Projeto e Análise de Algoritmos

A. G. Silva

Baseado nos materiais de
Souza, Silva, Lee, Rezende, Miyazawa – Unicamp
Ribeiro – FCUP
Manber, Introduction to Algorithms (1989) – Livro

13 de abril de 2018

Conteúdo programático

- Introdução (4 horas/aula)
- Notação Assintótica e Crescimento de Funções (4 horas/aula)
- Recorrências (4 horas/aula)
- Divisão e Conquista (12 horas/aula)
- Buscas (4 horas/aula)
- Grafos (4 horas/aula)
- Algoritmos Gulosos (8 horas aula)
- Programação Dinâmica (8 horas/aula)
- NP-Completude e Reduções (6 horas/aula)
- Algoritmos Aproximados e Busca Heurística (6 horas/aula)

Cronograma

- **02mar** – Apresentação da disciplina. Introdução.
- **09mar** – *Prova de proficiência/dispensa.*
- **16mar** – Notação assintótica. Recorrências.
- **23mar** – *Dia não letivo.* Exercícios.
- **30mar** – *Dia não letivo.* Exercícios.
- **06abr** – Recorrências. Divisão e conquista.
- **13abr** – Divisão e conquista. Ordenação.
- **20abr** – Divisão e conquista. Estatística de ordem.
- **27abr** – **Primeira avaliação.**
- **04mai** – Buscas. Grafos.
- **11mai** – Algoritmos gulosos.
- **18mai** – Algoritmos gulosos.
- **25mai** – Programação dinâmica.
- **01jun** – *Dia não letivo.* Exercícios.
- **08jun** – Programação dinâmica.
- **15jun** – NP-Completude e reduções.
- **22jun** – Exercícios *(copa)*.
- **29jun** – **Segunda avaliação.**
- **06jul** – **Avaliação substitutiva** *(opcional)*.

Divisão e Conquista

Projeto de Algoritmos por Divisão e Conquista

- **Dividir para conquistar:** uma tática de guerra aplicada ao projeto de algoritmos.
- Um algoritmo de divisão e conquista é aquele que resolve o problema desejado combinando as soluções parciais de (um ou mais) subproblemas, obtidas recursivamente.
- É mais um paradigma de projeto de algoritmos baseado no princípio da indução.
- Informalmente, podemos dizer que o paradigma incremental representa o projeto de algoritmos por indução fraca, enquanto o paradigma de divisão e conquista representa o projeto por indução forte.
- É natural, portanto, demonstrar a corretude de algoritmos de divisão e conquista por indução.

Algoritmo Genérico

DivisaoConquista(x)

▷ **Entrada:** A instância x
▷ **Saída:** Solução y do problema em questão para x
1. **se** x é suficientemente pequeno **então**
   ▷ *Solucao*(x) algoritmo para pequenas instâncias
2. **retorne** *Solucao*(x)
3. **senão**
   ▷ divisão
4. decomponha x em instâncias menores $x_1, x_2, \cdots, x_k$
5. **para** i **de** 1 **até** k **faça** $y_i := DivisaoConquista(x_i)$
   ▷ conquista
6. combine as soluções y_i para obter a solução y de x.
7. **retorne**(y)

Projeto por Divisão e Conquista - Exemplo 1
Exponenciação

Problema:

Calcular a^n, para todo real a e inteiro $n \geq 0$.

Primeira solução, por indução fraca:

- **Caso base:** $n = 0$; $a^0 = 1$.
- **Hipótese de indução:** *Suponha que, para qualquer inteiro* $n > 0$ *e real a, sei calcular* a^{n-1}.
- **Passo da indução:** Queremos provar que conseguimos calcular a^n, para $n > 0$. Por hipótese de indução, sei calcular a^{n-1}. Então, calculo a^n multiplicando a^{n-1} por a.

Exemplo 1 - Solução 1 - Algoritmo

Exponenciacao(a, n)

▷ **Entrada:** A base a e o expoente n.

▷ **Saída:** O valor de a^n.

1. **se** $n = 0$ **então**
2. **retorne**(1) {caso base}
3. **senão**
4. $an' := Exponenciacao(a, n - 1)$
5. $an := an' * a$
6. **retorne**(an)

Exemplo 1 - Solução 1 - Complexidade

Seja $T(n)$ o número de operações executadas pelo algoritmo para calcular a^n.

Então a relação de recorrência deste algoritmo é:

$$T(n) = \begin{cases} c_1, & n = 0 \\ T(n-1) + c_2, & n > 0, \end{cases}$$

onde c_1 e c_2 representam, respectivamente, o tempo (constante) executado na atribuição da base e multiplicação do passo.

Neste caso, não é difícil ver que

$$T(n) = c_1 + \sum_{i=1}^{n} c_2 = c_1 + nc_2 = \Theta(n).$$

Este algoritmo é linear no tamanho da entrada ?

Exemplo 1 - Solução 2 - Divisão e Conquista

Vamos agora projetar um algoritmo para o problema usando indução forte de forma a obter um algoritmo de divisão e conquista.

Segunda solução, por indução forte:

- **Caso base:** $n = 0$; $a^0 = 1$.
- **Hipótese de indução:** *Suponha que, para qualquer inteiro* $n > 0$ *e real a, sei calcular* a^k*, para todo* $k < n$.
- **Passo da indução:** Queremos provar que conseguimos calcular a^n, para $n > 0$. Por hipótese de indução sei calcular $a^{\lfloor \frac{n}{2} \rfloor}$. Então, calculo a^n da seguinte forma:

$$a^n = \begin{cases} \left(a^{\lfloor \frac{n}{2} \rfloor}\right)^2, & \text{se } n \text{ par;} \\ a \cdot \left(a^{\lfloor \frac{n}{2} \rfloor}\right)^2, & \text{se } n \text{ ímpar.} \end{cases}$$

Exemplo 1 - Solução 2 - Algoritmo

ExponenciacaoDC(a, n)

▷ **Entrada:** A base a e o expoente n.
▷ **Saída:** O valor de a^n.

```
1.  se n = 0 então
2.      retorne(1) {caso base}
3.  senão
        ▷ divisão
4.      an' := ExponenciacaoDC(a, n div 2)
        ▷ conquista
5.      an := an' * an'
6.      se (n mod 2) = 1
7.          an := an * a
8.  retorne(an)
```

Exemplo 1 - Solução 2 - Complexidade

- Seja $T(n)$ o número de operações executadas pelo algoritmo de divisão e conquista para calcular a^n.
- Então a relação de recorrência deste algoritmo é:

$$T(n) = \begin{cases} c_1, & n = 0 \\ T(\lfloor \frac{n}{2} \rfloor) + c_2, & n > 0, \end{cases}$$

- Não é difícil ver que $T(n) \in \Theta(\log n)$. Por quê?

Projeto por Divisão e Conquista - Exemplo 2
Busca Binária

Problema:

Dado um vetor ordenado A com n números reais e um real x, determinar a posição $1 \leq i \leq n$ tal que $A[i] = x$, ou que não existe tal i.

- O projeto de um algoritmo para este problema usando indução simples, nos leva a um algoritmo incremental de complexidade de pior caso $\Theta(n)$. Pense em como seria a indução !
- Se utilizarmos indução forte para projetar o algoritmo, podemos obter um algoritmo de divisão e conquista que nos leva ao algoritmo de busca binária. Pense na indução !
- Como o vetor está ordenado, conseguimos determinar, com apenas uma comparação, que *metade* das posições do vetor não pode conter o valor x.

Exemplo 2 - Algoritmo

BuscaBinaria(A, e, d, x)

▷ **Entrada:** Vetor A, delimitadores e e d do subvetor e x.
▷ **Saída:** Índice $1 \leq i \leq n$ tal que $A[i] = x$ ou $i = 0$.

```
1.  se e = d então se A[e] = x então retorne(e)
2.          senão retorne(0)
3.  senão
4.      i := (e + d) div 2
5.      se A[i] = x retorne(i)
6.      senão se A[i] > x
7.          i := BuscaBinaria(A, e, i − 1, x)
8.          senão {A[i] < x}
9.          i := BuscaBinaria(A, i + 1, d, x)
10. retorne(i)
```

Exemplo 2 - Complexidade

- O número de operações $T(n)$ executadas na busca binária no pior caso é:

$$T(n) = \begin{cases} c_1, & n = 1 \\ T(\lceil \frac{n}{2} \rceil) + c_2, & n > 1, \end{cases}$$

- Não é difícil ver que $T(n) \in \Theta(\log n)$. Por quê?
- O algoritmo de busca binária (divisão e conquista) tem complexidade de pior caso $\Theta(\log n)$, que é assintoticamente melhor que o algoritmo de busca linear (incremental).
- E se o vetor não estivesse ordenado, qual paradigma nos levaria a um algoritmo assintoticamente melhor ?

Projeto por Divisão e Conquista - Exemplo 3 - Máximo e Mínimo

Problema:

Dado um conjunto S de $n \geq 2$ números reais, determinar o maior e o menor elemento de S.

- Um algoritmo incremental para esse problema faz $2n - 3$ comparações: fazemos uma comparação no caso base e duas no passo.
- Será que um algoritmo de divisão e conquista seria melhor ?
- Um possível algoritmo de divisão e conquista seria:
  - Divida S em dois subconjuntos de mesmo tamanho S_1 e S_2 e solucione os subproblemas.
  - O máximo de S é o máximo dos máximos de S_1 e S_2 e o mínimo de S é o mínimo dos mínimos de S_1 e S_2.

Exemplo 3 - Complexidade

- Qual o número de comparações $T(n)$ efetuado por este algoritmo?

$$T(n) = \begin{cases} 1, & n = 2 \\ T(\lfloor \frac{n}{2} \rfloor) + T(\lceil \frac{n}{2} \rceil) + 2, & n > 2, \end{cases}$$

- Supondo que n é uma potência de 2, temos:

$$T(n) = \begin{cases} 1, & n = 2 \\ 2T(\frac{n}{2}) + 2, & n > 2, \end{cases}$$

- Neste caso, podemos provar que $T(n) = \frac{3}{2}n - 2$ usando o método da substituição (indução !).

Exemplo 3 - Complexidade

- **Caso Base:** $T(2) = 1 = 3 - 2$.
- **Hipótese de Indução:** Suponha, para $n = 2^{k-1}$, $k \geq 2$, que $T(n) = \frac{3}{2}n - 2$.
- **Passo de Indução:** Queremos provar para $n = 2^k$, $k \geq 2$, que $T(n) = \frac{3}{2}n - 2$.

$$\begin{aligned} T(n) &= 2T(\tfrac{n}{2}) + 2 \\ &= 2(\tfrac{3}{4}n - 2) + 2 \text{ (por h. i.)} \\ &= \tfrac{3}{2}n - 2. \end{aligned}$$

- É possível provar que $T(n) = \frac{3}{2}n - 2$ quando n não é potência de 2 ?

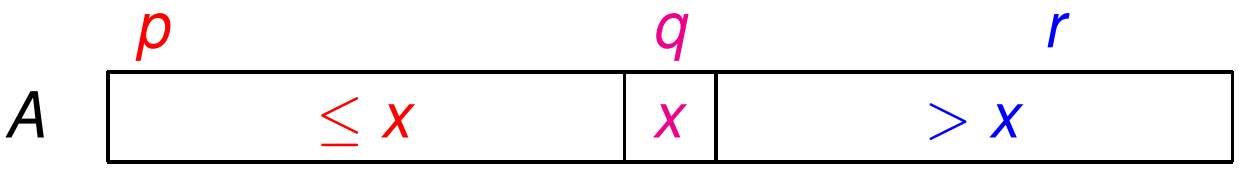
Exemplo 3 - Complexidade

- Assintoticamente, os dois algoritmos para este problema são equivalentes, ambos $\Theta(n)$.
- No entanto, o algoritmo de divisão e conquista permite que menos comparações sejam feitas. A estrutura hierárquica de comparações no retorno da recursão evita comparações desnecessárias.

QuickSort

O algoritmo QUICKSORT segue o paradigma de divisão-e-conquista.

Divisão: divida o vetor em dois subvetores $A[p \ldots q-1]$ e $A[q+1 \ldots r]$ tais que



$$A[p \ldots q-1] \leq A[q] < A[q+1 \ldots r]$$

Conquista: ordene os dois subvetores recursivamente usando o QUICKSORT;

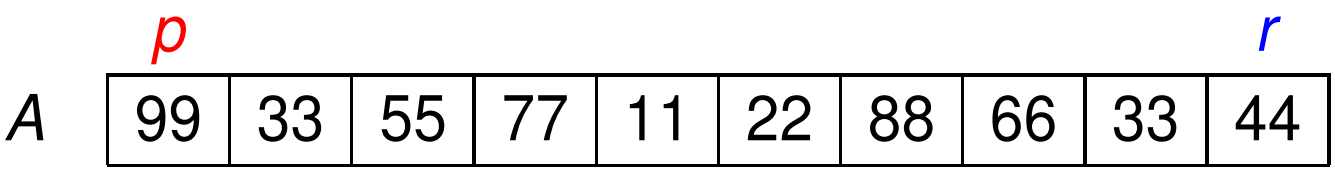
Combinação: nada a fazer, o vetor está ordenado.

Partição

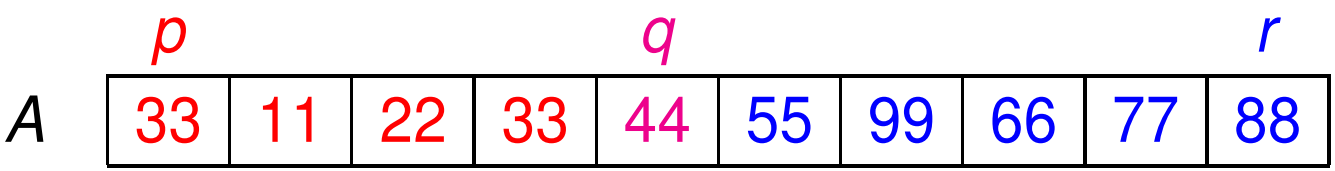
Problema: Rearranjar um dado vetor $A[p \ldots r]$ e devolver um índice q, $p \leq q \leq r$, tais que

$$A[p \ldots q-1] \leq A[q] < A[q+1 \ldots r]$$

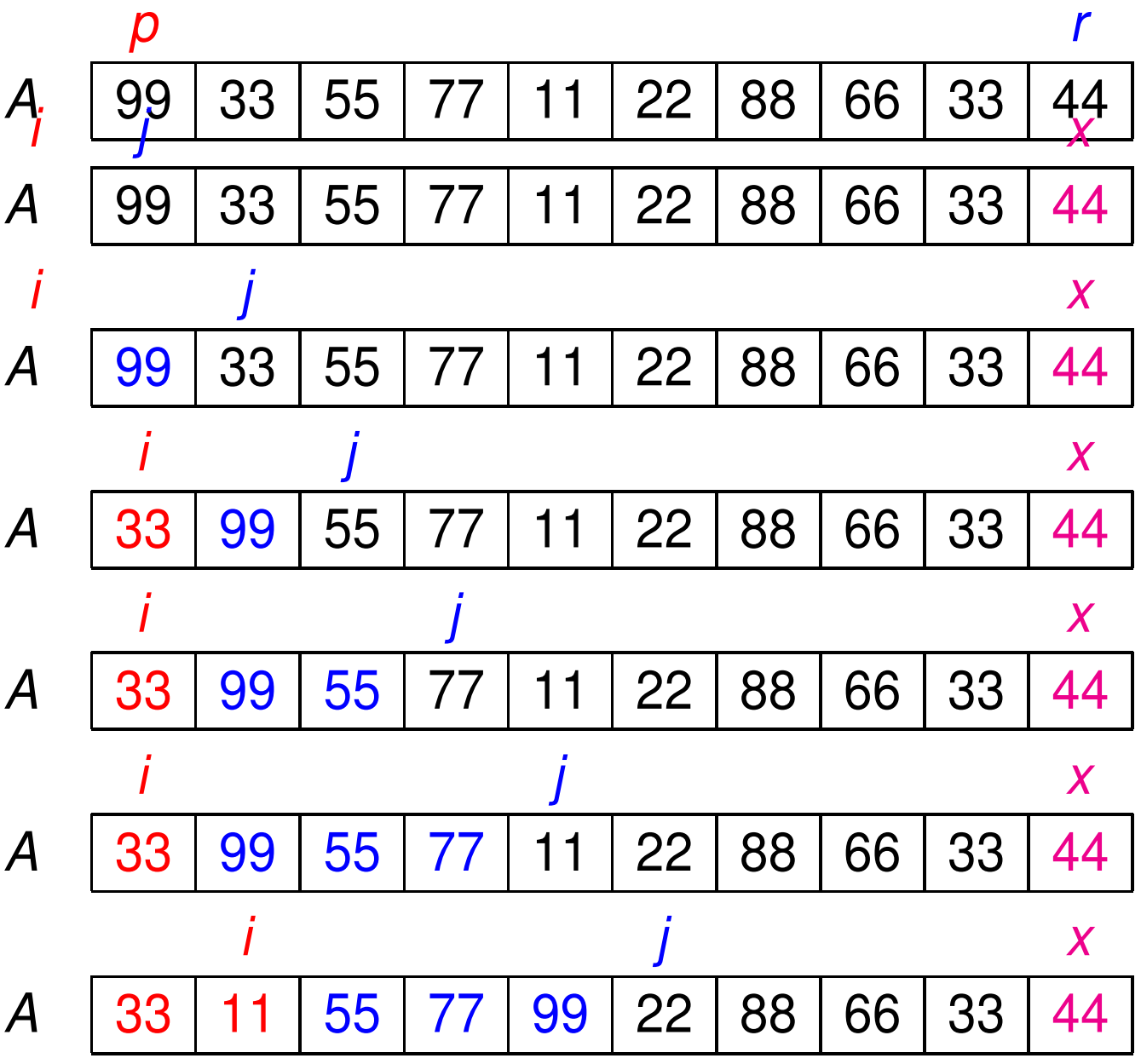
Entrada:



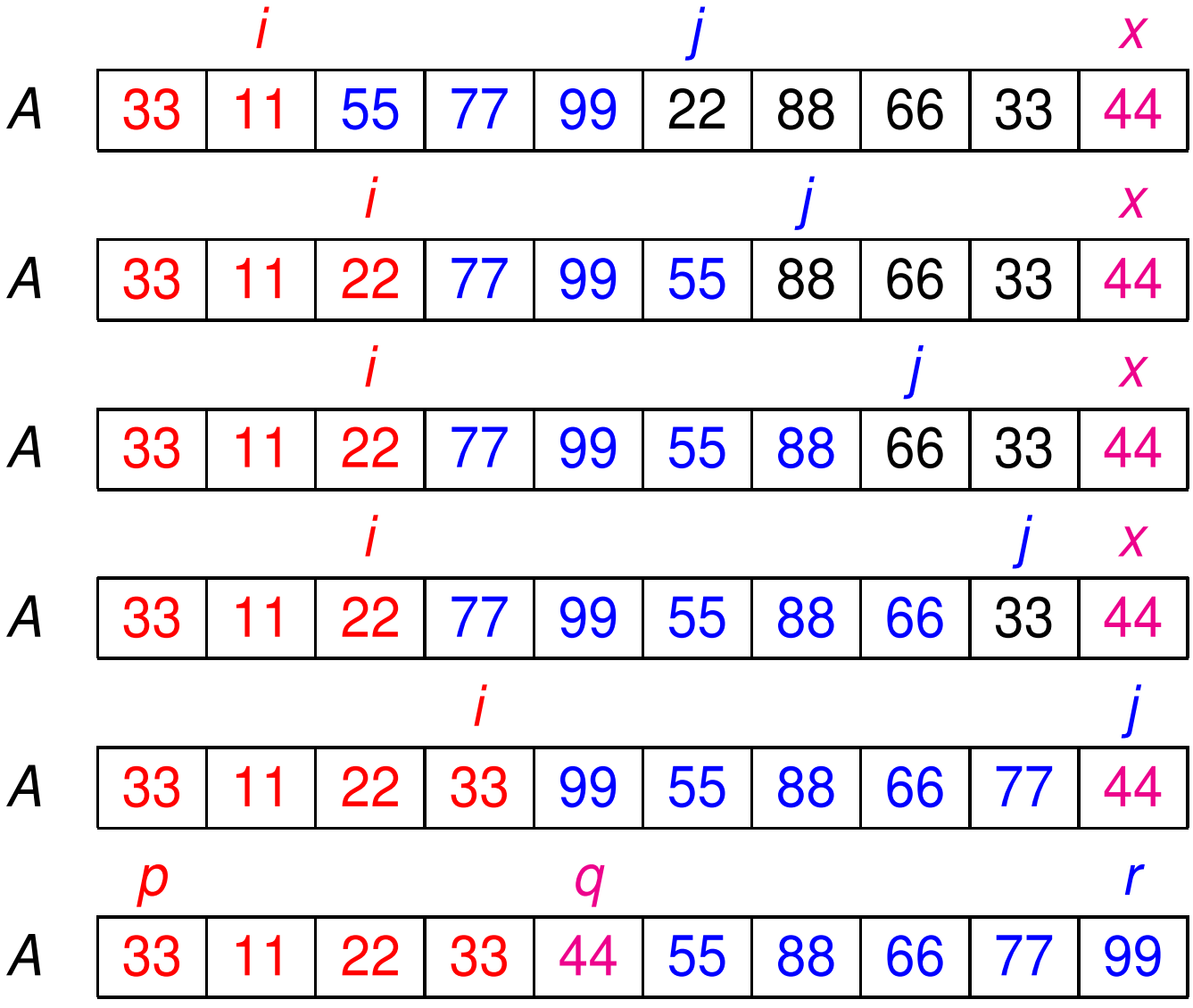
Saída:



Particione


p
r
A
99 33 55 77 11 22 88 66 33 44
i
j
x
A
99 33 55 77 11 22 88 66 33 44
i
j
x
A
99 33 55 77 11 22 88 66 33 44
i
j
x
A
33 99 55 77 11 22 88 66 33 44
i
j
x
A
33 99 55 77 11 22 88 66 33 44
i
j
x
A
33 99 55 77 11 22 88 66 33 44
i
j
x
A
33 11 55 77 99 22 88 66 33 44

Particione


i
j
x
A
33 11 55 77 99 22 88 66 33 44
i
j
x
A
33 11 22 77 99 55 88 66 33 44
i
j
x
A
33 11 22 77 99 55 88 66 33 44
i
j
x
A
33 11 22 77 99 55 88 66 33 44
i
j
A
33 11 22 33 99 55 88 66 77 44
p
q
r
A
33 11 22 33 44 55 88 66 77 99

Particione

Rearranja $A[p \ldots r]$ de modo que $p \le q \le r$ e
$A[p \ldots q-1] \le A[q] < A[q+1 \ldots r]$

PARTICIONE(A, p, r)

```
1  x ← A[r]   ▷ x é o "pivô"
2  i ← p−1
3  para j ← p até r − 1 faça
4       se A[j] ≤ x
5            então i ← i + 1
6                  A[i] ↔ A[j]
7  A[i+1] ↔ A[r]
8  devolva i + 1
```

Invariantes:

No começo de cada iteração da linha 3 vale que:
(1) $A[p \ldots i] \le x$ (2) $A[i+1 \ldots j-1] > x$ (3) $A[r] = x$

Complexidade de PARTICIONE

	PARTICIONE(A, p, r)	Tempo
1	$x \leftarrow A[r]$ $\triangleright$ x é o "pivô"	?
2	$i \leftarrow p-1$	?
3	**para** $j \leftarrow p$ **até** $r-1$ **faça**	?
4	**se** $A[j] \leq x$	?
5	**então** $i \leftarrow i+1$	?
6	$A[i] \leftrightarrow A[j]$	?
7	$A[i+1] \leftrightarrow A[r]$	?
8	**devolva** $i+1$	?

$T(n) =$ complexidade de tempo no pior caso sendo

$n := r - p + 1$

Complexidade de PARTICIONE

PARTICIONE(A, p, r)		Tempo
1	$x \leftarrow A[r]$ $\triangleright$ x é o "pivô"	$\Theta(1)$
2	$i \leftarrow p-1$	$\Theta(1)$
3	**para** $j \leftarrow p$ **até** $r-1$ **faça**	$\Theta(n)$
4	**se** $A[j] \leq x$	$\Theta(n)$
5	**então** $i \leftarrow i+1$	$O(n)$
6	$A[i] \leftrightarrow A[j]$	$O(n)$
7	$A[i+1] \leftrightarrow A[r]$	$\Theta(1)$
8	**devolva** $i+1$	$\Theta(1)$

$$T(n) = \Theta(2n+4) + O(2n) = \Theta(n)$$

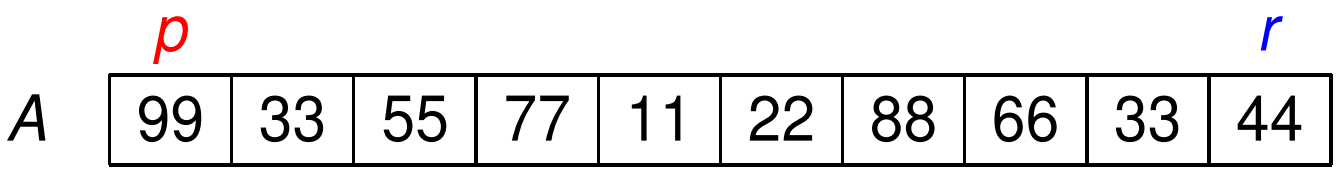
Conclusão:

A complexidade de PARTICIONE é $\Theta(n)$.

QuickSort

Rearranja um vetor $A[p \ldots r]$ em ordem crescente.

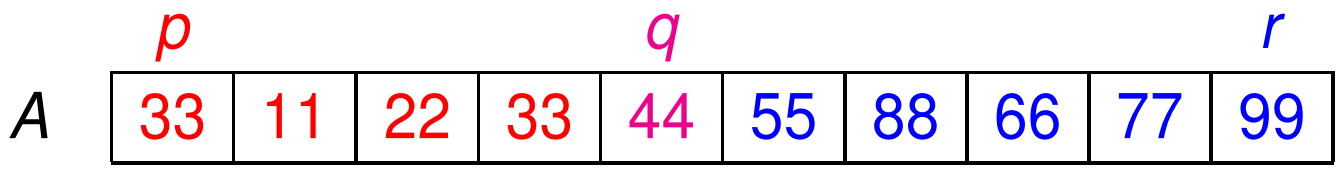
```
QUICKSORT(A, p, r)
1  se p < r
2     então q ← PARTICIONE(A, p, r)
3           QUICKSORT(A, p, q − 1)
4           QUICKSORT(A, q + 1, r)
```



QuickSort

Rearranja um vetor $A[p \ldots r]$ em ordem crescente.

```
QUICKSORT(A, p, r)
1  se p < r
2     então q ← PARTICIONE(A, p, r)
3           QUICKSORT(A, p, q − 1)
4           QUICKSORT(A, q + 1, r)
```



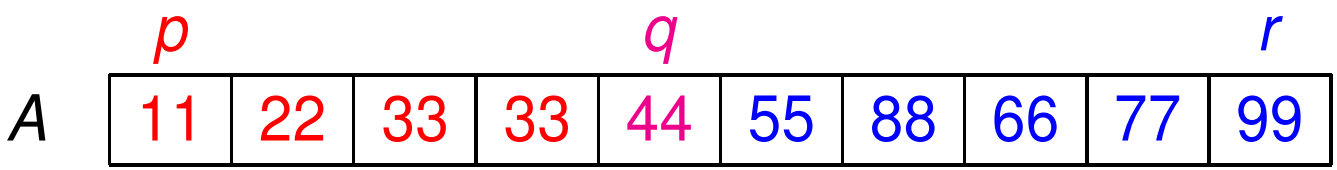
No começo da linha 3,

$$A[p \ldots q-1] \leq A[q] < A[q+1 \ldots r]$$

QuickSort

Rearranja um vetor $A[p \ldots r]$ em ordem crescente.

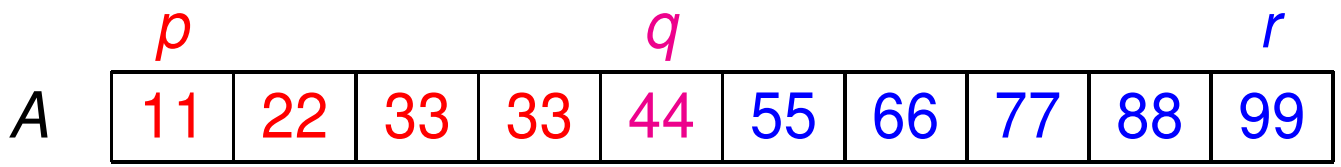
```
QUICKSORT(A, p, r)
1  se p < r
2     então q ← PARTICIONE(A, p, r)
3           QUICKSORT(A, p, q − 1)
4           QUICKSORT(A, q + 1, r)
```



QuickSort

Rearranja um vetor $A[p \ldots r]$ em ordem crescente.

```
QUICKSORT(A, p, r)
1  se p < r
2     então q ← PARTICIONE(A, p, r)
3           QUICKSORT(A, p, q − 1)
4           QUICKSORT(A, q + 1, r)
```



Complexidade de QUICKSORT

QUICKSORT(A, p, r)			Tempo
1	**se**	$p < r$	?
2		**então** $q \leftarrow$ PARTICIONE(A, p, r)	?
3		QUICKSORT$(A, p, q-1)$	?
4		QUICKSORT$(A, q+1, r)$	?

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

$n := r - p + 1$

Complexidade de QUICKSORT

QUICKSORT(A, p, r)			Tempo
1	**se**	$p < r$	$\Theta(1)$
2		**então** $q \leftarrow$ PARTICIONE(A, p, r)	$\Theta(n)$
3		QUICKSORT$(A, p, q-1)$	$T(k)$
4		QUICKSORT$(A, q+1, r)$	$T(n-k-1)$

$$T(n) = T(k) + T(n-k-1) + \Theta(n+1)$$

$$0 \leq k := q - p \leq n - 1$$

Recorrência

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

$$T(0) = \Theta(1)$$
$$T(1) = \Theta(1)$$
$$T(n) = T(k) + T(n-k-1) + \Theta(n) \quad \text{para } n = 2, 3, 4, \ldots$$

Recorrência grosseira:

$$T(n) = T(0) + T(n-1) + \Theta(n)$$

$T(n)$ é $\Theta(???)$.

Recorrência grosseira:

$$T(n) = T(0) + T(n-1) + \Theta(n)$$

$T(n)$ é $\Theta(n^2)$.

Recorrência cuidadosa

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

$$T(0) = \Theta(1)$$
$$T(1) = \Theta(1)$$
$$T(n) = \max_{0 \le k \le n-1} \{T(k) + T(n-k-1)\} + \Theta(n) \quad \text{para } n = 2, 3, 4, \ldots$$

$$T(n) = \max_{0 \le k \le n-1} \{T(k) + T(n-k-1)\} + bn$$

Quero mostrar que $T(n) = \Theta(n^2)$.

Demonstração – $T(n) = O(n^2)$

Vou provar que $T(n) \leq cn^2$ para n grande.

$$
\begin{aligned}
T(n) &= \max_{0 \leq k \leq n-1} \Big\{ T(k) + T(n-k-1) \Big\} + bn \\
&\leq \max_{0 \leq k \leq n-1} \Big\{ ck^2 + c(n-k-1)^2 \Big\} + bn \\
&= c \max_{0 \leq k \leq n-1} \Big\{ k^2 + (n-k-1)^2 \Big\} + bn \\
&= c(n-1)^2 + bn \qquad \triangleright \text{ exercício} \\
&= cn^2 - 2cn + c + bn \\
&\leq cn^2 \,,
\end{aligned}
$$

se $c > b/2$ e $n \geq c/(2c-b)$.

Continuação – $T(n) = \Omega(n^2)$

Agora vou provar que $T(n) \geq dn^2$ para n grande.

$$
\begin{aligned}
T(n) &= \max_{0 \leq k \leq n-1} \left\{ T(k) + T(n-k-1) \right\} + bn \\
&\geq \max_{0 \leq k \leq n-1} \left\{ dk^2 + d(n-k-1)^2 \right\} + bn \\
&= d \max_{0 \leq k \leq n-1} \left\{ k^2 + (n-k-1)^2 \right\} + bn \\
&= d(n-1)^2 + bn \\
&= dn^2 - 2dn + d + bn \\
&\geq dn^2 \,,
\end{aligned}
$$

se $d < b/2$ e $n \geq d/(2d-b)$.

Conclusão

$T(n)$ é $\Theta(n^2)$.

A complexidade de tempo do QUICKSORT no pior caso é $\Theta(n^2)$.

A complexidade de tempo do QUICKSORT é $O(n^2)$.

QuickSort no melhor caso

$M(n) :=$ complexidade de tempo no melhor caso

$$M(0) = \Theta(1)$$
$$M(1) = \Theta(1)$$
$$M(n) = \min_{0 \le k \le n-1} \{M(k) + M(n-k-1)\} + \Theta(n) \quad \text{para } n = 2, 3, 4, \ldots$$

Mostre que, para $n \geq 1$,

$$M(n) \geq \frac{(n-1)}{2} \lg \frac{n-1}{2}.$$

Isto implica que no melhor caso o QUICKSORT é $\Omega(n \lg n)$.

Que é o mesmo que dizer que o QUICKSORT é $\Omega(n \lg n)$.

QuickSort no melhor caso

No melhor caso k é aproximadamente $(n-1)/2$.

$$R(n) = R(\left\lfloor \frac{n-1}{2} \right\rfloor) + R(\left\lceil \frac{n-1}{2} \right\rceil) + \Theta(n)$$

Solução: $R(n)$ é $\Theta(n \lg n)$.

Humm, lembra a recorrência do MERGESORT...

Mais algumas conclusões

$M(n)$ é $\Theta(n \lg n)$.

O consumo de tempo do QUICKSORT no melhor caso é $\Omega(n \log n)$.

Mais precisamente, a complexidade de tempo do QUICKSORT no melhor caso é $\Theta(n \log n)$.

Caso médio

Apesar da complexidade de tempo do QUICKSORT no pior caso ser $\Theta(n^2)$, na prática ele é o algoritmo mais eficiente.

Mais precisamente, a complexidade de tempo do QUICKSORT no caso médio é mais próximo do melhor caso do que do pior caso.

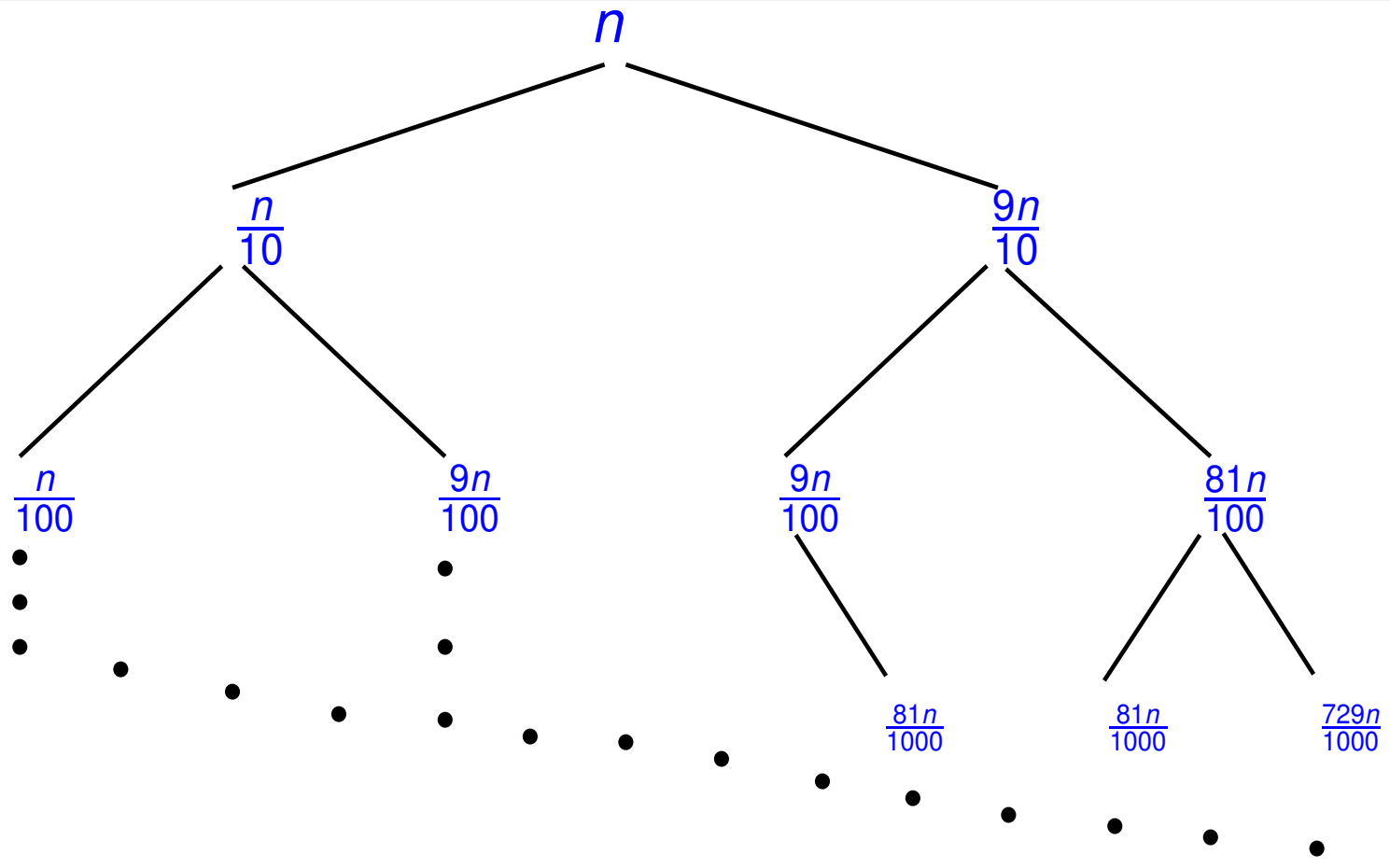
Por quê??

Suponha que (por sorte) o algoritmo PARTICIONE sempre divide o vetor na proporção $\frac{1}{9}$ para $\frac{9}{10}$. Então

$$T(n) = T(\left\lfloor \frac{n-1}{9} \right\rfloor) + T(\left\lceil \frac{9(n-1)}{10} \right\rceil) + \Theta(n)$$

Solução: $T(n)$ é $\Theta(n \lg n)$.

Árvore de recorrência



Número de níveis $\leq \log_{10/9} n$.

Em cada nível o custo é $\leq n$.

Custo total é $O(n \log n)$.

QuickSort Aleatório

O pior caso do QUICKSORT ocorre devido a uma escolha infeliz do pivô.

Um modo de minimizar este problema é usar aleatoriedade.

```
PARTICIONE-ALEATÓRIO(A, p, r)
1   i ← RANDOM(p, r)
2   A[i] ↔ A[r]
3   devolva PARTICIONE(A, p, r)

QUICKSORT-ALEATÓRIO(A, p, r)
1   se p < r
2      então q ← PARTICIONE-ALEATÓRIO(A, p, r)
3            QUICKSORT-ALEATÓRIO(A, p, q − 1)
4            QUICKSORT-ALEATÓRIO(A, q + 1, r)
```

Análise do caso médio

Recorrência para o caso médio do algoritmo QUICKSORT-ALEATÓRIO.

$T(n) =$ consumo de tempo médio do algoritmo QUICKSORT-ALEATÓRIO.

PARTICIONE-ALEATÓRIO rearranja o vetor A e devolve um índice q tal que $A[p \ldots q-1] \leq A[q]$ e $A[q+1 \ldots r] > A[q]$.

$$T(0) = \Theta(1)$$
$$T(1) = \Theta(1)$$
$$T(n) = \frac{1}{n}\left(\sum_{k=0}^{n-1}(T(k) + T(n-1-k))\right) + \Theta(n).$$

$T(n)$ é $\Theta(???)$.

Análise do caso médio

$$
\begin{aligned}
T(n) &= \frac{1}{n}\left(\sum_{k=0}^{n-1}(T(k)+T(n-1-k))\right)+cn \\
&= \frac{2}{n}\sum_{k=0}^{n-1}T(k)+cn.
\end{aligned}
$$

Vou mostrar que $T(n)$ é $O(n \lg n)$.

Vou mostrar que $T(n) \leq an \lg n + b$ para $n \geq 1$ onde $a, b > 0$ são constantes.

Demonstração

$$
\begin{aligned}
T(n) &\leq \frac{2}{n}\sum_{k=0}^{n-1} T(k) + cn \\
&\leq \frac{2}{n}\sum_{k=0}^{n-1} (ak \lg k + b) + cn \\
&= \frac{2a}{n}\sum_{k=1}^{n-1} k \lg k + 2b + cn
\end{aligned}
$$

Lema

$$\sum_{k=1}^{n-1} k \lg k \leq \frac{1}{2} n^2 \lg n - \frac{1}{8} n^2.$$

Demonstração

$$
\begin{aligned}
T(n) &= \frac{2a}{n}\sum_{k=1}^{n-1} k \lg k + 2b + cn \\
&\leq \frac{2a}{n}\left(\frac{1}{2}n^2 \lg n - \frac{1}{8}n^2\right) + 2b + cn \\
&= an\lg n - \frac{a}{4}n + 2b + cn \\
&= an\lg n + b + \left(cn + b - \frac{a}{4}n\right) \\
&\leq an\lg n + b,
\end{aligned}
$$

escolhendo a de modo que $\frac{a}{4}n \geq cn + b$ para $n \geq 1$.

Prova do Lema

$$
\begin{aligned}
\sum_{k=1}^{n-1} k \lg k &= \sum_{k=1}^{\lceil n/2 \rceil - 1} k \lg k + \sum_{k=\lceil n/2 \rceil}^{n-1} k \lg k \\
&\leq (\lg n - 1) \sum_{k=1}^{\lceil n/2 \rceil - 1} k + \lg n \sum_{k=\lceil n/2 \rceil}^{n-1} k \\
&= \lg n \sum_{k=1}^{n-1} k - \sum_{k=1}^{\lceil n/2 \rceil - 1} k \\
&\leq \frac{1}{2} n(n-1) \lg n - \frac{1}{2} \left(\frac{n}{2} - 1 \right) \frac{n}{2} \\
&\leq \frac{1}{2} n^2 \lg n - \frac{1}{8} n^2
\end{aligned}
$$

Conclusão

O consumo de tempo de QUICKSORT-ALEATÓRIO no caso médio é $O(n \lg n)$.

Exercício Mostre que $T(n) = \Omega(n \lg n)$.

Conclusão:

O consumo de tempo de QUICKSORT-ALEATÓRIO no caso médio é $\Theta(n \lg n)$.

Ordenação – outros métodos importantes

Algoritmos de ordenação

Algoritmos de ordenação:

- Insertion sort ✓
- Selection sort ✓
- Mergesort ✓
- Quicksort ✓
- Heapsort

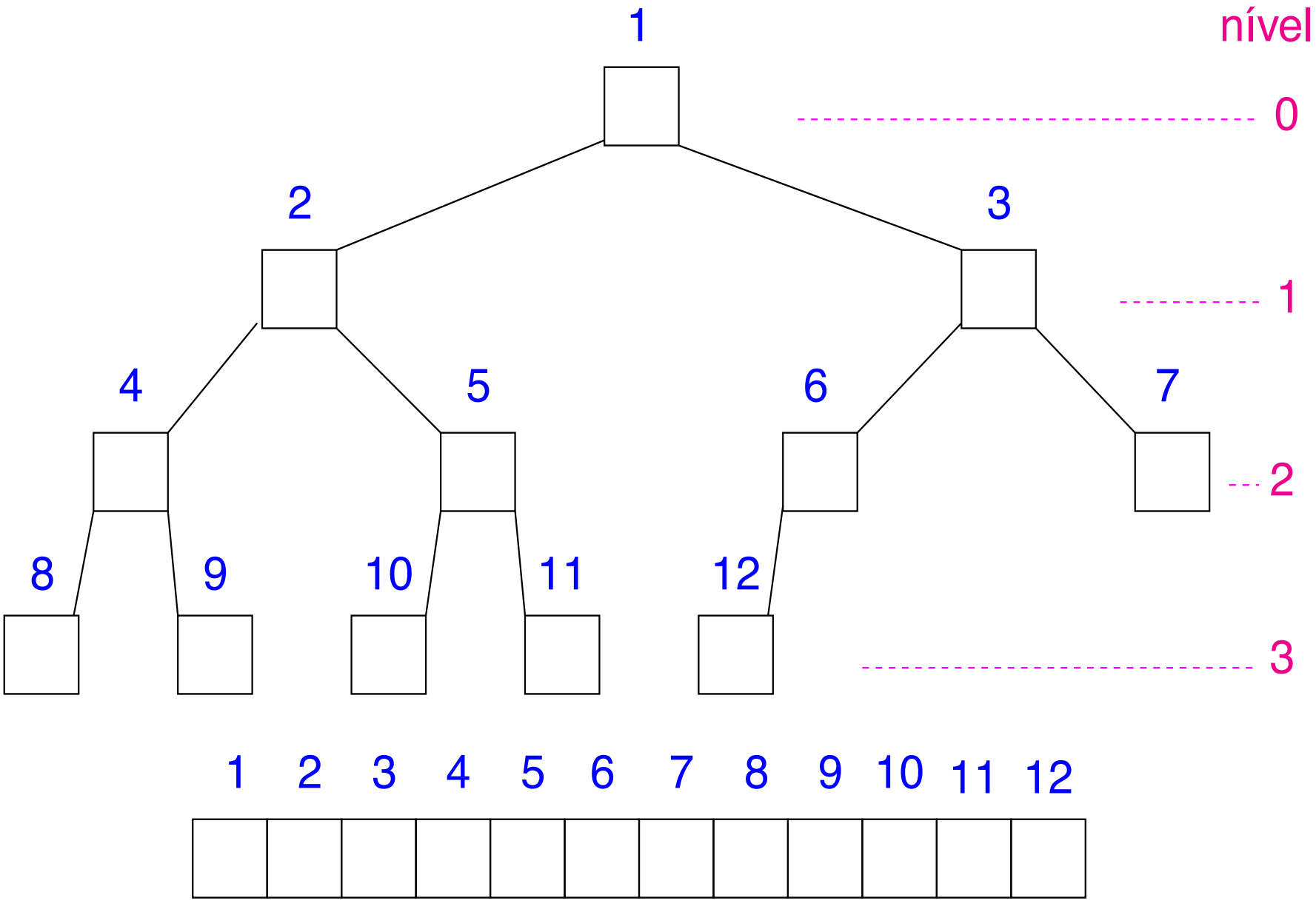
Algoritmos lineares:

- Counting sort
- Radix sort

Heapsort

- O *Heapsort* é um algoritmo de ordenação que usa uma estrutura de dados sofisticada chamada *heap*.
- A complexidade de pior caso é $\Theta(n \lg n)$.
- *Heaps* podem ser utilizados para implementar filas de prioridade que são extremamente úteis em outros algoritmos.
- Um *heap* é um vetor A que simula uma árvore binária completa, com exceção possivelmente do último nível.

Heaps



Heaps

Considere um vetor $A[1 \ldots n]$ representando um heap.

- Cada posição do vetor corresponde a um nó do heap.
- O pai de um nó i é $\lfloor i/2 \rfloor$.
- O nó 1 não tem pai.

Heaps

- Um nó i tem
  $2i$ como filho esquerdo e
  $2i + 1$ como filho direito.
- Naturalmente, o nó i
  tem filho esquerdo apenas se $2i \leq n$ e
  tem filho direito apenas se $2i + 1 \leq n$.
- Um nó i é uma folha se não tem filhos, ou seja, se $2i > n$.
- As folhas são $\lfloor n/2 \rfloor + 1, \ldots, n - 1, n$.

Níveis

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

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

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

Prova: Se p é o nível do nó i, então

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

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

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

Altura

A altura de um nó *i* é o maior comprimento de um caminho de *i* a uma folha.

Os nós que têm altura zero são as folhas.

Qual é a altura de um nó *i*?

Altura

A altura de um nó i é o comprimento da seqüência

$$2\,i, 2^2\,i, 2^3\,i, \ldots, 2^h\,i$$

onde $2^h\,i \leq n < 2^{(h+1)}\,i$.

Assim,

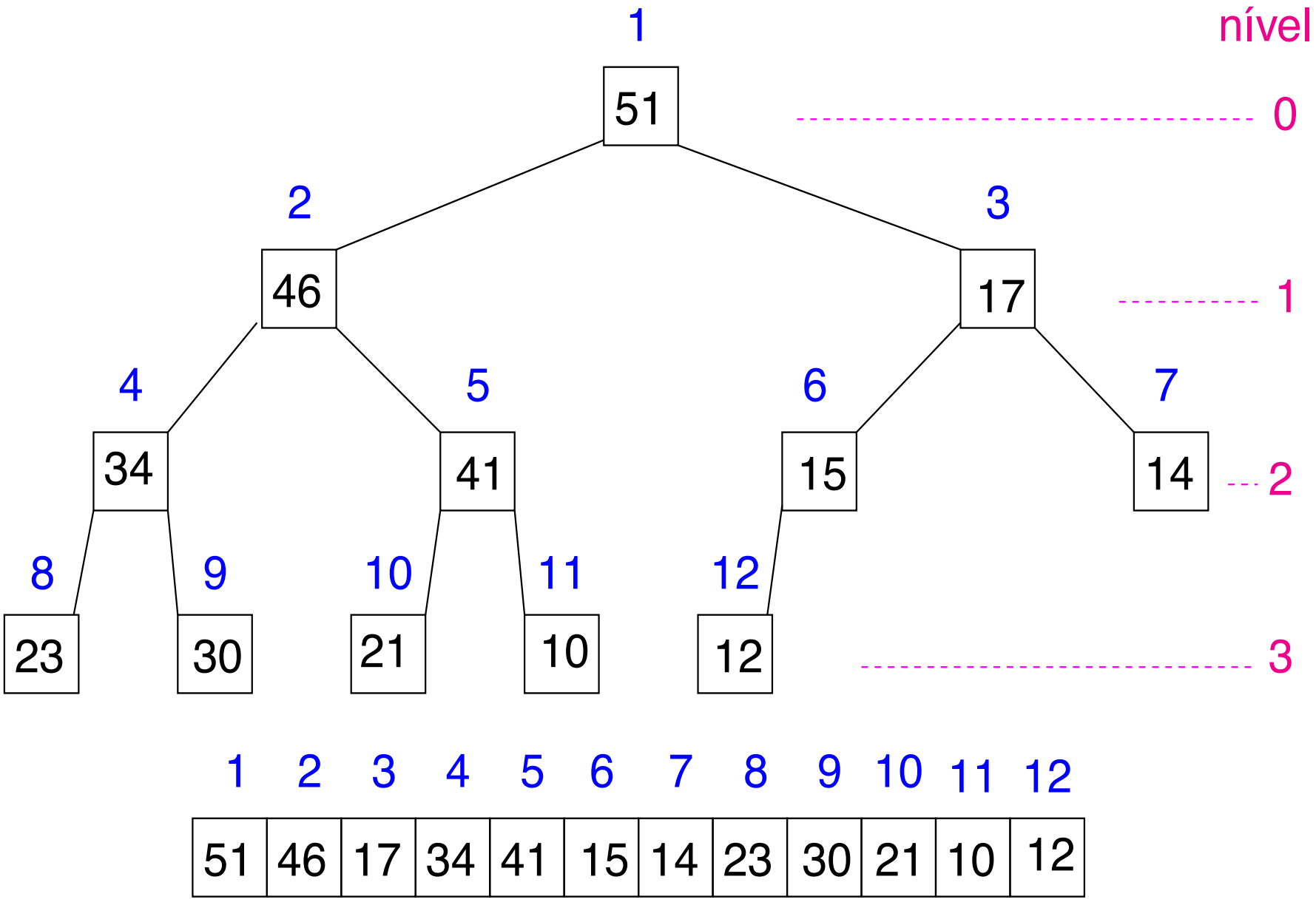
$$\begin{array}{rcccl} 2^h\,i & \leq & n & < & 2^{h+1}\,i \quad \Rightarrow \\ 2^h & \leq & n/i & < & 2^{h+1} \quad \Rightarrow \\ h & \leq & \lg(n/i) & < & h+1 \end{array}$$

Portanto, a altura de i é $\lfloor \lg(n/i) \rfloor$.

Max-heaps

- Um nó i satisfaz a propriedade de (max-)heap se $A[\lfloor i/2 \rfloor] \geq A[i]$ (ou seja, pai $\geq$ filho).
- Uma árvore binária completa é um *max-heap* se todo nó distinto da raiz satisfaz a propriedade de heap.
- O máximo ou maior elemento de um max-heap está na raiz.

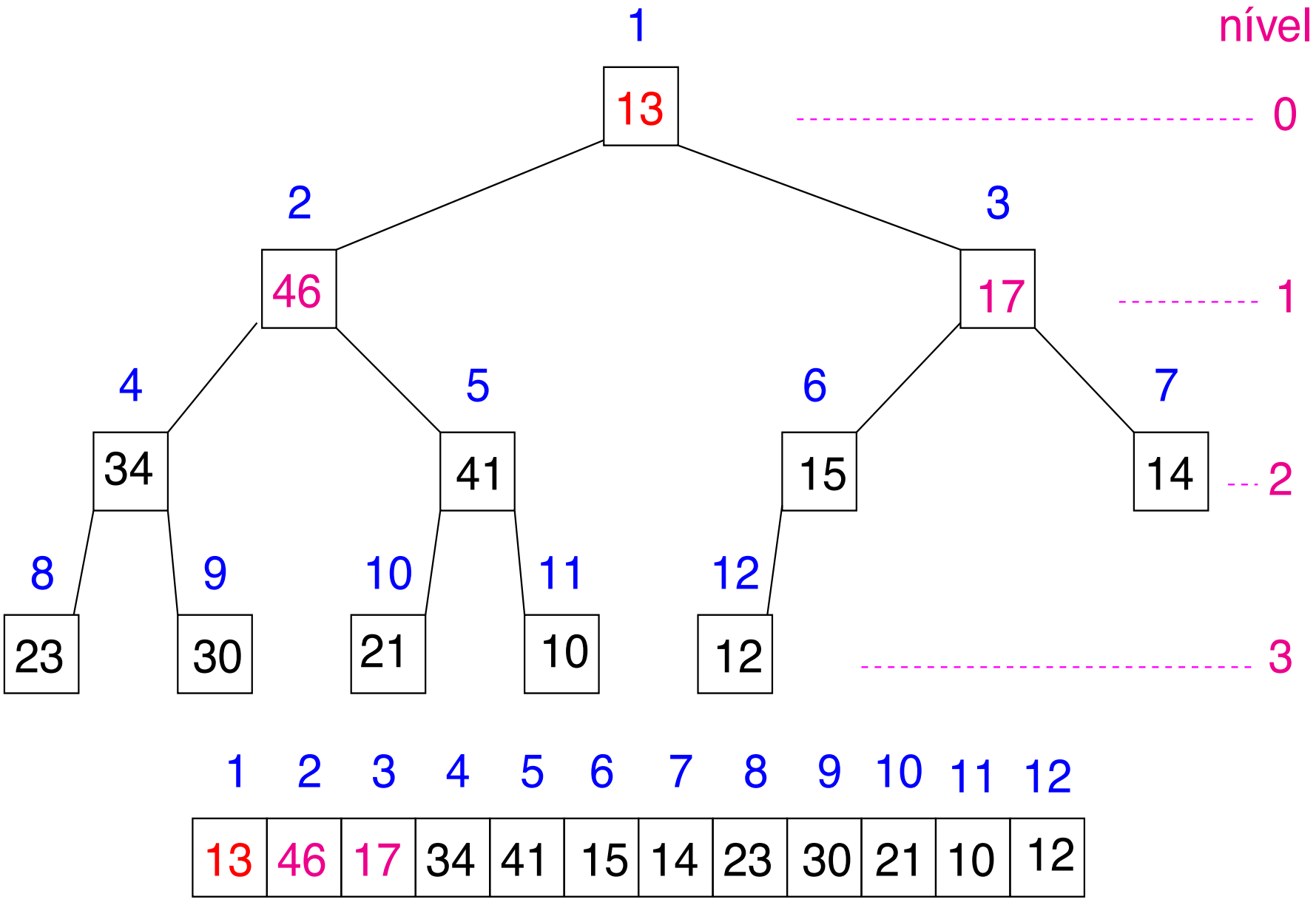
Max-heap


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

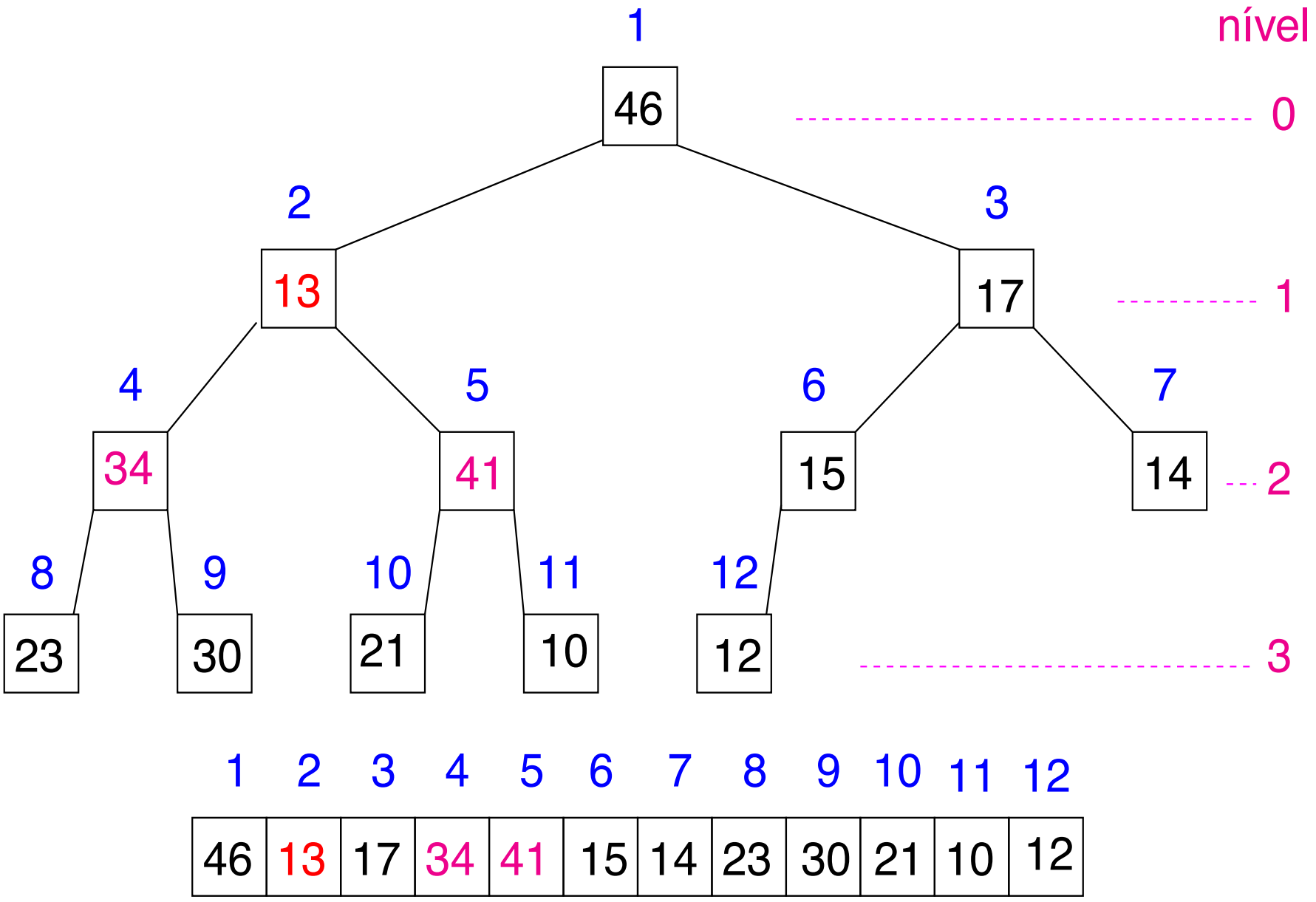
Min-heaps

- Um nó i satisfaz a propriedade de (min-)heap se $A[\lfloor i/2 \rfloor] \leq A[i]$ (ou seja, pai $\leq$ filho).
- Uma árvore binária completa é um *min-heap* se todo nó distinto da raiz satisfaz a propriedade de min-heap.
- Vamos nos concentrar apenas em max-heaps.
- Os algoritmos que veremos podem ser facilmente modificados para trabalhar com min-heaps.

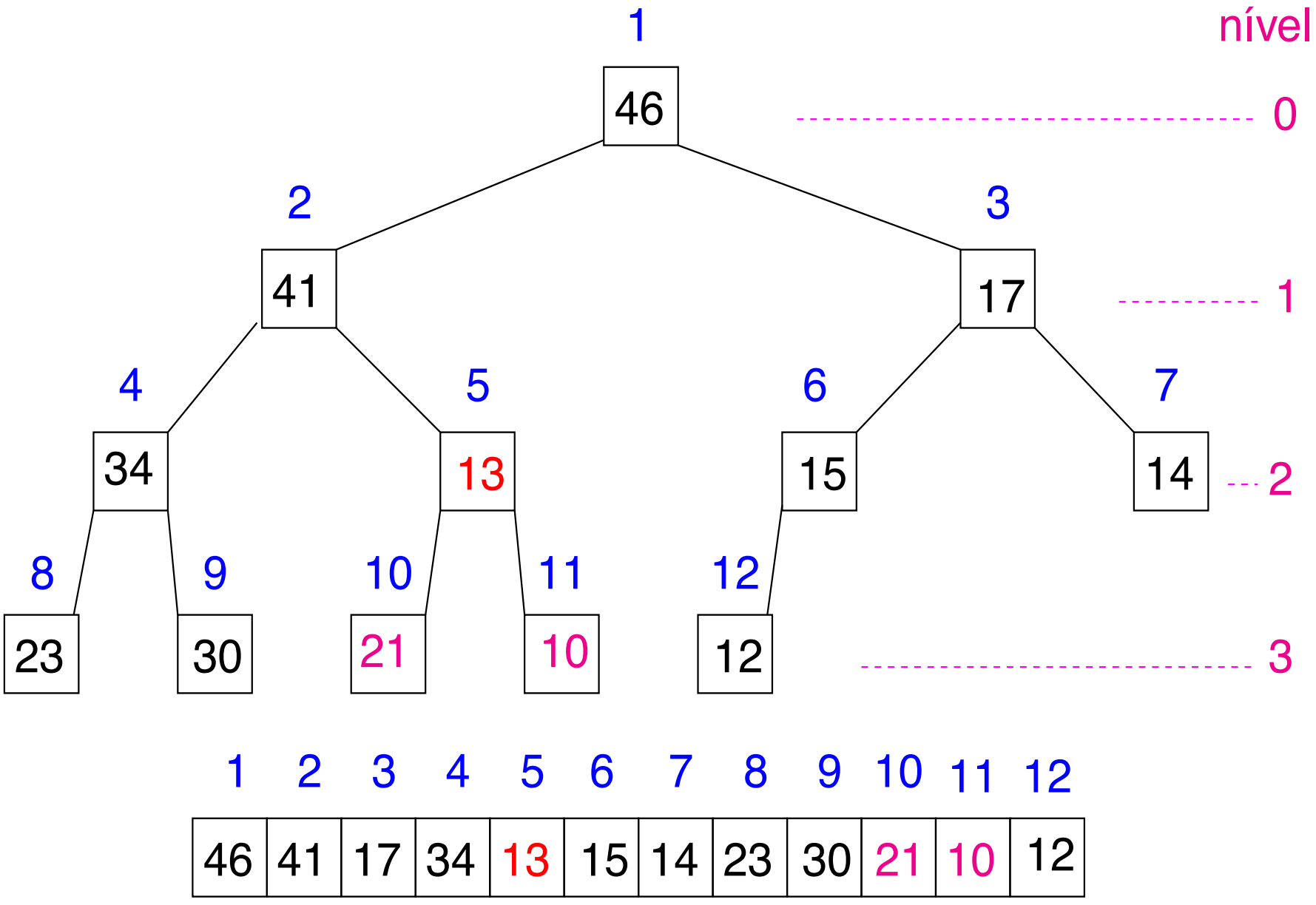
Manipulação de max-heap



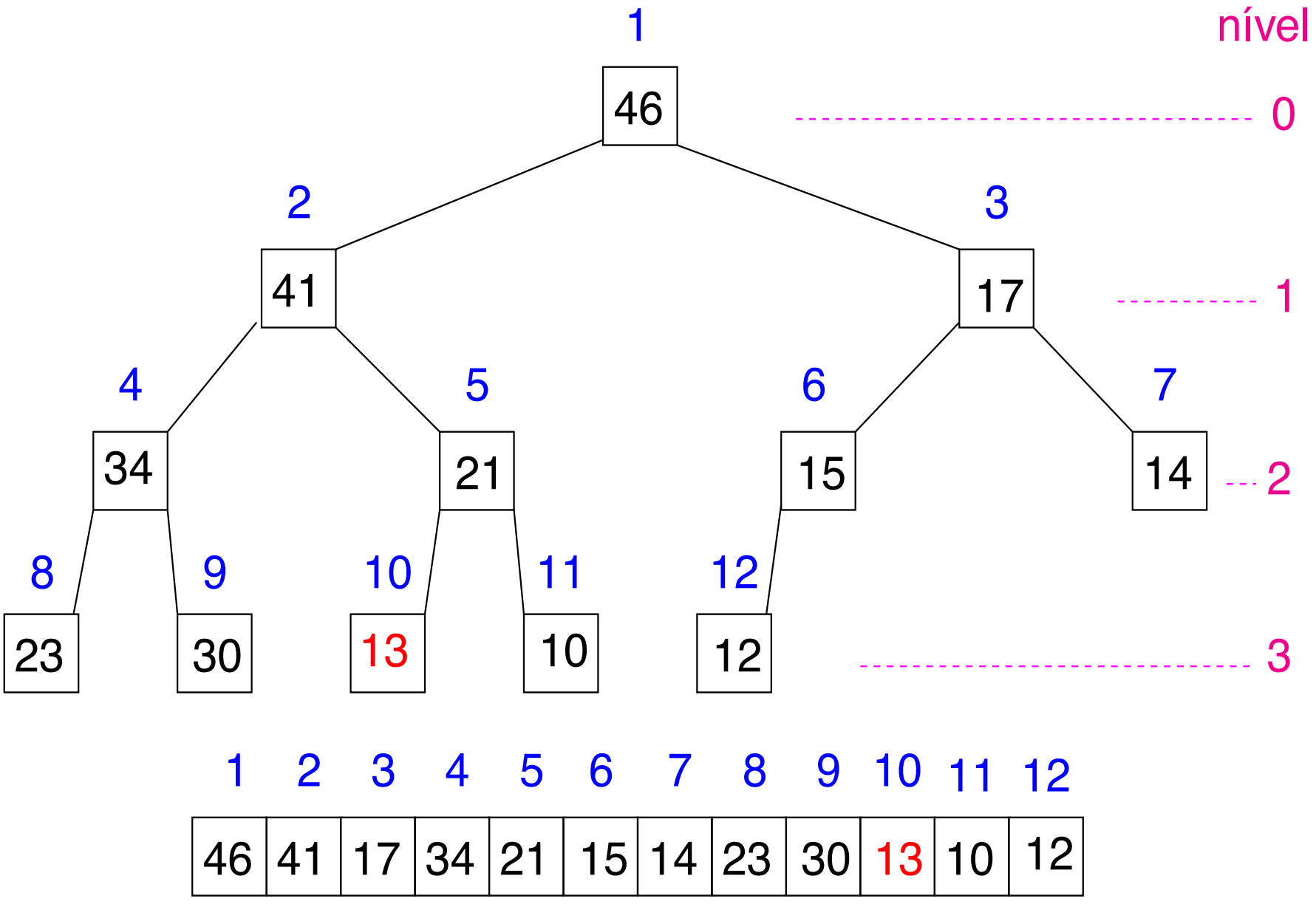
Manipulação de max-heap



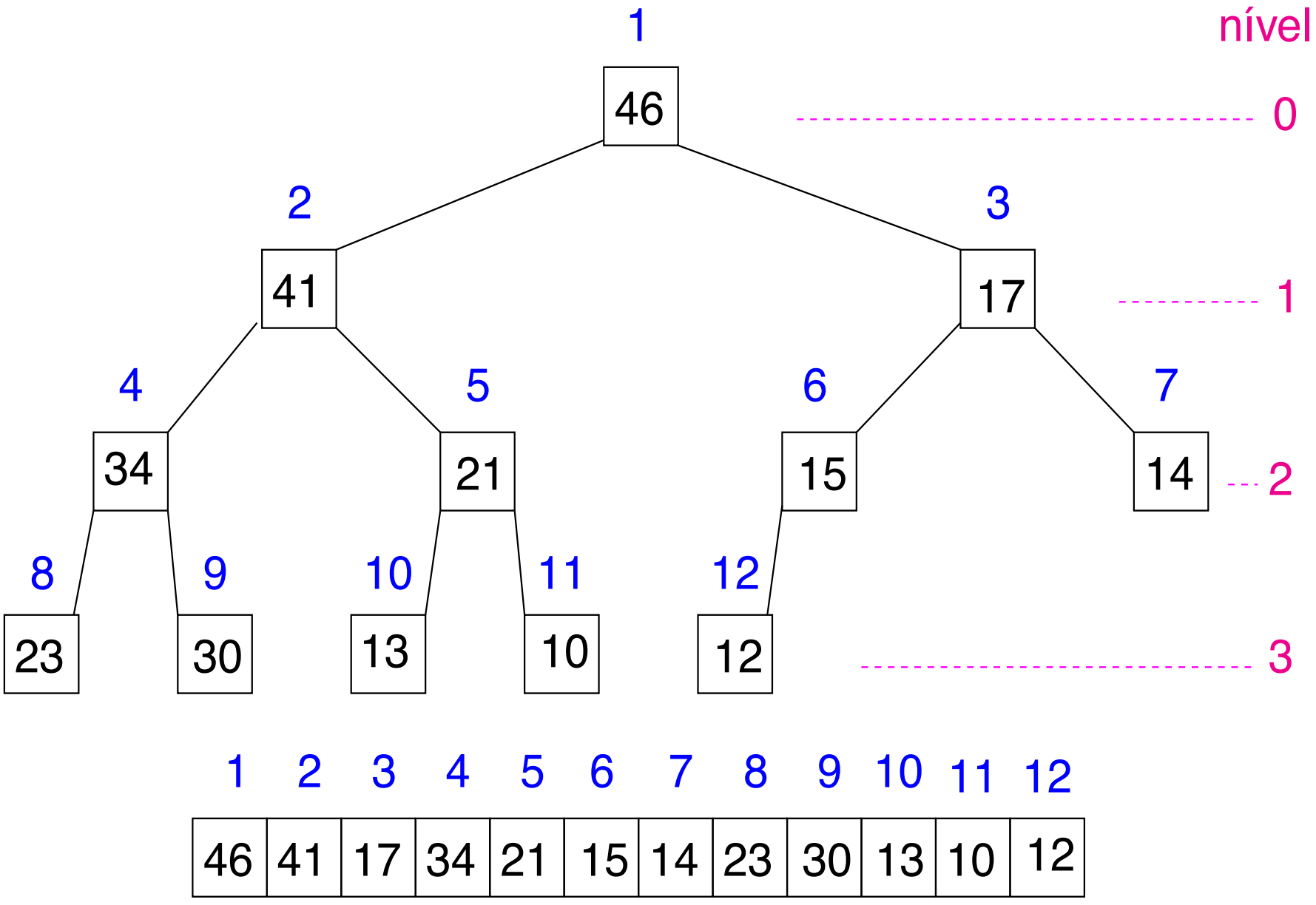
Manipulação de max-heap



Manipulação de max-heap



Manipulação de max-heap



Manipulação de max-heap

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

MAX-HEAPIFY(A, n, i)

```
 1  e ← 2i
 2  d ← 2i + 1
 3  se e ≤ n  e A[e] > A[i]
 4        então maior ← e
 5        senão maior ← i
 6  se d ≤ n e A[d] > A[maior]
 7        então maior ← d
 8  se maior ≠ i
 9        então A[i] ↔ A[maior]
10              MAX-HEAPIFY(A, n, maior)
```

Corretude de MaxHeapify

A corretude de Max-Heapify segue por indução na altura h do nó i.

Base: para $h = 1$, o algoritmo funciona.

Hipótese de indução: Max-Heapify funciona para heaps de altura $< h$.

Passo de indução:

A variável maior na linha 8 guarda o índice do maior elemento entre $A[i]$, $A[2i]$ e $A[2i + 1]$.
Após a troca na linha 9, temos $A[2i], A[2i + 1] \leq A[i]$.

O algoritmo Max-Heapify transforma a subárvore com raiz maior em um max-heap (hipótese de indução).

Corretude de MaxHeapify

Passo de indução:

A variável maior na linha 8 guarda o índice do maior elemento entre $A[i]$, $A[2i]$ e $A[2i+1]$.
Após a troca na linha 9, temos $A[2i], A[2i+1] \leq A[i]$.

O algoritmo Max-Heapify transforma a subárvore com raiz maior em um max-heap (hipótese de indução).

A subárvore cuja raiz é o irmão de maior continua sendo um max-heap.

Logo, a subárvore com raiz i torna-se um max-heap e portanto, o algoritmo Max-Heapify está correto.

Complexidade de MaxHeapify

MAX-HEAPIFY(A, n, i)		Tempo
1	$e \leftarrow 2i$	?
2	$d \leftarrow 2i + 1$	?
3	**se** $e \leq n$ **e** $A[e] > A[i]$	?
4	**então** maior $\leftarrow e$	?
5	**senão** maior $\leftarrow i$	?
6	**se** $d \leq n$ e $A[d] > A[\text{maior}]$	?
7	**então** maior $\leftarrow d$	?
8	**se** maior $\neq i$	?
9	**então** $A[i] \leftrightarrow A[\text{maior}]$	?
10	MAX-HEAPIFY(A, n, maior)	?

h := altura de i = $\lfloor \lg \frac{n}{i} \rfloor$

$T(h)$:= complexidade de tempo no pior caso

Complexidade de MaxHeapify

MAX-HEAPIFY(A, n, i)		Tempo
1	$e \leftarrow 2i$	$\Theta(1)$
2	$d \leftarrow 2i + 1$	$\Theta(1)$
3	**se** $e \leq n$ **e** $A[e] > A[i]$	$\Theta(1)$
4	**então** maior $\leftarrow e$	$O(1)$
5	**senão** maior $\leftarrow i$	$O(1)$
6	**se** $d \leq n$ e $A[d] > A[\text{maior}]$	$\Theta(1)$
7	**então** maior $\leftarrow d$	$O(1)$
8	**se** maior $\neq i$	$\Theta(1)$
9	**então** $A[i] \leftrightarrow A[\text{maior}]$	$O(1)$
10	MAX-HEAPIFY(A, n, maior)	$T(h-1)$

h := altura de $i = \lfloor \lg \frac{n}{i} \rfloor$

$T(h) \leq T(h-1) + \Theta(5) + O(2)$.

Complexidade de MaxHeapify

h := altura de $i = \lfloor \lg \frac{n}{i} \rfloor$

$T(h)$:= complexidade de tempo no pior caso

$T(h) \leq T(h-1) + \Theta(1)$

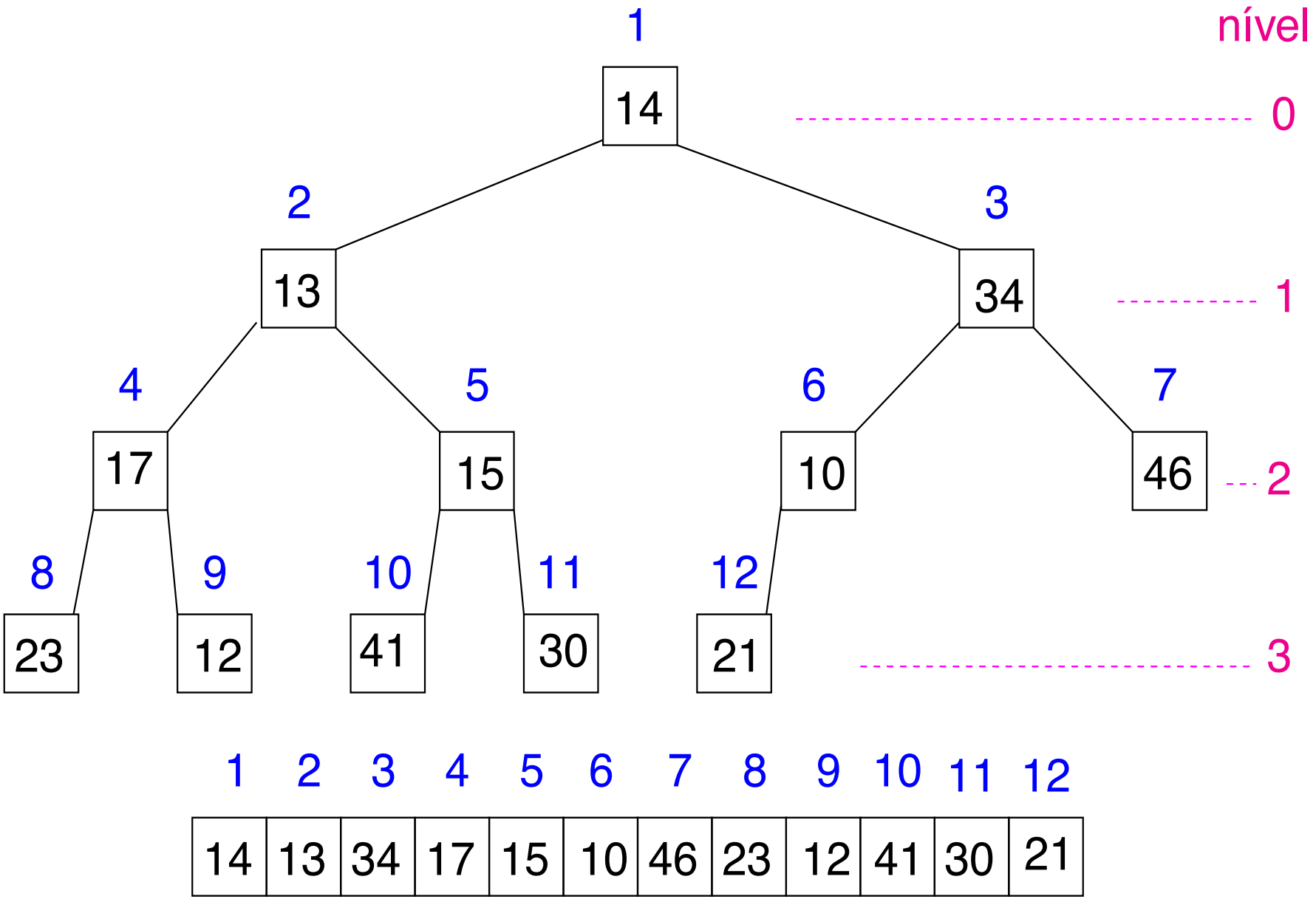
Solução assintótica: $T(n)$ é ???.

Solução assintótica: $T(n)$ é $O(h)$.

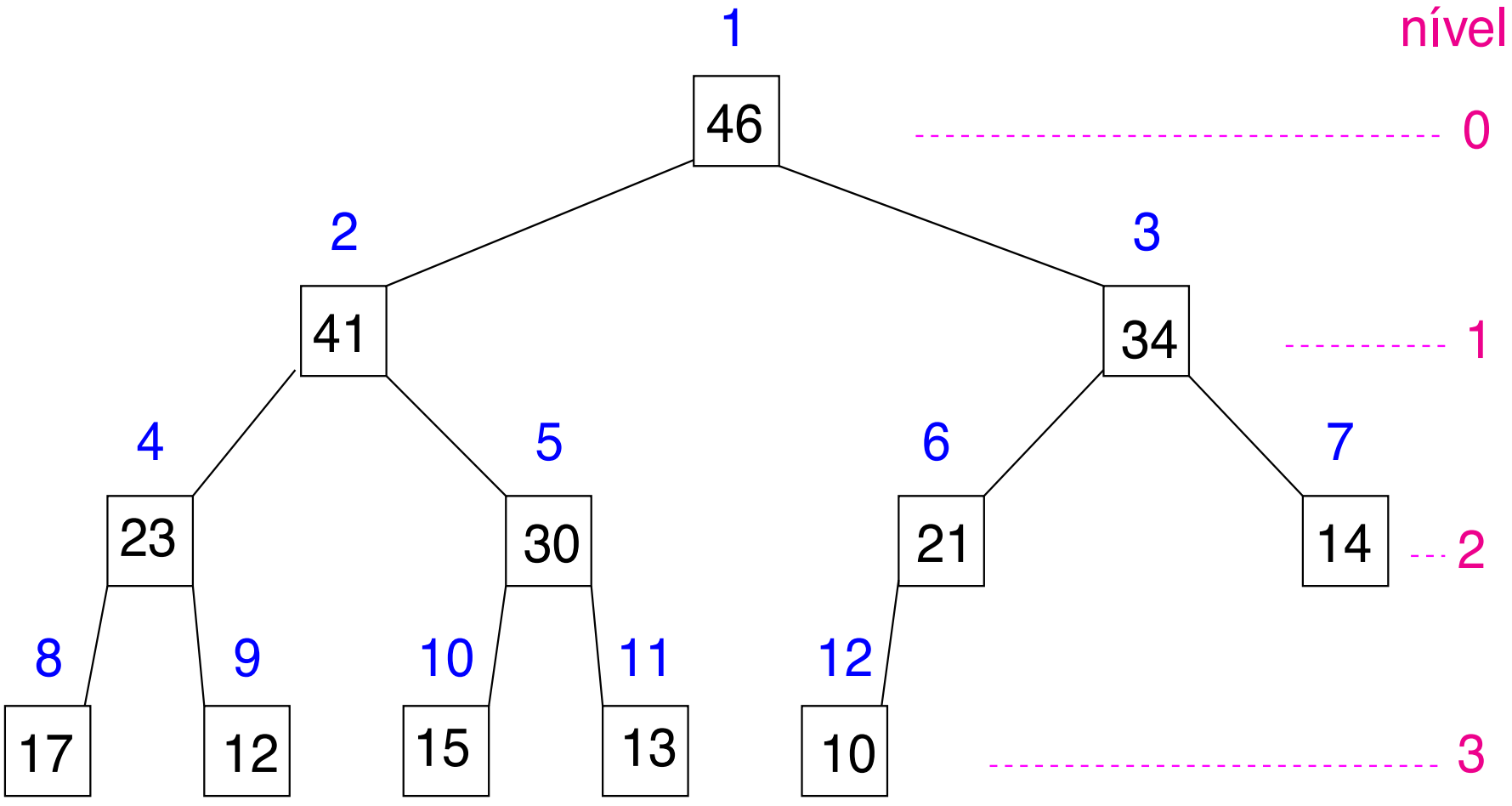
Como $h \leq \lg n$, podemos dizer que:

O consumo de tempo do algoritmo Max-Heapify é $O(\lg n)$ (ou melhor ainda, $O(\lg \frac{n}{i})$).

Construção de um max-heap



Construção de um max-heap



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

Construção de um max-heap

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

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

Invariante:

No início de cada iteração, $i + 1, \ldots, n$ são raízes de max-heaps.

$T(n) =$ complexidade de tempo no pior caso

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

Construção de um max-heap

Análise mais cuidadosa: $T(n)$ é $O(n)$.

- Na iteração i são feitas $O(h_i)$ comparações e trocas no pior caso, onde h_i é a altura da subárvore de raiz i.
- Seja $S(h)$ a soma das alturas de todos os nós de uma árvore binária completa de altura h.
- A altura de um heap é $\lfloor \lg n \rfloor + 1$.

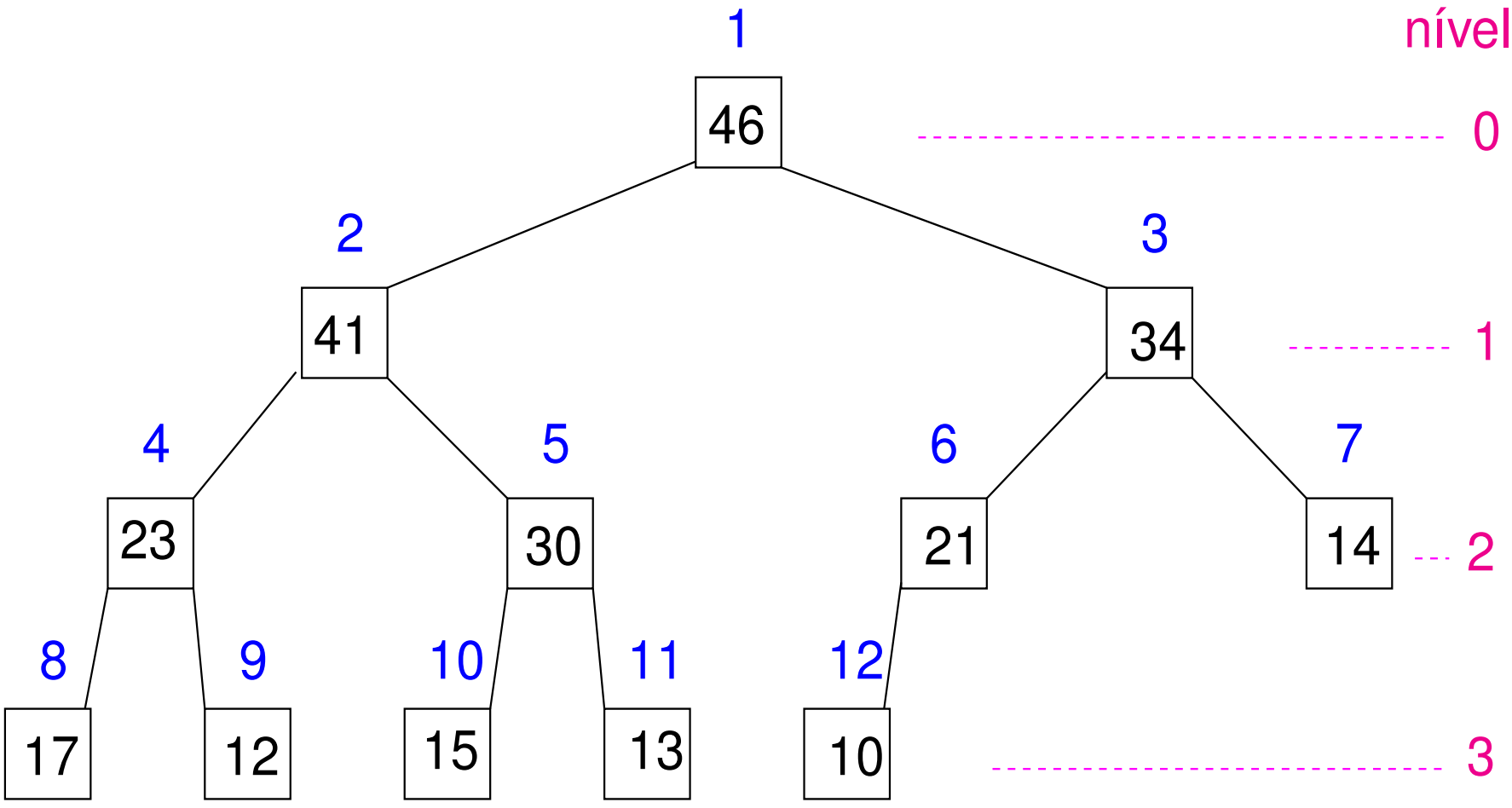
  A complexidade de BUILDMAXHEAP é $T(n) = O(S(\lg n))$.

Construção de um max-heap

- Pode-se provar por indução que $S(h) = 2^{h+1} - h - 2$.
- Logo, a complexidade de BUILDMAXHEAP é $T(n) = O(S(\lg n)) = O(n)$.

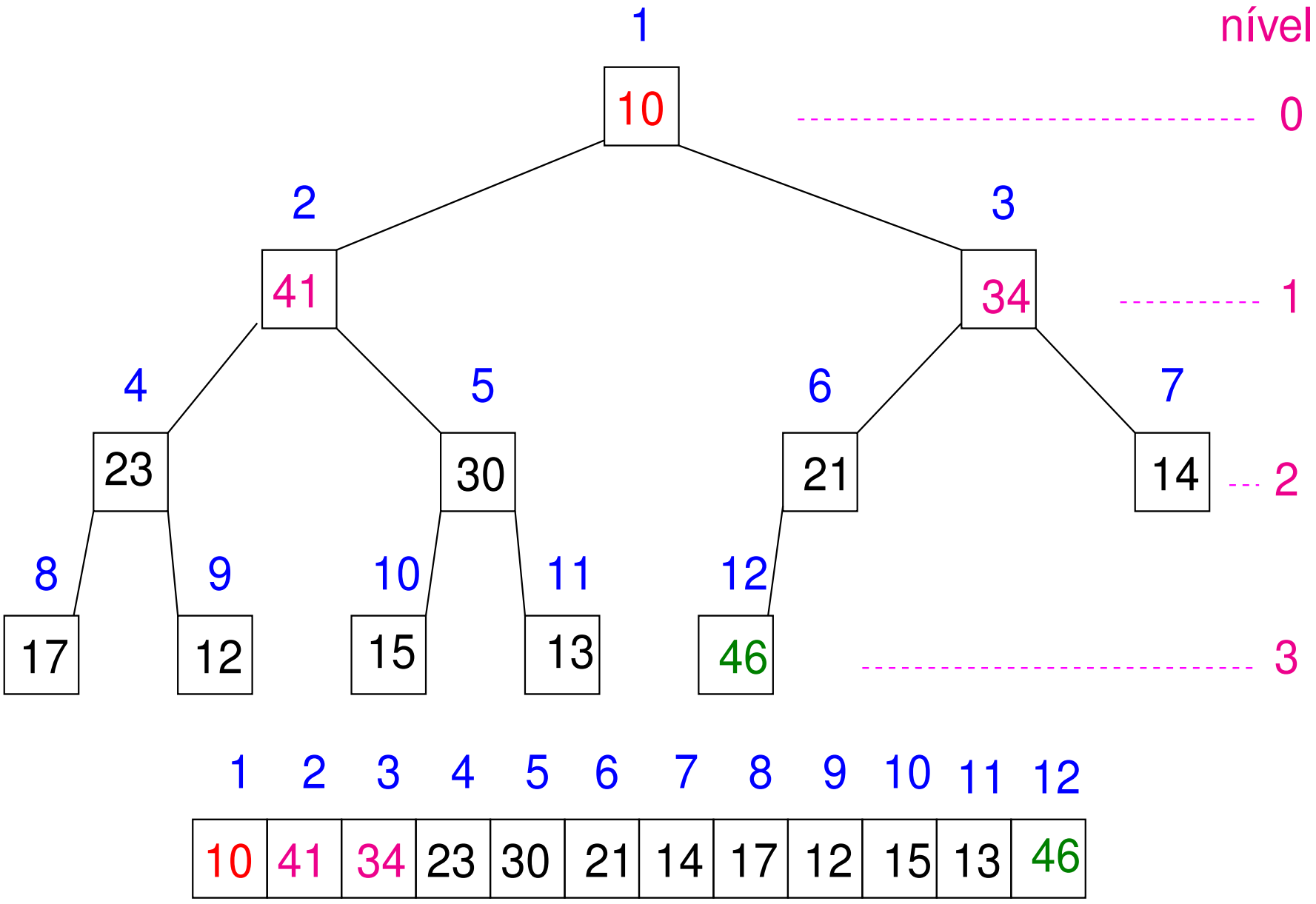
  Mais precisamente, $T(n) = \Theta(n)$. (Por quê?)
- Veja no CLRS uma prova diferente deste fato.

HeapSort

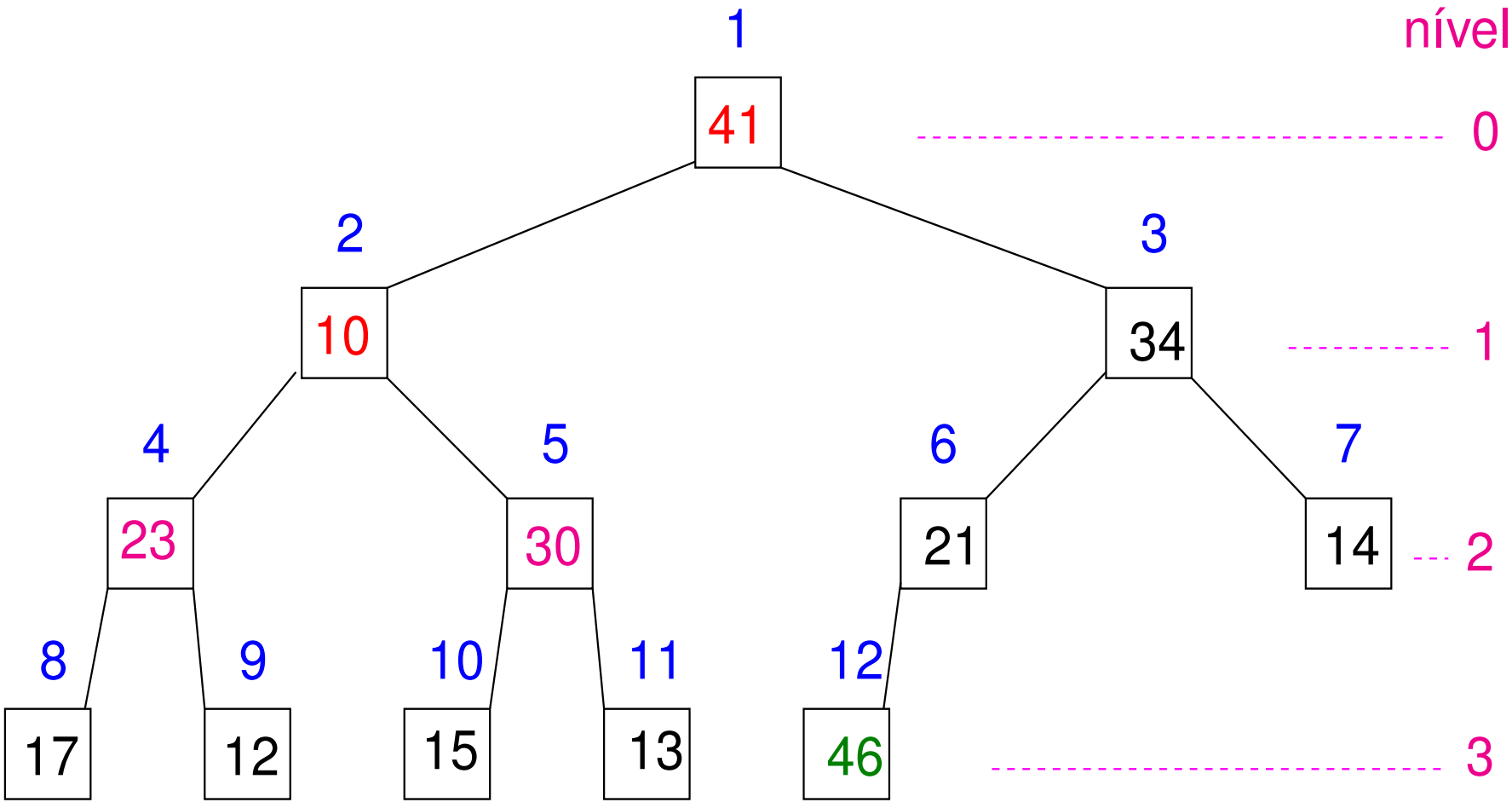


1	2	3	4	5	6	7	8	9	10	11	12
46	41	34	23	30	21	14	17	12	15	13	10

HeapSort

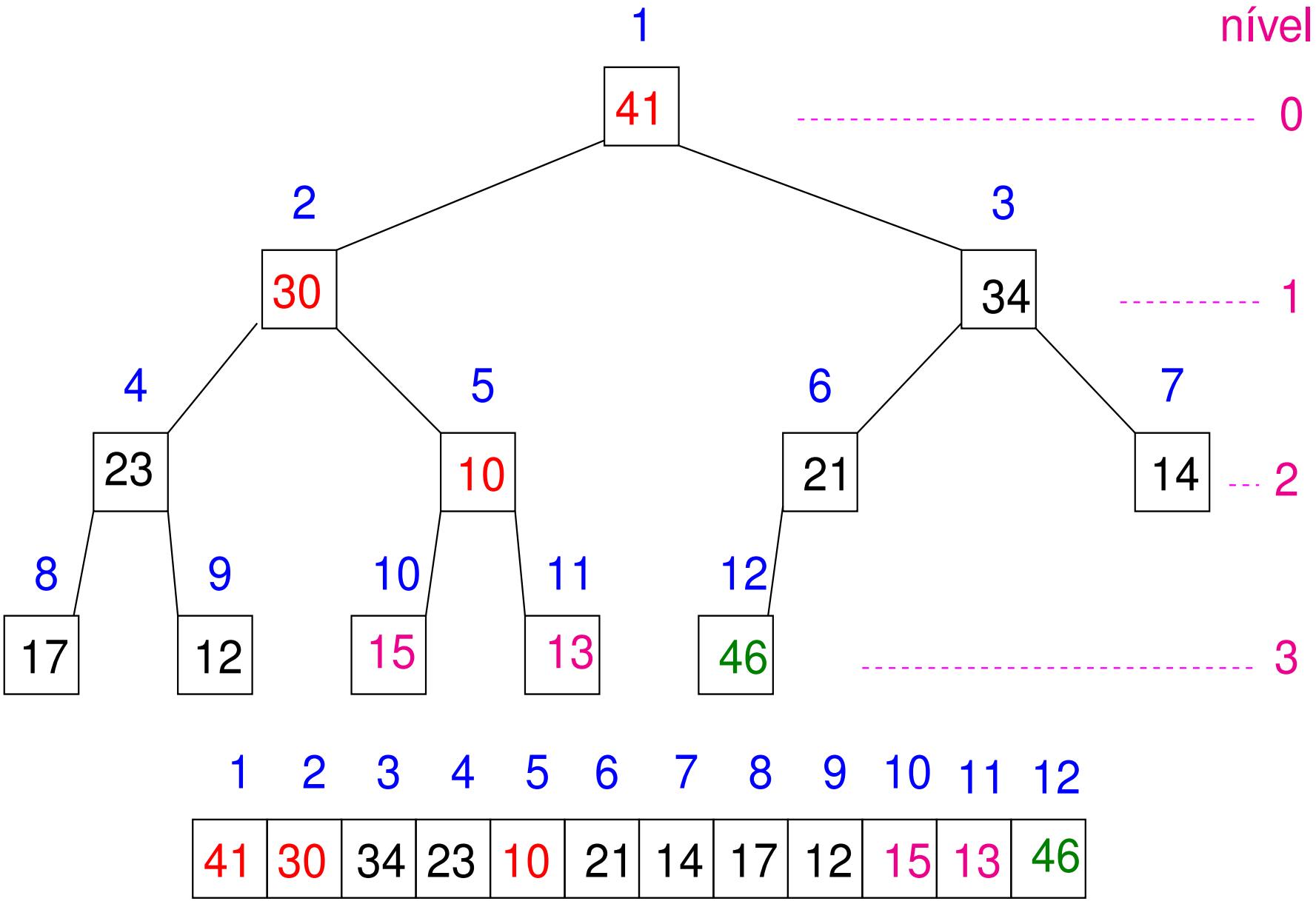

1
nível
10
0
2
3
41
34
1
4
5
6
7
23
30
21
14
2
8
9
10
11
12
17
12
15
13
46
3
1 2 3 4 5 6 7 8 9 10 11 12
10 41 34 23 30 21 14 17 12 15 13 46

HeapSort

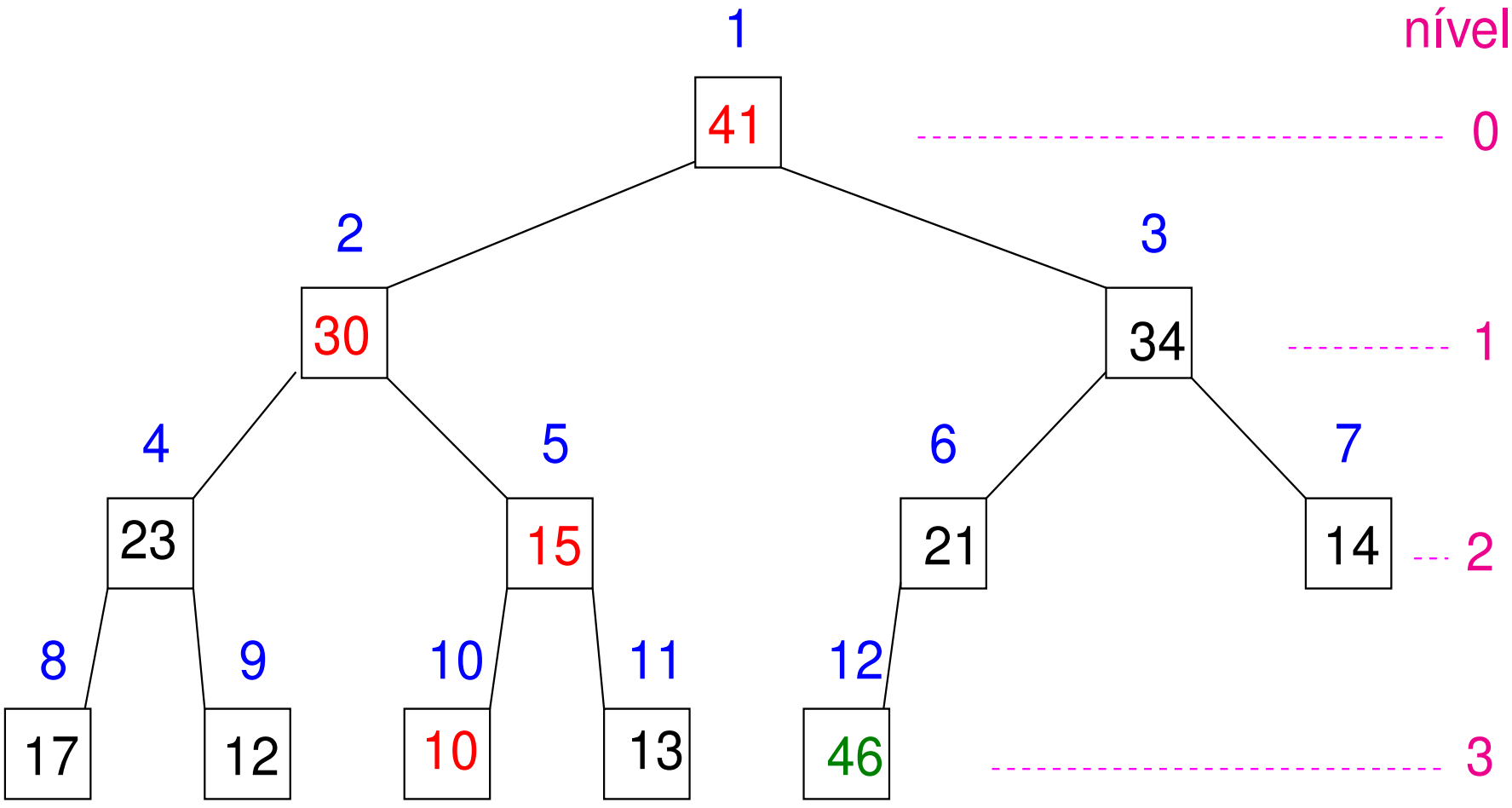


1	2	3	4	5	6	7	8	9	10	11	12
41	10	34	23	30	21	14	17	12	15	13	46

HeapSort

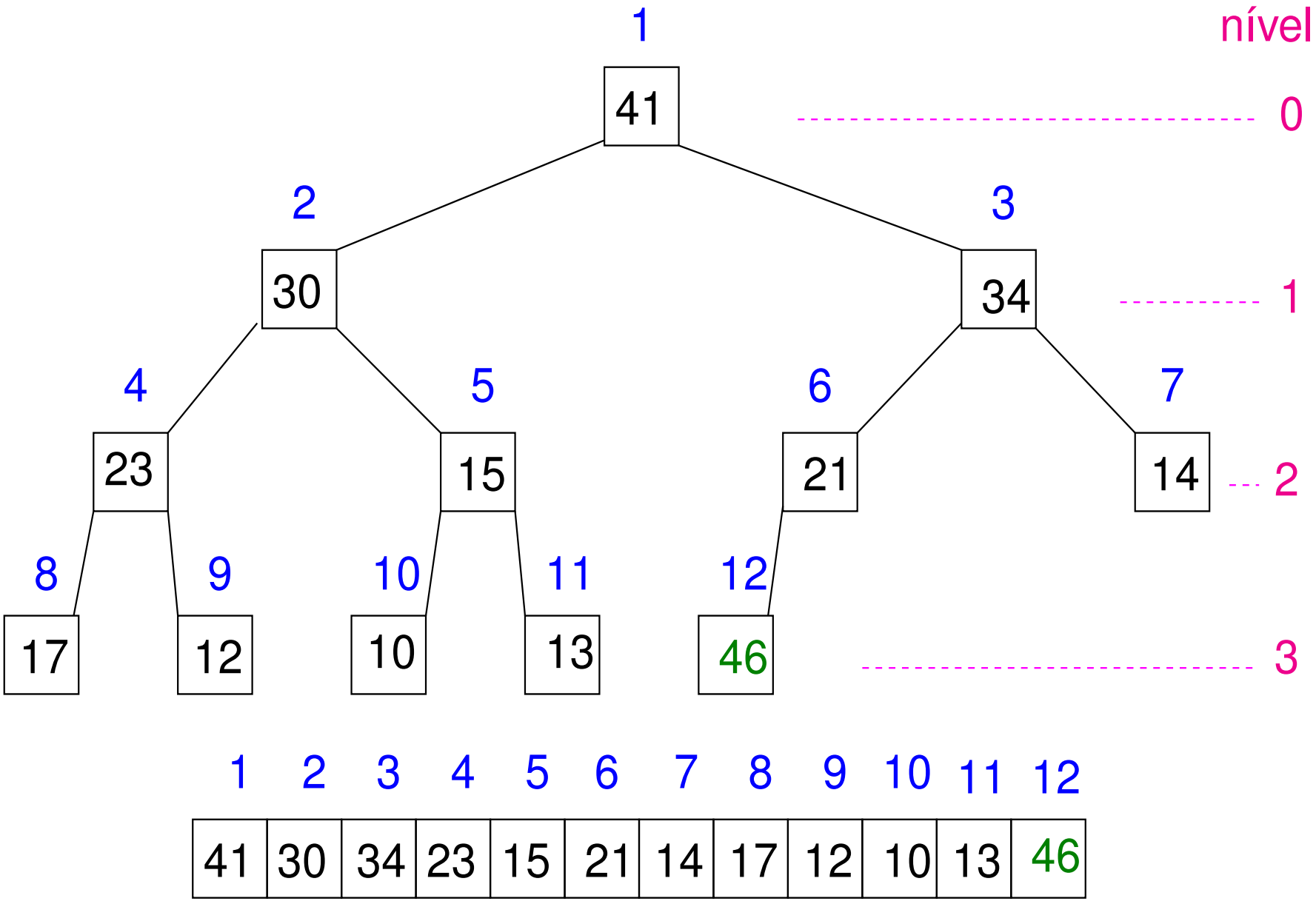

1
nível
41
0
2
3
30
34
1
4
5
6
7
23
10
21
14
2
8
9
10
11
12
17
12
15
13
46
3
1 2 3 4 5 6 7 8 9 10 11 12
41 30 34 23 10 21 14 17 12 15 13 46

HeapSort

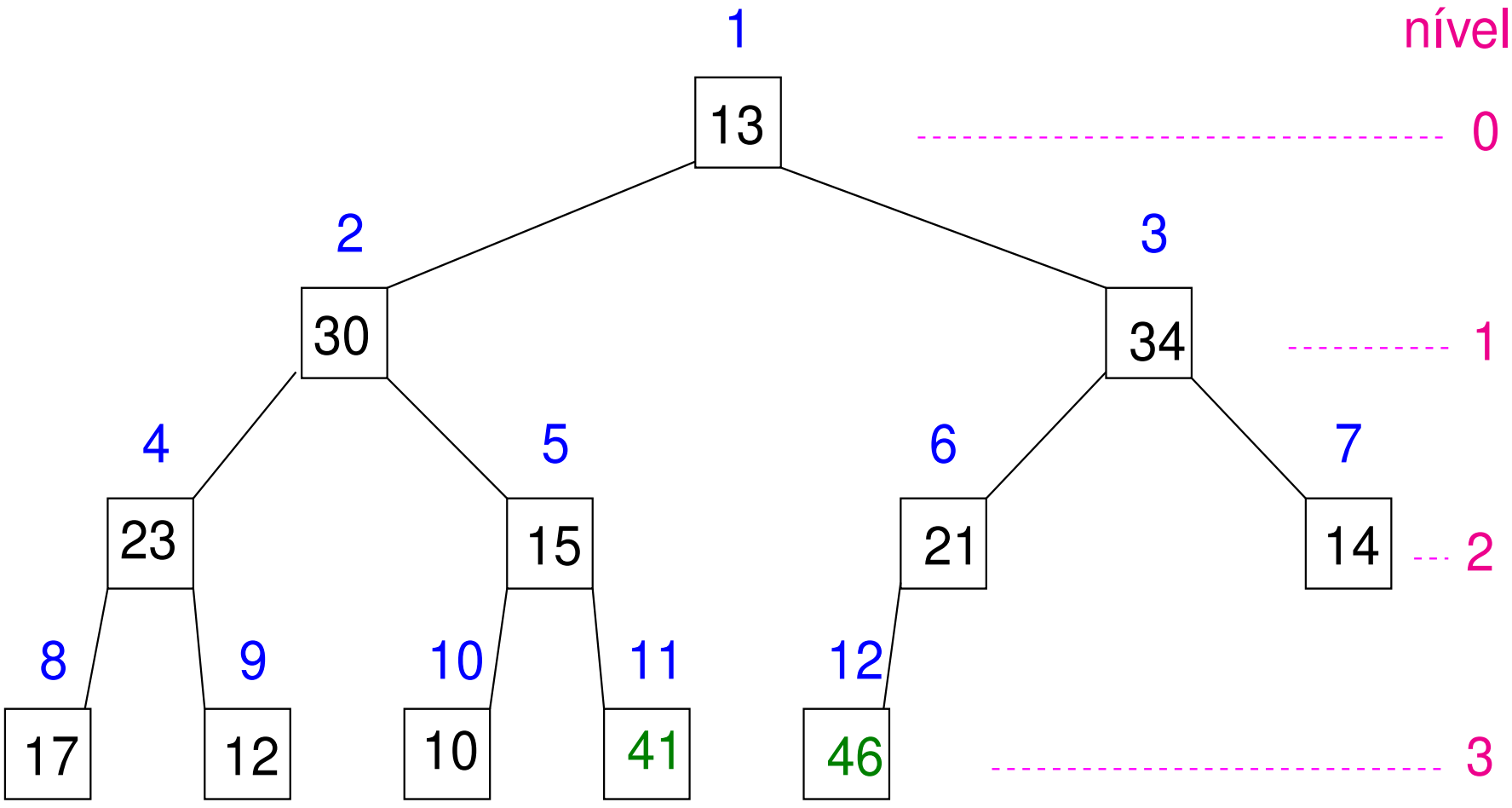


1	2	3	4	5	6	7	8	9	10	11	12
41	30	34	23	15	21	14	17	12	10	13	46

HeapSort

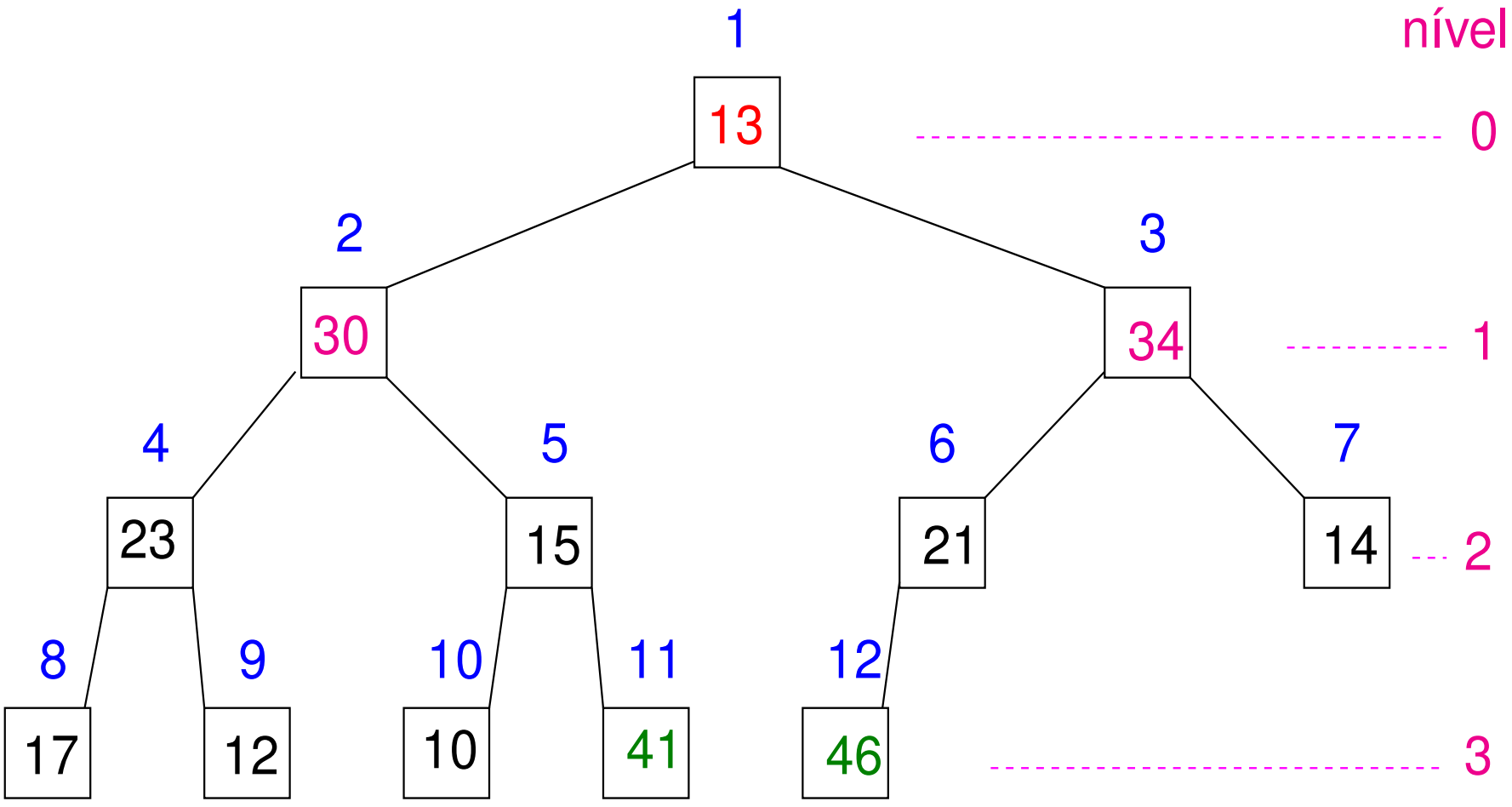

1
nível
41
0
2
3
30
34
1
4
5
6
7
23
15
21
14
2
8
9
10
11
12
17
12
10
13
46
3
1 2 3 4 5 6 7 8 9 10 11 12
41 30 34 23 15 21 14 17 12 10 13 46

HeapSort



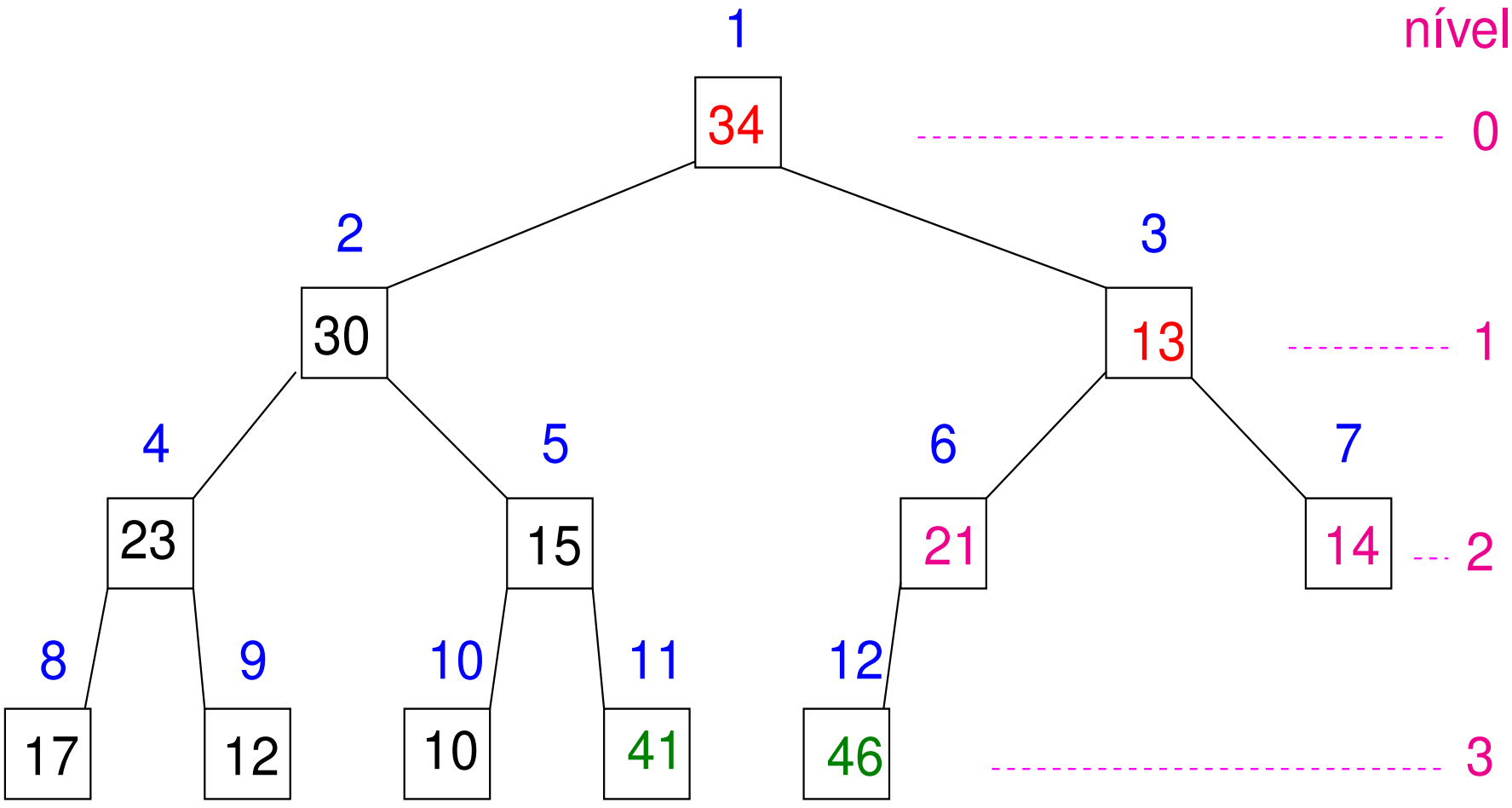
1	2	3	4	5	6	7	8	9	10	11	12
13	30	34	23	15	21	14	17	12	10	41	46

HeapSort



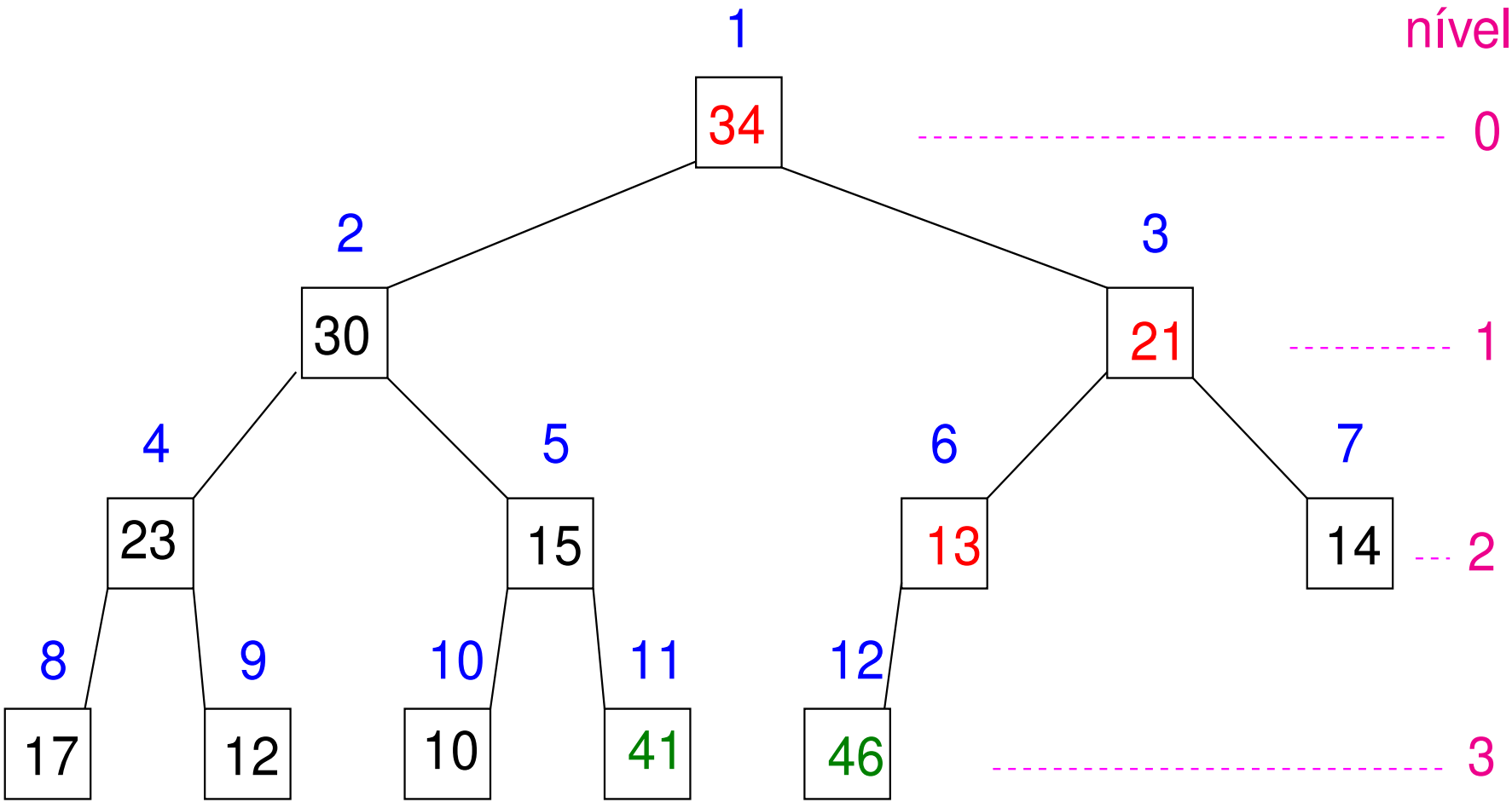
1	2	3	4	5	6	7	8	9	10	11	12
13	30	34	23	15	21	14	17	12	10	41	46

HeapSort



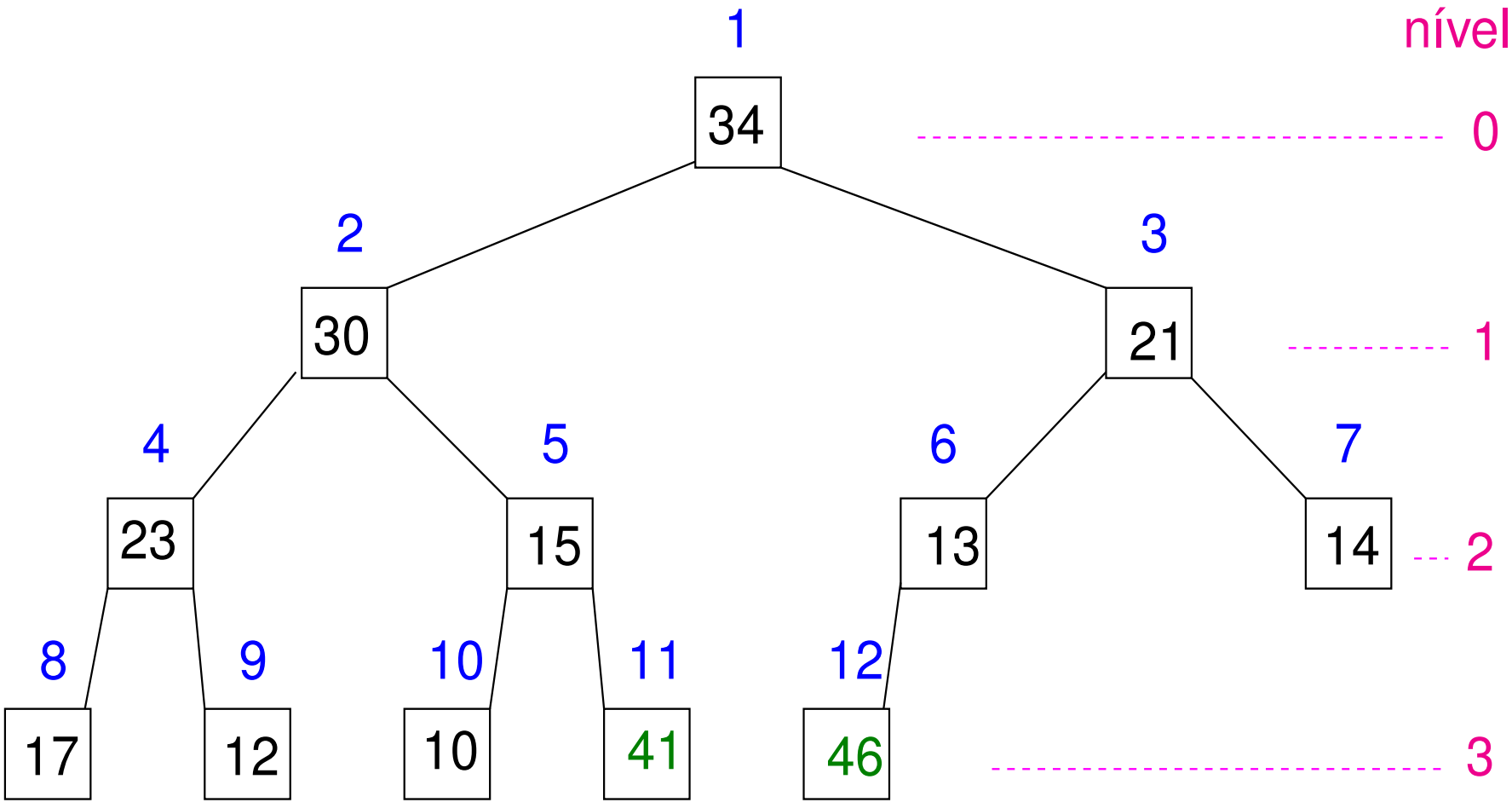
1	2	3	4	5	6	7	8	9	10	11	12
34	30	13	23	15	21	14	17	12	10	41	46

HeapSort



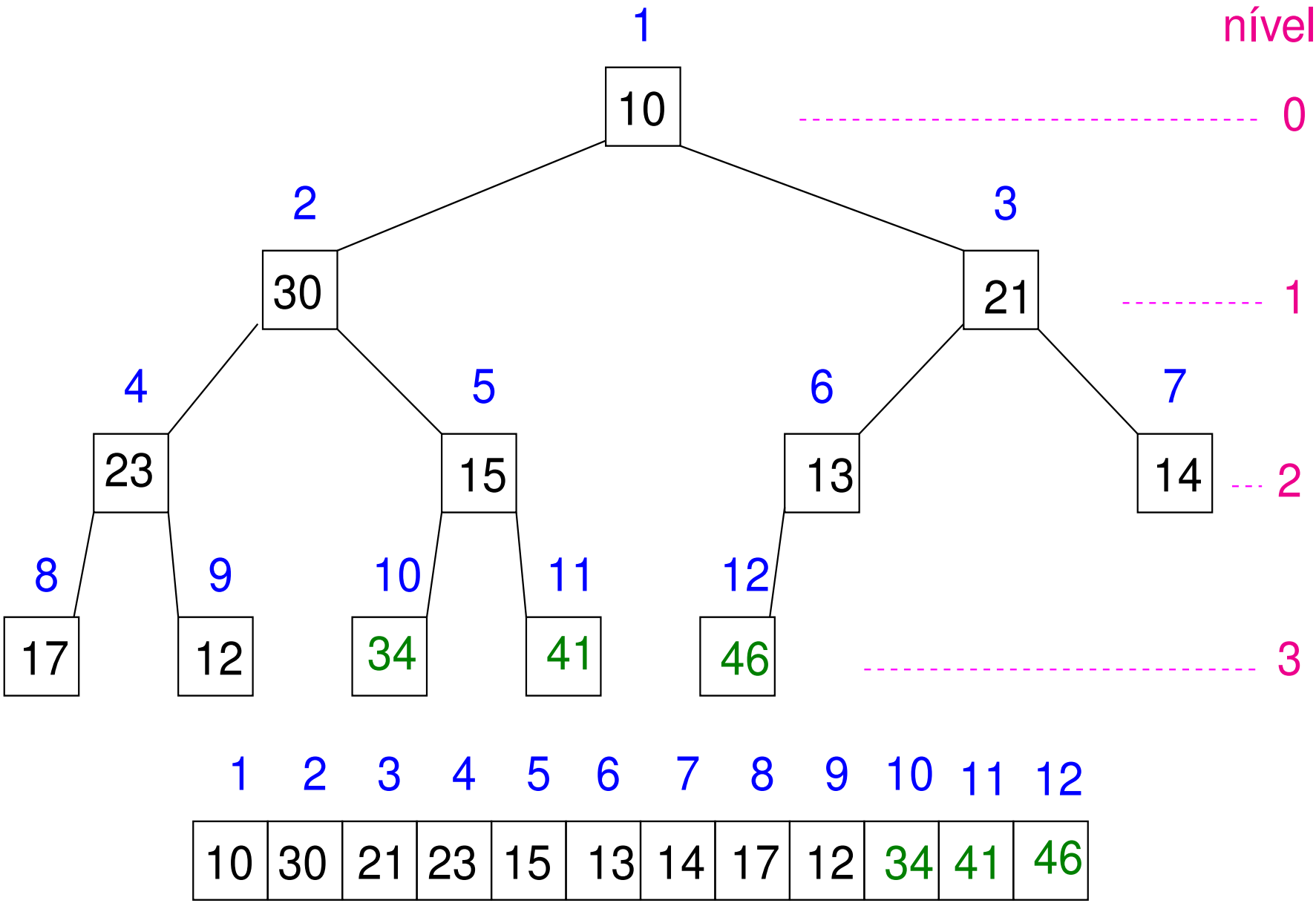
1	2	3	4	5	6	7	8	9	10	11	12
34	30	21	23	15	13	14	17	12	10	41	46

HeapSort

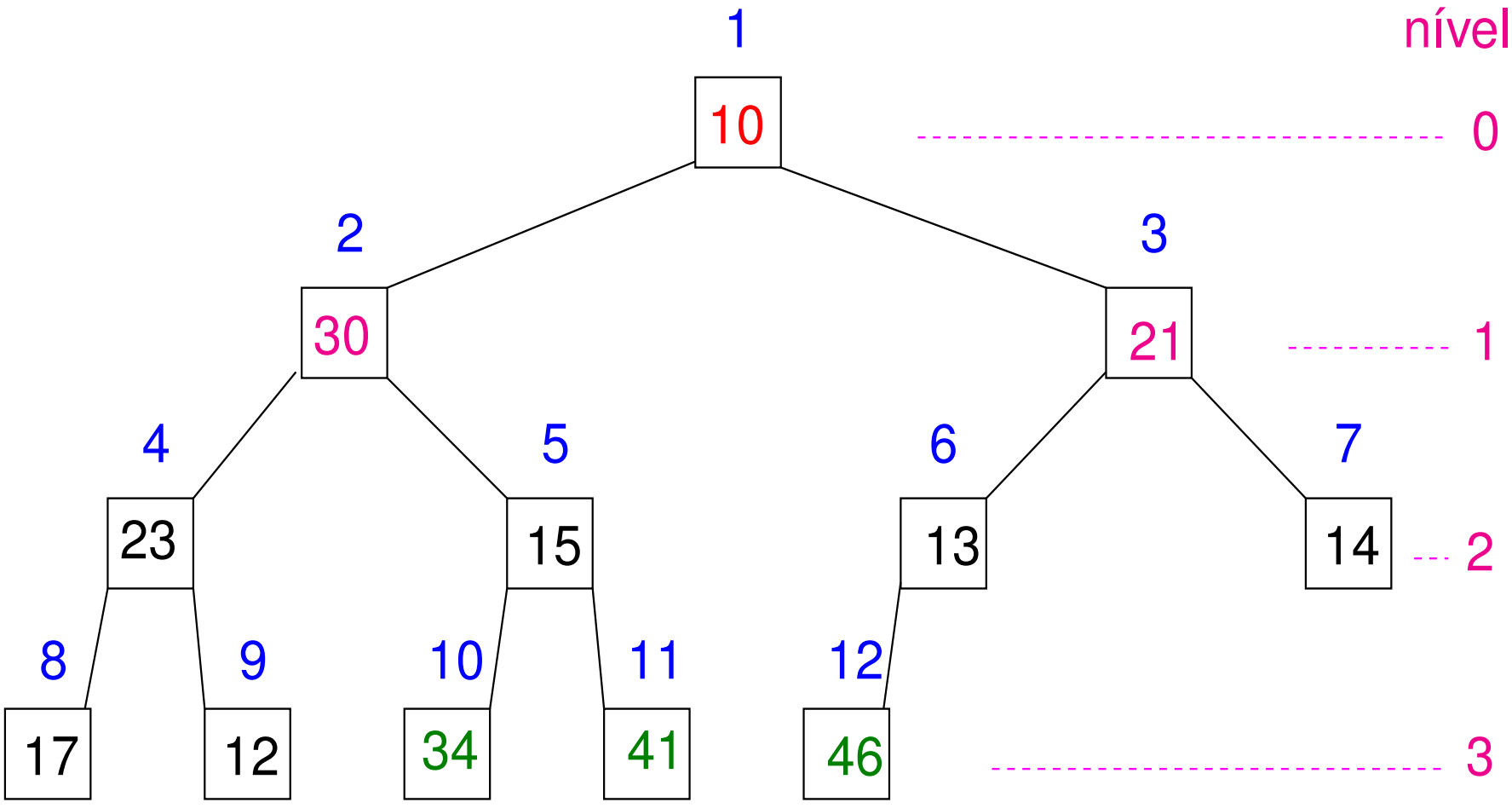


1	2	3	4	5	6	7	8	9	10	11	12
34	30	21	23	15	13	14	17	12	10	41	46

HeapSort

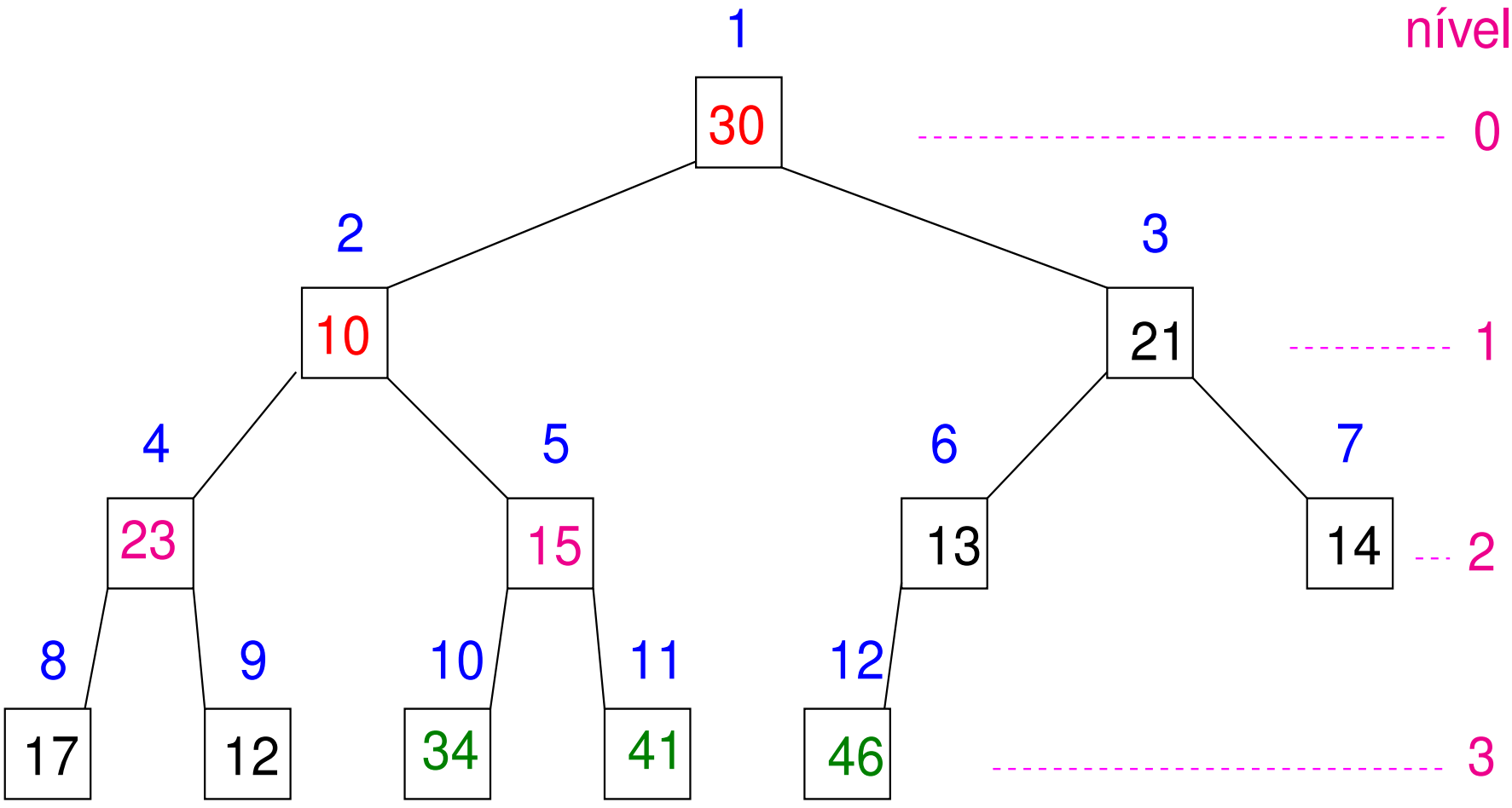

1
nível
10
0
2
3
30
21
1
4
5
6
7
23
15
13
14
2
8
9
10
11
12
17
12
34
41
46
3
1 2 3 4 5 6 7 8 9 10 11 12
10 30 21 23 15 13 14 17 12 34 41 46

HeapSort



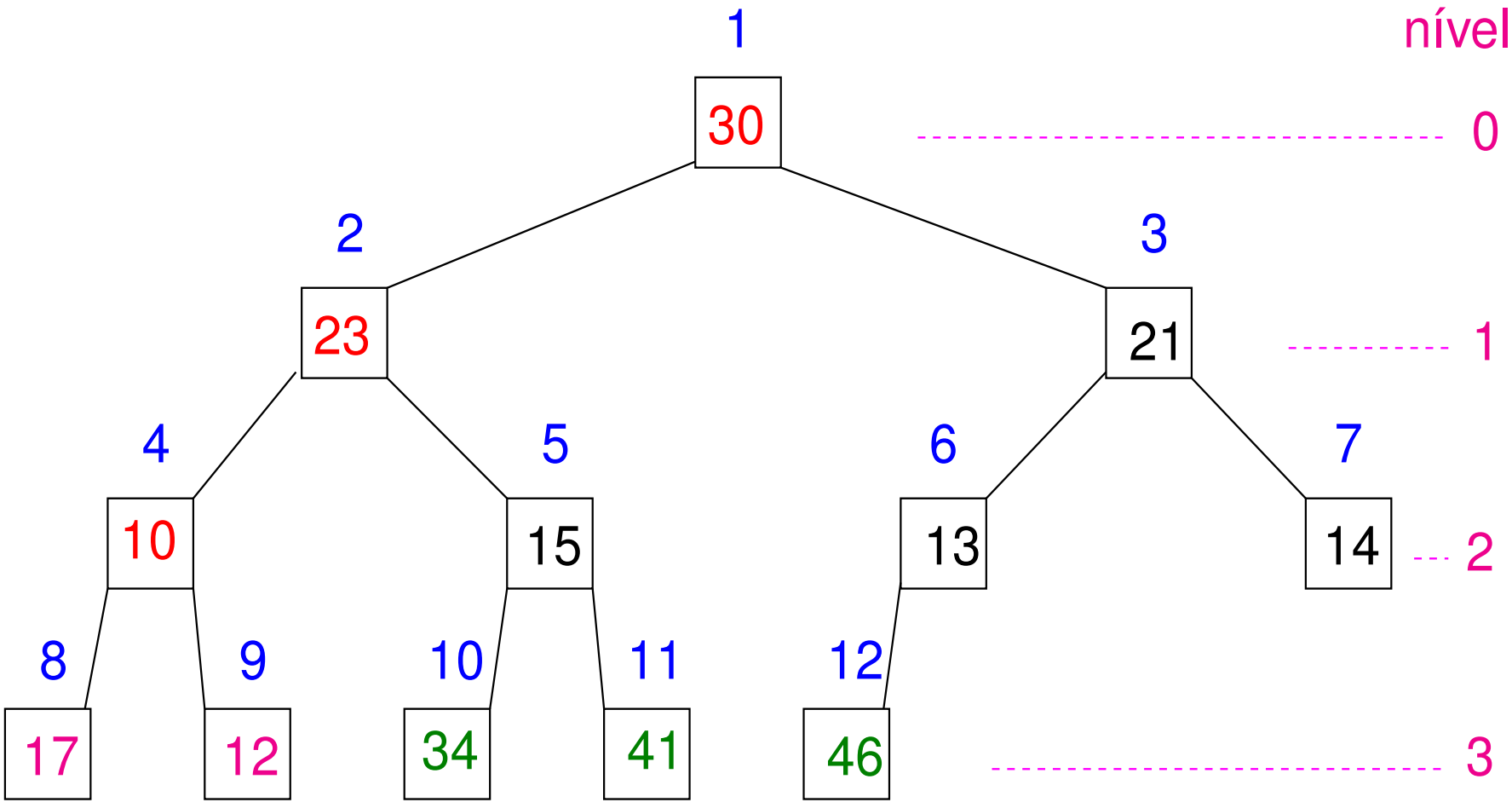
1	2	3	4	5	6	7	8	9	10	11	12
10	30	21	23	15	13	14	17	12	34	41	46

HeapSort



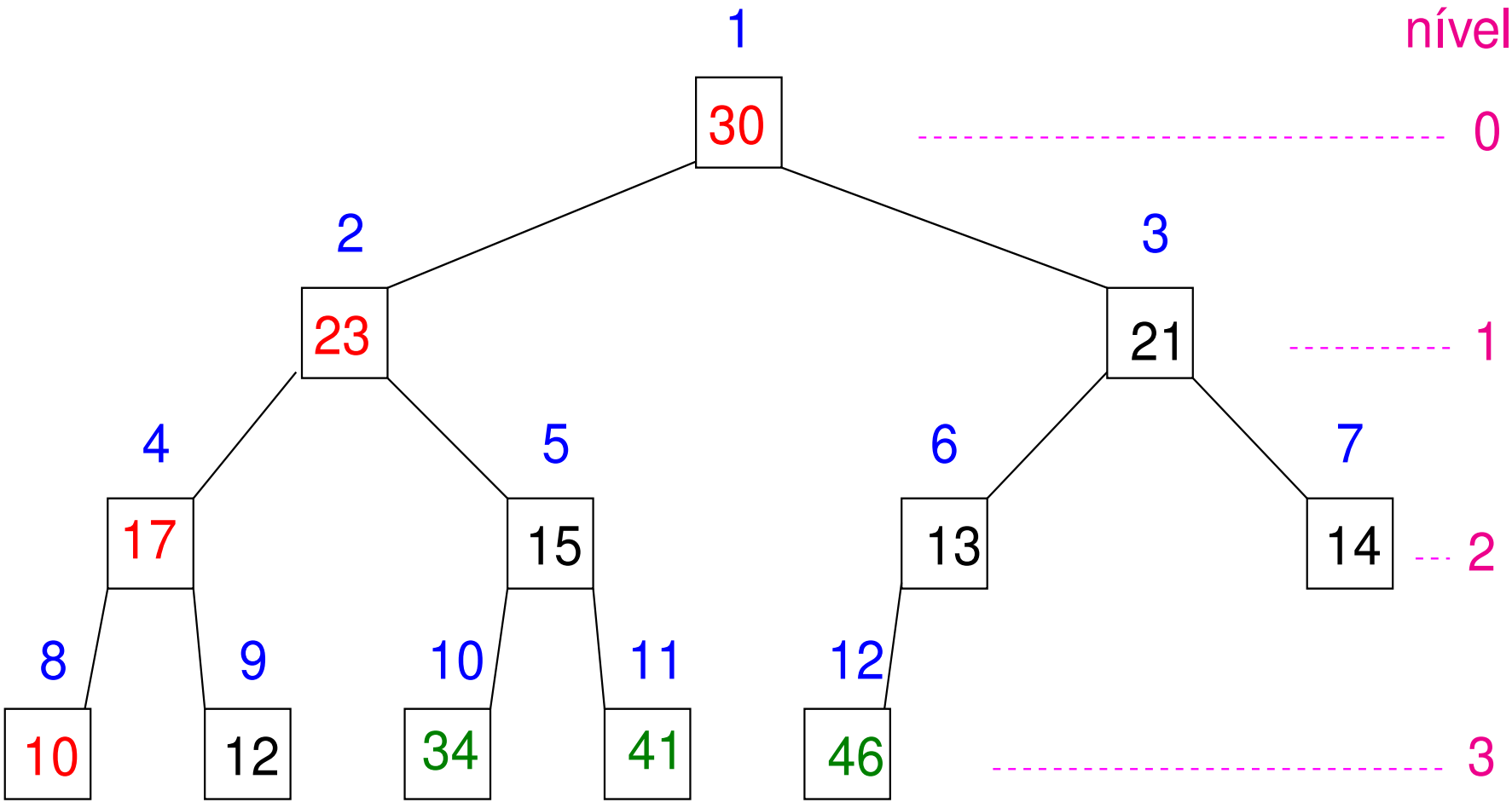
1	2	3	4	5	6	7	8	9	10	11	12
30	10	21	23	15	13	14	17	12	34	41	46

HeapSort



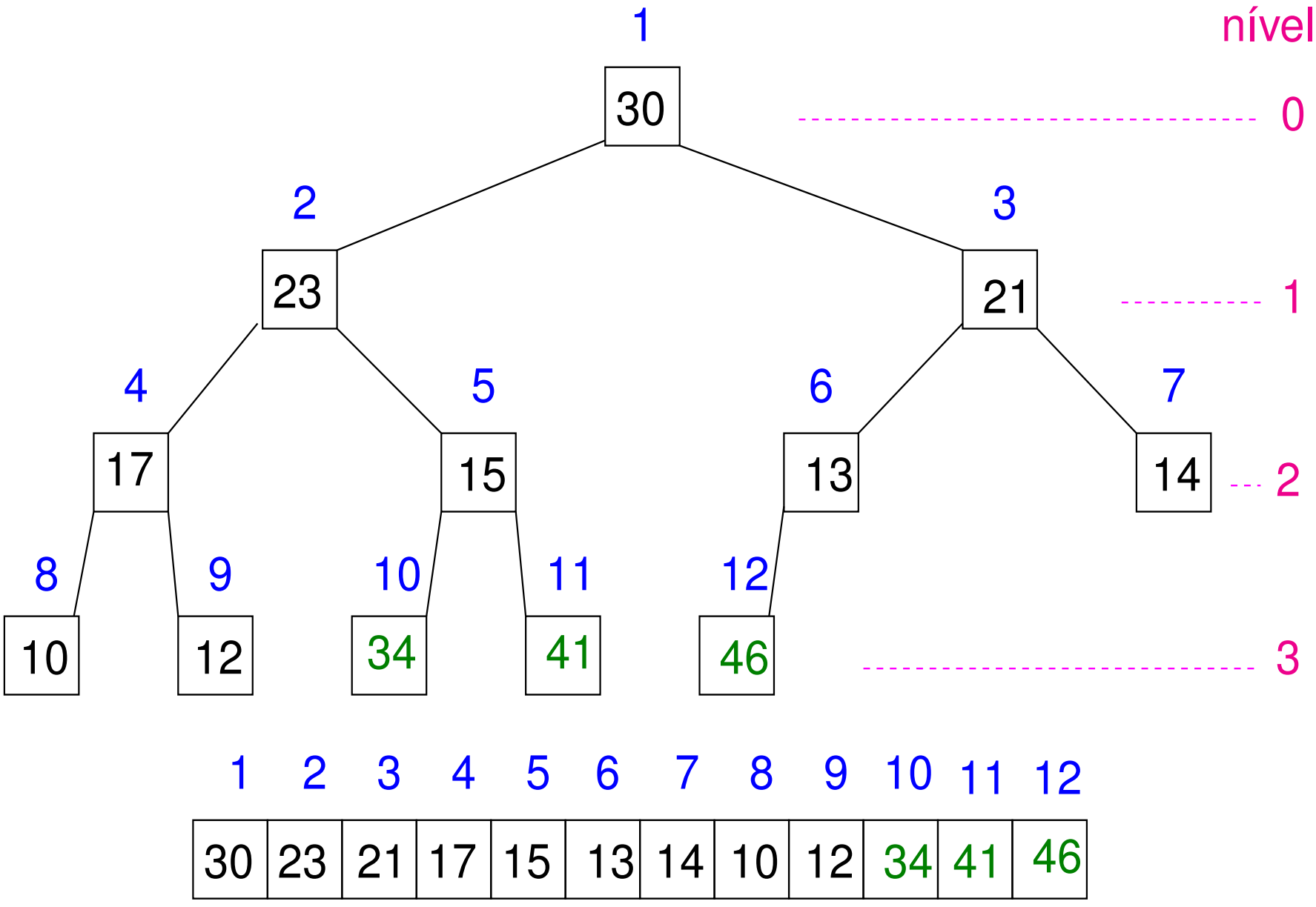
1	2	3	4	5	6	7	8	9	10	11	12
30	23	21	10	15	13	14	17	12	34	41	46

HeapSort

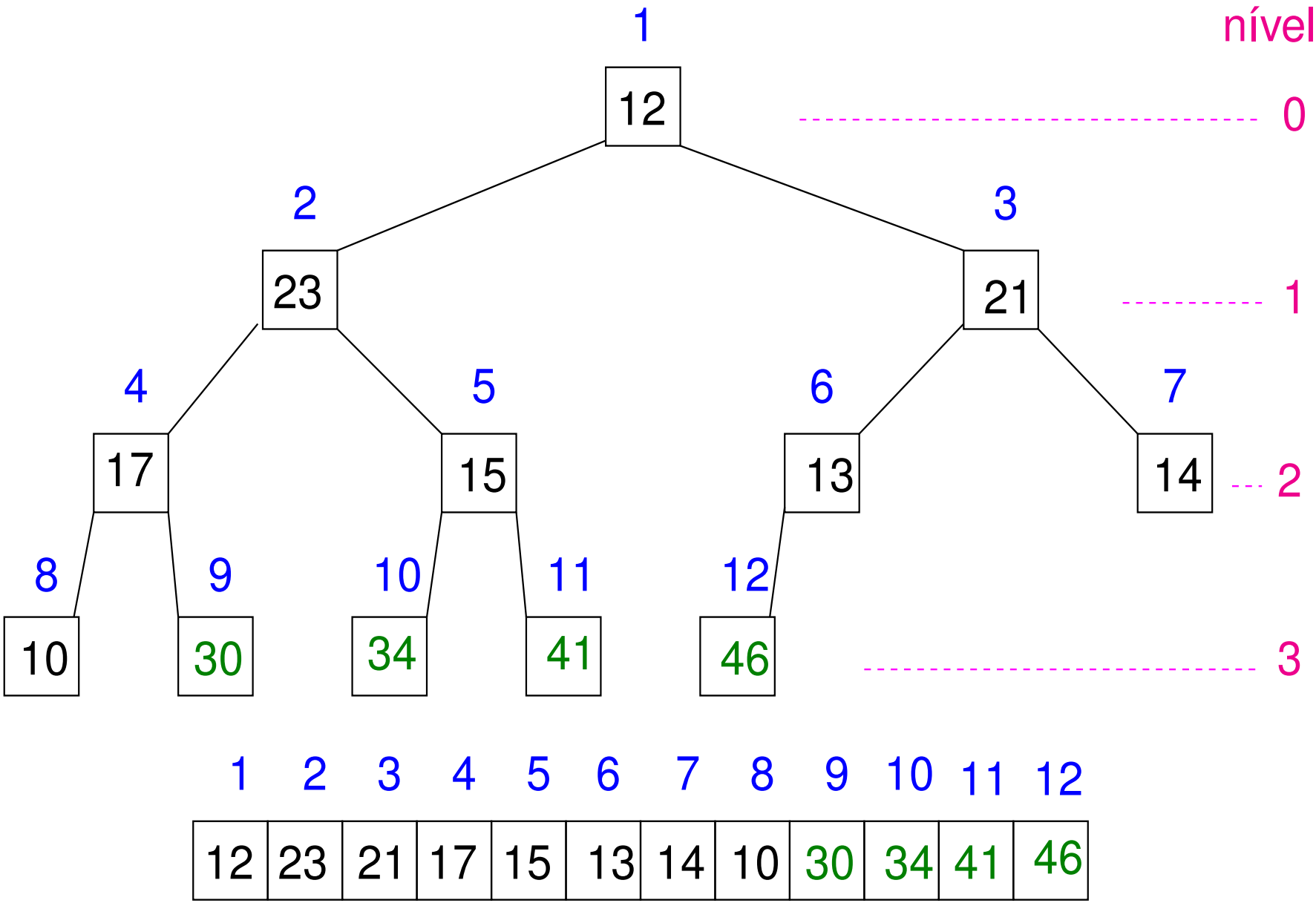


1	2	3	4	5	6	7	8	9	10	11	12
30	23	21	17	15	13	14	10	12	34	41	46

HeapSort


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

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

```
HEAPSORT(A, n)
1  BUILD-MAX-HEAP(A, n)
2  m ← n
3  para i ← n decrescendo até 2 faça
4      A[1] ↔ A[i]
5      m ← m − 1
6      MAX-HEAPIFY(A, m, 1)
```

Invariantes:

No início de cada iteração na linha 3 vale que:

1. $A[m \ldots n]$ é crescente;
2. $A[1 \ldots m] \leq A[m+1]$;
3. $A[1 \ldots m]$ é um max-heap.

HeapSort

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

	HEAPSORT(A, n)	Tempo
1	BUILD-MAX-HEAP(A, n)	?
2	$m \leftarrow n$	?
3	**para** $i \leftarrow n$ **decrescendo até** 2 **faça**	?
4	$A[1] \leftrightarrow A[i]$	?
5	$m \leftarrow m - 1$	?
6	MAX-HEAPIFY$(A, m, 1)$	?

$T(n) =$ complexidade de tempo no pior caso

HeapSort

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

HEAPSORT(A, n)	Tempo
1 BUILD-MAX-HEAP(A, n)	$\Theta(n)$
2 $m \leftarrow n$	$\Theta(1)$
3 **para** $i \leftarrow n$ **decrescendo até** 2 **faça**	$\Theta(n)$
4 $A[1] \leftrightarrow A[i]$	$\Theta(n)$
5 $m \leftarrow m - 1$	$\Theta(n)$
6 MAX-HEAPIFY$(A, m, 1)$	$nO(\lg n)$

$T(n) = ??$ $T(n) = nO(\lg n) + \Theta(4n + 1) = O(n \lg n)$

A complexidade de HEAPSORT no pior caso é $O(n \lg n)$.

Como seria a complexidade de tempo no melhor caso?

Filas com prioridades

Uma fila com prioridades é um tipo abstrato de dados que consiste de uma coleção S de itens, cada um com um valor ou prioridade associada.

Algumas operações típicas em uma fila com prioridades são:

MAXIMUM(S): devolve o elemento de S com a maior prioridade;

EXTRACT-MAX(S): remove e devolve o elemento em S com a maior prioridade;

INCREASE-KEY(S, x, p): aumenta o valor da prioridade do elemento x para p; e

INSERT(S, x, p): insere o elemento x em S com prioridade p.

Implementação com max-heap

HEAP-MAX(A, n)
1 **devolva** $A[1]$

Complexidade de tempo: $\Theta(1)$.

HEAP-EXTRACT-MAX(A, n)
1 $\triangleright$ $n \geq 1$
2 max $\leftarrow$ $A[1]$
3 $A[1] \leftarrow A[n]$
4 `cor` $\leftarrow n - 1$
5 MAX-HEAPIFY (A, n, 1)
6 **devolva** max

Complexidade de tempo: $O(\lg n)$.

Implementação com max-heap

HEAP-INCREASE-KEY($A, i, prior$)
1 ▷ Supõe que $prior \geq A[i]$
2 $A[i] \leftarrow prior$
3 **enquanto** $i > 1$ e $A[\lfloor i/2 \rfloor] < A[i]$ **faça**
4 $A[i] \leftrightarrow A[\lfloor i/2 \rfloor]$
5 $i \leftarrow \lfloor i/2 \rfloor$

Complexidade de tempo: $O(\lg n)$.

MAX-HEAP-INSERT($A, n, prior$)
1 $n \leftarrow n + 1$
2 $A[n] \leftarrow -\infty$
3 HEAP-INCREASE-KEY($A, n, prior$)

Complexidade de tempo: $O(\lg n)$.

Ordenação em Tempo Linear

Algoritmos lineares para ordenação

Os seguintes algoritmos de ordenação têm complexidade $O(n)$:

- Counting Sort: Elementos são números inteiros "pequenos"; mais precisamente, inteiros $x \in O(n)$.
- Radix Sort: Elementos são números inteiros de comprimento máximo constante, isto é, independente de n.
- Bucket Sort: Elementos são números reais uniformemente distribuídos no intervalo $[0..1)$.

Counting Sort

- Considere o problema de ordenar um vetor $A[1 \ldots n]$ de inteiros quando se sabe que todos os inteiros estão no intervalo entre 0 e k.
- Podemos ordenar o vetor simplesmente contando, para cada inteiro i no vetor, quantos elementos do vetor são menores que i.
- É exatamente o que faz o algoritmo *Counting Sort*.

Counting Sort

```
COUNTING-SORT(A, B, n, k)
1  para i ← 0 até k faça
2      C[i] ← 0

3  para j ← 1 até n faça
4      C[A[j]] ← C[A[j]] + 1
   ▷ C[i] é o número de js tais que A[j] = i

5  para i ← 1 até k faça
6      C[i] ← C[i] + C[i − 1]
   ▷ C[i] é o número de js tais que A[j] ≤ i

7  para j ← n decrescendo até 1 faça
8      B[C[A[j]]] ← A[j]
9      C[A[j]] ← C[A[j]] − 1
```

Counting Sort - Complexidade

- Qual a complexidade do algoritmo COUNTING-SORT?
- O algoritmo não faz comparações entre elementos de A!
- Sua complexidade deve ser medida em função do número das outras operações, aritméticas, atribuições, etc.
- Claramente, a complexidade de COUNTING-SORT é $O(n+k)$. Quando $k \in O(n)$, ele tem complexidade $O(n)$.

Há algo de errado com o limite inferior de $\Omega(n \log n)$ para ordenação?

Algoritmos *in-place* e *estáveis*

- Algoritmos de ordenação podem ser ou não *in-place* ou *estáveis*.
- Um algoritmo de ordenação é in-place se a memória adicional requerida é independente do tamanho do vetor que está sendo ordenado.
- Exemplos: QUICKSORT e HEAPSORT são métodos de ordenação *in-place*, já MERGESORT e COUNTING-SORT não são.
- Um método de ordenação é estável se elementos iguais ocorrem no vetor ordenado na mesma ordem em que são passados na entrada.
- Exemplos: COUNTING-SORT e QUICKSORT são exemplos de métodos estáveis (desde que certos cuidados sejam tomados na implementação). HEAPSORT não é.

Radix Sort

- Considere agora o problema de ordenar um vetor $A[1 \ldots n]$ inteiros quando se sabe que todos os inteiros podem ser representados com apenas d dígitos, onde d é uma constante.
- Por exemplo, os elementos de A podem ser CEPs, ou seja, inteiros de 8 dígitos.

Radix Sort

- Poderíamos ordenar os elementos do vetor dígito a dígito da seguinte forma:
  - Separamos os elementos do vetor em grupos que compartilham o mesmo dígito mais significativo.
  - Em seguida, ordenamos os elementos em cada grupo pelo mesmo método, levando em consideração apenas os $d - 1$ dígitos menos significativos.
- Esse método funciona, mas requer o uso de bastante memória adicional para a organização dos grupos e subgrupos.

Radix Sort

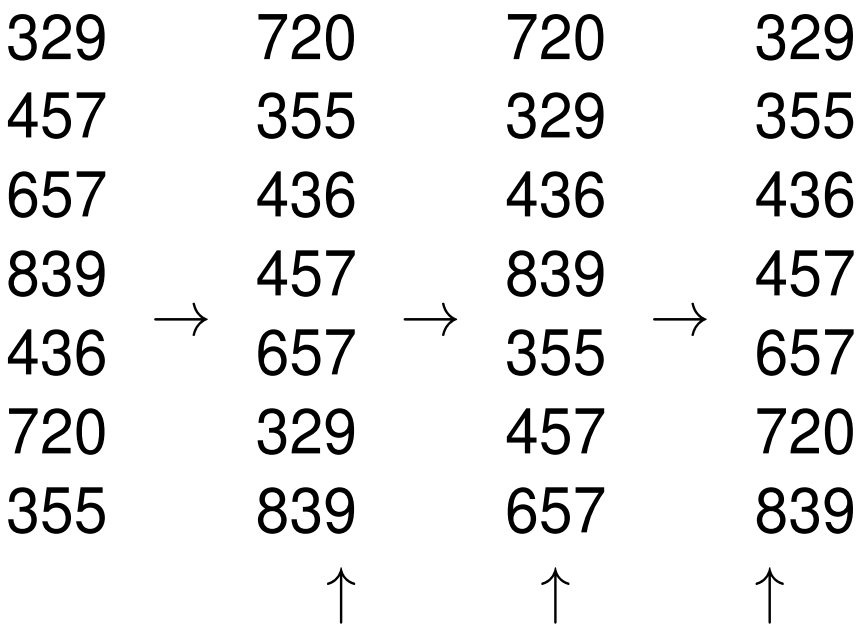
- Podemos evitar o uso excessivo de memória adicional começando pelo dígito menos significativo.
- É isso o que faz o algoritmo Radix Sort.
- Para que Radix Sort funcione corretamente, ele deve usar um método de ordenação *estável*.
- Por exemplo, o COUNTING-SORT.

Radix Sort

Suponha que os elementos do vetor A a ser ordenado sejam números inteiros de até d dígitos. O *Radix Sort* é simplesmente:

```
RADIX-SORT(A, n, d)
1   para i ← 1 até d faça
2       Ordene A[1 . . . n] pelo i-ésimo dígito
        usando um método estável
```

Radix Sort - Exemplo



Radix Sort - Corretude

O seguinte argumento indutivo garante a corretude do algoritmo:

- Hipótese de indução: os números estão ordenados com relação aos $i - 1$ dígitos menos significativos.
- O que acontece ao ordenarmos pelo i-ésimo dígito?
- Se dois números têm i-ésimo dígitos distintos, o de menor i-ésimo dígito aparece antes do de maior i-ésimo dígito.
- Se ambos possuem o mesmo i-ésimo dígito, então a ordem dos dois também estará correta pois o método de ordenação é estável e, pela HI, os dois elementos já estavam ordenados segundo os $i - 1$ dígitos menos significativos.

Radix Sort - Complexidade

- Qual é a complexidade de RADIX-SORT?
- Depende da complexidade do algoritmo estável usado para ordenar cada dígito.
- Se essa complexidade for $\Theta(f(n))$, obtemos uma complexidade total de $\Theta(d\ f(n))$.
- Como d é constante, a complexidade é então $\Theta(f(n))$.
- Se o algoritmo estável for, por exemplo, o COUNTING-SORT, obtemos a complexidade $\Theta(n+k)$.
- Se $k \in O(n)$, isto resulta em uma complexidade linear em n.

E o limite inferior de $\Omega(n \log n)$ para ordenação?

Radix Sort - Complexidade

- Em contraste, um algoritmo por comparação como o MERGESORT teria complexidade $\Theta(n \lg n)$.
- Assim, RADIX-SORT é mais vantajoso que MERGESORT quando $d < \lg n$, ou seja, o número de dígitos for menor que $\lg n$.
- Se n for um limite superior para o maior valor a ser ordenado, então $O(\log n)$ é uma estimativa para a quantidade de dígitos dos números.
- Isso significa que não há diferença significativa entre o desempenho do MERGESORT e do RADIX-SORT?

Radix Sort - Complexidade

- O nome *Radix Sort* vem da **base** (em inglês *radix*) em que intepretamos os dígitos.
- A vantagem de se usar RADIX-SORT fica evidente quando interpretamos os dígitos de forma mais geral que simplesmente 0..9.
- Tomemos o seguinte exemplo: suponha que desejemos ordenar um conjunto de $n = 2^{20}$ números de 64 bits. Então, MERGESORT faria cerca de $n \lg n = 20 \times 2^{20}$ comparações e usaria um vetor auxiliar de tamanho 2^{20}.

Radix Sort - Complexidade

- Agora suponha que interpretamos cada número do como tendo $d = 4$ dígitos em base $k = 2^{16}$, e usarmos RADIX-SORT com o *Counting Sort* como método estável.

  Então a complexidade de tempo seria da ordem de $d(n + k) = 4(2^{20} + 2^{16})$ operações, bem menor que 20×2^{20} do MERGESORT. Mas, note que utilizamos dois vetores auxiliares, de tamanhos 2^{16} e 2^{20}.
- Se o uso de memória auxiliar for muito limitado, então o melhor mesmo é usar um algoritmo de ordenação por comparação *in-place*.
- Note que é possível usar o *Radix Sort* para ordenar outros tipos de elementos, como datas, palavras em ordem lexicográfica e qualquer outro tipo que possa ser visto como uma d-upla ordenada de itens comparáveis.