BC1424
Algoritmos e Estruturas de Dados I

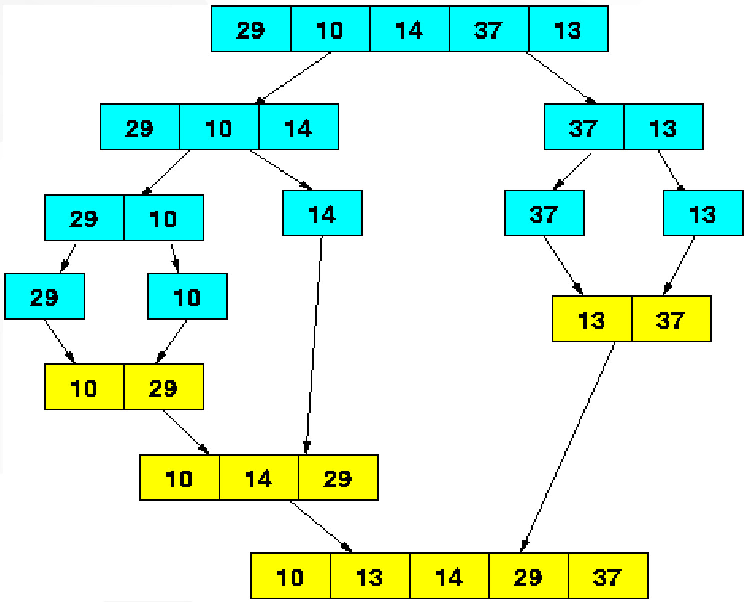
Aula 13:
Métodos eficientes de ordenação
(Heap Sort)

Prof. Jesús P. Mena-Chalco
jesus.mena@ufabc.edu.br

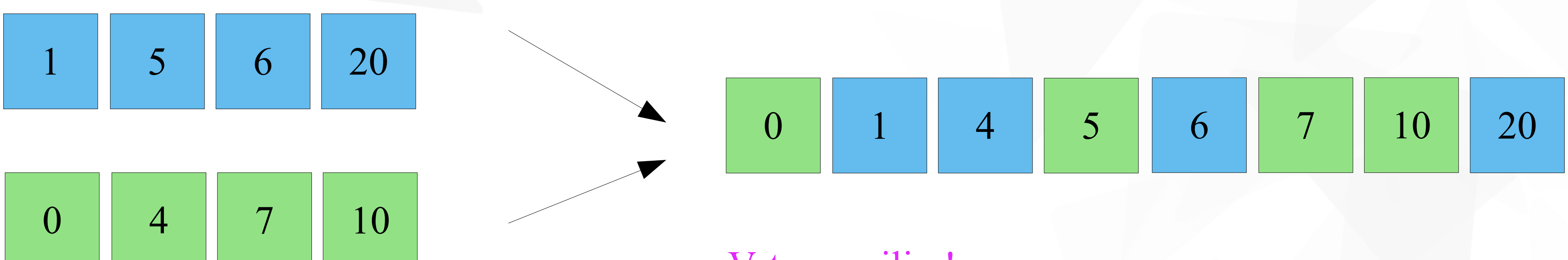
1Q-2015

Da última aula

Merge sort



MergeSort ilustra uma técnica (ou paradigma ou estratégia) de concepção de algoritmos eficientes: divisão e conquista.



Vetor auxiliar!

```c
void Intercala(int p, int q, int r, int v[]) {
    int i, j, k;
    int *w = (int *)malloc( (r-p)*sizeof(int) );

    i = p;
    j = q;
    k = 0;

    while (i<q && j<r) {
        if (v[i]<v[j])
            w[k++] = v[i++];
        else
            w[k++] = v[j++];
    }

    while(i<q)
        w[k++] = v[i++];

    while(j<r)
        w[k++] = v[j++];

    for (i=p; i<r; i++)
        v[i] = w[i-p];

    free(w);
}
```

Complexidade computacional, baseado em comparações, no pior caso = ***r-p***

Merge sort

```
void MergeSort (int p, int r, int v[]) {
    if (p<r-1) {
        int q = (p+r)/2;
        MergeSort(p, q, v);
        MergeSort(q, r, v);
        Intercala(p, q, r, v);
    }
}
```

Complexidade computacional, baseado em comparações, O (n lg(n))

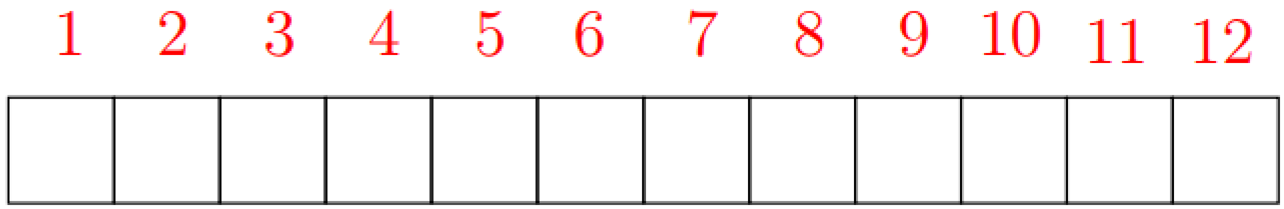
onde n = r-p;

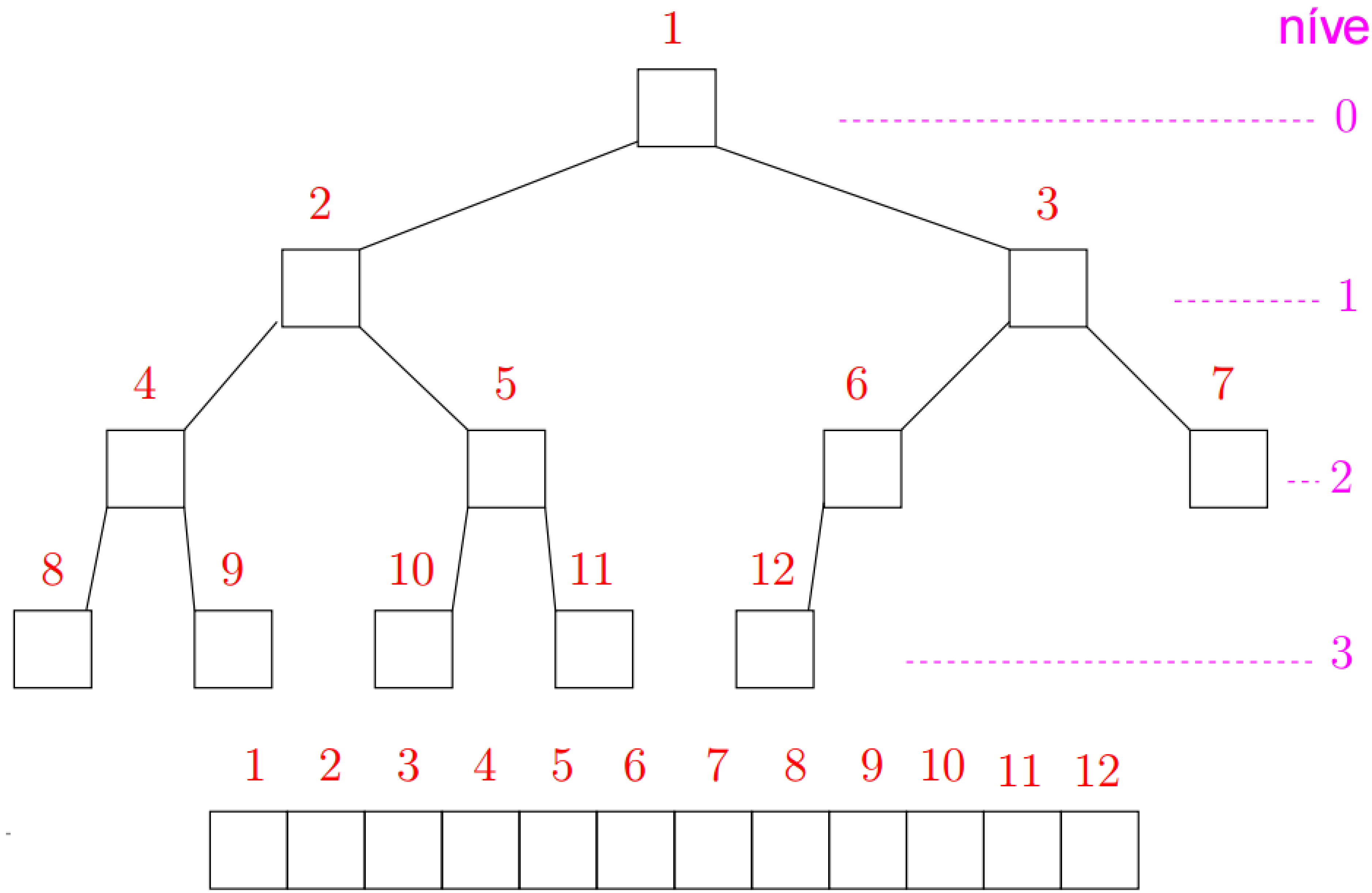
Merge sort

```
int main()
{
    int v[] = {10, -9, 2, 99, 3, 20, 111, -900};
    int n=sizeof(v)/sizeof(v[0]);

    ImprimeVetor(v, n);
    MergeSort(0, n, v);
    ImprimeVetor(v, n);
}
```

```
10 -9 2 99 3 20 111 -900
-900 -9 2 3 10 20 99 111
```

Representação de árvores em vetores


1 2 3 4 5 6 7 8 9 10 11 12



nível
1
2
3
4
5
6
7
8
9
10
11
12
0
1
2
3
1 2 3 4 5 6 7 8 9 10 11 12

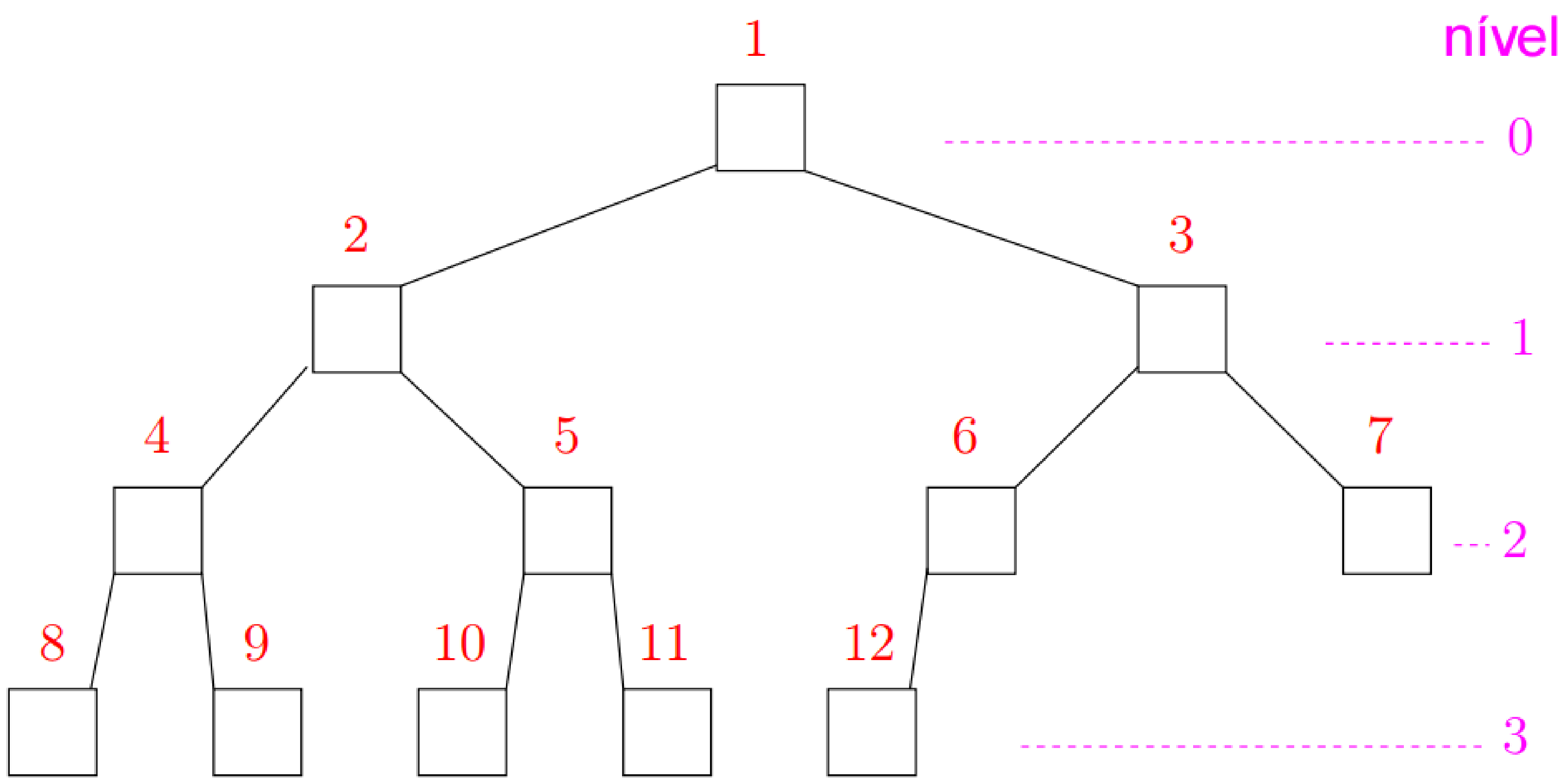
Pais e filhos

- `A[1..m]` é um vetor representado por uma árvore
- Diremos que para qualquer índice ou nó `i`:
  - `floor(i/2)` é o pai de `i`.
  - 2*i é o filho esquerdo de `i`.
  - 2*i+1 é o filho direito de `i`.
- O nó `1` não tem pai e é chamado de nó raiz.
- Um nó `i` só tem filho esquerdo se $2*i \leq m$.
- Um nó `i` só tem filho direito se $2*i+1 \leq m$.
- Um no `i` é um nó folha, se não tem filhos: $2*i > m$.

Pais e filhos

- Todo nó `i` é raiz da sub-árvore formada por:

$$A[i, 2i, 2i+1, 4i, 4i+1, 4i+2, 4i+3, 8i, \ldots, 8i+7, \ldots]$$



Propriedades

`A[1..m]` é um vetor representado por uma árvore

filho esquerdo de i:	$2\,i$
filho direito de i:	$2\,i+1$
pai de i:	$\lfloor i/2 \rfloor$
nível da raiz:	0
nível de i:	$\lfloor \lg i \rfloor$
altura da raiz:	$\lfloor \lg m \rfloor$
altura da árvore:	$\lfloor \lg m \rfloor$
altura de i:	$\lfloor \lg(m/i) \rfloor$
altura de uma folha:	0
total de nós de altura h	$\leq \lceil m/2^{h+1} \rceil$

Max-heap

Um vetor é um Max-heap se todo pai é pelo menos tão valioso quanto qualquer dos seus dois filhos.

Max-heap

Um vetor $A[1 \dots m]$ é um max-heap se

$$A[\lfloor i/2 \rfloor] \geq A[i]$$

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

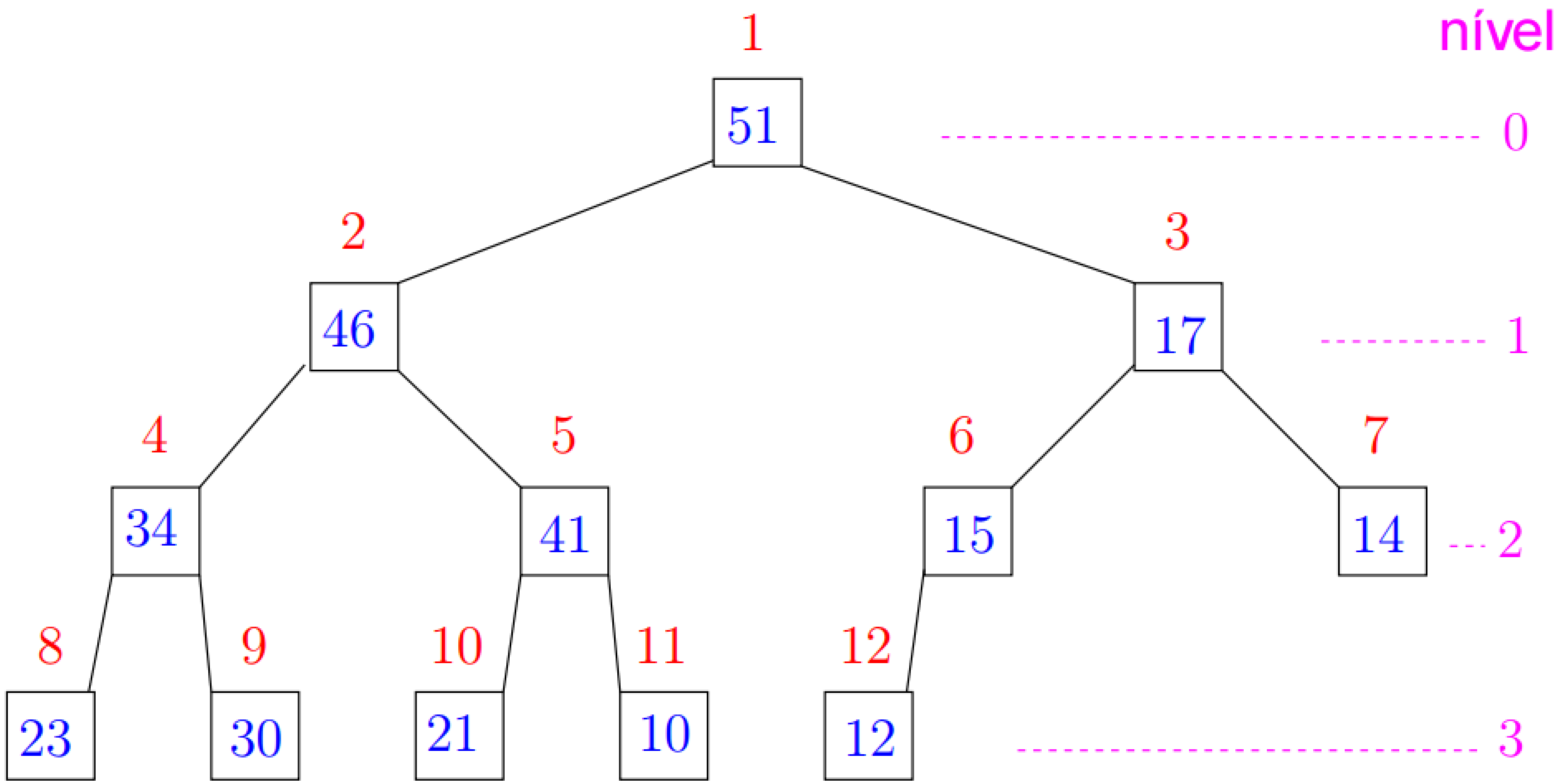
De uma forma mais geral, $A[j \dots m]$ é um max-heap se

$$A[\lfloor i/2 \rfloor] \geq A[i]$$

para todo $i = 2j, 2j+1, 4j, \dots, 4j+3, 8j, \dots, 8j+7, \dots$.

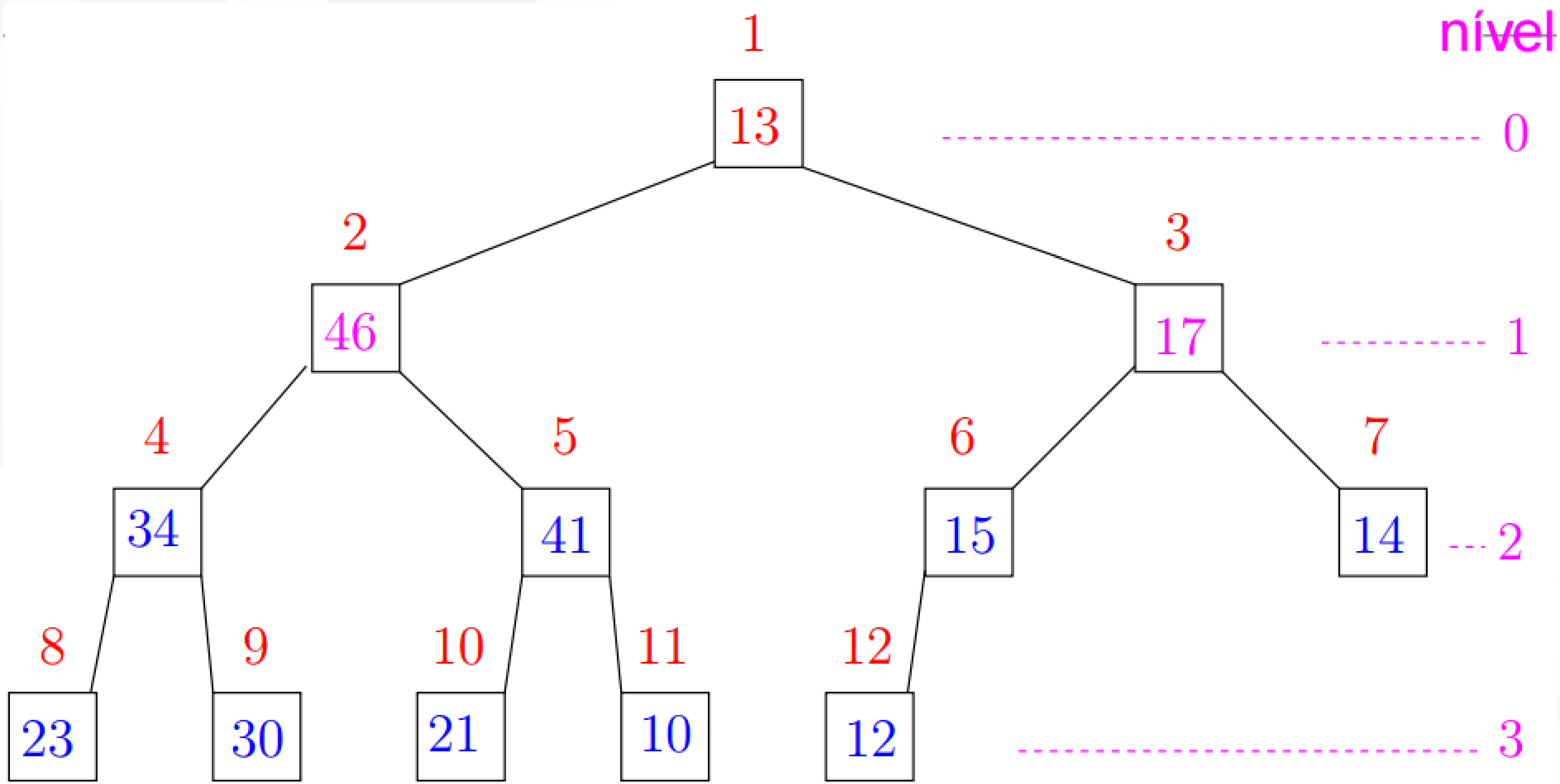
Neste caso também diremos que a subárvore com raiz j é um max-heap.

Max-heap



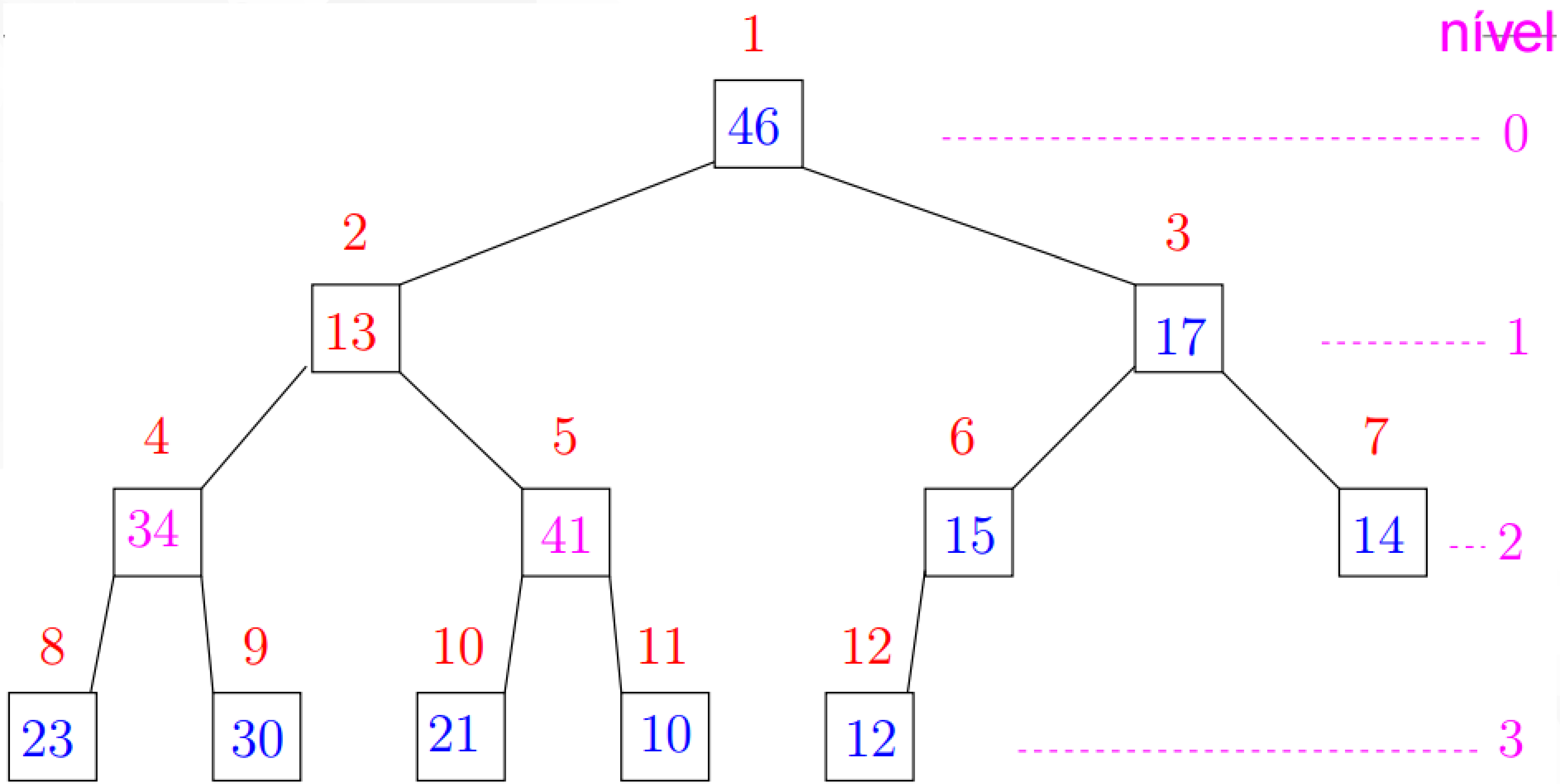
1	2	3	4	5	6	7	8	9	10	11	12
51	46	17	34	41	15	14	23	30	21	10	12

Manipulação de Max-heap



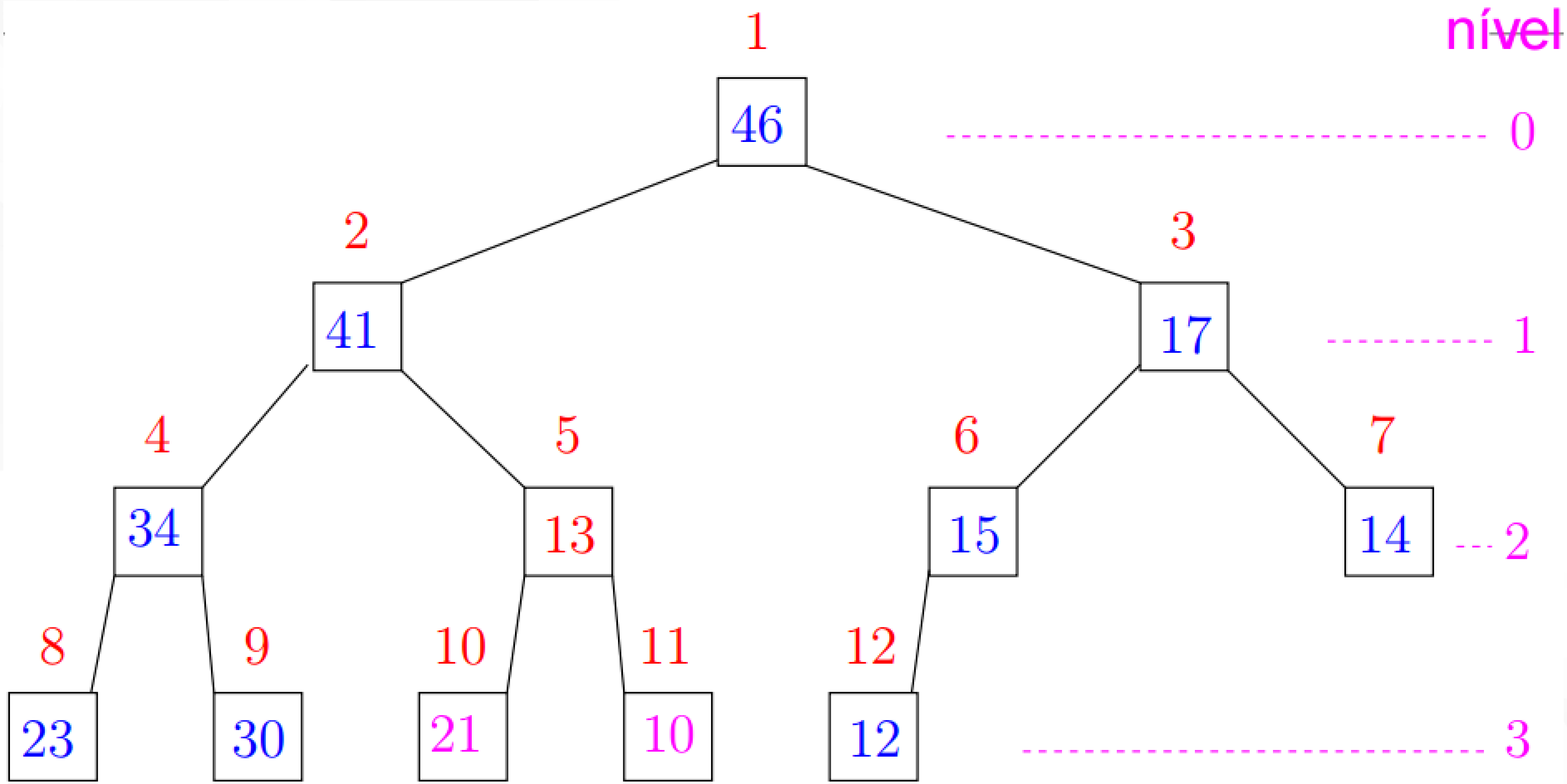
1	2	3	4	5	6	7	8	9	10	11	12
13	46	17	34	41	15	14	23	30	21	10	12

Manipulação de Max-heap



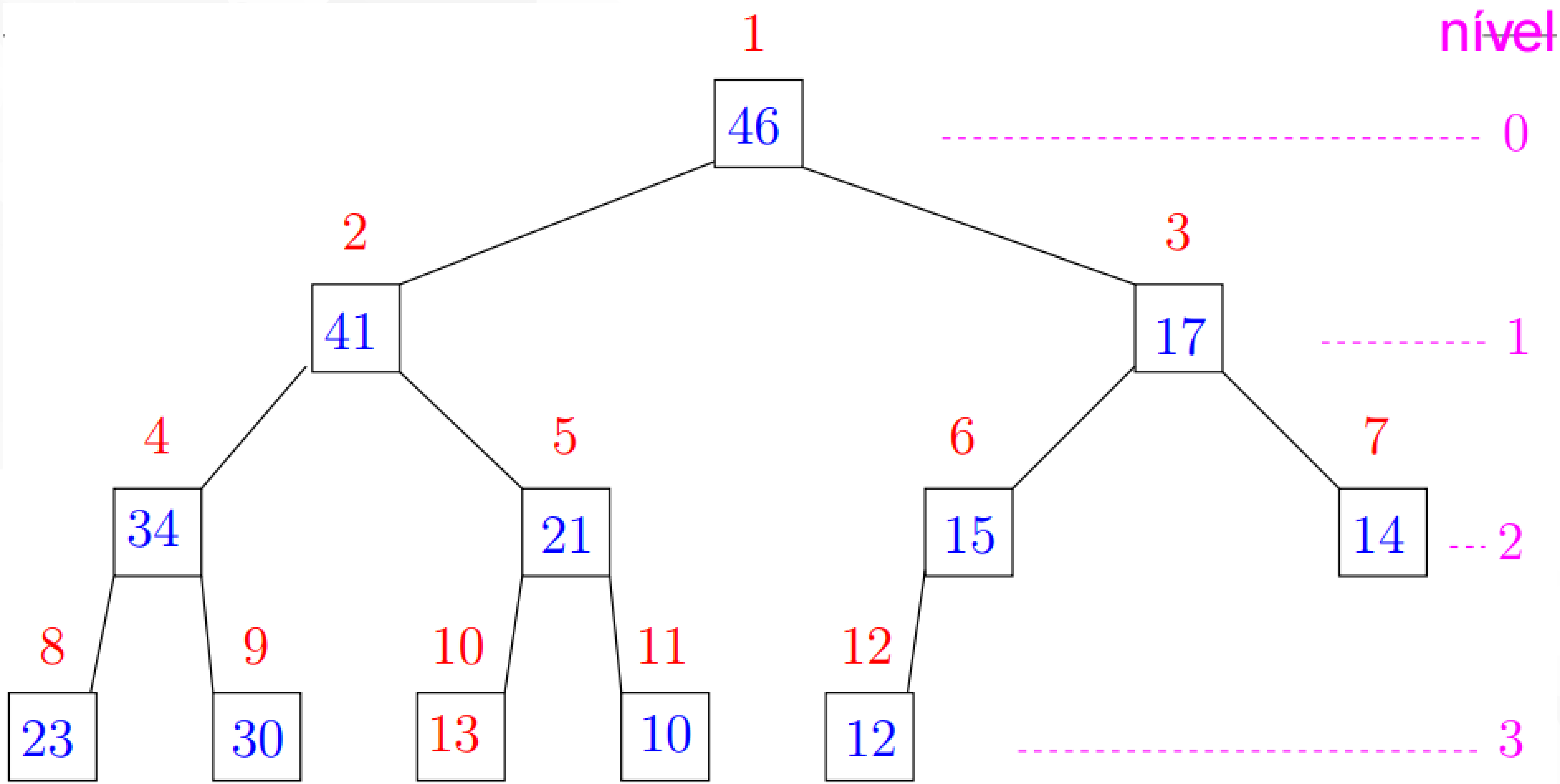
1	2	3	4	5	6	7	8	9	10	11	12
46	13	17	34	41	15	14	23	30	21	10	12

Manipulação de Max-heap



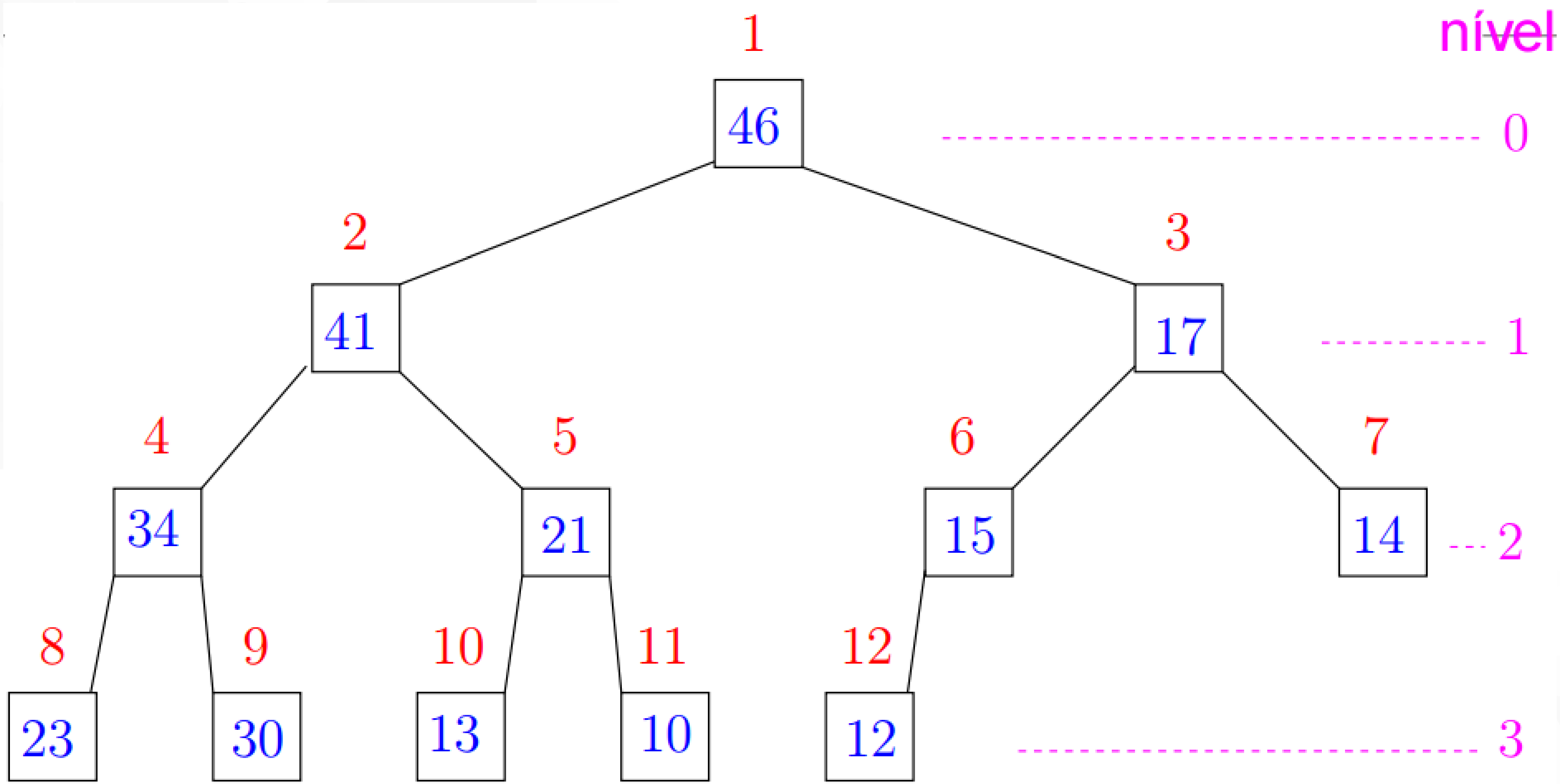
1	2	3	4	5	6	7	8	9	10	11	12
46	41	17	34	13	15	14	23	30	21	10	12

Manipulação de Max-heap



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

Manipulação de Max-heap



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

Max-Heapify

- Implemente a função MaxHeapify.
- **Dica**: Pense em uma abordagem recursiva

Assinatura:

```
void MaxHeapify (int A[], int m, int i)
```

Max-Heapify

Recebe $A[1\,.\,.\,m]$ e $i \geq 1$ tais que subárvores com raiz $2i$ e $2i+1$ são max-heaps e rearranja A de modo que subárvore com raiz i seja max-heap.

MAX-HEAPIFY (A, m, i)

$e \leftarrow 2i$
$d \leftarrow 2i+1$
se $e \leq m$ e $A[e] > A[i]$
    então $maior \leftarrow e$
    senão $maior \leftarrow i$
se $d \leq m$ e $A[d] > A[maior]$
    então $maior \leftarrow d$
se $maior \neq i$
    então $A[i] \leftrightarrow A[maior]$
        MAX-HEAPIFY $(A, m, maior)$

Max-Heapify

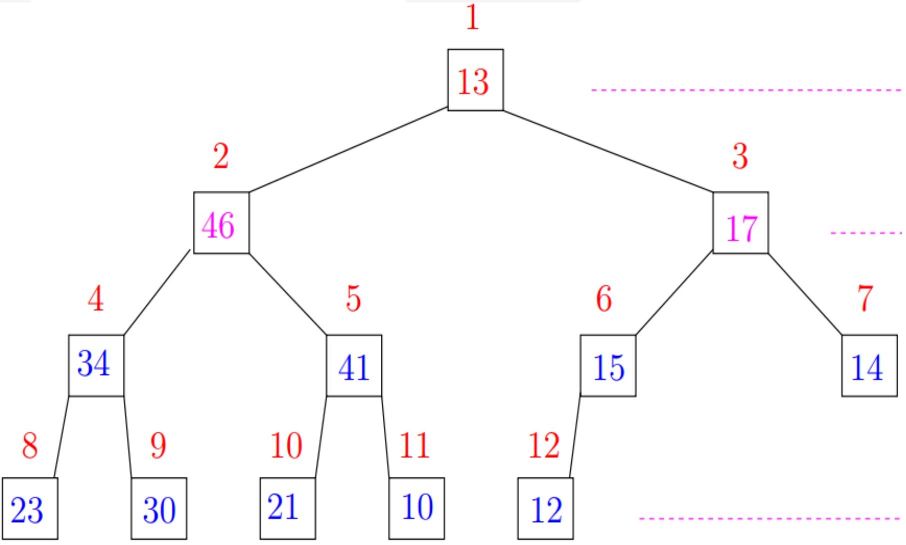
```
/* Funcao que recebe um vetor A[1..m] e                    *
 * rearranja-o de modo que a subarvore i seja max-heap */

void MaxHeapify (int A[], int m, int i) {
    int e, d, maior, aux;

    e = 2*i;
    d = 2*i+1;

    if (e<=m && A[e]>A[i])
        maior = e;
    else
        maior = i;

    if (d<=m && A[d]>A[maior])
        maior = d;

    if (maior!=i) {
        aux = A[maior];
        A[maior] = A[i];
        A[i] = aux;

        MaxHeapify(A, m, maior);
    }
}
```

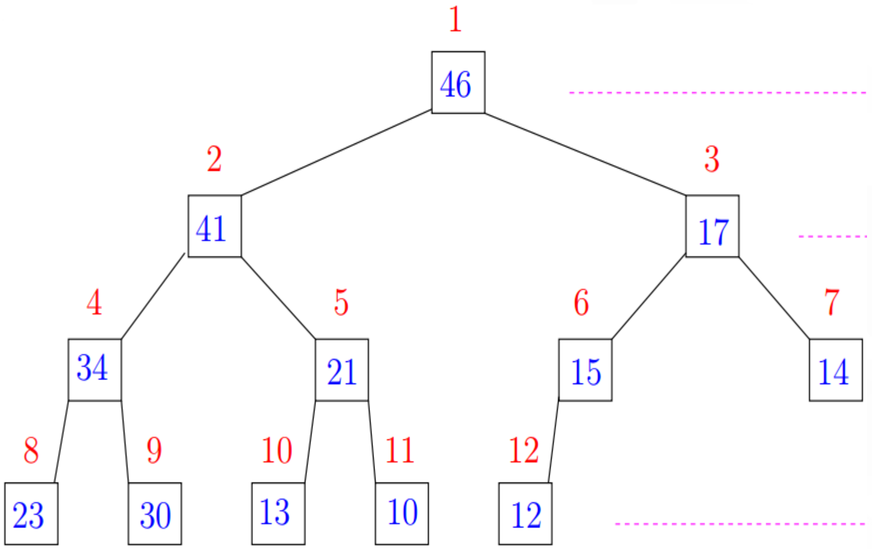
Consumo de tempo é proporcional a altura da árvore = *lg(m)*

heapify.c (Tidia)

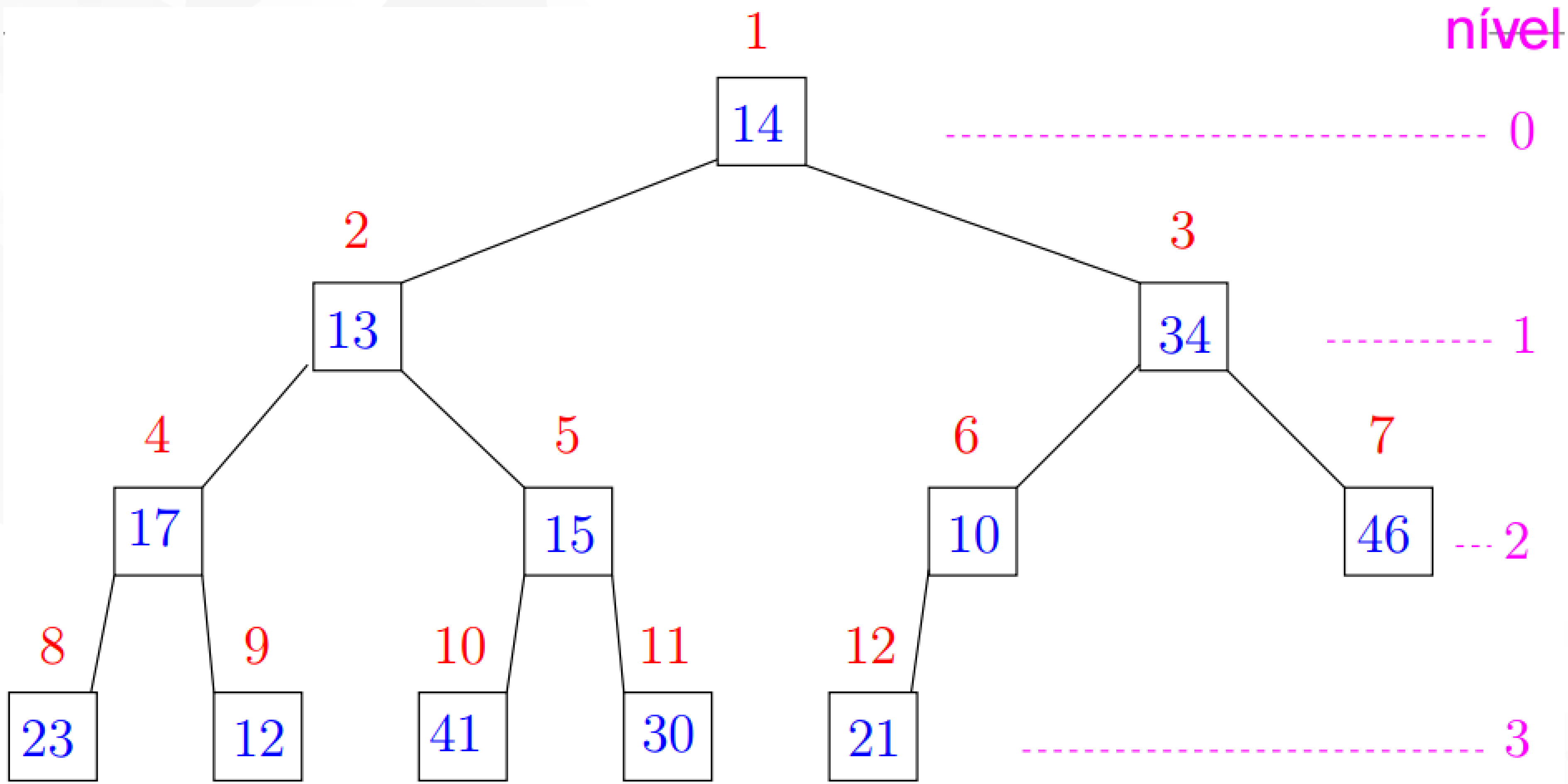
```c
int main()
{
    int A[] = {-1,13,46,17,34,41,15,14,23,30,21,10,12};
    int m=sizeof(A)/sizeof(A[0]);

    ImprimeVetor(A, m);
    MaxHeapify(A, m, 1);
    ImprimeVetor(A, m);
}
```

```
-1 13 46 17 34 41 15 14 23 30 21 10 12
-1 46 41 17 34 21 15 14 23 30 13 10 12
```

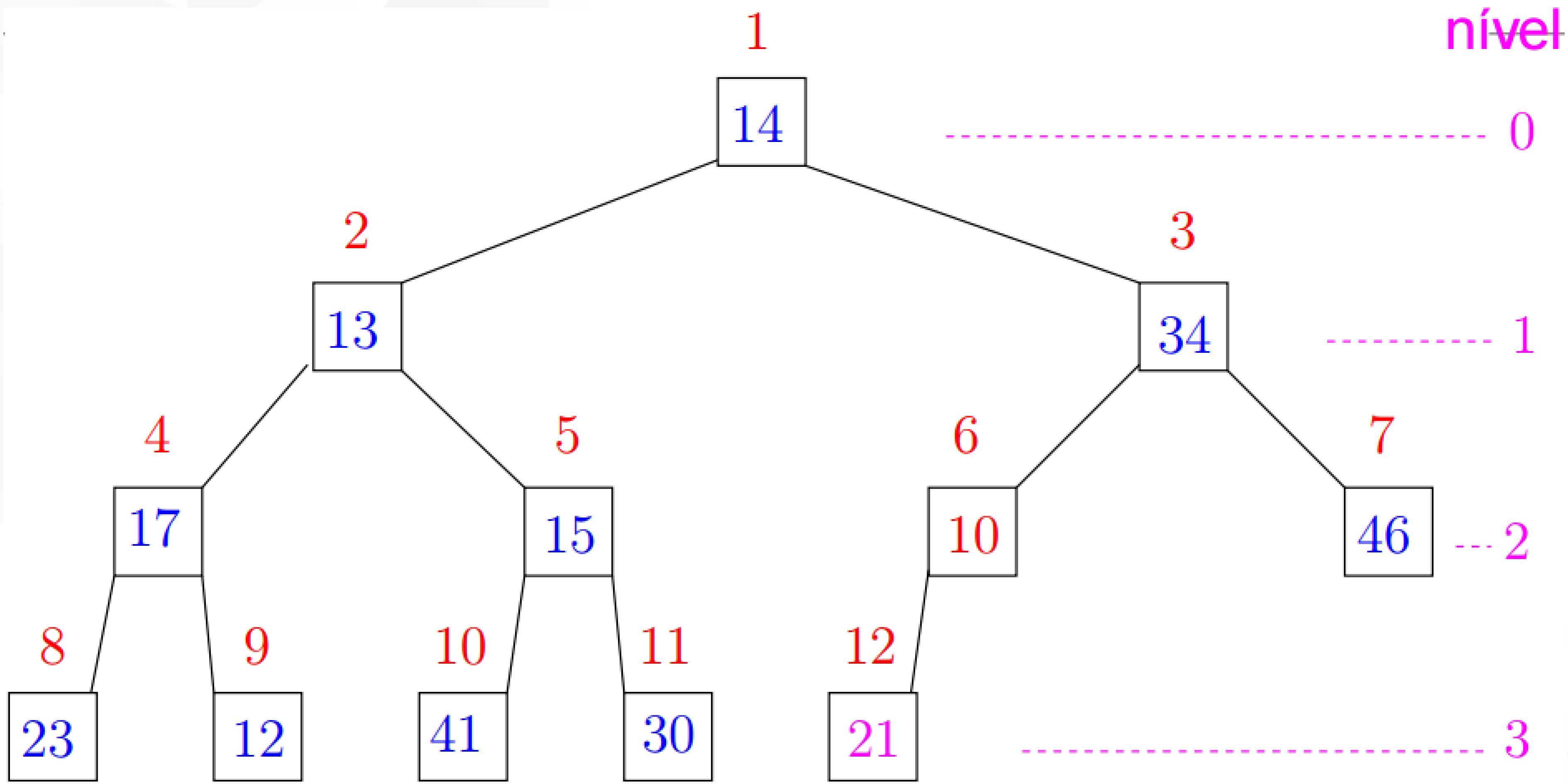




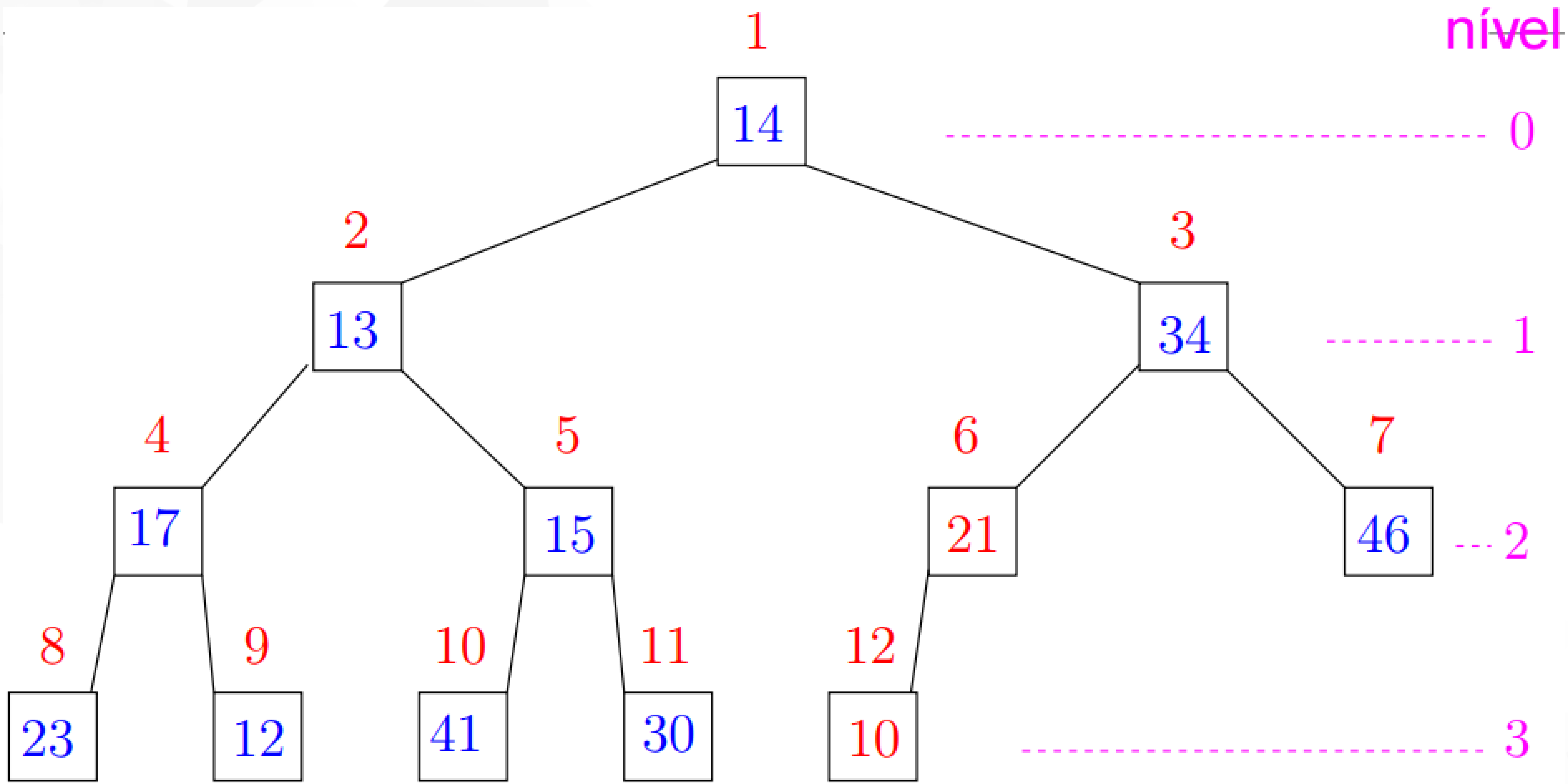
Construção de um Heap-Max



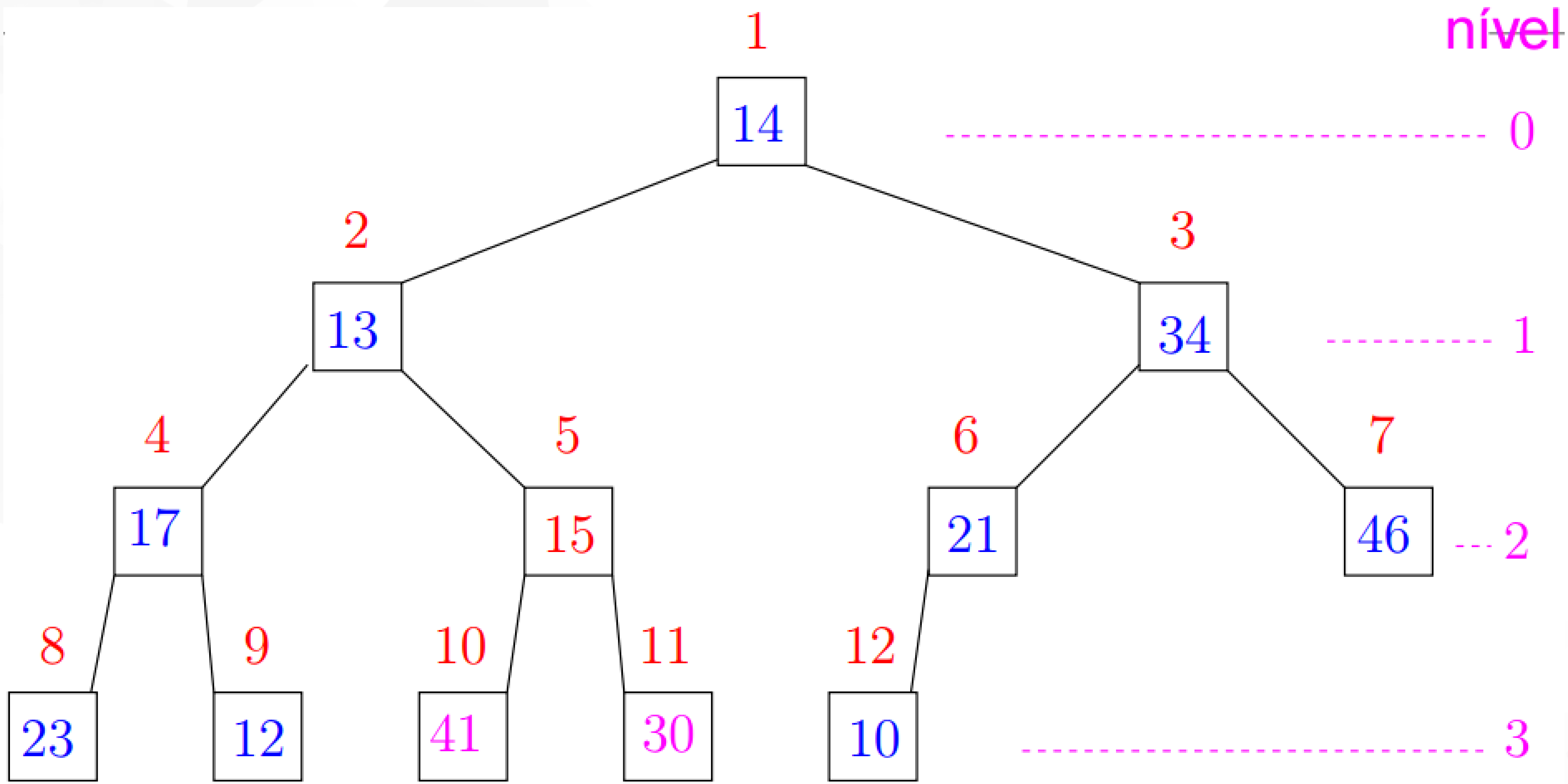
1	2	3	4	5	6	7	8	9	10	11	12
14	13	34	17	15	10	46	23	12	41	30	21



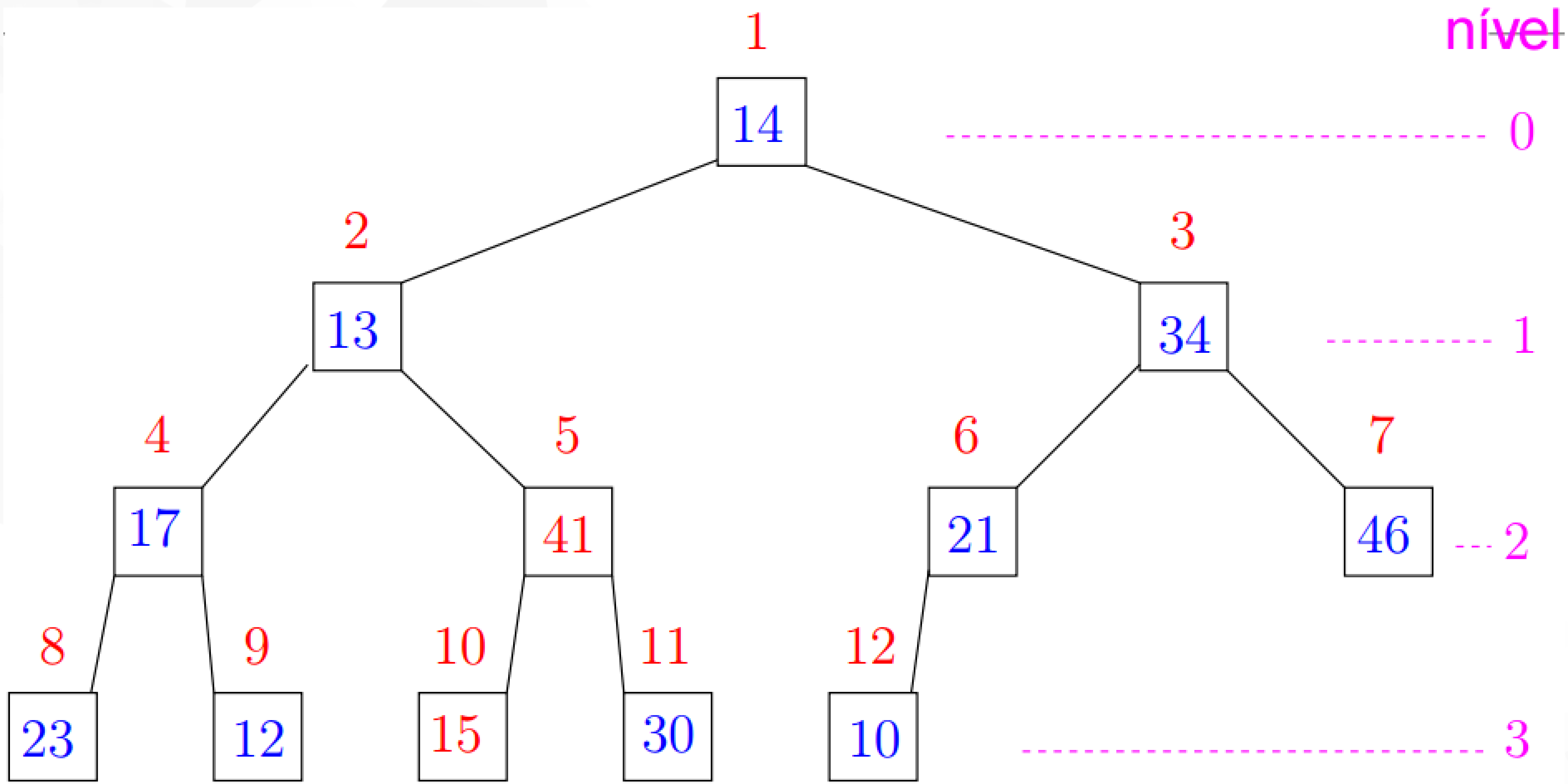
1	2	3	4	5	6	7	8	9	10	11	12
14	13	34	17	15	10	46	23	12	41	30	21



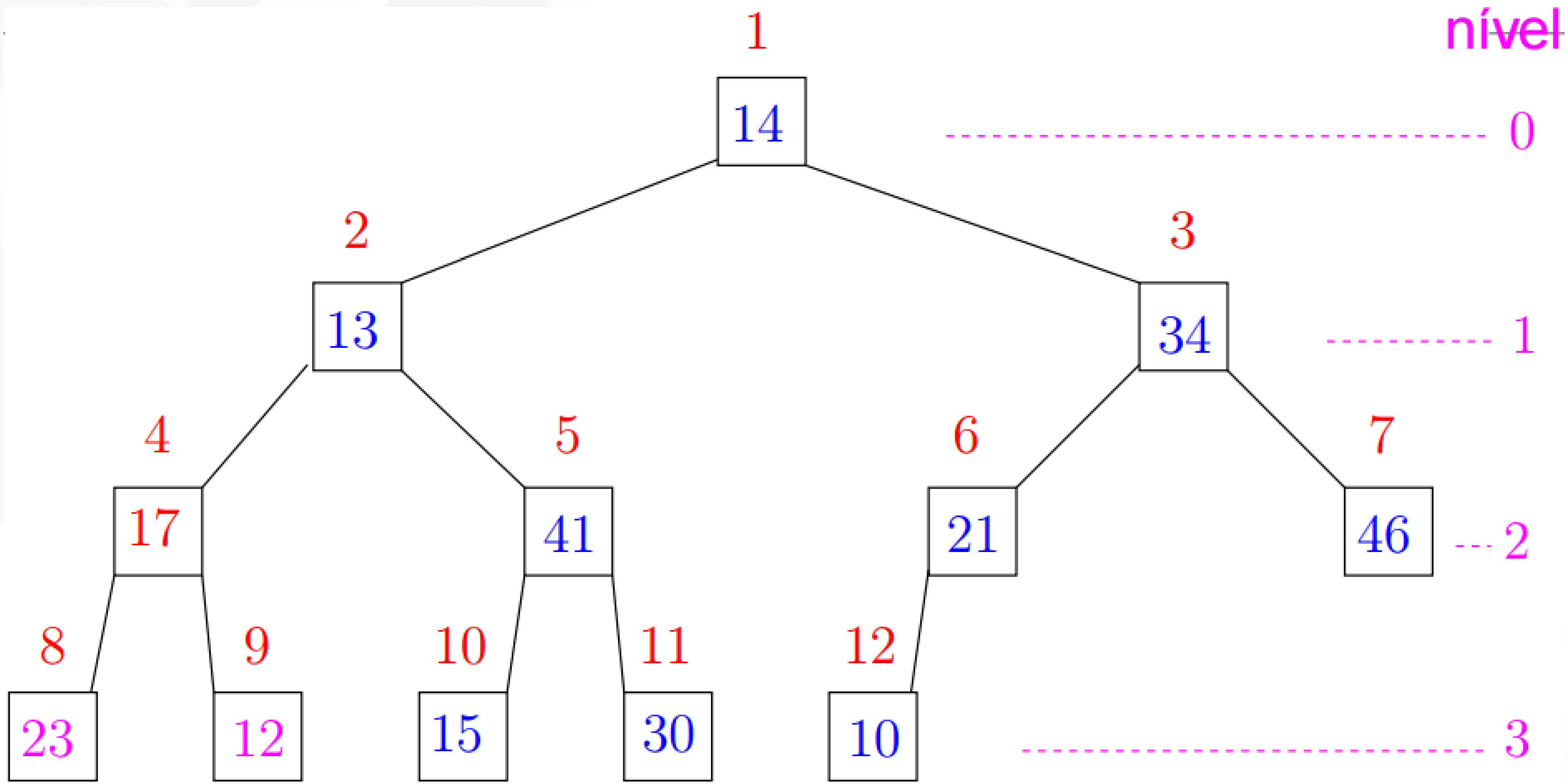
1	2	3	4	5	6	7	8	9	10	11	12
14	13	34	17	15	21	46	23	12	41	30	10



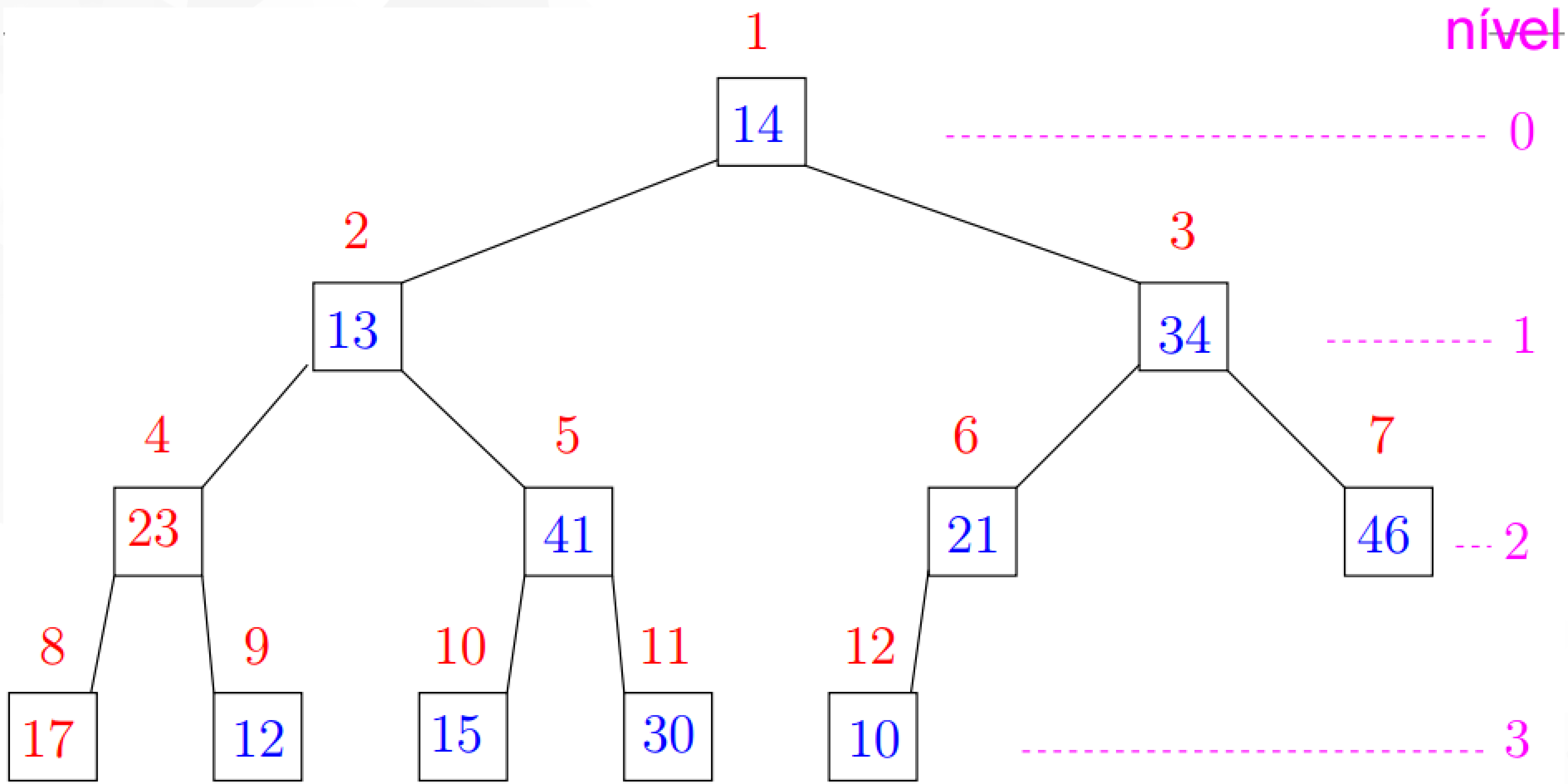
1	2	3	4	5	6	7	8	9	10	11	12
14	13	34	17	15	21	46	23	12	41	30	10



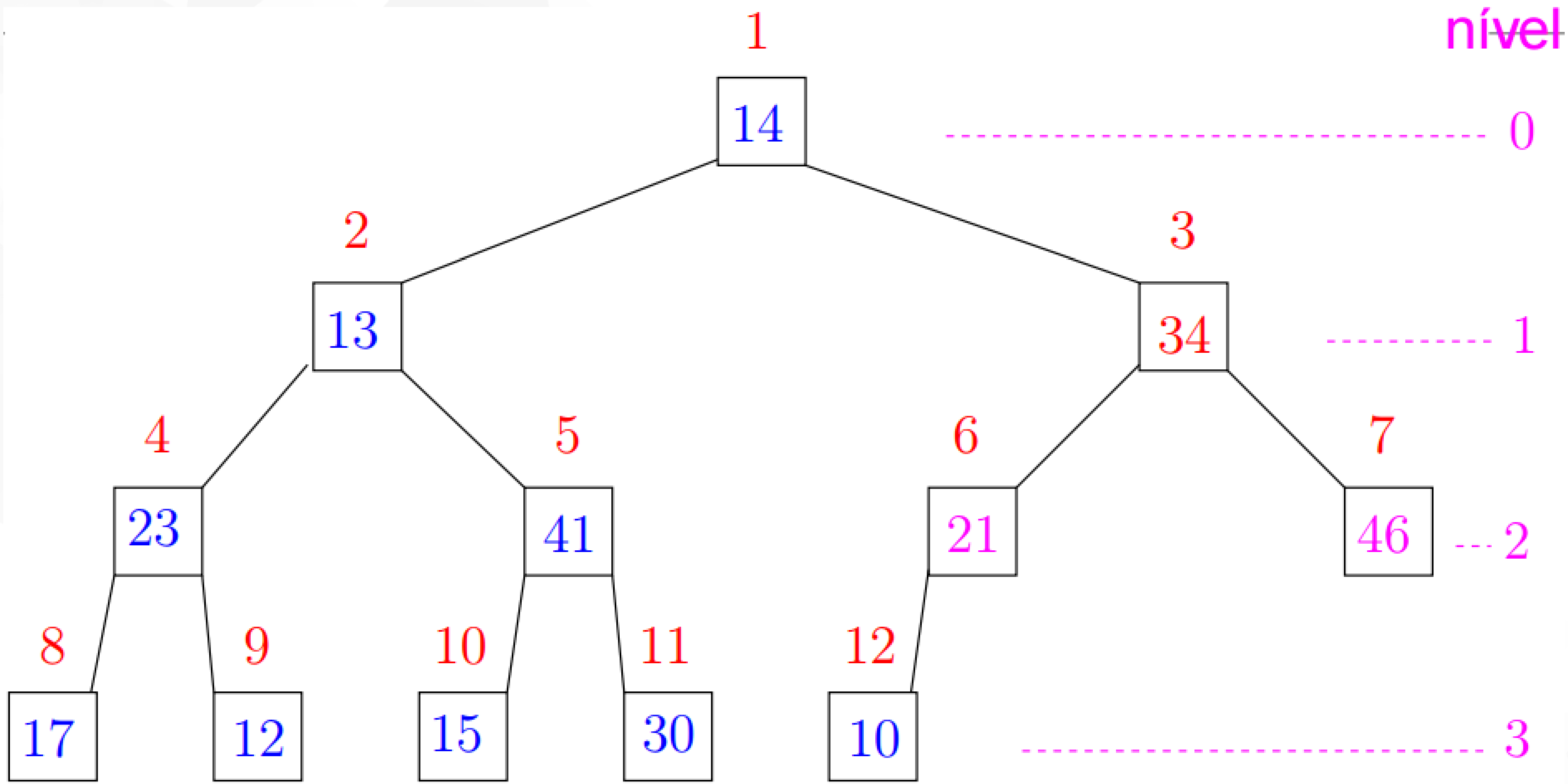
1	2	3	4	5	6	7	8	9	10	11	12
14	13	34	17	41	21	46	23	12	15	30	10



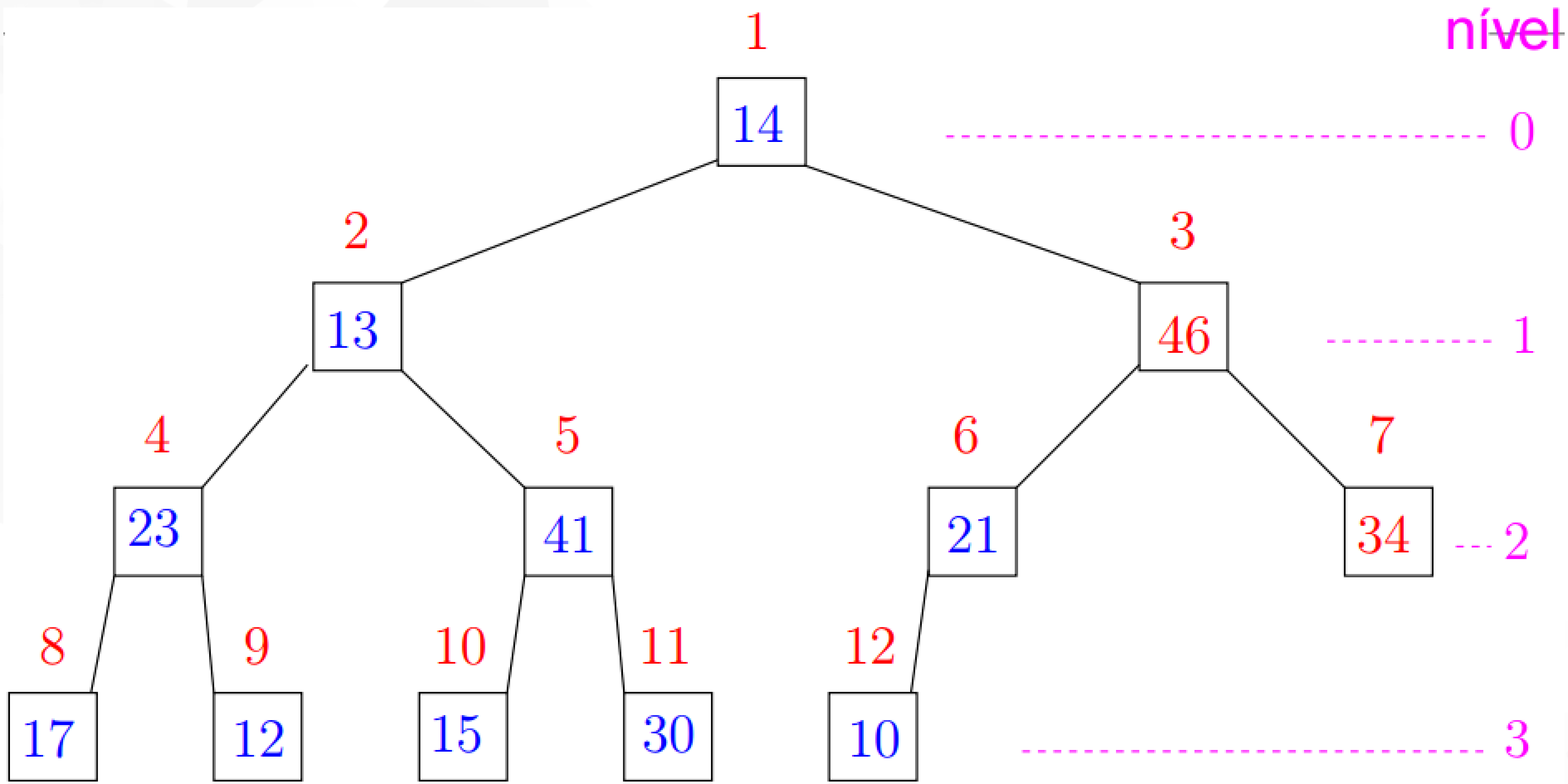
1	2	3	4	5	6	7	8	9	10	11	12
14	13	34	17	41	21	46	23	12	15	30	10



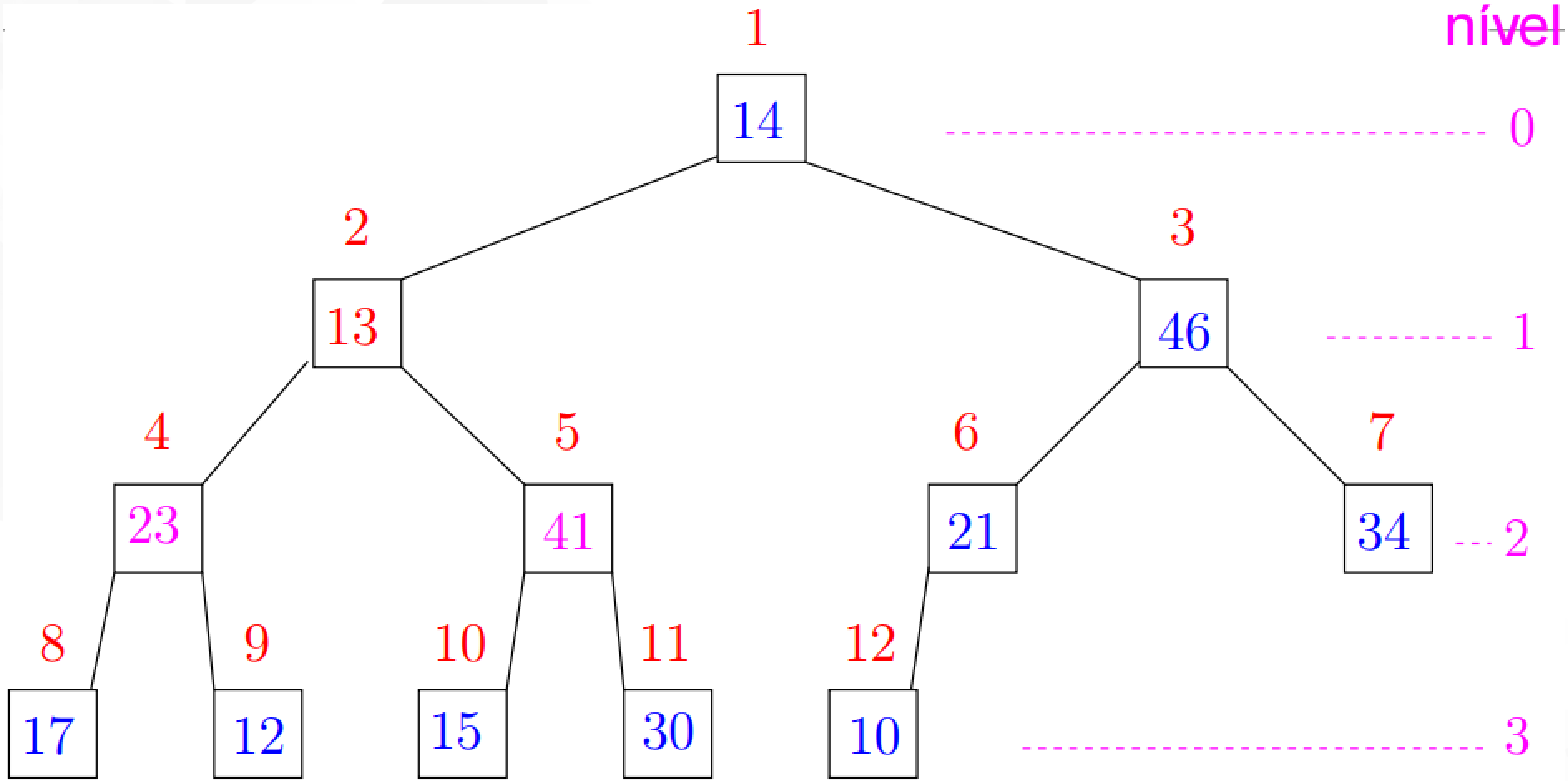
1	2	3	4	5	6	7	8	9	10	11	12
14	13	34	23	41	21	46	17	12	15	30	10



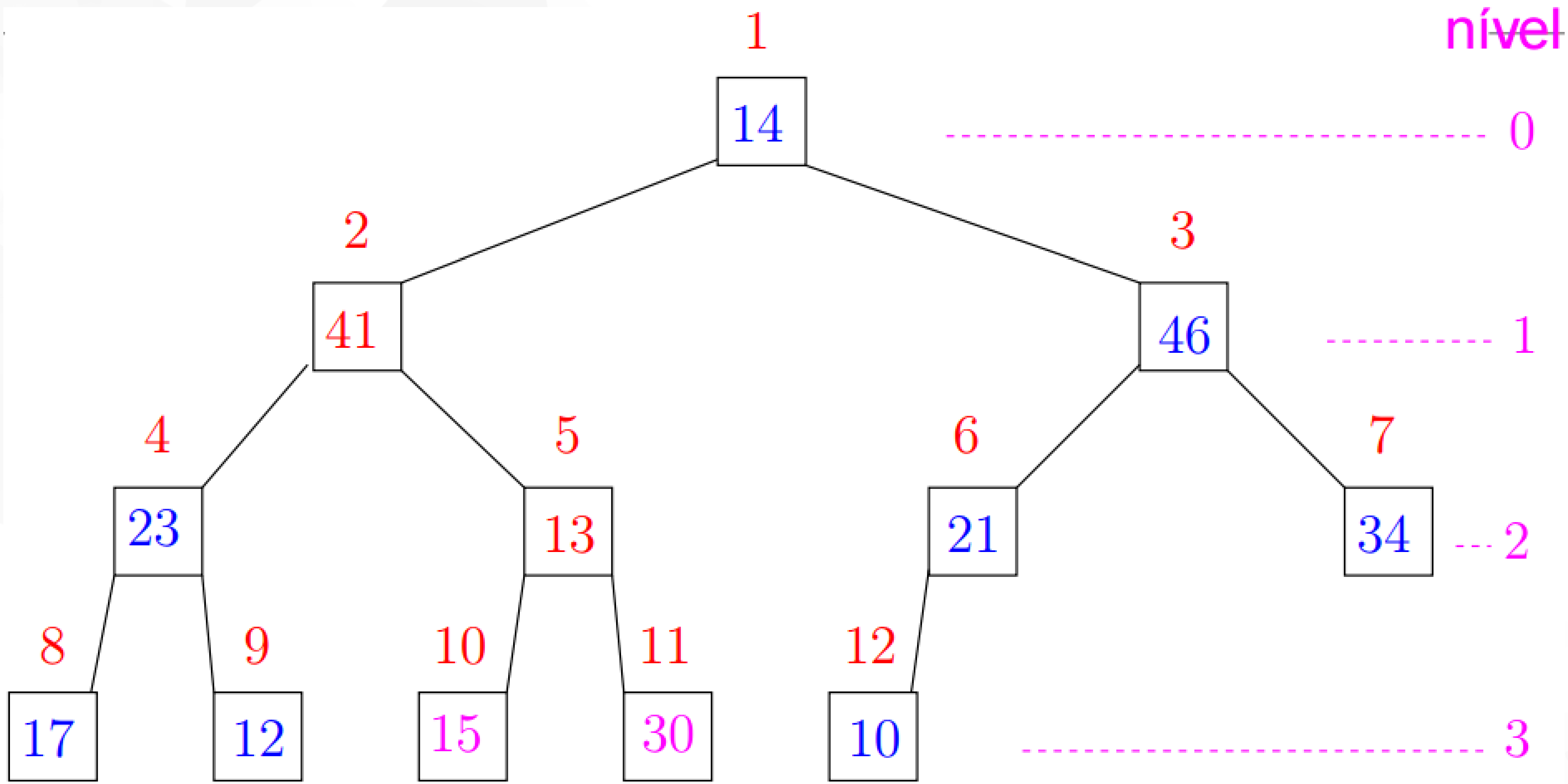
1	2	3	4	5	6	7	8	9	10	11	12
14	13	34	23	41	21	46	17	12	15	30	10



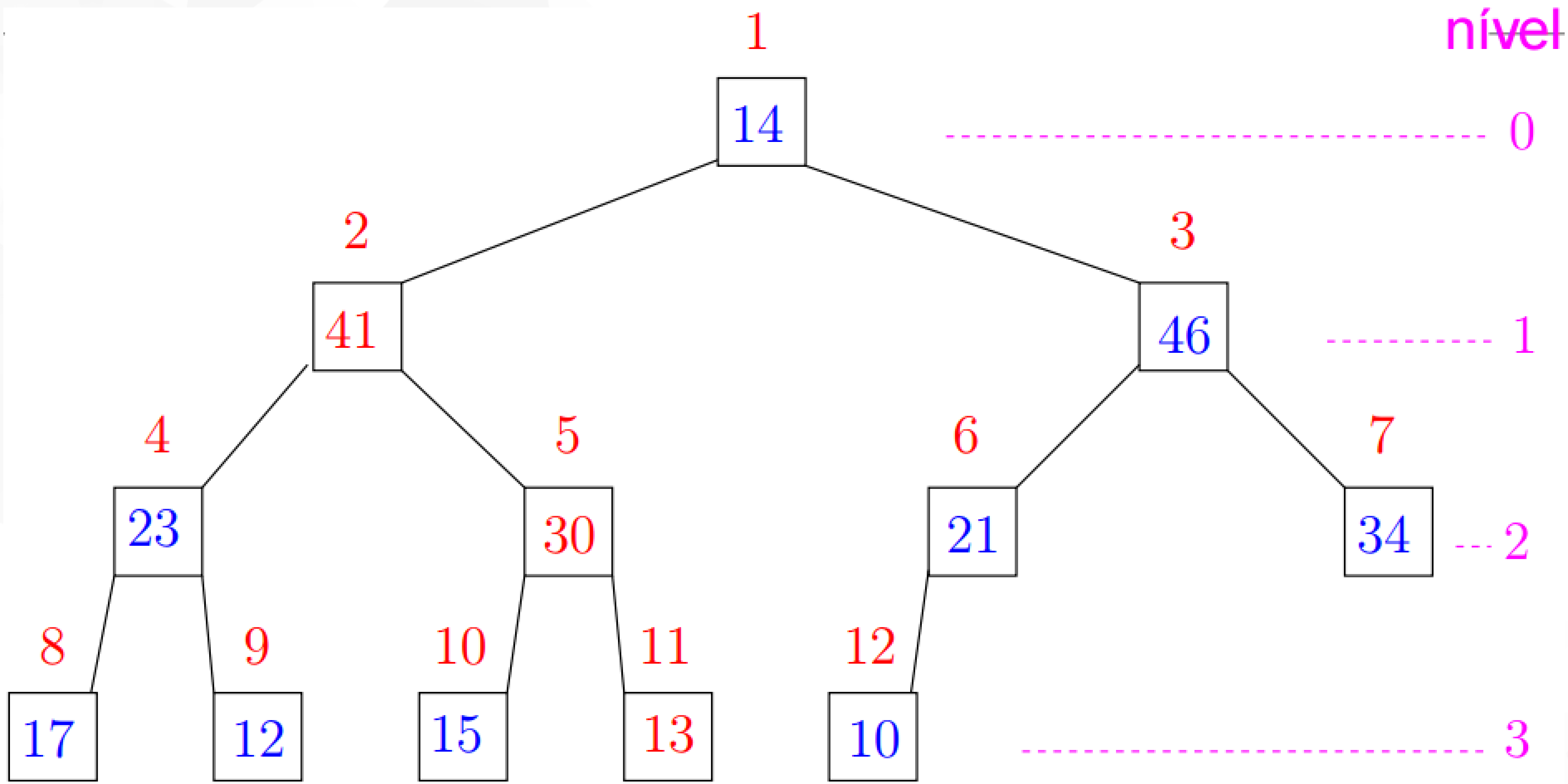
1	2	3	4	5	6	7	8	9	10	11	12
14	13	46	23	41	21	34	17	12	15	30	10



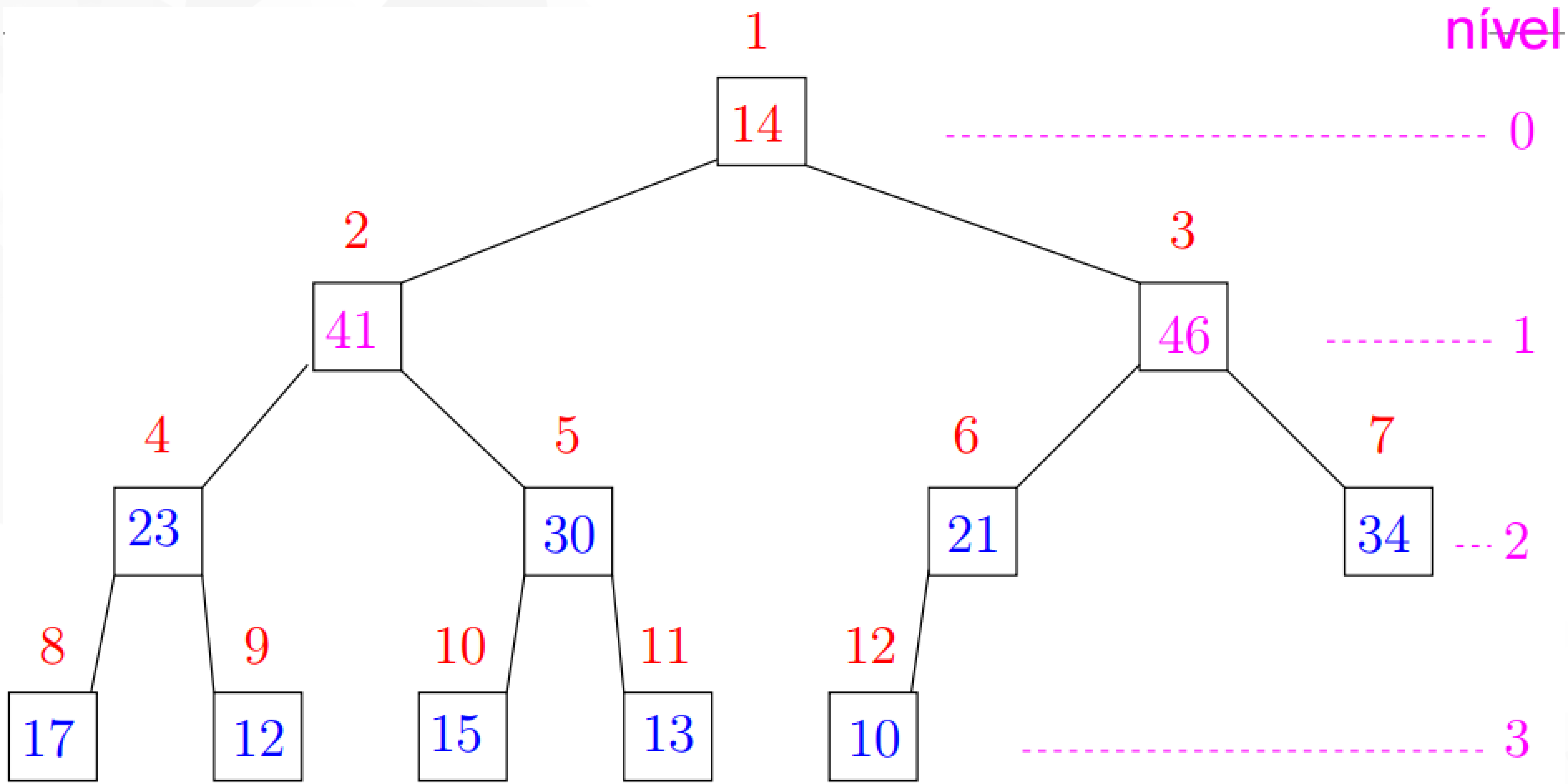
1	2	3	4	5	6	7	8	9	10	11	12
14	13	46	23	41	21	34	17	12	15	30	10



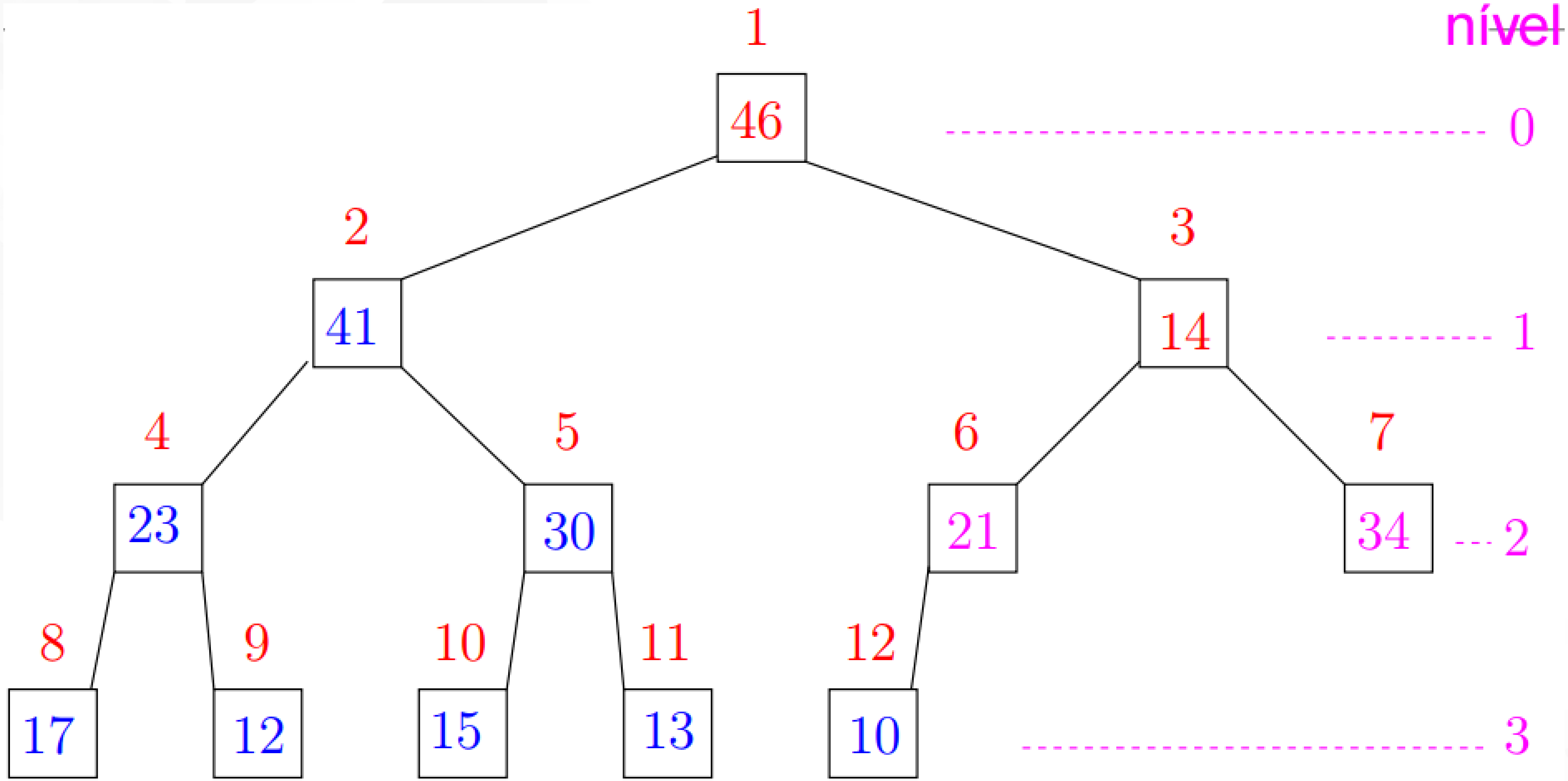
1	2	3	4	5	6	7	8	9	10	11	12
14	41	46	23	13	21	34	17	12	15	30	10



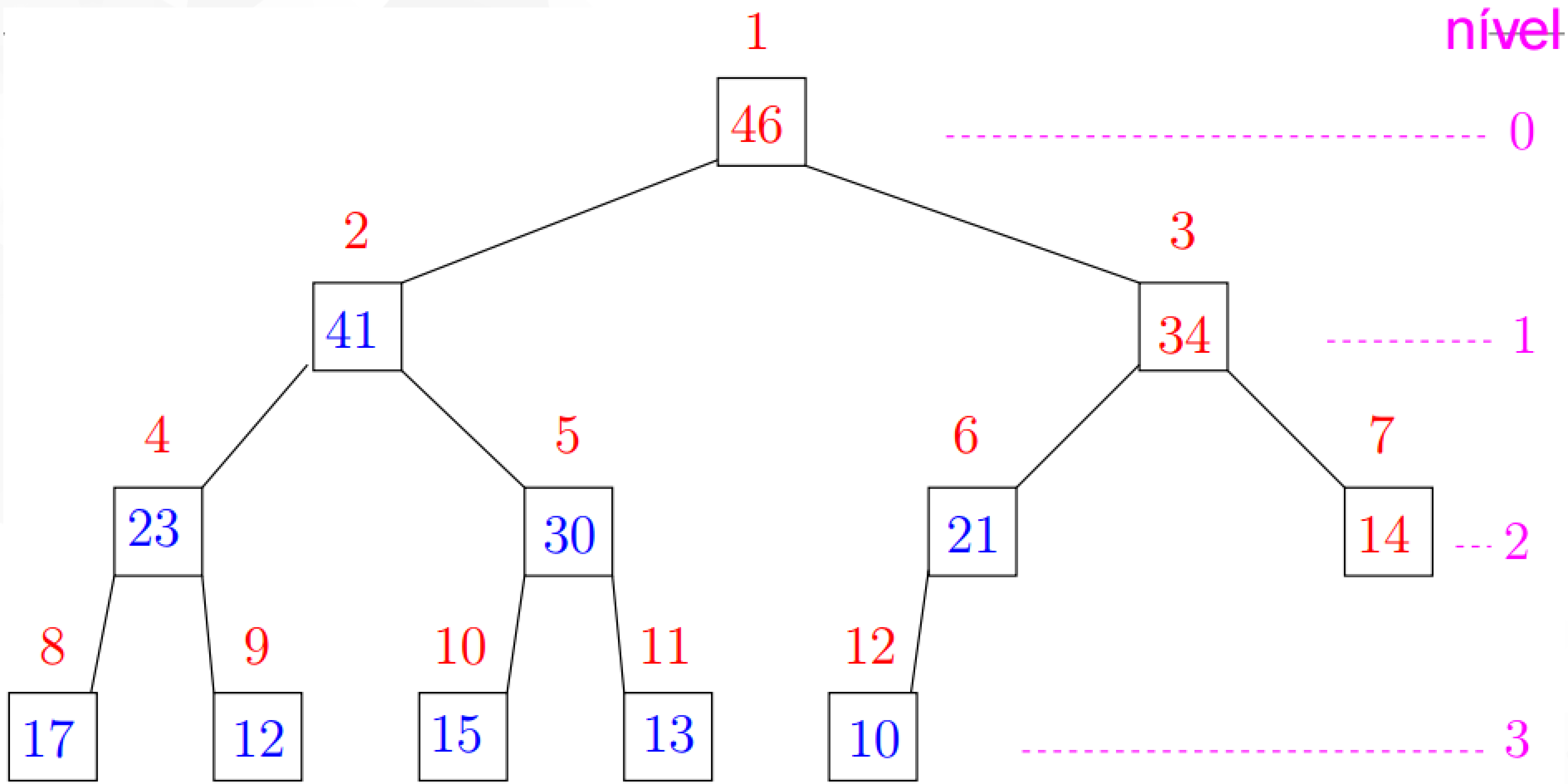
1	2	3	4	5	6	7	8	9	10	11	12
14	41	46	23	30	21	34	17	12	15	13	10



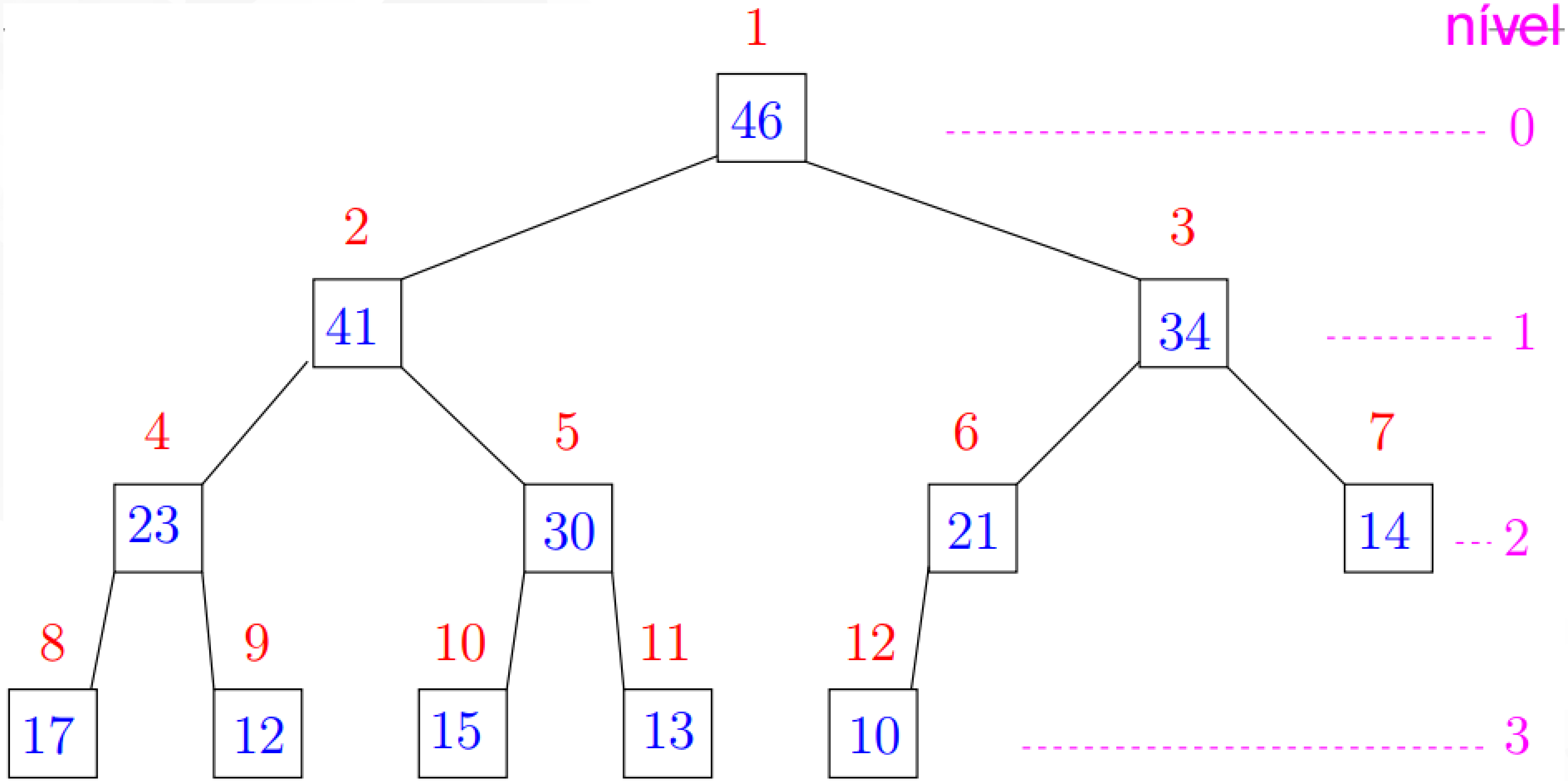
1	2	3	4	5	6	7	8	9	10	11	12
14	41	46	23	30	21	34	17	12	15	13	10



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



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



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

Construção de um Max-Heap

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

BUILD-MAX-HEAP (A, n)
    para $i \leftarrow \lfloor n/2 \rfloor$ **decrescendo até** 1 **faça**
        MAX-HEAPIFY (A, n, i)

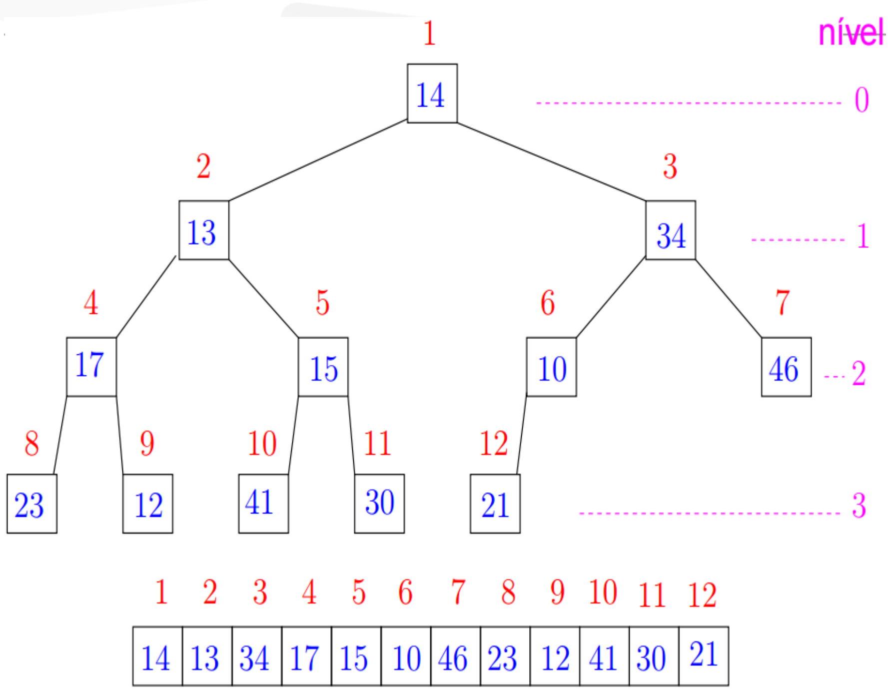
maxheap.c (Tidia)

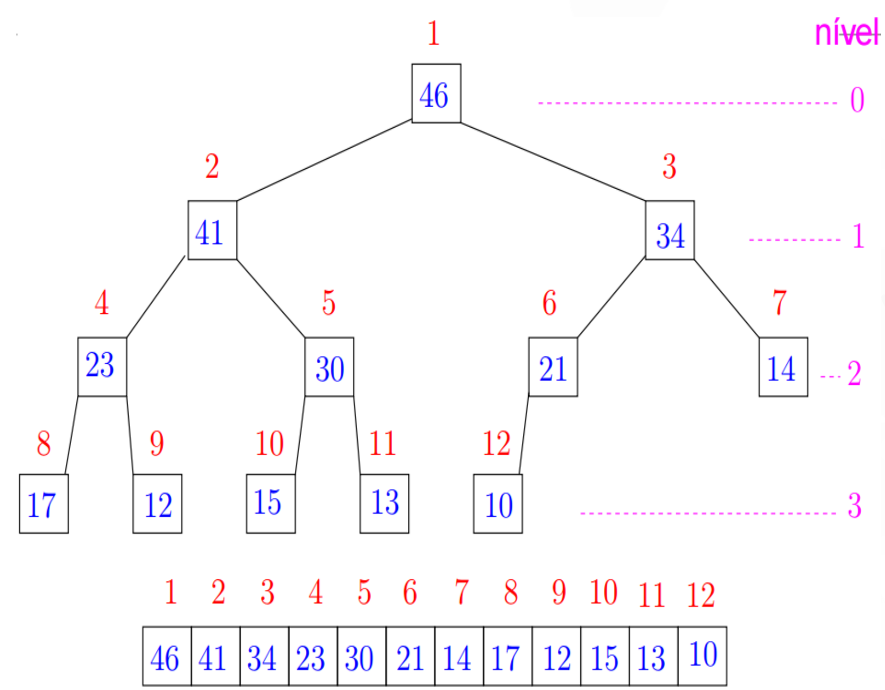
```c
void BuildMaxHeap(int A[], int n) {
    int i;
    for (i=n/2; i>=1; i--)
        MaxHeapify (A, n, i);
}
```

maxheap.c (Tidia)

```c
void BuildMaxHeap(int A[], int n) {
    int i;
    for (i=n/2; i>=1; i--)
        MaxHeapify (A, n, i);
}
```

```
-1 14 13 34 17 15 10 46 23 12 41 30 21
-1 46 41 34 23 30 21 14 17 12 15 13 10
```





maxheap.c (Tidia)

```
void BuildMaxHeap(int A[], int n) {
    int i;
    for (i=n/2; i>=1; i--)
        MaxHeapify (A, n, i);
}
```

$T(n)$:= consumo de tempo no pior caso

Análise grosseira: $T(n)$ é $\frac{n}{2}\,\mathrm{O}(\lg n) = \mathrm{O}(n \lg n)$.

Análise mais cuidadosa: $T(n)$ é $\mathrm{O}(n)$

$T(n)$ é $\mathrm{O}(n)$

Prova: O consumo de BuildMaxHeap (A, n, i) é proporcional a $h = \lfloor \lg \frac{n}{i} \rfloor$. Logo,

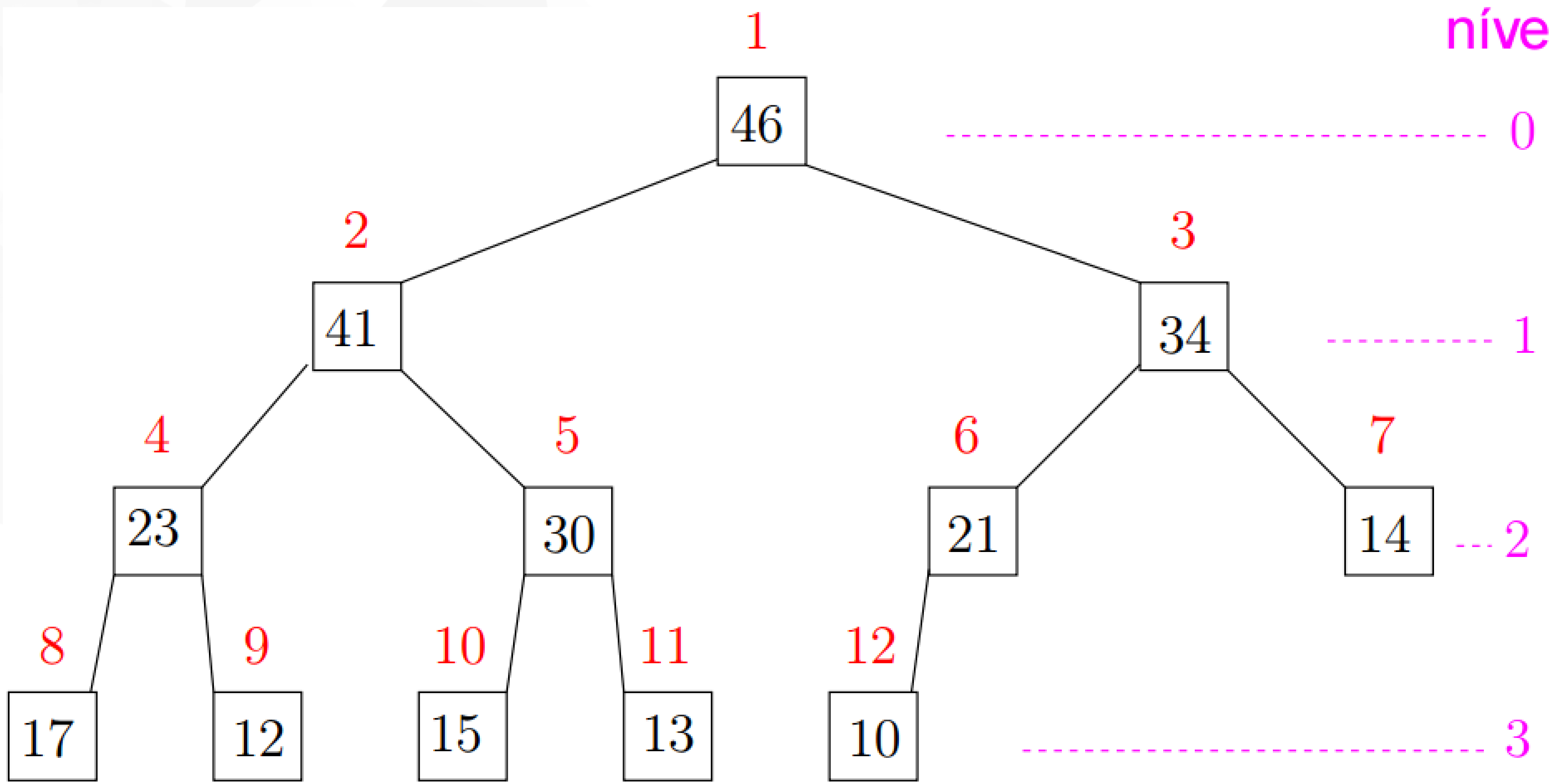
$$
\begin{aligned}
T(n) &= \sum_{h=1}^{\lfloor \lg n \rfloor} \left\lceil \frac{n}{2^{h+1}} \right\rceil h \\
&\leq \sum_{h=1}^{\lfloor \lg n \rfloor} \frac{n}{2^h} h \\
&\leq n \left(\frac{1}{2^1} + \frac{2}{2^2} + \frac{3}{2^3} + \cdots + \frac{\lfloor \lg n \rfloor}{2^{\lfloor \lg n \rfloor}} \right) \\
&< n \frac{1/2}{(1-1/2)^2} \\
&= 2n.
\end{aligned}
$$

Heap Sort

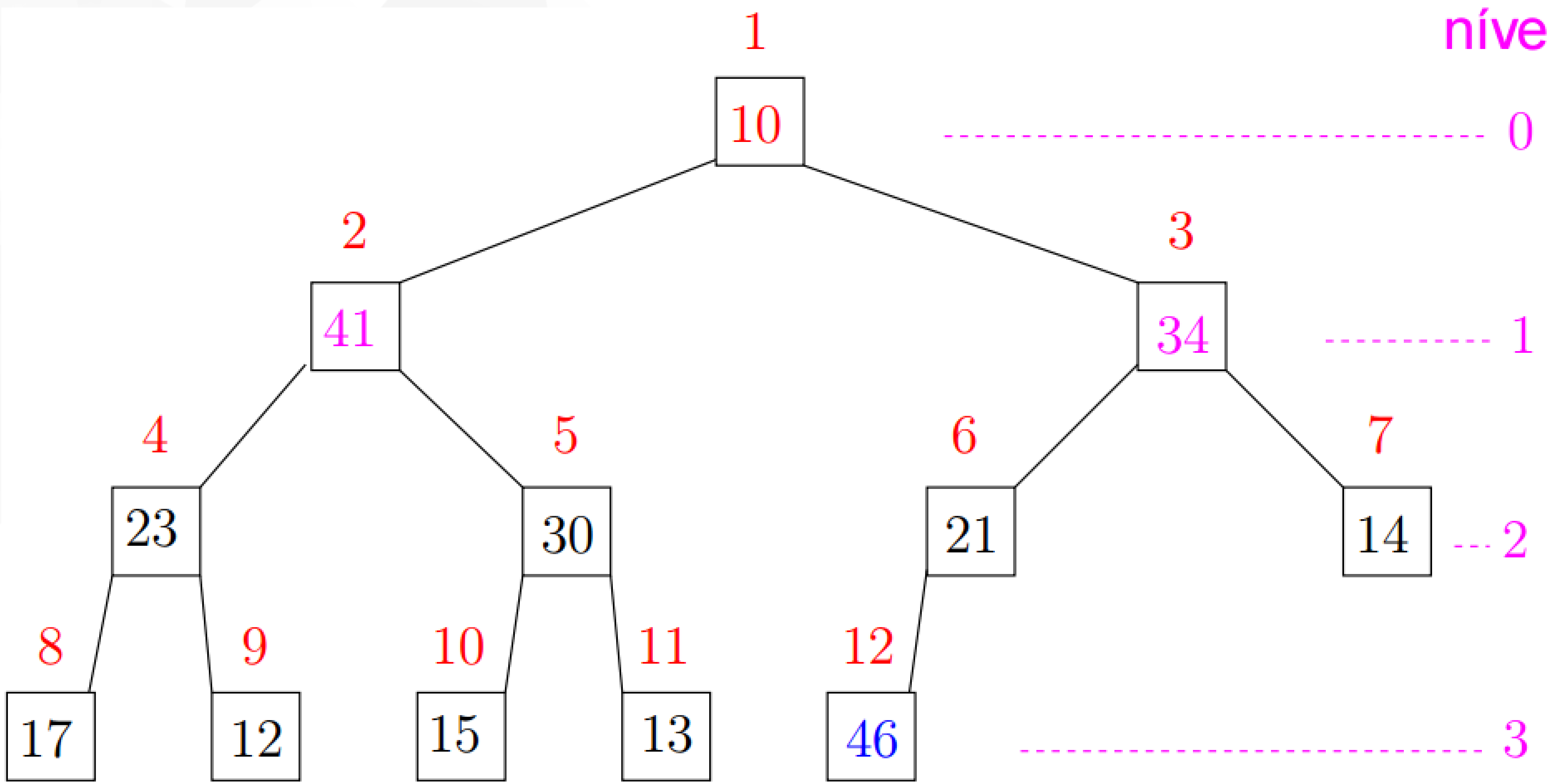
HeapSort ilustra o uso de estrutura de dados no projeto de algoritmos eficientes

Heap Sort

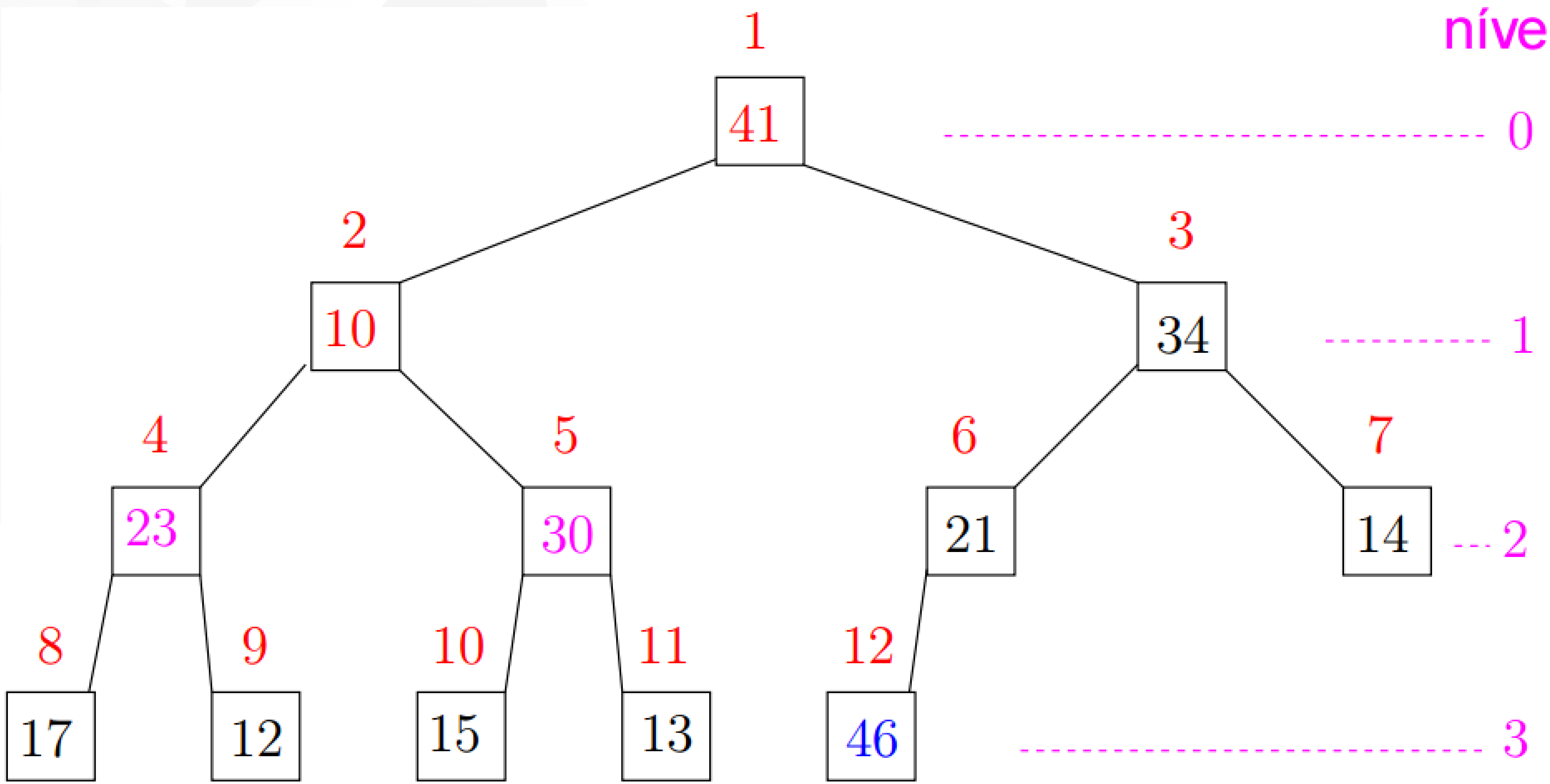
- O algoritmo Heapsort permite rearranjar um vetor em ordem crescente (ou decrescente)
- Este algoritmo é mais rápido que os algoritmos estudados anteriormente.
- **Ao contrário do MergeSort não é necessário um vetor auxiliar**



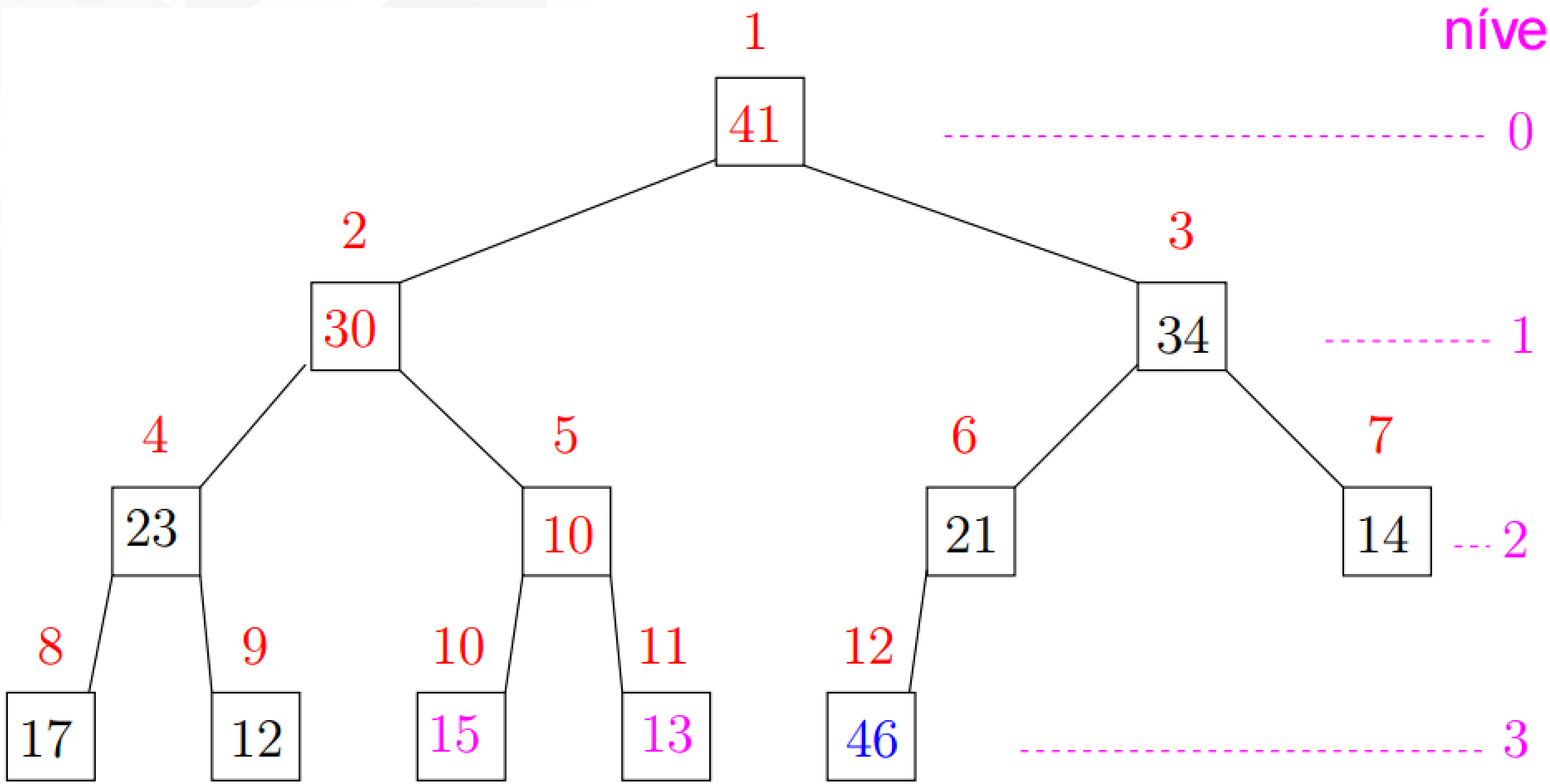
1	2	3	4	5	6	7	8	9	10	11	12
46	41	34	23	30	21	14	17	12	15	13	10



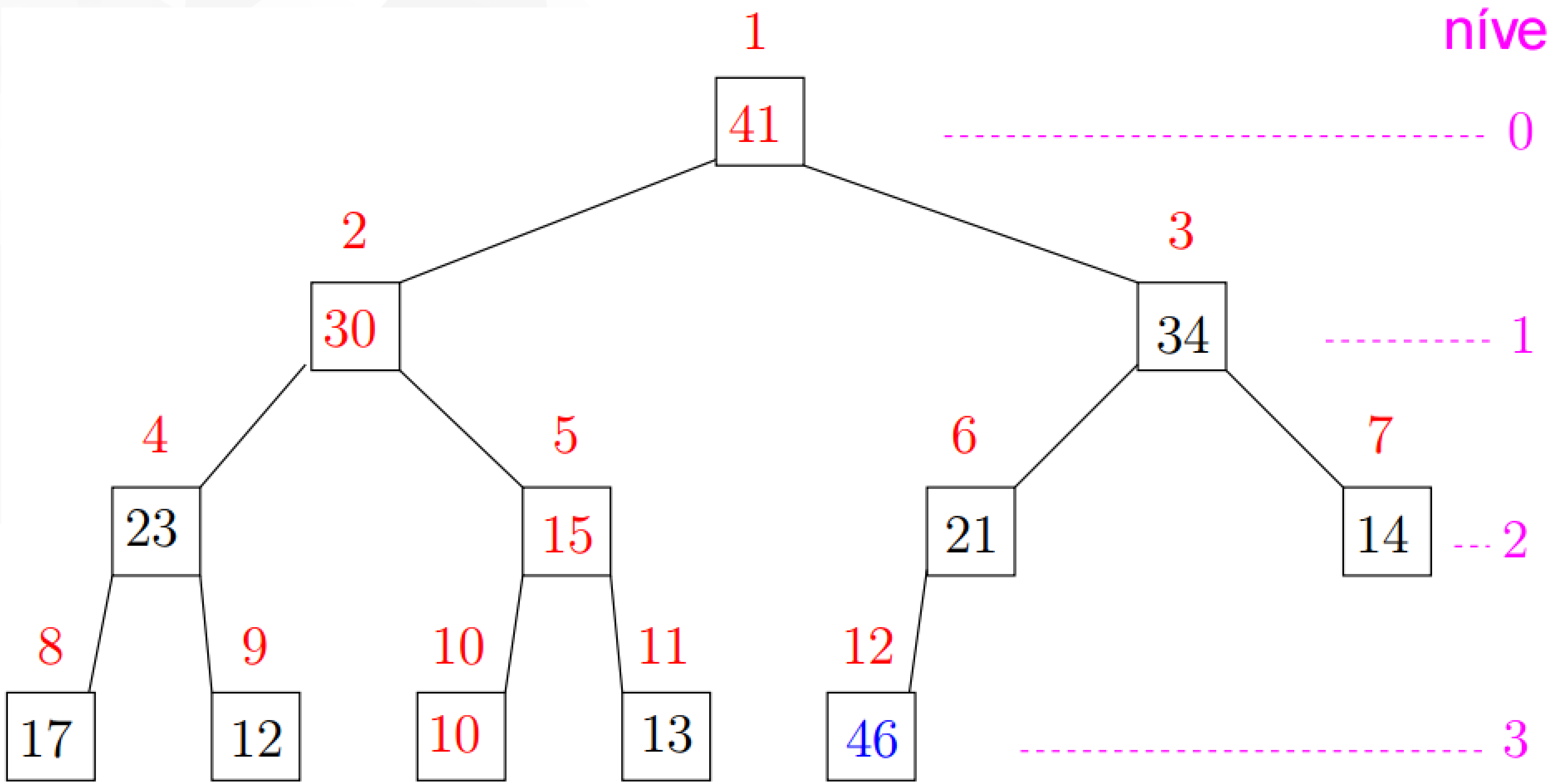
1	2	3	4	5	6	7	8	9	10	11	12
10	41	34	23	30	21	14	17	12	15	13	46



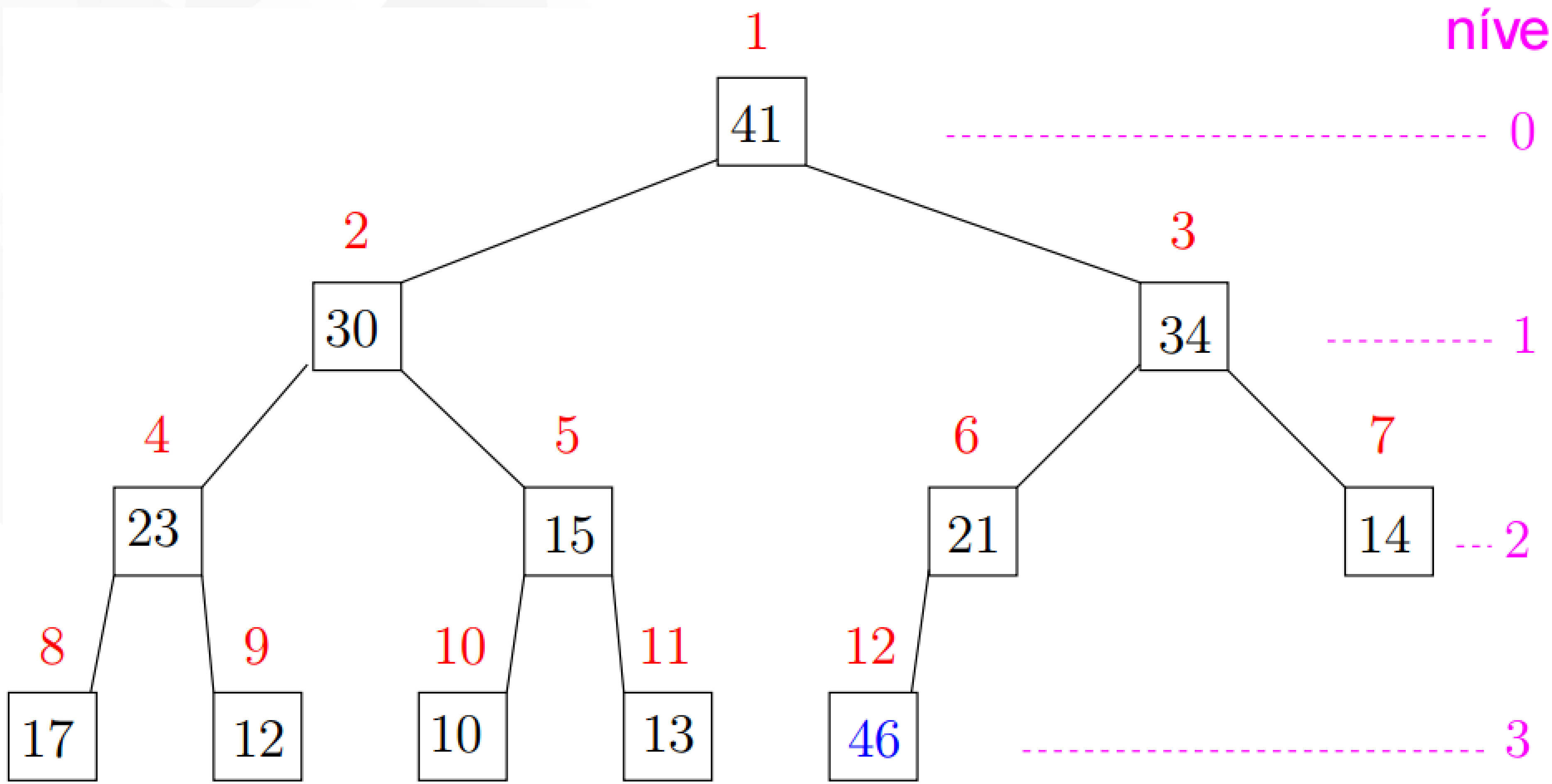
1	2	3	4	5	6	7	8	9	10	11	12
41	10	34	23	30	21	14	17	12	15	13	46



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



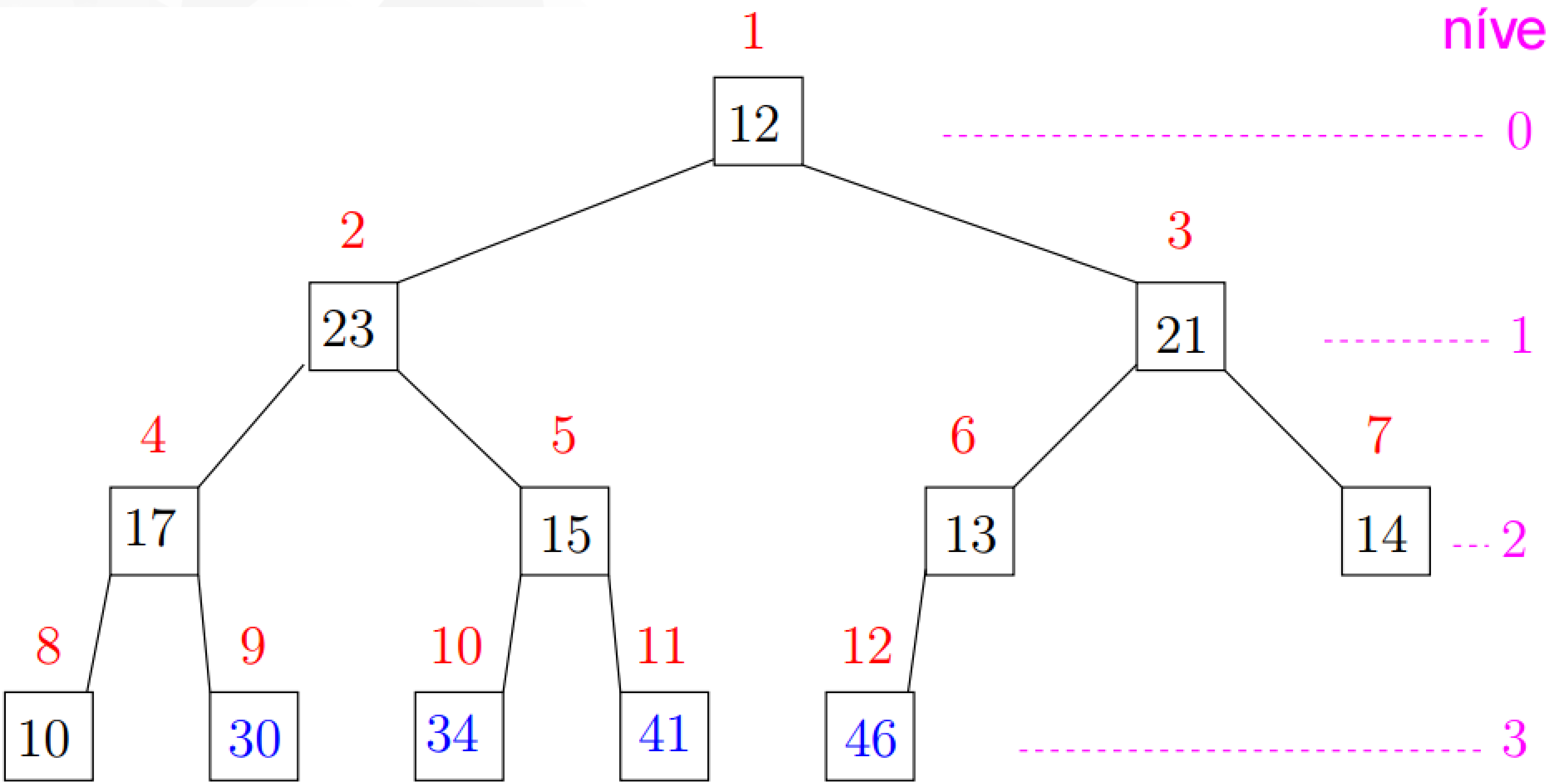
1	2	3	4	5	6	7	8	9	10	11	12
41	30	34	23	15	21	14	17	12	10	13	46



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

Após algumas iterações

i: m $\rightarrow$ 2



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

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

HEAPSORT (A, n)

BUILD-MAX-HEAP (A, n) $\quad \triangleright$ pré-processamento

$m \leftarrow n$

para $i \leftarrow n$ **decrescendo até** 2 **faça**

$\quad A[1] \leftrightarrow A[i]$

$\quad m \leftarrow m - 1$

$\quad$ MAX-HEAPIFY $(A, m, 1)$

heapsort.c (Tidia)

```c
void HeapSort(int A[], int n) {
    int i, m, aux;

    BuildMaxHeap(A, n);
    m = n;

    for (i=n; i>=2; i--) {
        aux = A[i];
        A[i] = A[1];
        A[1] = aux;

        m = m-1;
        MaxHeapify (A, m, 1);
    }
}
```

O consumo de tempo do algoritmo HEAPSORT é $O(n \lg n)$.

Recursos computacionais

- Sortvis: http://sortvis.org/
- Sorting: http://sorting.at/
- Data struture visualizations
  https://www.cs.usfca.edu/~galles/visualization/Algorithms.html
- 15 Sorting Algorithms in 6 Minutes
  https://www.youtube.com/watch?v=kPRA0W1kECg