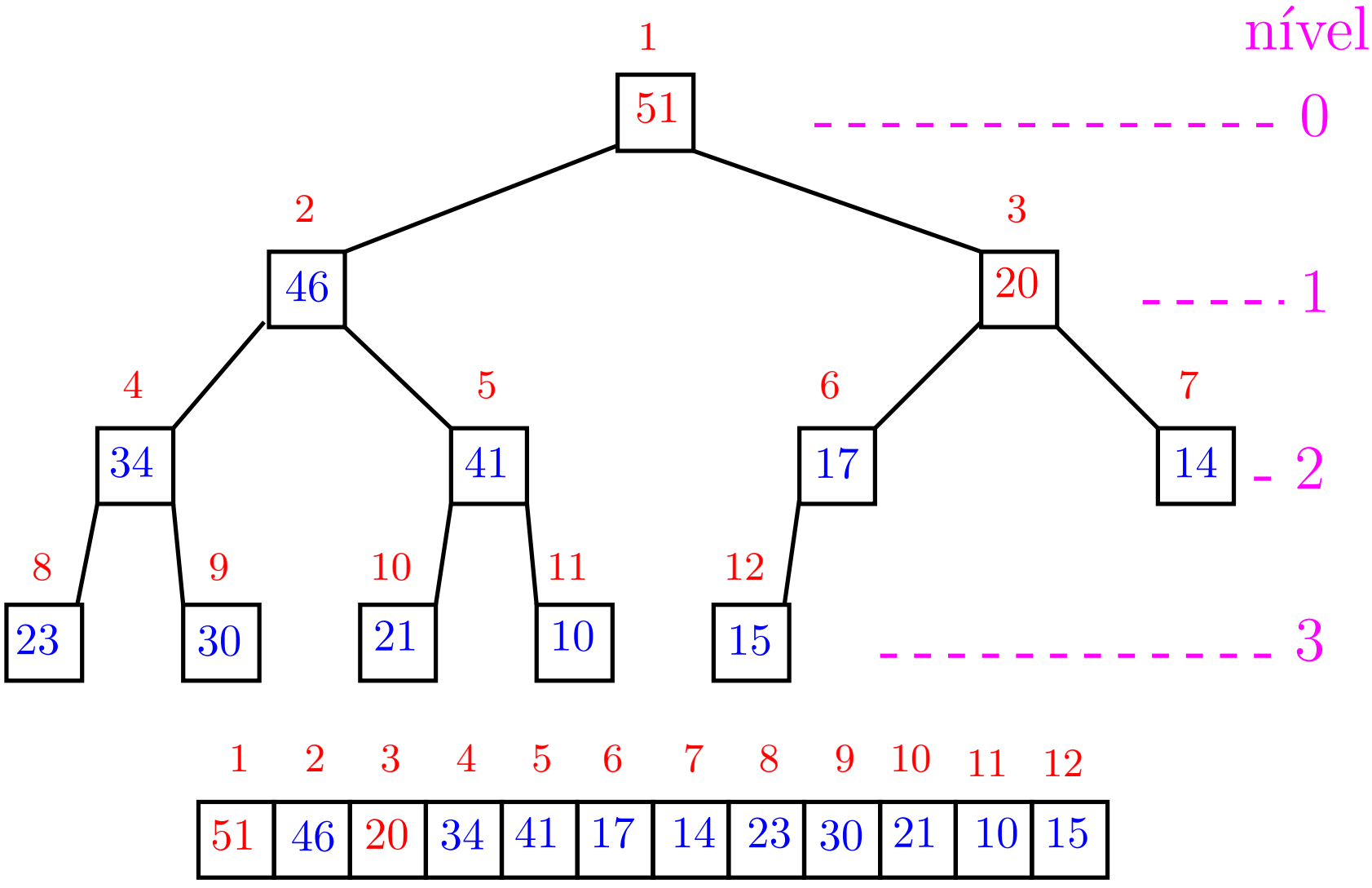
swim()


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
20
3
1 2 3 4 5 6 7 8 9 10 11 12
51 46 17 34 41 15 14 23 30 21 10 20

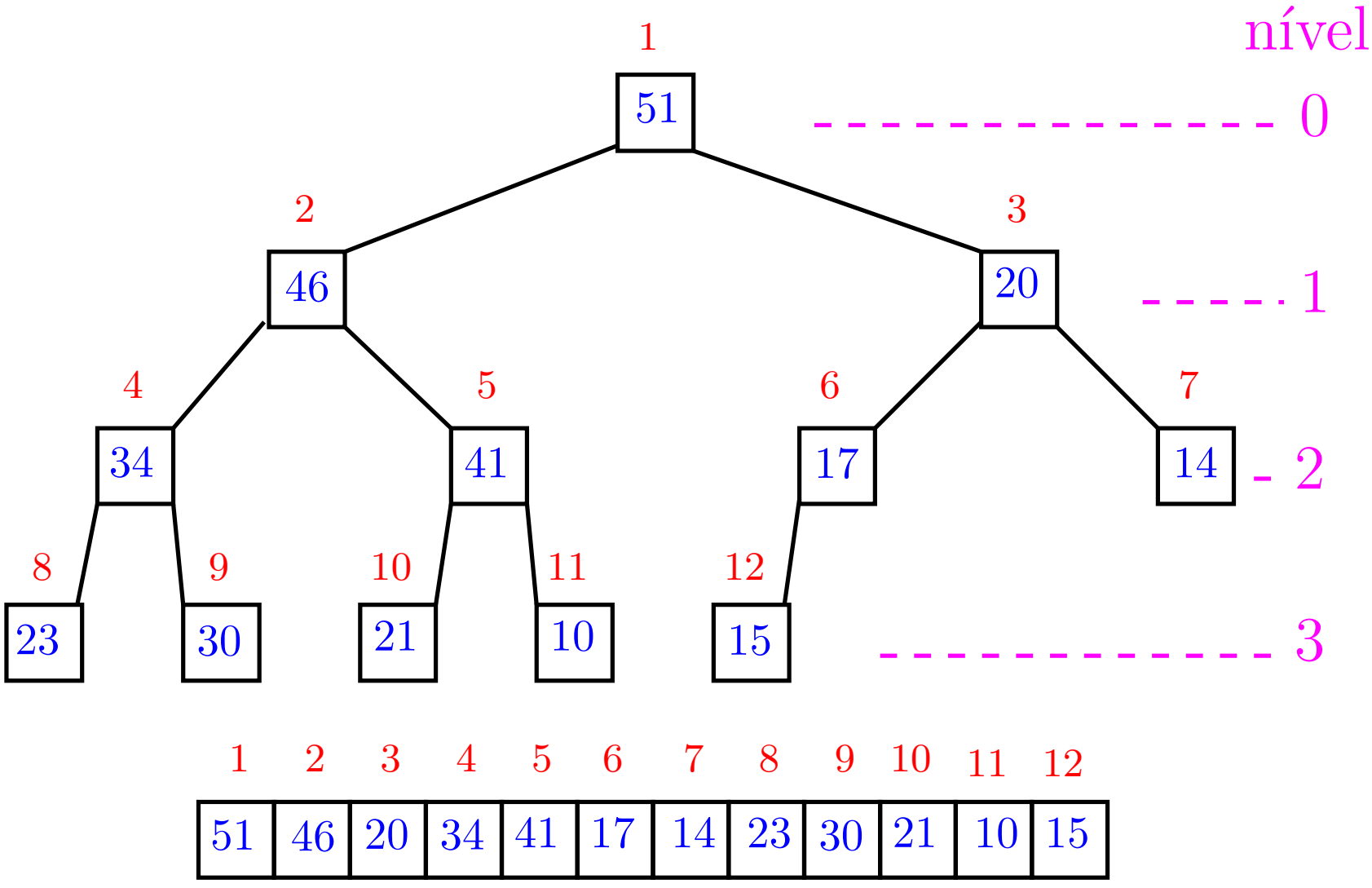
swim()


nível
1
51
0
2
46
3
17
1
4
34
5
41
6
20
7
14
2
8
23
9
30
10
21
11
10
12
15
3
1 2 3 4 5 6 7 8 9 10 11 12
51 46 17 34 41 20 14 23 30 21 10 15

swim()


nível
1
51
0
2
46
3
20
1
4
34
5
41
6
17
7
14
2
8
23
9
30
10
21
11
10
12
15
3
1 2 3 4 5 6 7 8 9 10 11 12
51 46 20 34 41 17 14 23 30 21 10 15

swim()


nível
1
51
0
2
46
3
20
1
4
34
5
41
6
17
7
14
2
8
23
9
30
10
21
11
10
12
15
3
1 2 3 4 5 6 7 8 9 10 11 12
51 46 20 34 41 17 14 23 30 21 10 15

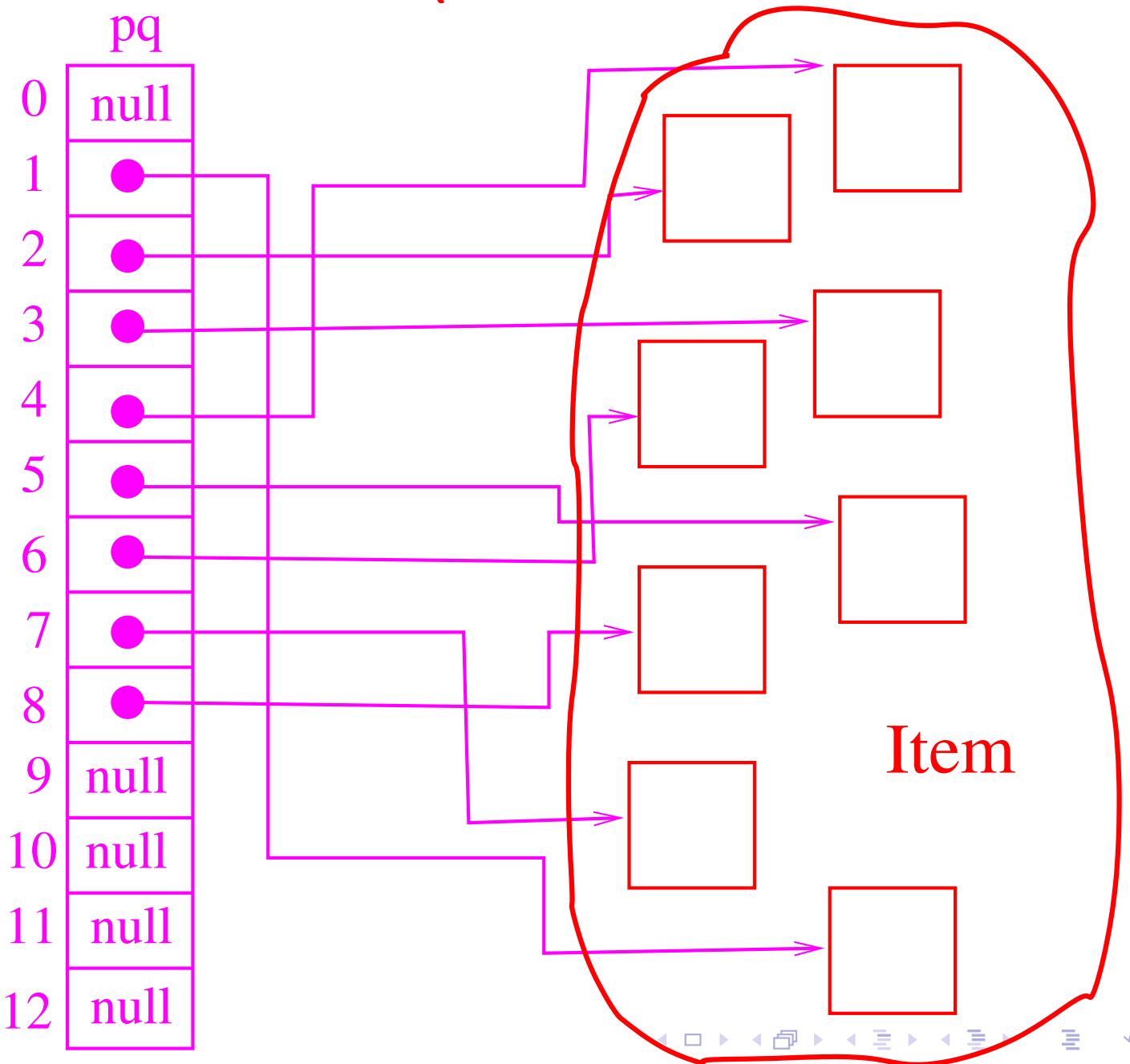
Função swim()

Função que recebe um max-heap a[1..m-1] e rearranja o vetor a[1..m] de modo que seja um max-heap.

```
private static
void swim (int f, Comparable a[]){
1   int p = f/2; Item x;
2   while (p > 1 && less(a[p],a[f])) {
3       x = a[p]; a[p] = a[f]; a[f] = x;
4       f = p; p = f/2; // sobe
    }
 }
```

Class MaxPQ: estrutura

n = 8



MaxPQ: less() e exch()

```
private boolean less(int i, int j) {
   return pq[i].compareTo(pq[j]) < 0;
}

private void exch(int i, int j) {
   Item t = pq[i];
   pq[i] = pq[j];
   pq[j] = t;
}
```

Class MaxPQ: esqueleto

```
public class MaxPQ<Item
                 extends Comparable<Item>> {
   private Item[] pq; // heap em pq[1..n]
   private int n = 0;
   public MaxPQ(int max) {...}
   public boolean isEmpty() {...}
   public int size() {...}
   public void insert(Item item) {...}
   public Item delMax() {...}
   private void swim(int k) {...}
   private void sink(int k) {...}
   private boolean less(int i, int j){...}
   private void exch(int i, int j){...}
}
```

MaxPQ: isEmpty() e size()

```
// construtor
public MaxPQ(int maxN) {
   pq = (Item[]) new Object[maxN+1];
}

public boolean isEmpty() {
   return n == 0;
}

public int size() {
   return n;
}
```

MaxPQ: insert() e delMax()

```
public void insert(Item item) {
   pq[++n] = item;
   swim(n);
}

public Item delMax() {
   Item max = pq[1];
   exch(1, n--);
   pq[n+1] = null; // avoid loitering
   sink(1);
   return max;
}
```

MaxPQ: swim() e sink()

```
private void swim(int f) {
   while (f > 1 && less(f/2, f)) {
       exch(f/2, f);
       f = f/2;
   }
}
private void sink(int p) {
   while (2*p <= n) {
       int f = 2*p;
       if (f < n && less(f, f+1)) f++;
       if (!less(p, f)) break;
       exch(p, f);
       p = f;
   }
```

MaxPQ: less() e exch()

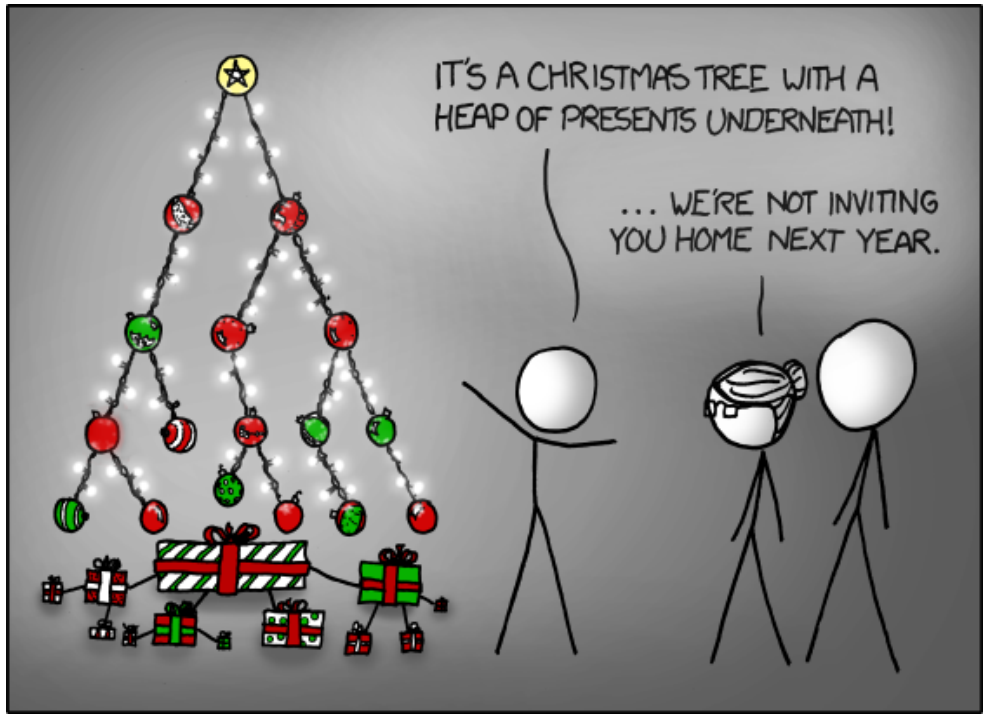
```
private boolean less(int i, int j) {
   return pq[i].compareTo(pq[j]) < 0;
}

private void exch(int i, int j) {
   Item t = pq[i];
   pq[i] = pq[j];
   pq[j] = t;
}
```

Conclusão

O consumo de tempo das operações envolvendo uma fila priorizada implementada como um heap é $\mathrm{O}(\lg \mathtt{n})$, onde `n` é o número de itens na fila.

O consumo de tempo dos métodos da classe `MaxPQ` é $\mathrm{O}(\lg \mathtt{n})$, onde `n` é o número de itens na fila.

PQ com itens mutáveis



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

Filas priorizadas, PF, Priority queues, S&W

PQ com itens mutáveis

Não sei se **PQ com itens mutáveis** é um bom nome para o que S&W chamam de *index priority queues*.

Em algumas aplicações é razoável permitirmos que o cliente altere a prioridade de um item que já esta na fila.

Uma maneira de lidar com isso é associar um único índice a cada item.

Já comentamos essa estratégia quando tratamos de union-find.

API PQ-máximo mutável

```
public class IndexMinPQ<Item extends
                        Comparable<Item>>
```

public class	IndexMinPQ	
	IndexMinPQ(int maxN)	
void	insert(int k, Item item)	insere
void	change(int k, Item item)	muda item
boolean	contains(int k)	k está associado?
void	delete(int k)	remove k e o item
Item	min()	menor item
int	minIndex()	índice do maior ite
int	delMin()	remove maior item
		retorna seu índice
boolean	isEmpty()	está vazia?
int	size()	número de itens

Cliente IndexMinPQ: class Multiway

No código a seguir, IMiPQ é uma abreviatura para IndexMinPQ.

Multiway é uma das classes do algs4.

O programa intercala (merge) arquivos ordenados.

Os nomes dos arquivos (streams) são lidos da linha de comando.

A linhas dos streams são lidos da entrada padrão.

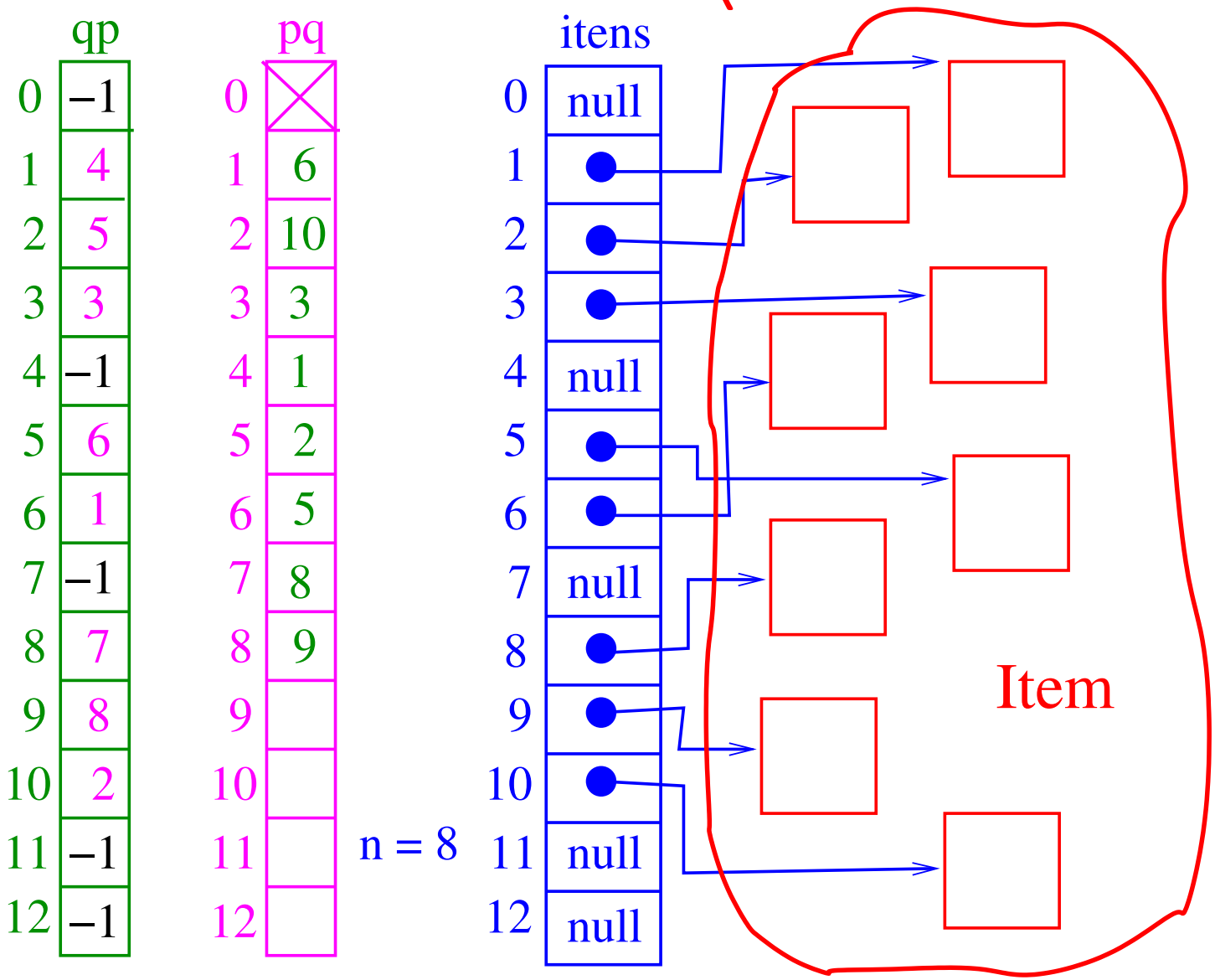
Cliente IndexMinPQ: class Multiway

```
/* Reads sorted text files;
* merges them into a sorted output;
* writes the results to StdOut.
*/
public static void main(String[] args) {
   int n = args.length;
   In[] streams = new In[n];
   for (int i = 0; i < n; i++)
       streams[i] = new In(args[i]);
   merge(streams);
}
```

Cliente IndexMinPQ: class Multiway

```
private static void merge(In[] streams) {
   int n = streams.length;
   IMiPQ<String> pq= new IMiPQ<String>(n);
   for (int i = 0; i < n; i++)
       if (!streams[i].isEmpty())
       pq.insert(i,streams[i].readString());
       while (!pq.isEmpty()) {
           StdOut.print(pq.min()+ " ");
           int i = pq.delMin();
           if(!streams[i].isEmpty())
              pq.insert(i,streams[i].readString()
       }
   StdOut.println();
}
```

Class IndexMinPQ: estrutura



IndexMinPQ: greater() e exch()

```
private boolean greater(int i, int j) {
   Item itemI = itens[pq[i]];
   Item itemJ = itens[pq[j]];
   return itemI.compareTo(itemJ) > 0;
}

private void exch(int i, int j) {
   Item t = pq[i];
   pq[i] = pq[j];
   pq[j] = t;
   // para consistência
   qp[pq[i]] = i;
   qp[pq[j]] = j;
}
```

Class IndexMinPQ: esqueleto

```
public class IndexMinPQ<Item
             extends Comparable<Item>> {
   private int[] pq; // heap em pq[1..n]
   // qp[pq[i]] = pq[qp[i]] = i
   private int[] qp;
   private Item[] itens;
   private int n = 0;

   public IndexMinPQ(int max) {...}
   public boolean isEmpty() {...}
   public int size() {...}
   public Item min() {...}
   public int delMin() {...}
   public int minIndex() {...}
}
```

Class IndexMinPQ: esqueleto

```
public class IndexMinPQ<Item
             extends Comparable<Item>> {
   public void insert(int k, Item a) {...}
   public void delete(int k) {...}
   public void change(int k, Item item) {}
   public boolean contains(int k) {...}

   // métodos administrativos
   private void swim(int k) {...}
   private void sink(int k) {...}
   private boolean less(int i, int j){...}
   private void exch(int i, int j){...}
}
```

IndexMinPQ: construtor

```
public IndexMinPQ(int maxN) {
   itens= (Item[]) new Comparable[maxN+1];
   pq = new int[maxN+1];
   qp = new int[maxN+1];
   for (int i = 0; i <= maxN; i++) {
       qp[i] = -1;
   }
}
```

IndexMinPQ: isEmpty() e size()

```
public boolean isEmpty() {
   return n == 0;
}

public int size() {
   return n;
}
```

IndexMinPQ: insert() e contains()

```
public void insert(int k, Item item) {
   n++;
   itens[k] = item;
   pq[n] = k;
   qp[k] = n;
   swim(n);
}
public boolean contains(int k) {
   return qp[k] != -1;
}
```

IndexMinPQ: delMin() e min()

```
public int delMin() {
   int indexOfMin = pq[1];
   exch(1, n--);
   sink(1);
   itens[pq[n+1]] = null; // loitering
   qp[pq[n+1]] = -1;
   return indexOfMin;
}

public Item min() {
   return itens[pq[1]];
}
```

IndexMinPQ: minIndex() e change()

```
public int minIndex() {
   return pq[1];
}

public void change(int k, Item item) {
   itens[k] = item;
   swim(qp[k]);
   sink(qp[k]);
}
```

IndexMinPQ: delete()

```
public void delete(int k) {
   int j = pq[n];
   exch(qp[k],n--);
   // arruma heap
   sink(qp[j]);
   swim(qp[j]);
   // destroi o rastros de k
   itens[k] = null;
   qp[k] = -1;
}
```

IndexMinPQ: swim() e sink()

```
private void swim(int f) {
   while (f > 1 && greater(f/2, f)) {
      exch(f/2, f);
      f = f/2;
   }
}
private void sink(int p) {
   while (2*p <= n) {
      int f = 2*p;
      if (f < n && greater(f, f+1)) f++;
      if (!greater(p, f)) break;
      exch(p, f);
      p = f;
   }
```

Conclusão

O consumo de tempo das operações envolvendo uma fila priorizada com itens mutáveis é $\mathrm{O}(\lg \mathtt{n})$, onde `n` é o número de itens na fila.

O consumo de tempo dos métodos da classe `IndexMinPQ` é $\mathrm{O}(\lg \mathtt{n})$, onde `n` é o número de itens na fila.