MC458 — Projeto e Análise de Algoritmos I

C.C. de Souza C.N. da Silva O. Lee

Antes de mais nada...

- Uma versão anterior deste conjunto de slides foi preparada por Cid Carvalho de Souza e Cândida Nunes da Silva para uma instância anterior desta disciplina.
- O que vocês tem em mãos é uma versão modificada preparada para atender a meus gostos.
- Nunca é demais enfatizar que o material é apenas um guia e não deve ser usado como única fonte de estudo. Para isso consultem a bibliografia (em especial o CLR ou CLRS).

Orlando Lee

Agradecimentos (Cid e Cândida)

- Várias pessoas contribuíram direta ou indiretamente com a preparação deste material.
- Algumas destas pessoas cederam gentilmente seus arquivos digitais enquanto outras cederam gentilmente o seu tempo fazendo correções e dando sugestões.
- Uma lista destes "colaboradores" (em ordem alfabética) é dada abaixo:
  - Célia Picinin de Mello
  - José Coelho de Pina
  - Orlando Lee
  - Paulo Feofiloff
  - Pedro Rezende
  - Ricardo Dahab
  - Zanoni Dias

Ordenação

O problema da ordenação

Problema:

Rearranjar um vetor $A[1 \ldots n]$ de inteiros de modo que fique em ordem crescente.

Ou simplesmente:

O problema da ordenação

Problema:

Rearranjar um vetor $A[1 \ldots n]$ de inteiros de modo que fique em ordem crescente.

Ou simplesmente:

Problema:

Ordenar um vetor $A[1 \ldots n]$ de inteiros.

O problema da ordenação

Problema:

Rearranjar um vetor $A[1\,..\,n]$ de inteiros de modo que fique em ordem crescente.

Ou simplesmente:

Problema:

Ordenar um vetor $A[1\,..\,n]$ de inteiros.

Observação: de fato, os algoritmos que veremos são capazes de ordenar qualquer sequência de elementos comparáveis.

Algoritmos de ordenação

Veremos vários algoritmos de ordenação:

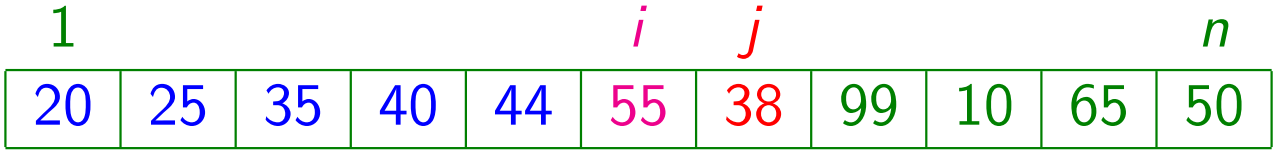
- *Insertion sort*
- *Selection sort*
- *Mergesort*
- *Heapsort*
- *Quicksort*

Insertion sort – iteração genérica

$chave = 38$

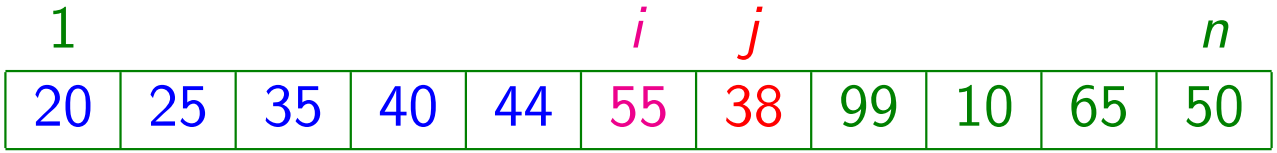
Insertion sort – iteração genérica

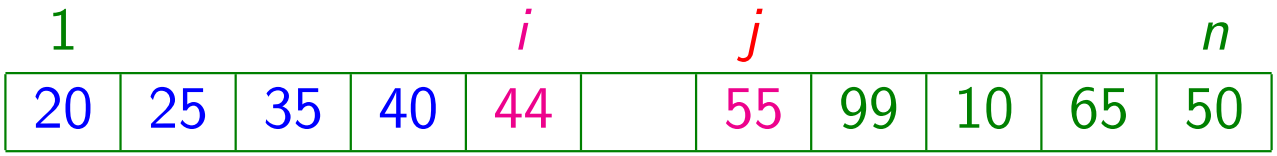
$chave = 38$



Insertion sort – iteração genérica

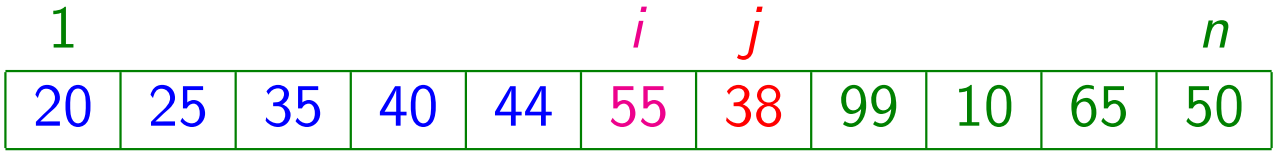
$chave = 38$

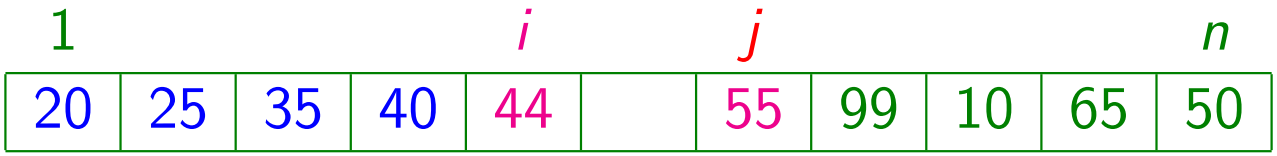


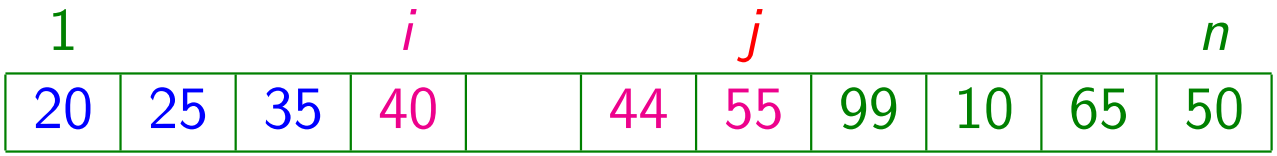


Insertion sort – iteração genérica

$chave = 38$

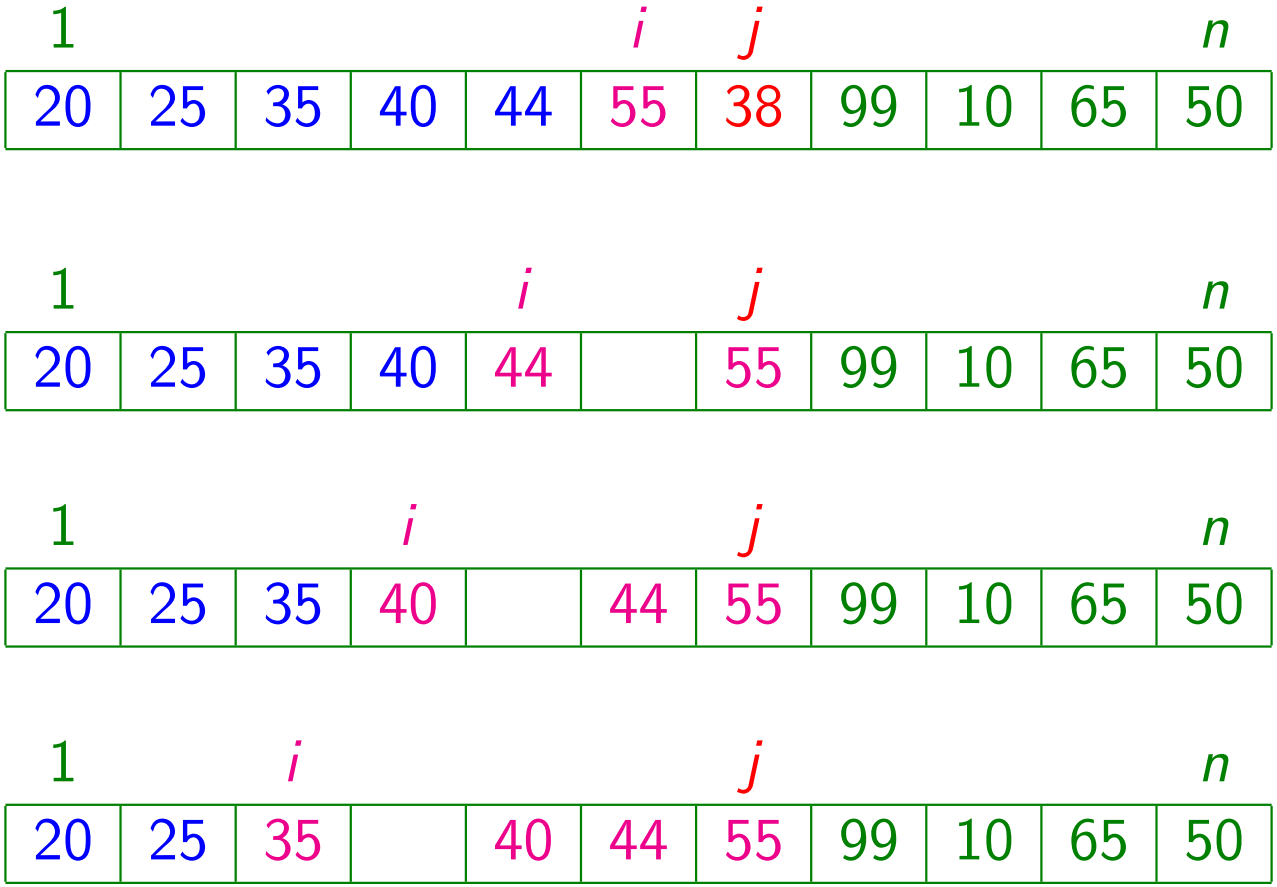






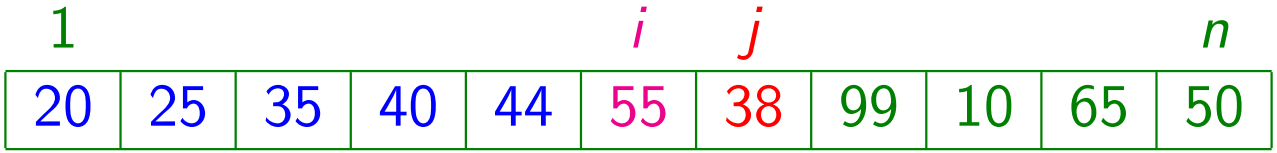
Insertion sort – iteração genérica

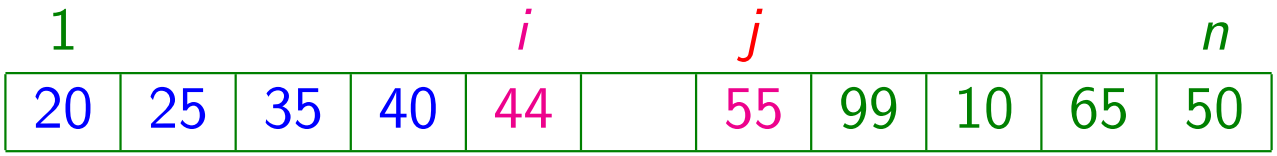
$chave = 38$

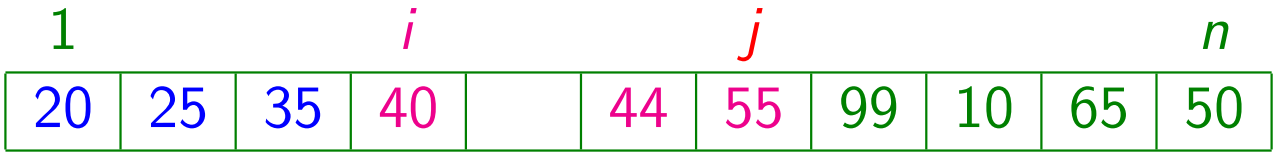


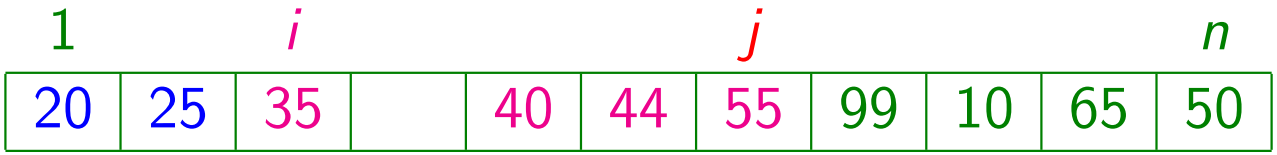
Insertion sort – iteração genérica

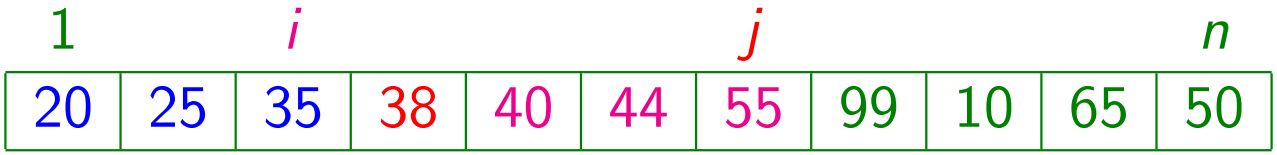
$chave = 38$





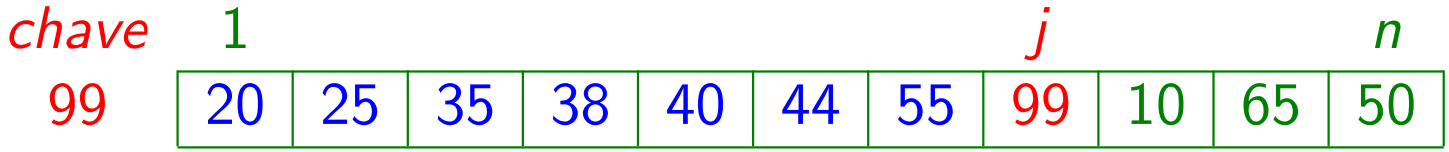




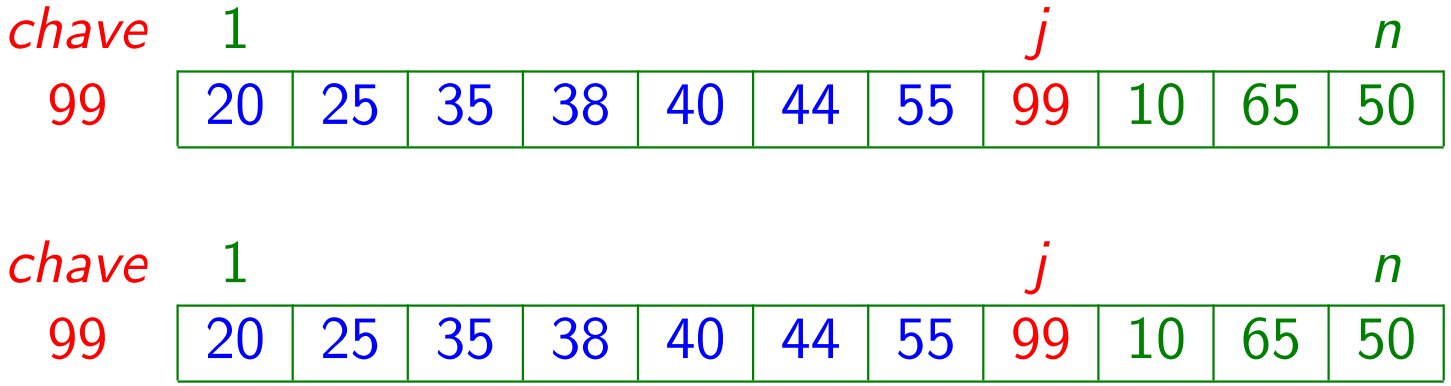


Insertion sort

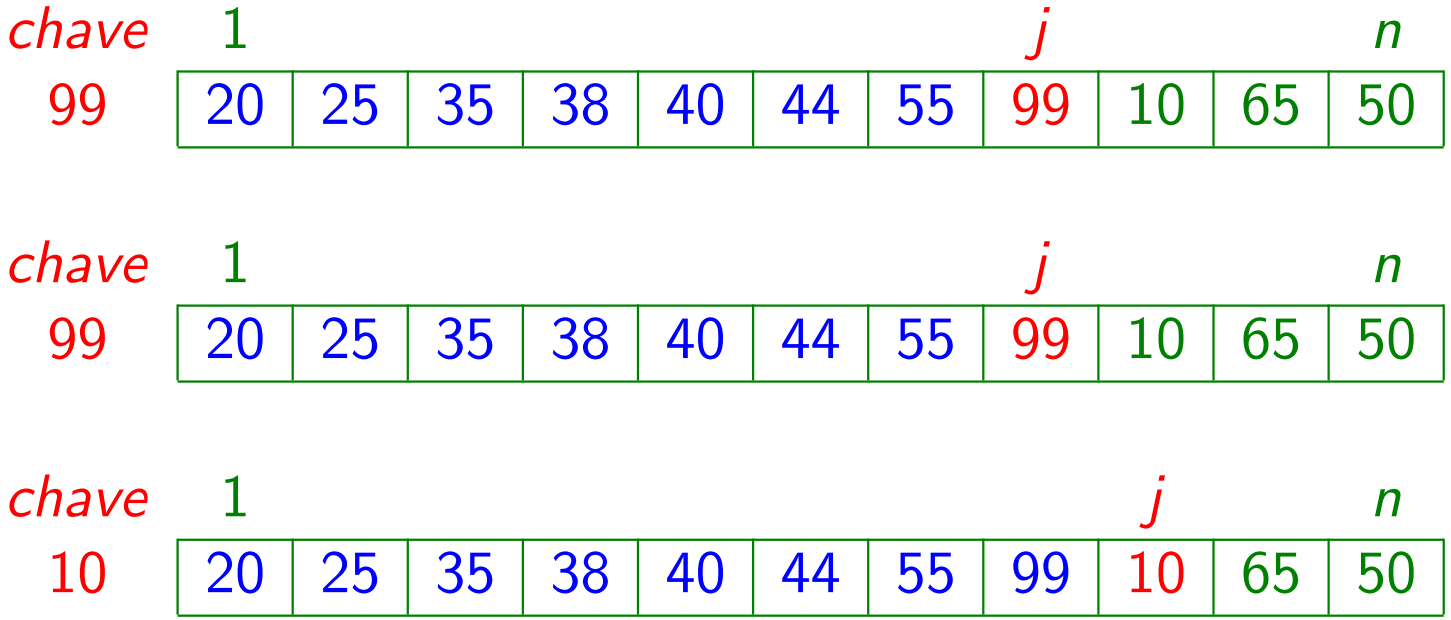
Insertion sort



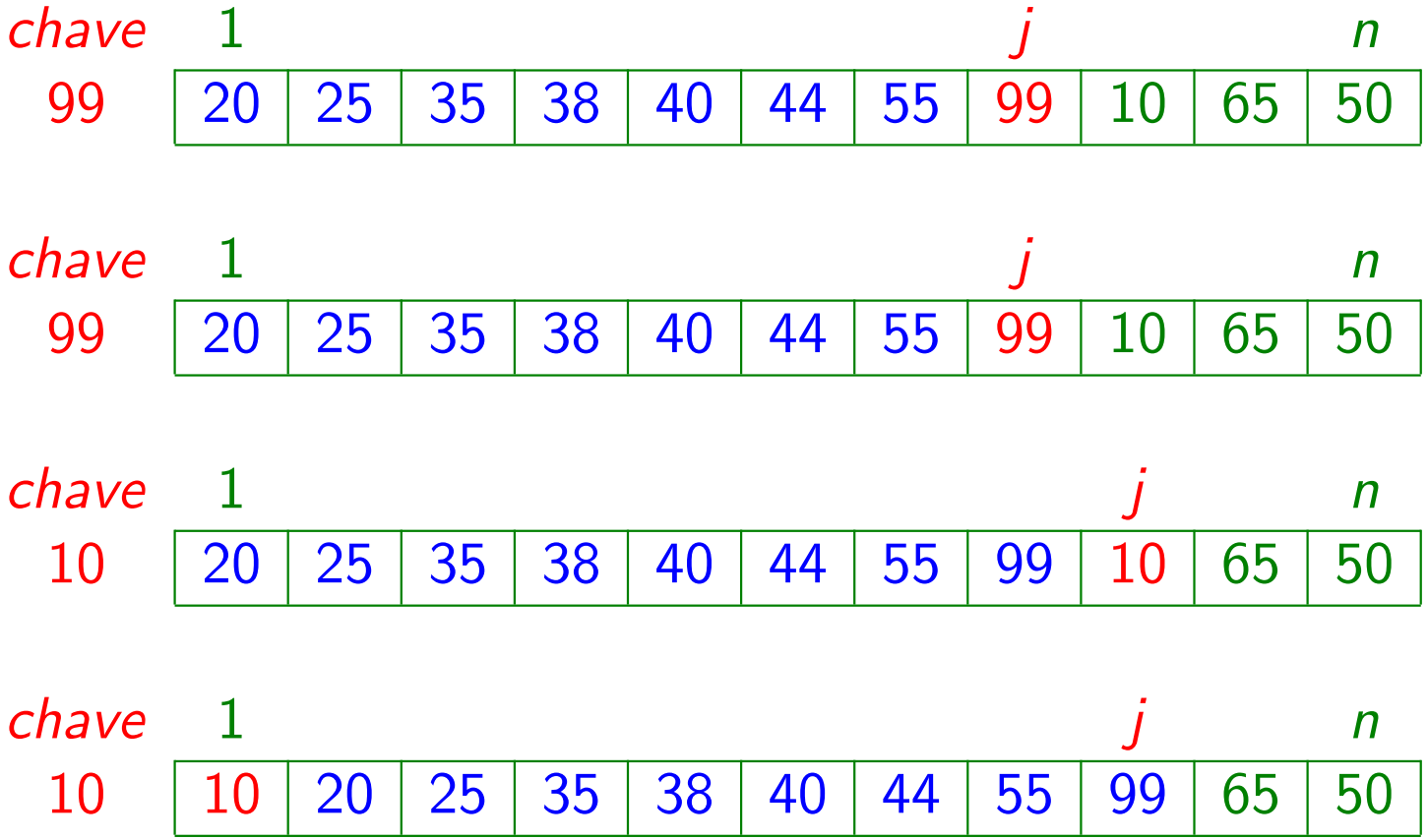
Insertion sort



Insertion sort

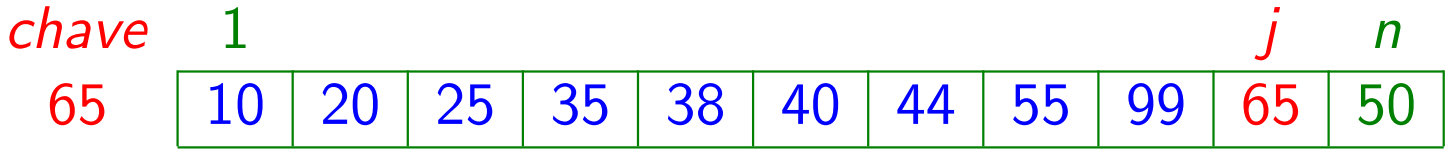


Insertion sort

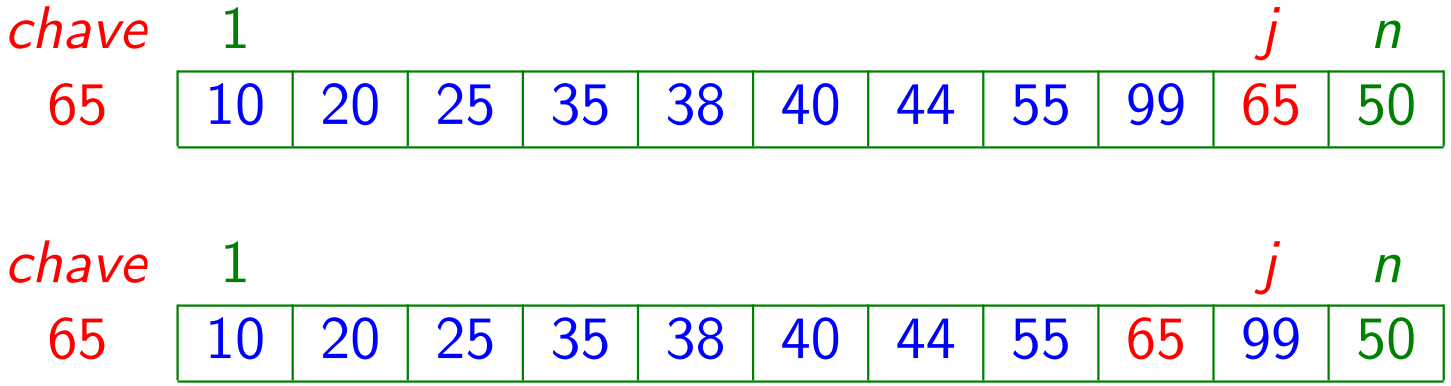


Insertion sort

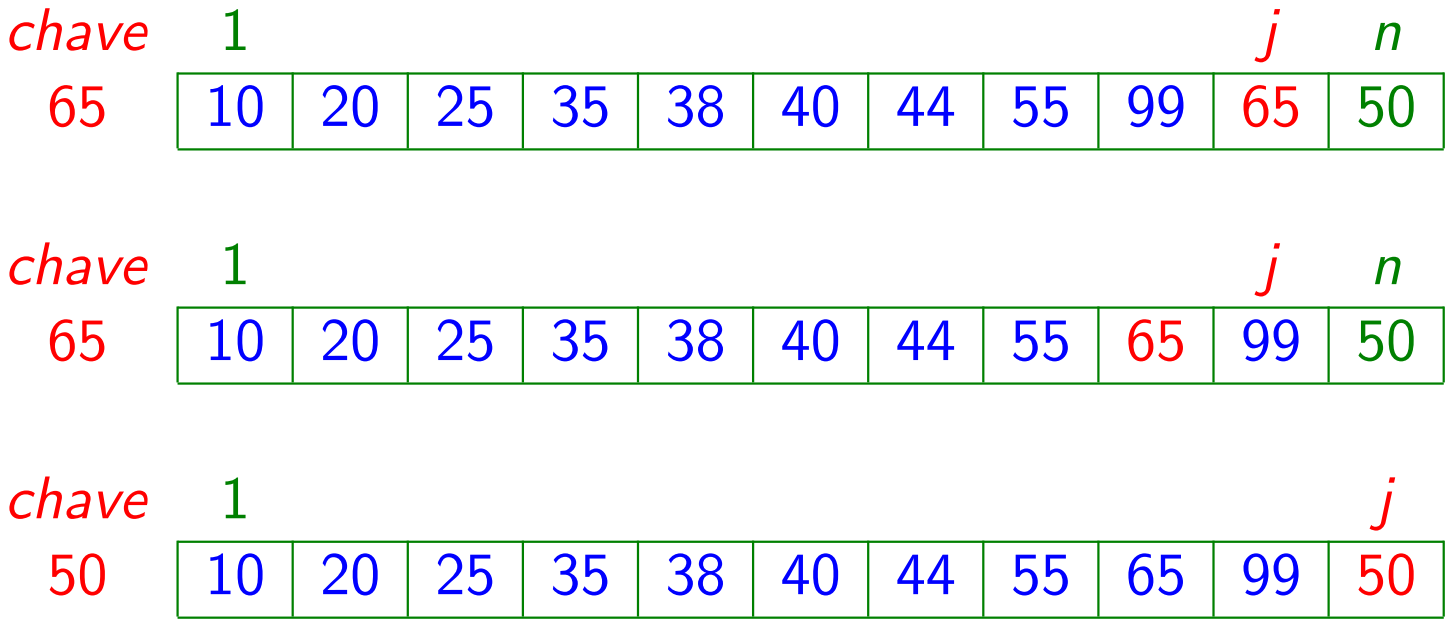
Insertion sort



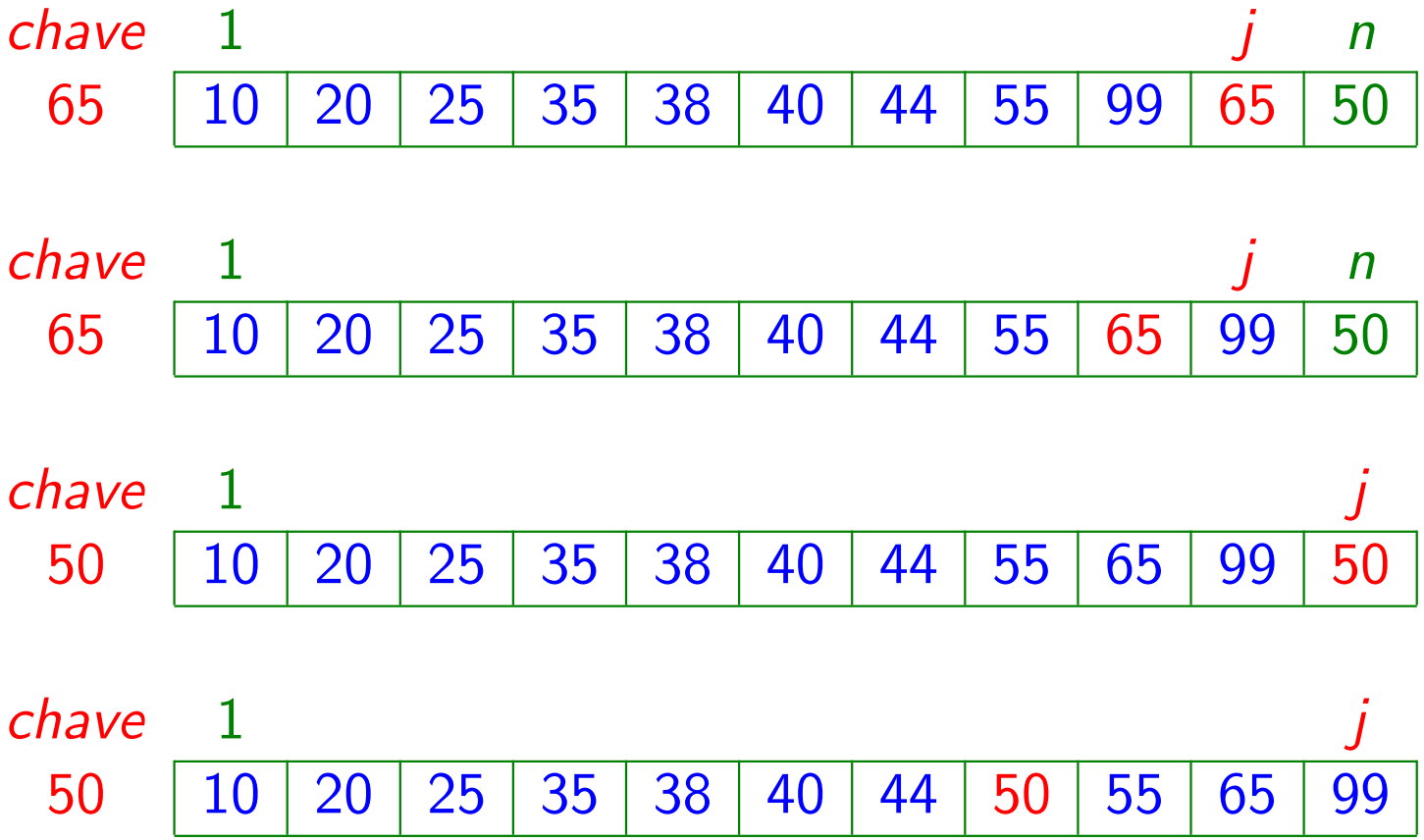
Insertion sort



Insertion sort



Insertion sort



Insertion sort

Insertion sort

- **Idéia:** a cada passo mantemos o subvetor $A[1\,..\,j-1]$ ordenado e inserimos o elemento $A[j]$ neste subvetor.

Insertion sort

- **Idéia:** a cada passo mantemos o subvetor $A[1\,..\,j-1]$ ordenado e inserimos o elemento $A[j]$ neste subvetor.
- Repetimos o processo para $j = 2, \ldots, n$ e ordenamos o vetor.

Insertion sort

- **Idéia:** a cada passo mantemos o subvetor $A[1\,..\,j-1]$ ordenado e inserimos o elemento $A[j]$ neste subvetor.
- Repetimos o processo para $j = 2, \ldots, n$ e ordenamos o vetor.

Primeira iteração: $j = 2$

1	j								n
25	44	20	35	40	55	38	99	10	65	50

Insertion sort – pseudo-código

```
INSERTION-SORT(A, n)
1   para j ← 2 até n faça
2       chave ← A[j]
3       ▷ Insere A[j] no subvetor ordenado A[1..j − 1]
4       i ← j − 1
5       enquanto i ≥ 1  e A[i] > chave faça
6           A[i + 1] ← A[i]
7           i ← i − 1
8       A[i + 1] ← chave
```

Insertion sort – pseudo-código

```
INSERTION-SORT(A, n)
1   para j ← 2 até n faça
2       chave ← A[j]
3       ▷ Insere A[j] no subvetor ordenado A[1..j − 1]
4       i ← j − 1
5       enquanto i ≥ 1  e A[i] > chave faça
6           A[i + 1] ← A[i]
7           i ← i − 1
8       A[i + 1] ← chave
```

Veremos como demonstrar a corretude de INSERTION-SORT usando a técnica de prova por invariante de laços.

Insertion sort – pseudo-código

```
INSERTION-SORT(A, n)
1   para j ← 2 até n faça
2       chave ← A[j]
3       ▷ Insere A[j] no subvetor ordenado A[1..j − 1]
4       i ← j − 1
5       enquanto i ≥ 1  e A[i] > chave faça
6           A[i + 1] ← A[i]
7           i ← i − 1
8       A[i + 1] ← chave
```

Veremos como demonstrar a corretude de INSERTION-SORT usando a técnica de prova por invariante de laços.

Depois analisaremos a complexidade de INSERTION-SORT.

Corretude da ordenação por inserção

Corretude da ordenação por inserção

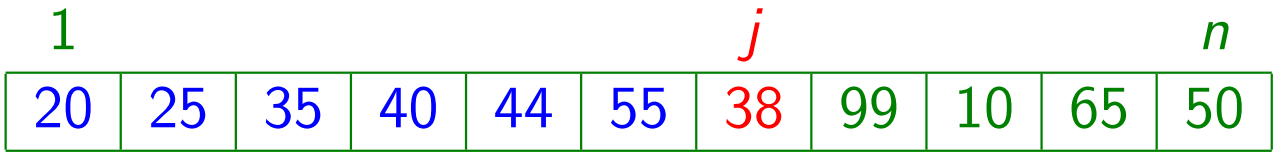
Invariante principal: (i1)

no começo de cada iteração do laço **para** das linhas 1–8, o subvetor $A[1\,..\,j-1]$ está ordenado.

Corretude da ordenação por inserção

Invariante principal: (i1)

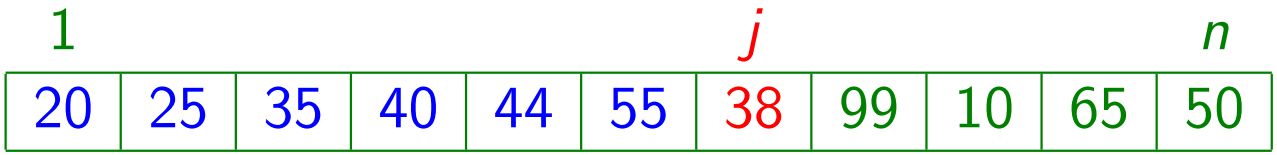
no começo de cada iteração do laço **para** das linhas 1–8, o subvetor $A[1\,..\,j-1]$ está ordenado.



Corretude da ordenação por inserção

Invariante principal: (i1)

no começo de cada iteração do laço **para** das linhas 1–8, o subvetor $A[1\,..\,j-1]$ está ordenado.

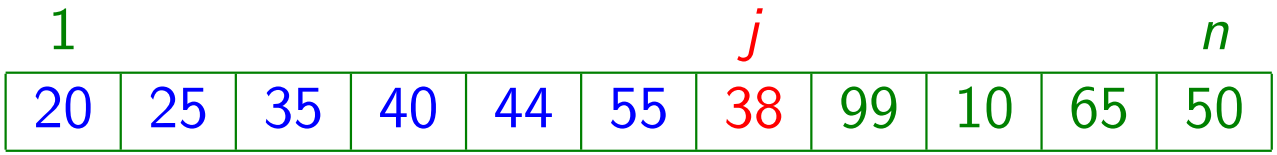


- Suponha que o invariante vale.

Corretude da ordenação por inserção

Invariante principal: (i1)

no começo de cada iteração do laço **para** das linhas 1–8, o subvetor $A[1\,..\,j-1]$ está ordenado.

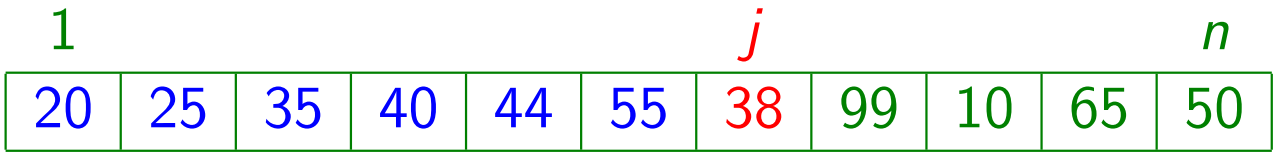


- Suponha que o invariante vale.
- Então a corretude do algoritmo é "evidente". **Por quê?**

Corretude da ordenação por inserção

Invariante principal: (i1)

no começo de cada iteração do laço **para** das linhas 1–8, o subvetor $A[1\,..\,j-1]$ está ordenado.



- Suponha que o invariante vale.
- Então a corretude do algoritmo é "evidente". **Por quê?**
- No ínicio da última iteração temos $j = n+1$. Assim, do invariante segue que o (sub)vetor $A[1\,..\,n]$ está ordenado!

Melhorando a argumentação

```
INSERTION-SORT(A, n)
1   para j ← 2 até n faça
2       chave ← A[j]
3       ▷ Insere A[j] no subvetor ordenado A[1 . . . j − 1]
4       i ← j − 1
5       enquanto i ≥ 1  e A[i] > chave faça
6           A[i + 1] ← A[i]
7           i ← i − 1
8       A[i + 1] ← chave
```

Melhorando a argumentação

```
INSERTION-SORT(A, n)
1   para j ← 2 até n faça
2       chave ← A[j]
3       ▷ Insere A[j] no subvetor ordenado A[1 . . . j − 1]
4       i ← j − 1
5       enquanto i ≥ 1  e A[i] > chave faça
6           A[i + 1] ← A[i]
7           i ← i − 1
8       A[i + 1] ← chave
```

Um invariante forte: (i1')

no começo de cada iteração do laço **para** das linhas 1–8, o subvetor $A[1\,..\,j-1]$ é uma permutação ordenada do subvetor original $A[1\,..\,j-1]$.

Esboço da demonstração de (i1')

Esboço da demonstração de (i1')

1. Validade na primeira iteração: neste caso, temos $j = 2$ e o invariante simplesmente afirma que $A[1\,..\,1]$ está ordenado, o que é evidente.

Esboço da demonstração de (i1')

1. Validade na primeira iteração: neste caso, temos $j = 2$ e o invariante simplesmente afirma que $A[1\,..\,1]$ está ordenado, o que é evidente.
2. Validade de uma iteração para a seguinte: segue da discussão anterior. O algoritmo empurra os elementos maiores que a chave para seus lugares corretos e ela é colocada no espaço vazio.

Esboço da demonstração de (i1')

1. Validade na primeira iteração: neste caso, temos $j = 2$ e o invariante simplesmente afirma que $A[1\,..\,1]$ está ordenado, o que é evidente.
2. Validade de uma iteração para a seguinte: segue da discussão anterior. O algoritmo empurra os elementos maiores que a chave para seus lugares corretos e ela é colocada no espaço vazio.

   Observação: Uma demonstração mais formal deste fato exigiria outros invariantes auxiliares para o laço principal e para o laço interno. (veja CLRS)

Esboço da demonstração de (i1')

1. Validade na primeira iteração: neste caso, temos $j = 2$ e o invariante simplesmente afirma que $A[1\,..\,1]$ está ordenado, o que é evidente.
2. Validade de uma iteração para a seguinte: segue da discussão anterior. O algoritmo empurra os elementos maiores que a chave para seus lugares corretos e ela é colocada no espaço vazio.

   Observação: Uma demonstração mais formal deste fato exigiria outros invariantes auxiliares para o laço principal e para o laço interno. (veja CLRS)
3. Resposta correta: na última iteração, temos $j = n + 1$ e logo $A[1\,..\,n]$ está ordenado com os *elementos originais* do vetor. Portanto, o algoritmo é correto.

Complexidade de tempo de Insertion sort

INSERTION-SORT(A, n)	Tempo
1 **para** $j \leftarrow 2$ **até** n **faça**	?
2 $\quad chave \leftarrow A[j]$	?
3 $\quad \triangleright$ Insere $A[j]$ em $A[1\,..\,j-1]$	?
4 $\quad i \leftarrow j-1$	?
5 $\quad$ **enquanto** $i \geq 1$ **e** $A[i] > chave$ **faça**	?
6 $\quad\quad A[i+1] \leftarrow A[i]$	?
7 $\quad\quad i \leftarrow i-1$	?
8 $\quad A[i+1] \leftarrow chave$	?

Consumo de tempo no pior caso: ?

Complexidade de tempo de Insertion sort

INSERTION-SORT(A, n)	Tempo
1 **para** $j \leftarrow 2$ **até** n **faça**	$\Theta(n)$
2 $\quad chave \leftarrow A[j]$	$\Theta(n)$
3 $\quad \triangleright$ Insere $A[j]$ em $A[1\,..\,j-1]$	
4 $\quad i \leftarrow j-1$	$\Theta(n)$
5 $\quad$ **enquanto** $i \geq 1$ **e** $A[i] > chave$ **faça**	$nO(n) = O(n^2)$
6 $\quad\quad A[i+1] \leftarrow A[i]$	$nO(n) = O(n^2)$
7 $\quad\quad i \leftarrow i-1$	$nO(n) = O(n^2)$
8 $\quad A[i+1] \leftarrow chave$	$O(n)$

Consumo de tempo: $O(n^2)$

Insertion sort

Insertion sort

- Complexidade de tempo no pior caso: $\Theta(n^2)$
  Vetor em ordem decrescente
  $\Theta(n^2)$ comparações
  $\Theta(n^2)$ movimentações

Insertion sort

- Complexidade de tempo no pior caso: $\Theta(n^2)$
  Vetor em ordem decrescente
  $\Theta(n^2)$ comparações
  $\Theta(n^2)$ movimentações
- Complexidade de tempo no melhor caso: $\Theta(n)$
  (vetor em ordem crescente)
  $\Theta(n)$ comparações
  zero movimentações

Insertion sort

- Complexidade de tempo no pior caso: $\Theta(n^2)$
  Vetor em ordem decrescente
  $\Theta(n^2)$ comparações
  $\Theta(n^2)$ movimentações
- Complexidade de tempo no melhor caso: $\Theta(n)$
  (vetor em ordem crescente)
  $\Theta(n)$ comparações
  zero movimentações
- Complexidade de espaço/consumo espaço: $\Theta(n)$

Conhecimento geral

Conhecimento geral

- Para vetores com no máximo 10 elementos, o melhor algoritmo de ordenação costuma ser *Insertion sort*.

Conhecimento geral

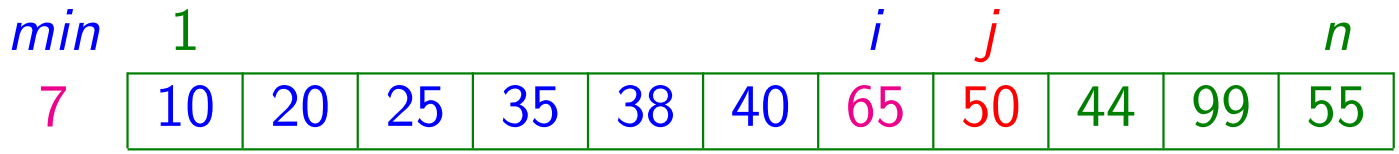
- Para vetores com no máximo 10 elementos, o melhor algoritmo de ordenação costuma ser *Insertion sort*.
- Para um vetor que está quase ordenado, *Insertion sort* também é a melhor escolha.

Conhecimento geral

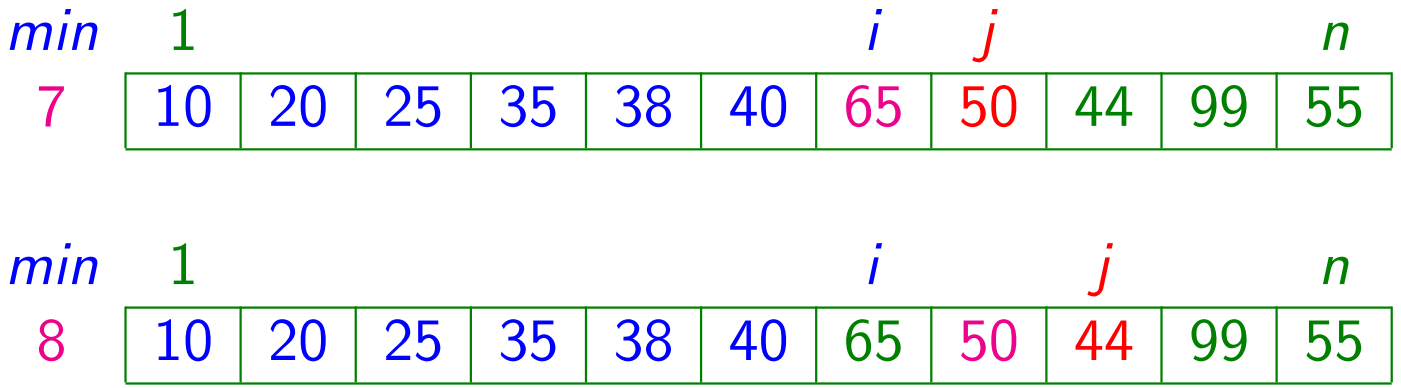
- Para vetores com no máximo 10 elementos, o melhor algoritmo de ordenação costuma ser *Insertion sort*.
- Para um vetor que está quase ordenado, *Insertion sort* também é a melhor escolha.
- Algoritmos super-eficientes assintoticamente tendem a fazer muitas movimentações, enquanto *Insertion sort* faz poucas movimentações quando o vetor está quase ordenado.

Selection sort

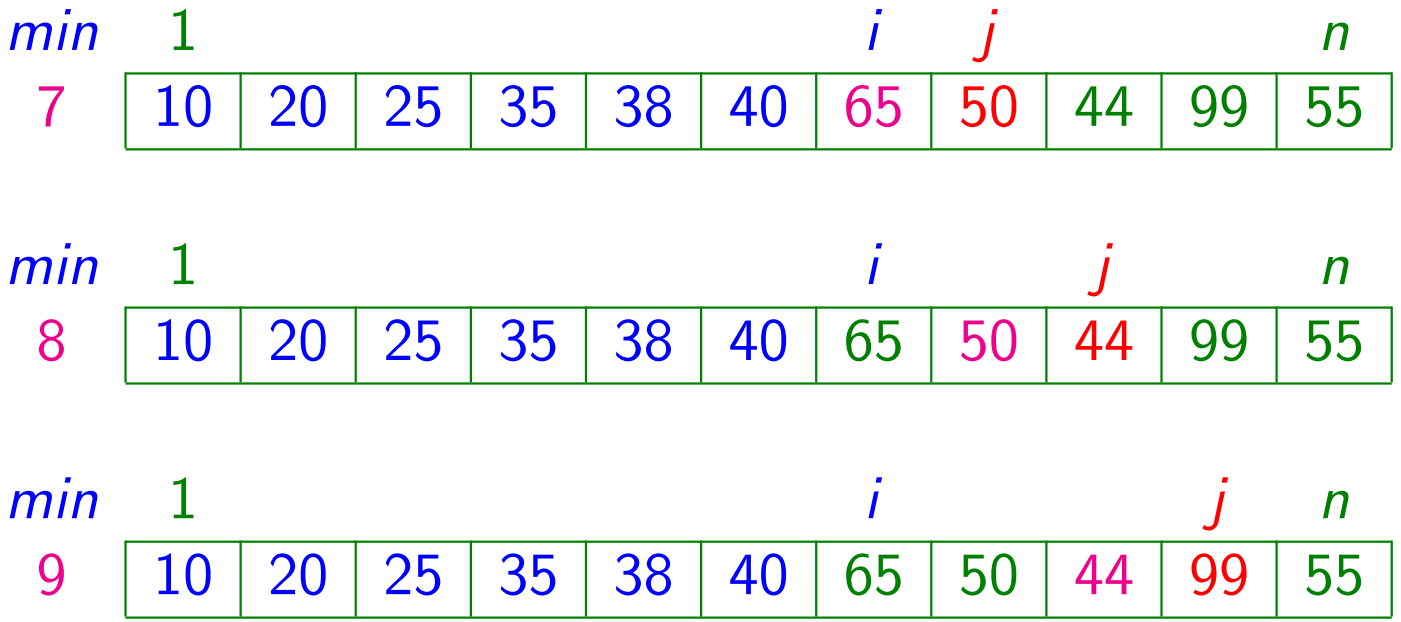
Selection sort



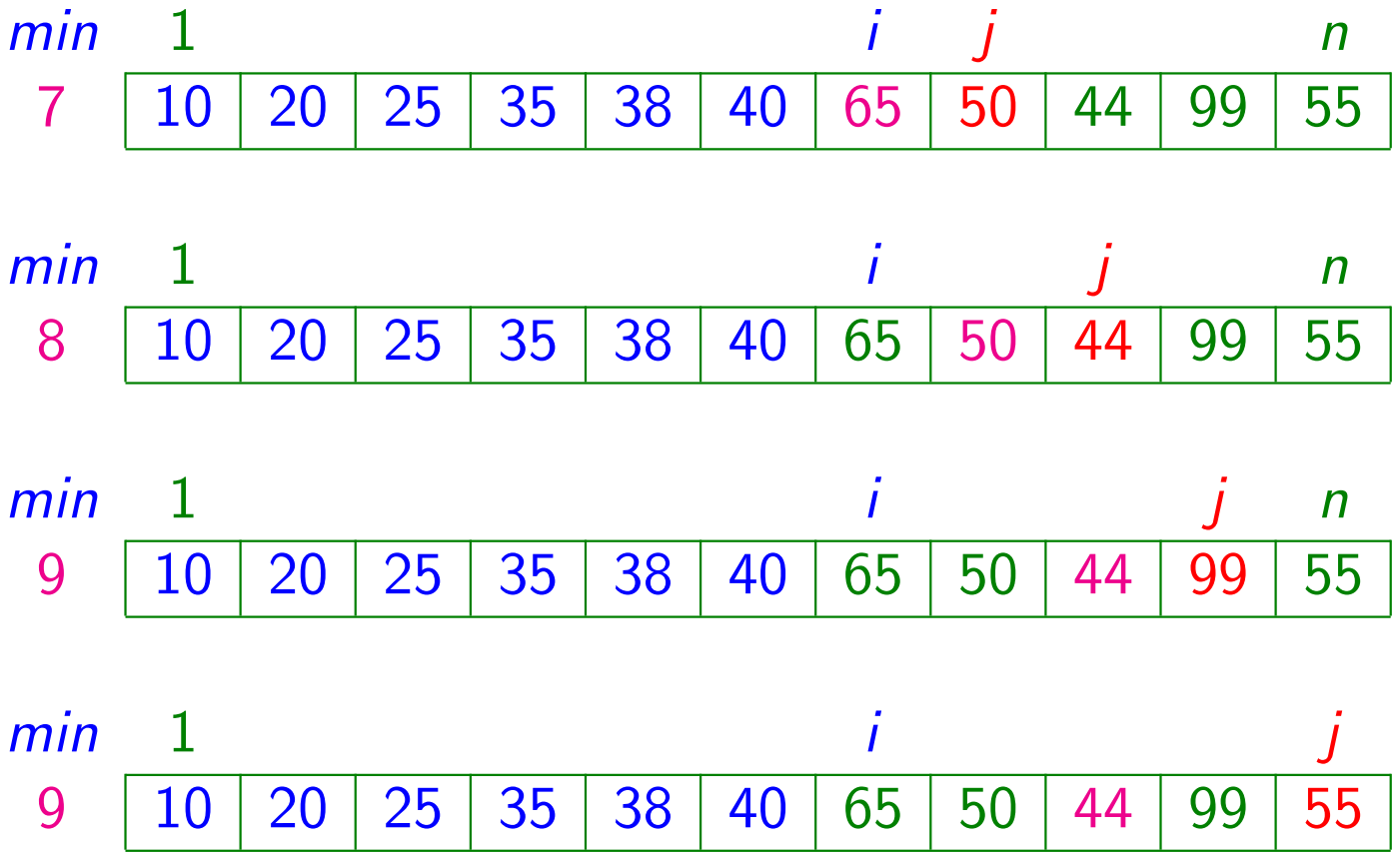
Selection sort



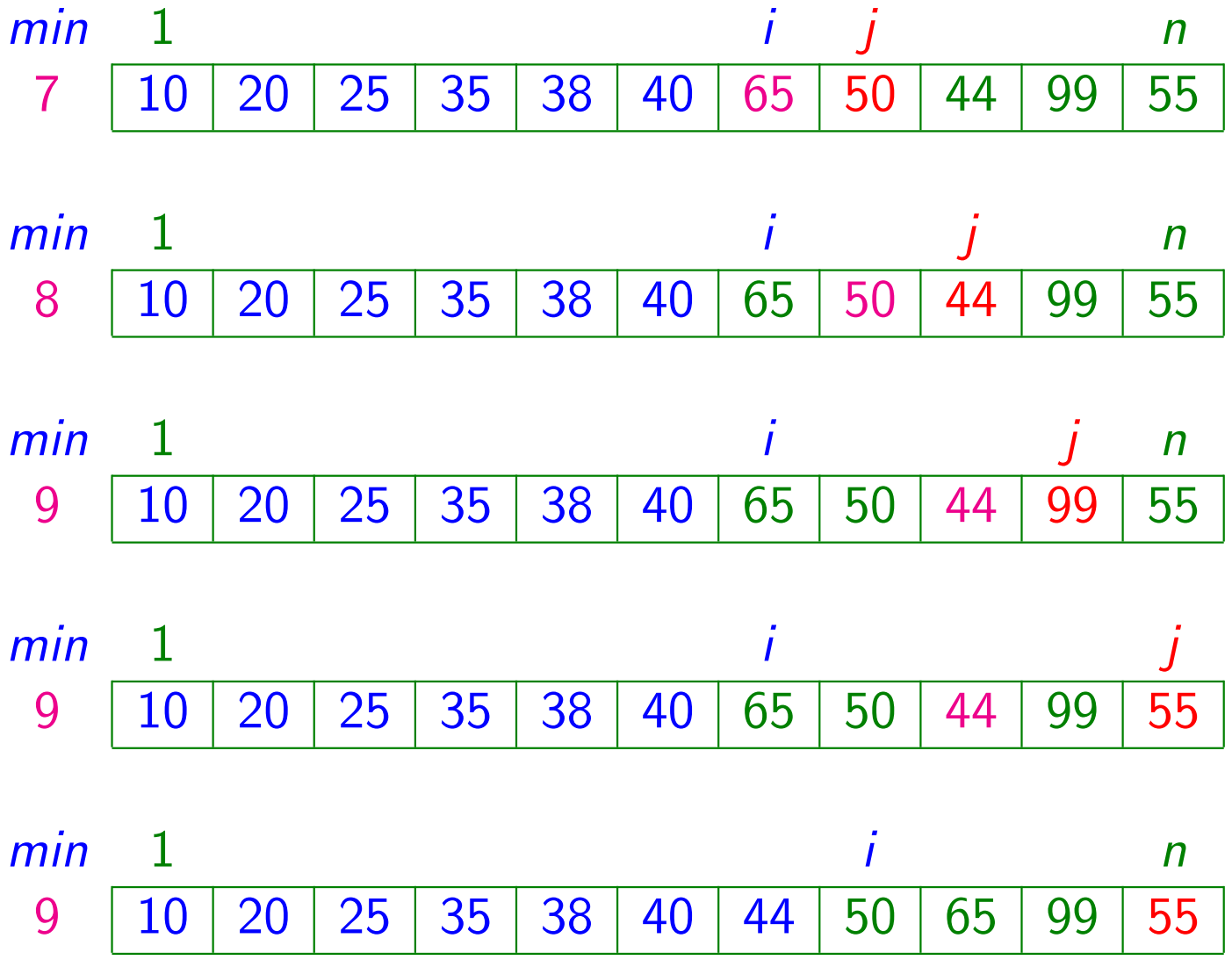
Selection sort



Selection sort

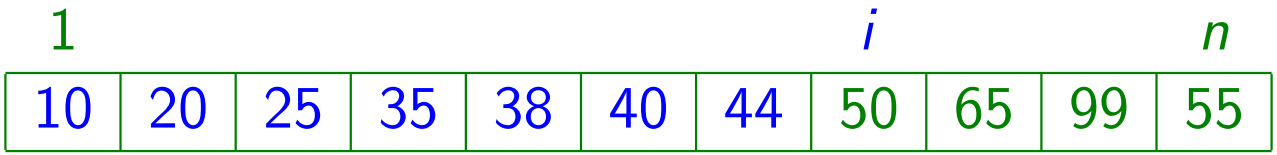


Selection sort

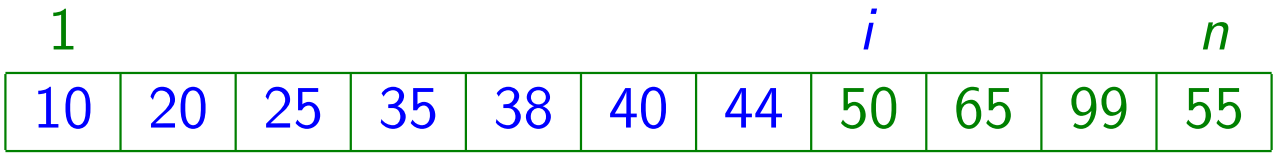


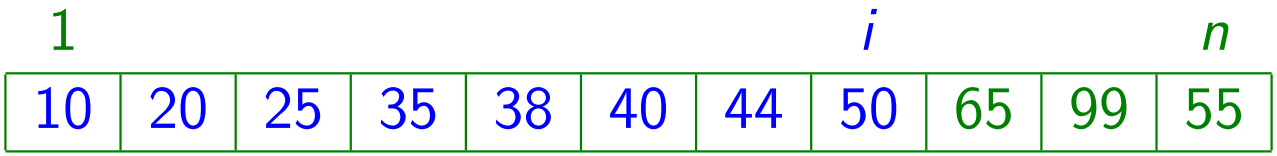
Selection sort

Selection sort

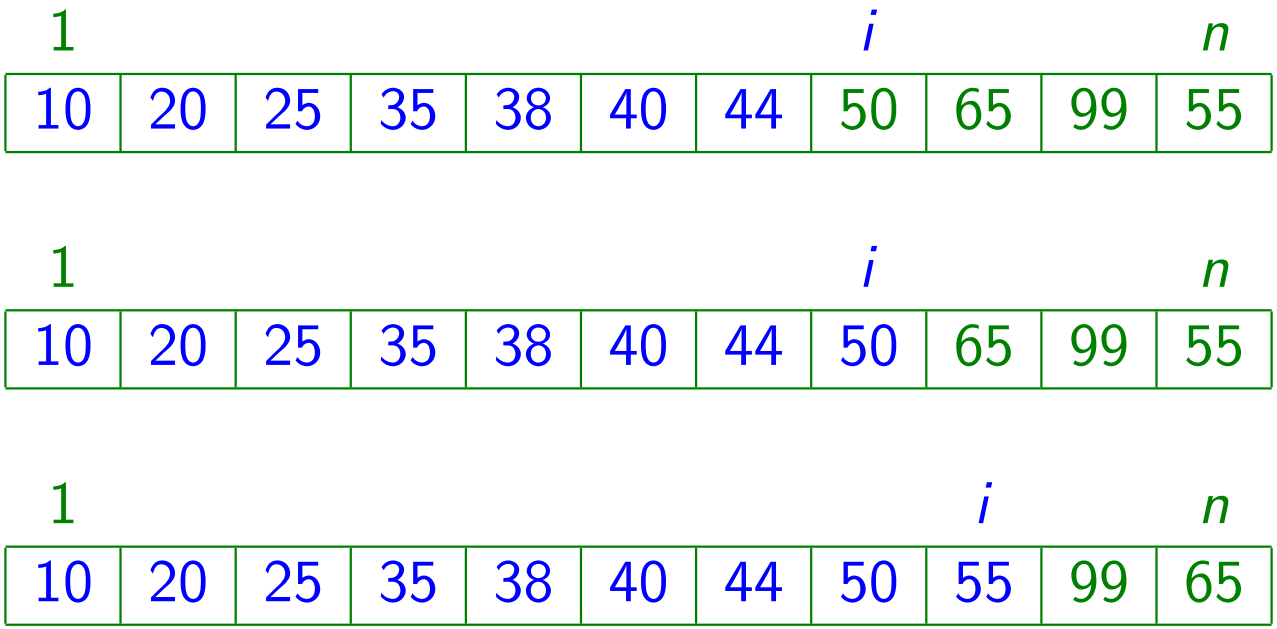


Selection sort

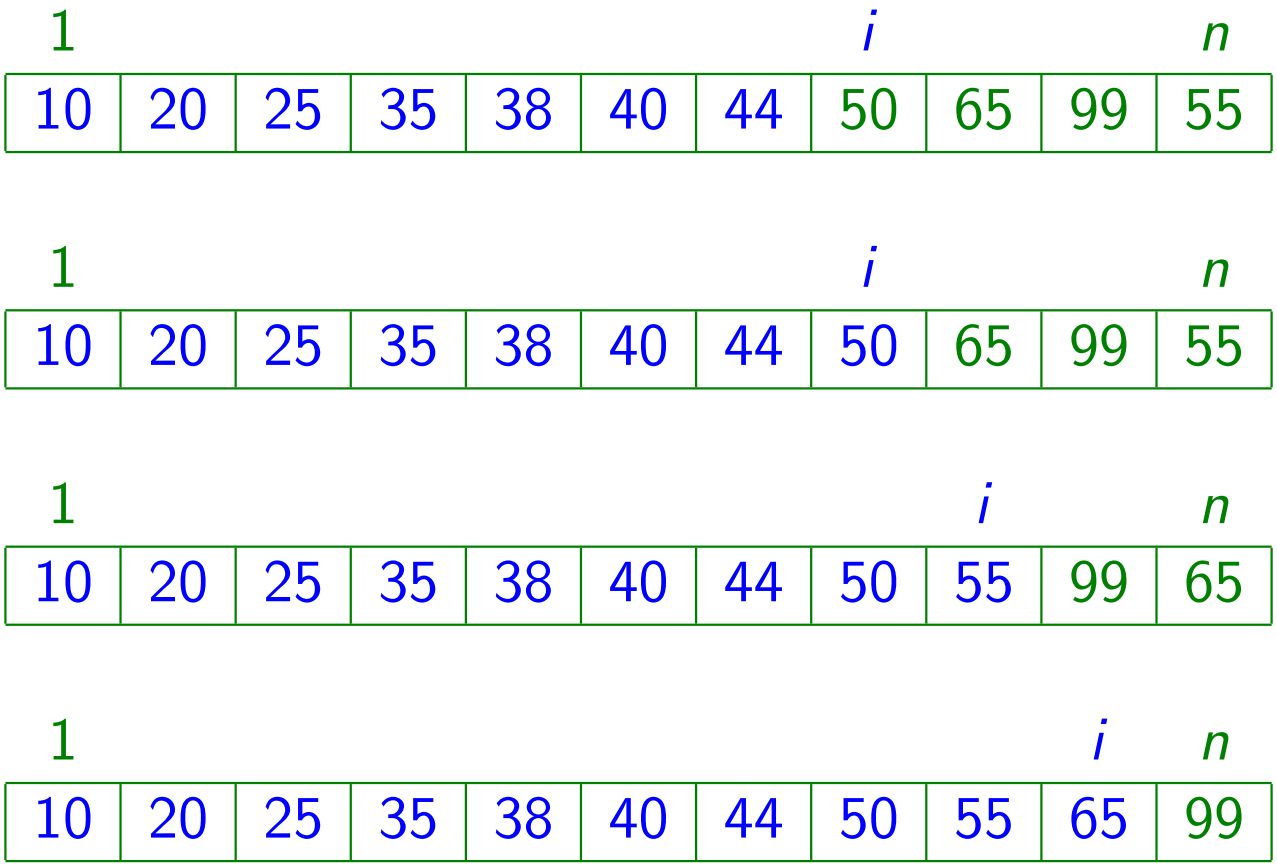




Selection sort



Selection sort



Selection sort

Selection sort

- Mantemos um subvetor $A[1 \ldots i-1]$ tal que:

Selection sort

- Mantemos um subvetor $A[1\,..\,i-1]$ tal que:
  1. $A[1\,..\,i-1]$ está **ordenado** e

Selection sort

- Mantemos um subvetor $A[1\,..\,i-1]$ tal que:
  1. $A[1\,..\,i-1]$ está **ordenado** e
  2. $A[1\,..\,i-1] \leq A[i\,..\,n]$.

Selection sort

- Mantemos um subvetor $A[1\,..\,i-1]$ tal que:
  1. $A[1\,..\,i-1]$ está **ordenado** e
  2. $A[1\,..\,i-1] \leq A[i\,..\,n]$.

  A cada passo encontramos a posição *min* do menor elemento em $A[i\,..\,n]$ e trocamos $A[min]$ e $A[i]$ de lugar.

Selection sort

- Mantemos um subvetor $A[1\,..\,i-1]$ tal que:
  1. $A[1\,..\,i-1]$ está **ordenado** e
  2. $A[1\,..\,i-1] \leq A[i\,..\,n]$.

  A cada passo encontramos a posição *min* do menor elemento em $A[i\,..\,n]$ e trocamos $A[min]$ e $A[i]$ de lugar.
- Repetimos o processo para $i = 1, \ldots, n-1$ e ordenamos o vetor.

Selection sort

- Mantemos um subvetor $A[1\,..\,i-1]$ tal que:
  1. $A[1\,..\,i-1]$ está **ordenado** e
  2. $A[1\,..\,i-1] \leq A[i\,..\,n]$.

  A cada passo encontramos a posição *min* do menor elemento em $A[i\,..\,n]$ e trocamos $A[min]$ e $A[i]$ de lugar.
- Repetimos o processo para $i = 1, \ldots, n-1$ e ordenamos o vetor.

Primeira iteração: $i = 1$

Selection sort – pseudo-código

```
SELECTION-SORT(A, n)
1   para i ← 1 até n − 1 faça
2       min ← i
3       para j ← i + 1 até n faça
4           se A[j] < A[min] então min ← j
5       A[i] ↔ A[min]
```

Selection sort – pseudo-código

```
SELECTION-SORT(A, n)
1   para i ← 1 até n − 1 faça
2       min ← i
3       para j ← i + 1 até n faça
4           se A[j] < A[min] então min ← j
5       A[i] ↔ A[min]
```

Invariantes:

1. $A[1\,..\,i-1]$ está ordenado,
2. $A[1\,..\,i-1] \leq A[i\,..\,n]$.

Complexidade de Selection sort

SELECTION-SORT(A, n)	Tempo
1 **para** $i \leftarrow 1$ **até** $n-1$ **faça**	?
2 $min \leftarrow i$	?
3 **para** $j \leftarrow i+1$ **até** n **faça**	?
4 **se** $A[j] < A[min]$ **então** $min \leftarrow j$	?
5 $A[i] \leftrightarrow A[min]$	?

Consumo de tempo no pior caso: ?

Complexidade de Selection sort

SELECTION-SORT(A, n)		Tempo
1	**para** $i \leftarrow 1$ **até** $n-1$ **faça**	$\Theta(n)$
2	$min \leftarrow i$	$\Theta(n)$
3	**para** $j \leftarrow i+1$ **até** n **faça**	$\Theta(n^2)$
4	**se** $A[j] < A[min]$ **então** $min \leftarrow j$	$\Theta(n^2)$
5	$A[i] \leftrightarrow A[min]$	$\Theta(n)$

Consumo de tempo no pior caso: $O(n^2)$

Selection sort

Selection sort

- Complexidade de tempo no pior caso: $\Theta(n^2)$
  $\Theta(n^2)$ comparações
  $\Theta(n)$ movimentações

Selection sort

- Complexidade de tempo no pior caso: $\Theta(n^2)$
  $\Theta(n^2)$ comparações
  $\Theta(n)$ movimentações
- Complexidade de tempo no melhor caso: $\Theta(n^2)$
  Mesmo que o pior caso.

Selection sort

- Complexidade de tempo no pior caso: $\Theta(n^2)$
  $\Theta(n^2)$ comparações
  $\Theta(n)$ movimentações
- Complexidade de tempo no melhor caso: $\Theta(n^2)$
  Mesmo que o pior caso.
- Complexidade de espaço/consumo espaço: $\Theta(n)$

Mergesort

O algoritmo *Mergesort* é um exemplo clássico de paradigma de divisão-e-conquista.

Mergesort

O algoritmo *Mergesort* é um exemplo clássico de paradigma de divisão-e-conquista.

- Divisão: divida o vetor de n elementos em subvetores de tamanhos $\lceil n/2 \rceil$ e $\lfloor n/2 \rfloor$.

Mergesort

O algoritmo *Mergesort* é um exemplo clássico de paradigma de divisão-e-conquista.

- Divisão: divida o vetor de n elementos em subvetores de tamanhos $\lceil n/2 \rceil$ e $\lfloor n/2 \rfloor$.
- Conquista: recursivamente ordene cada subvetor.

Mergesort

O algoritmo *Mergesort* é um exemplo clássico de paradigma de divisão-e-conquista.

- Divisão: divida o vetor de n elementos em subvetores de tamanhos $\lceil n/2 \rceil$ e $\lfloor n/2 \rfloor$.
- Conquista: recursivamente ordene cada subvetor.
- Combinação: intercale os subvetores ordenados para obter o vetor ordenado.

Mergesort: pseudo-código

```
MergeSort(A, p, r)
1   se p < r
2       então q ← ⌊(p + r)/2⌋
3             MergeSort(A, p, q)
4             MergeSort(A, q + 1, r)
5             Intercala(A, p, q, r)
```

Mergesort: pseudo-código

```
MergeSort(A, p, r)
1   se p < r
2       então q ← ⌊(p + r)/2⌋
3             MergeSort(A, p, q)
4             MergeSort(A, q + 1, r)
5             Intercala(A, p, q, r)
```

A complexidade de MergeSort é dada pela recorrência:

$$T(n) = T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + O(f(n)),$$

onde $f(n)$ é a complexidade de Intercala.

Intercalação

Intercalação

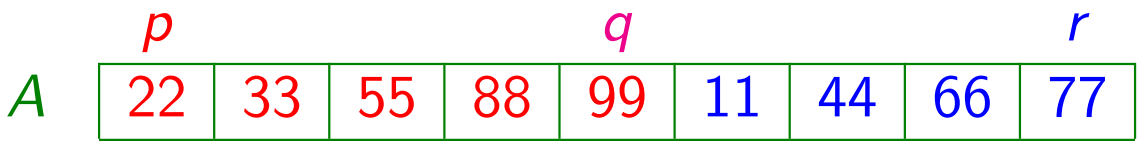
O que significa intercalar dois (sub)vetores ordenados?

Intercalação

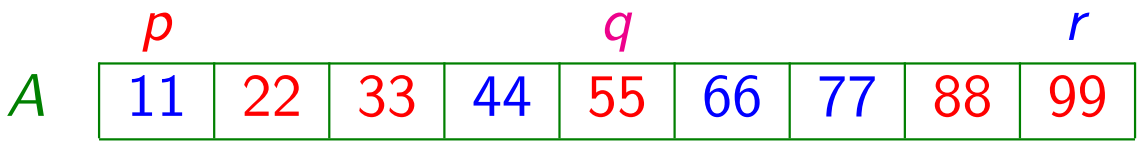
O que significa intercalar dois (sub)vetores ordenados?

Problema: Dados $A[p\,..\,q]$ e $A[q{+}1\,..\,r]$ crescentes, rearranjar $A[p\,..\,r]$ de modo que ele fique em ordem crescente.

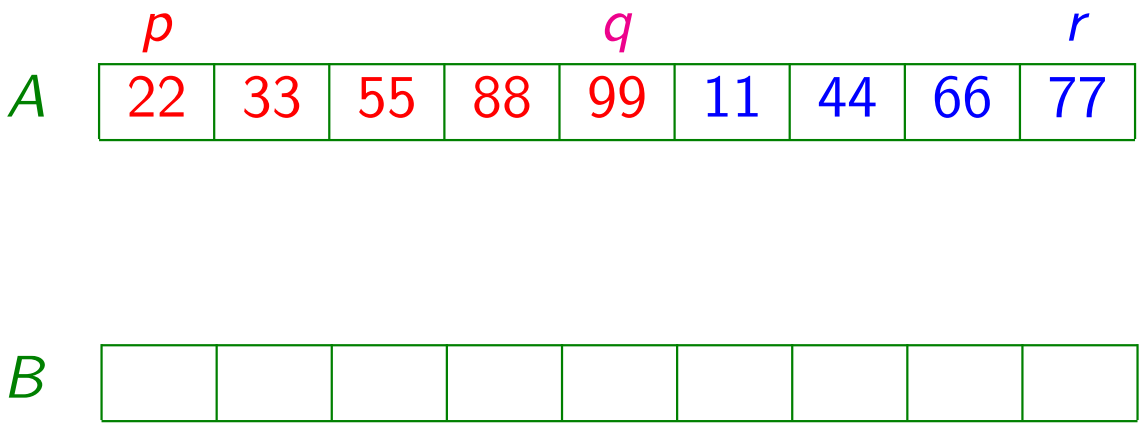
Entrada:



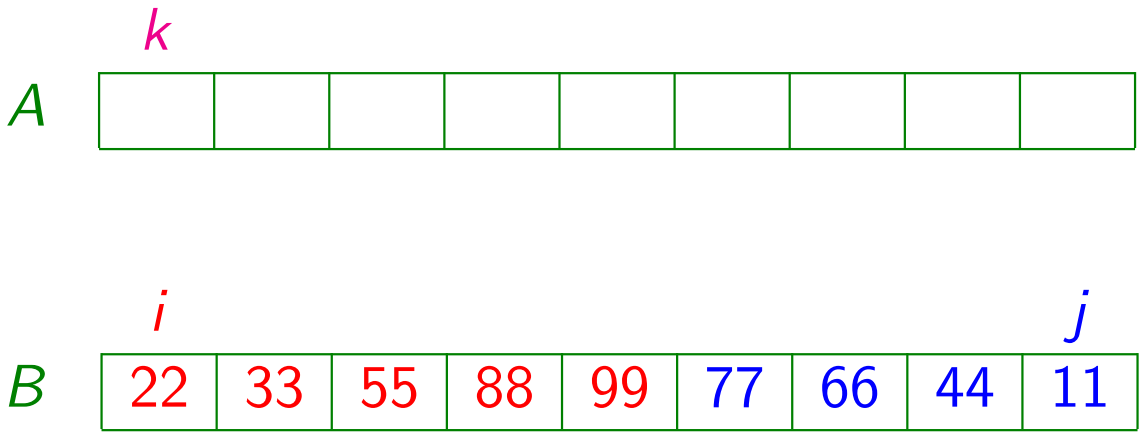
Saída:



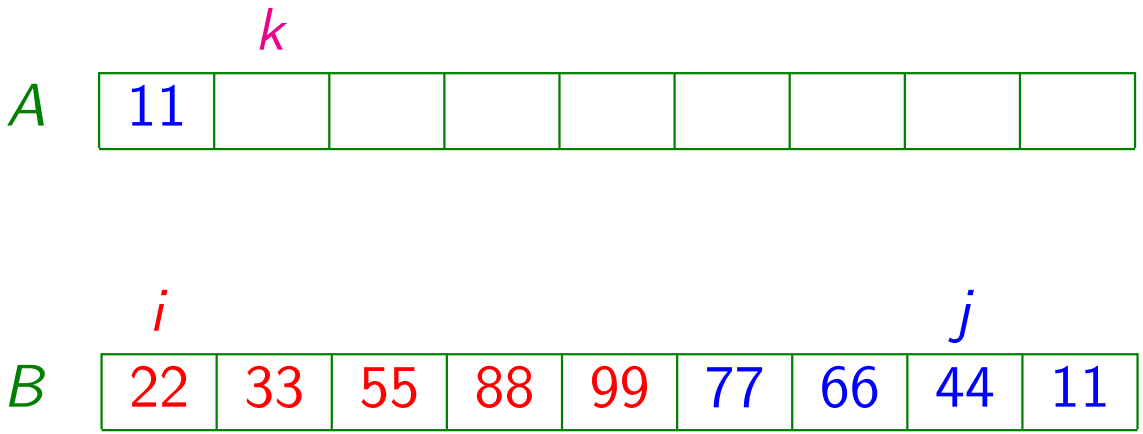
Intercalação



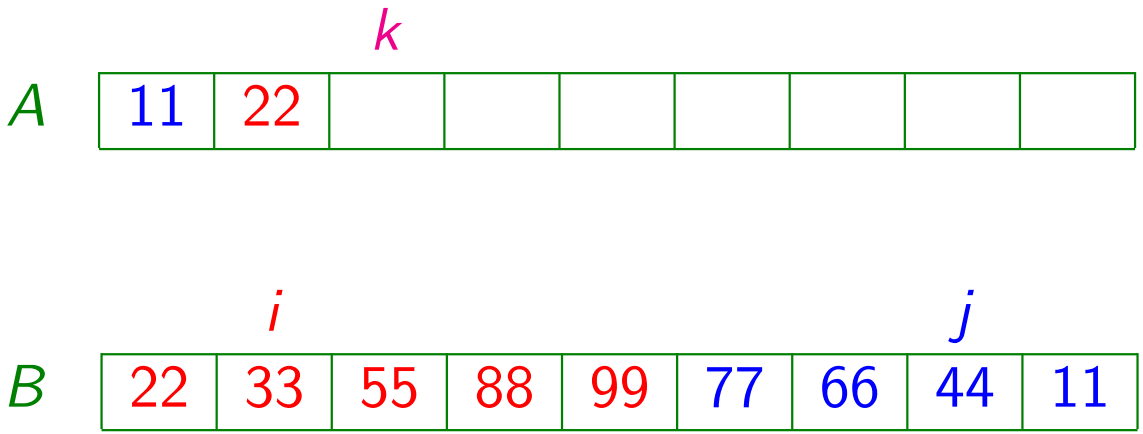
Intercalação



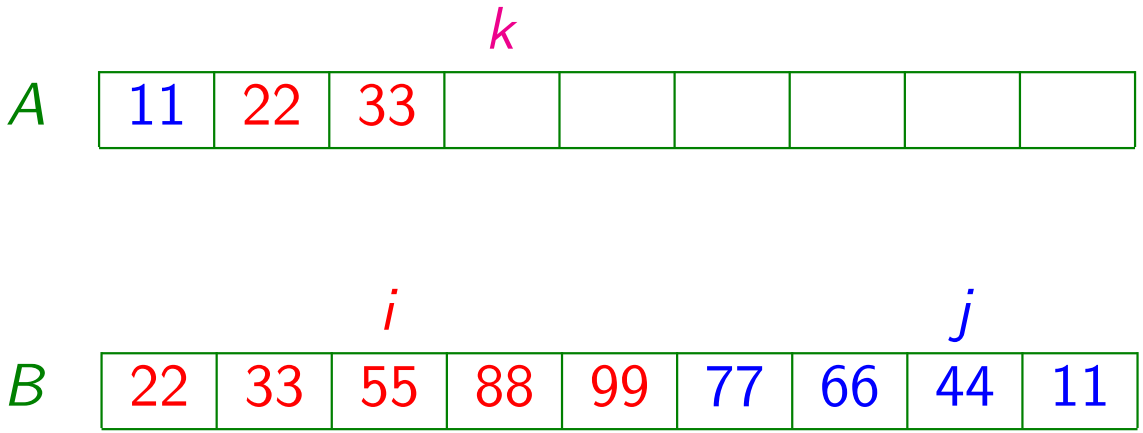
Intercalação



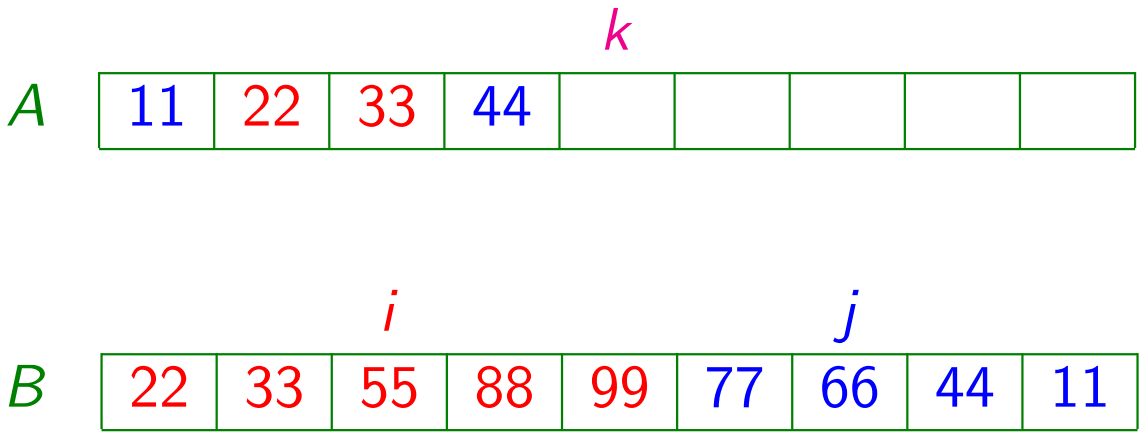
Intercalação



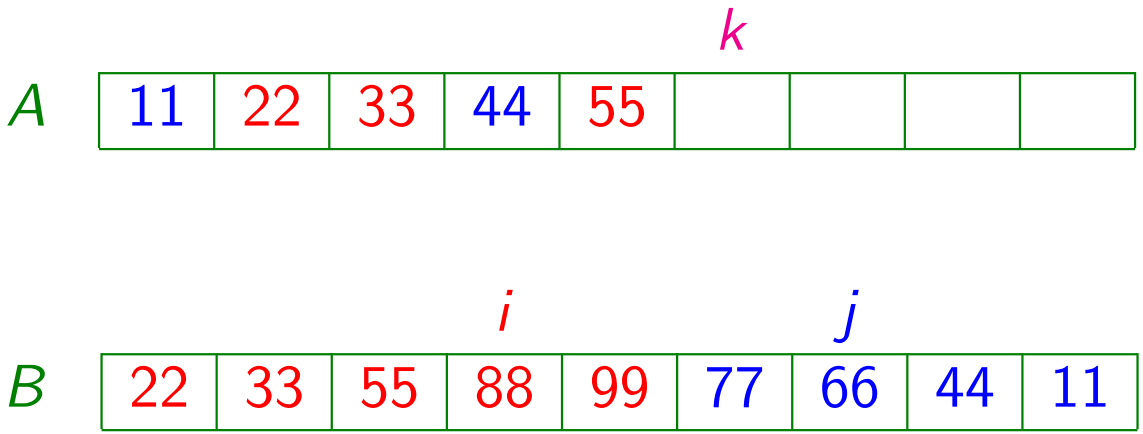
Intercalação



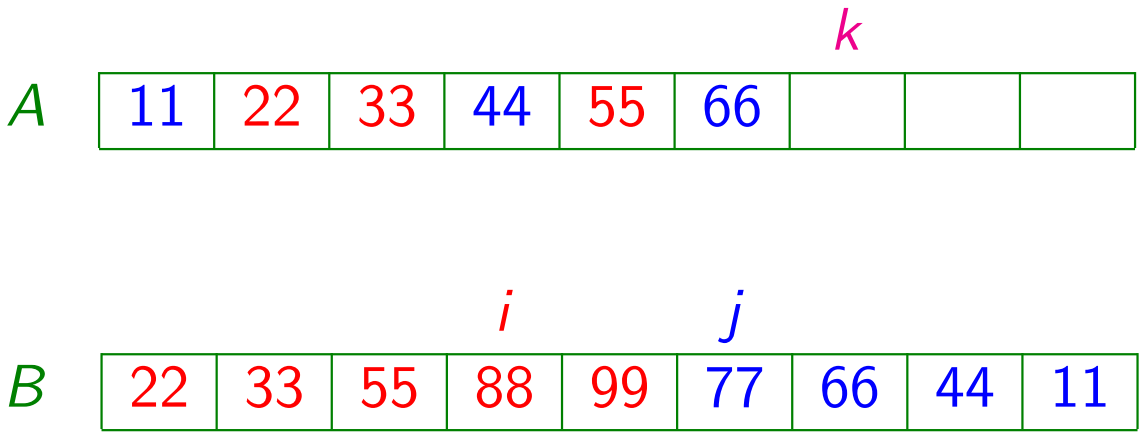
Intercalação



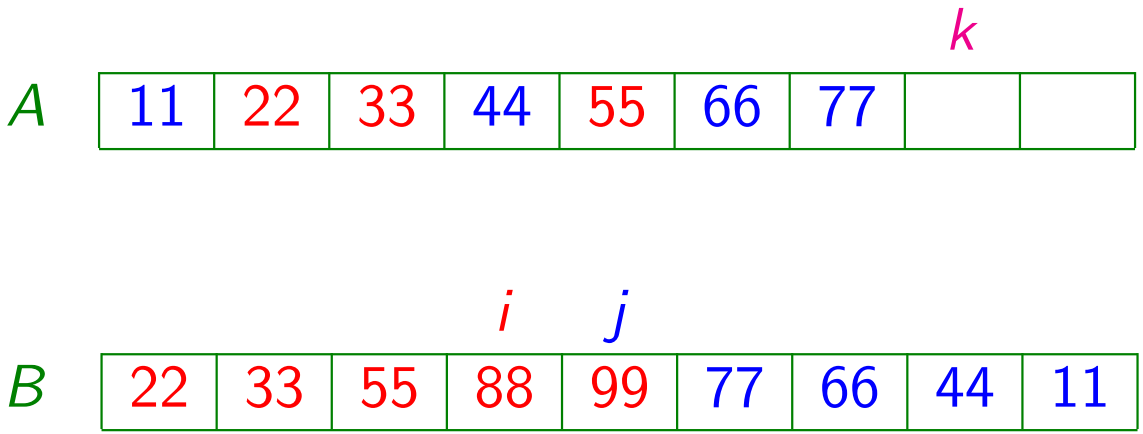
Intercalação



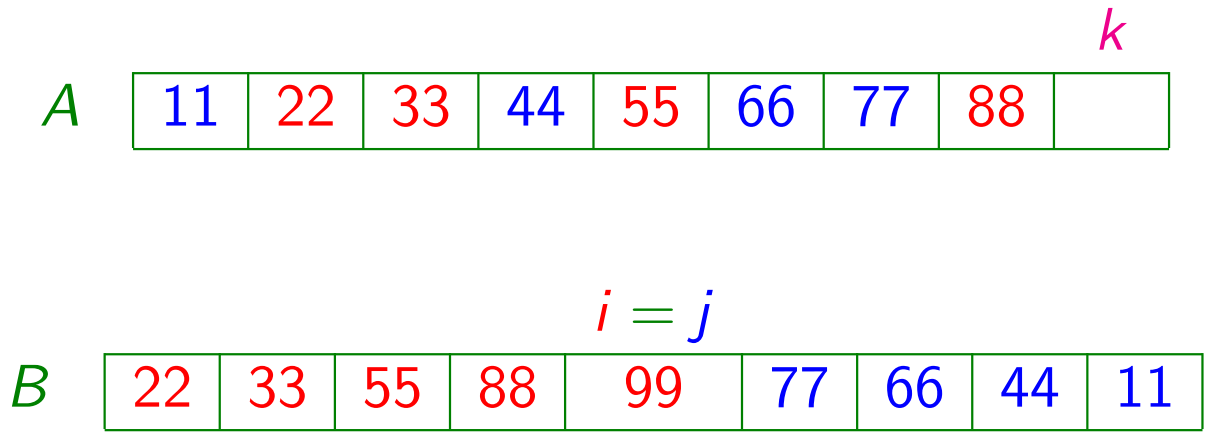
Intercalação



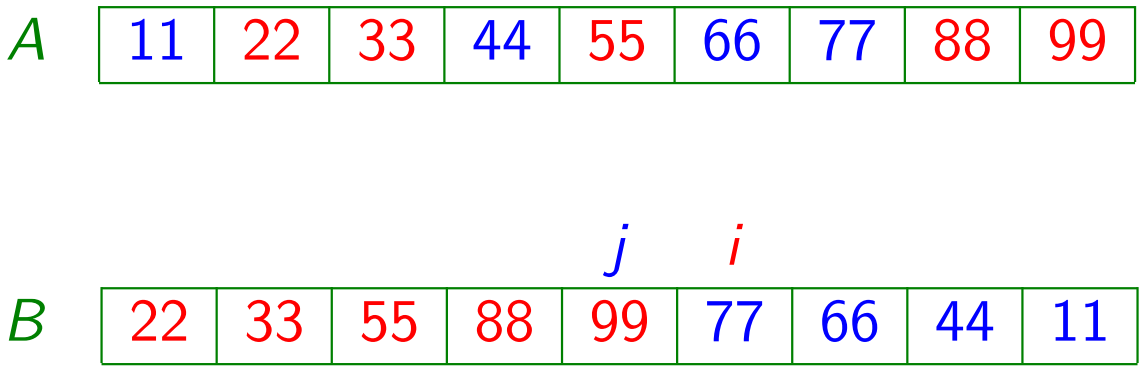
Intercalação



Intercalação



Intercalação



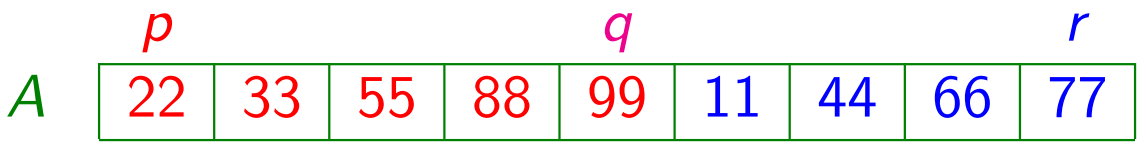
Pseudo-código

Pseudo-código

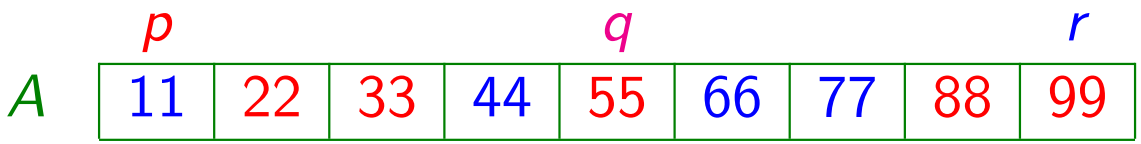
```
INTERCALA(A, p, q, r)
 1   para i ← p até q faça
 2       B[i] ← A[i]
 3   para j ← q + 1 até r faça
 4       B[r + q + 1 − j] ← A[j]
 5   i ← p
 6   j ← r
 7   para k ← p até r faça
 8       se B[i] ≤ B[j]
 9           então A[k] ← B[i]
10                 i ← i + 1
11           senão A[k] ← B[j]
12                 j ← j − 1
```

Complexidade de Intercala

Entrada:



Saída:



Tamanho da entrada: $n = r - p + 1$

Consumo de tempo: $\Theta(n)$

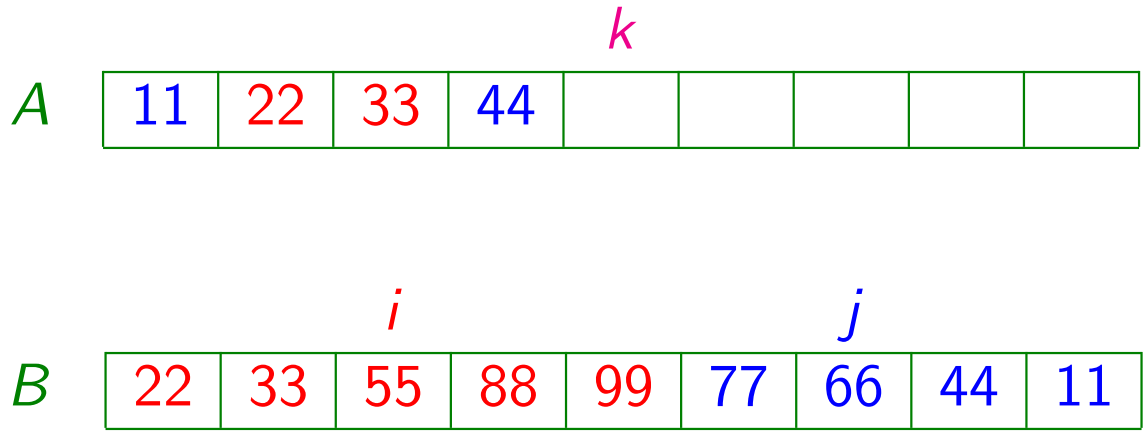
Corretude de Intercala

Corretude de Intercala

Invariante principal de Intercala:

No começo de cada iteração do laço das linhas 7–12, vale que:

1. $A[p\,..\,k-1]$ está ordenado,
2. $A[p\,..\,k-1]$ contém todos os elementos de $B[p\,..\,i-1]$ e de $B[j+1\,..\,r]$,
3. $B[i] \geq A[k-1]$ e $B[j] \geq A[k-1]$.



Corretude de Intercala

Invariante principal de Intercala:

No começo de cada iteração do laço das linhas 7–12, vale que:

1. $A[p\,..\,k-1]$ está ordenado,
2. $A[p\,..\,k-1]$ contém todos os elementos de $B[p\,..\,i-1]$ e de $B[j+1\,..\,r]$,
3. $B[i] \geq A[k-1]$ e $B[j] \geq A[k-1]$.

Exercício. Prove que a afirmação acima é de fato um invariante de Intercala.

Exercício. (fácil) Mostre usando o invariante acima que Intercala é correto.

Corretude do Mergesort

```
MergeSort(A, p, r)
1   se p < r
2       então q ← ⌊(p + r)/2⌋
3             MergeSort(A, p, q)
4             MergeSort(A, q + 1, r)
5             Intercala(A, p, q, r)
```

Corretude do Mergesort

```
MergeSort(A, p, r)
1   se p < r
2       então q ← ⌊(p + r)/2⌋
3             MergeSort(A, p, q)
4             MergeSort(A, q + 1, r)
5             Intercala(A, p, q, r)
```

O algoritmo está correto?

Corretude do Mergesort

```
MergeSort(A, p, r)
1   se p < r
2       então q ← ⌊(p + r)/2⌋
3             MergeSort(A, p, q)
4             MergeSort(A, q + 1, r)
5             Intercala(A, p, q, r)
```

O algoritmo está correto?

A corretude do algoritmo MergeSort apoia-se na corretude do algoritmo Intercala e segue facilmente **por indução** em $n := r - p + 1$.

Corretude do Mergesort

Corretude do Mergesort

Base: MergeSort ordena vetores de tamanho 0 ou 1.

Corretude do Mergesort

Base: MergeSort ordena vetores de tamanho 0 ou 1.

Hipótese de indução: MergeSort ordena vetores com menos que n elementos.

Corretude do Mergesort

Base: MergeSort ordena vetores de tamanho 0 ou 1.

Hipótese de indução: MergeSort ordena vetores com menos que n elementos.

Passo de indução: por hipótese de indução, MergeSort ordena os dois subvetores (de tamanho $\lceil n/2 \rceil$ e $\lfloor n/2 \rfloor$).

Corretude do Mergesort

Base: MergeSort ordena vetores de tamanho 0 ou 1.

Hipótese de indução: MergeSort ordena vetores com menos que n elementos.

Passo de indução: por hipótese de indução, MergeSort ordena os dois subvetores (de tamanho $\lceil n/2 \rceil$ e $\lfloor n/2 \rfloor$).

Pela corretude de Intercala, segue que o vetor resultante da intercalação é um vetor ordenado de n elementos.

Corretude do Mergesort

Base: MergeSort ordena vetores de tamanho 0 ou 1.

Hipótese de indução: MergeSort ordena vetores com menos que n elementos.

Passo de indução: por hipótese de indução, MergeSort ordena os dois subvetores (de tamanho $\lceil n/2 \rceil$ e $\lfloor n/2 \rfloor$).

Pela corretude de Intercala, segue que o vetor resultante da intercalação é um vetor ordenado de n elementos.

Portanto, MergeSort é correto.

Complexidade de Mergesort

```
MergeSort(A, p, r)
1   se p < r
2       então q ← ⌊(p + r)/2⌋
3             MergeSort(A, p, q)
4             MergeSort(A, q + 1, r)
5             Intercala(A, p, q, r)
```

Complexidade de Mergesort

```
MergeSort(A, p, r)
1   se p < r
2      então q ← ⌊(p + r)/2⌋
3            MergeSort(A, p, q)
4            MergeSort(A, q + 1, r)
5            Intercala(A, p, q, r)
```

$T(n)$: complexidade de pior caso de MergeSort

Complexidade de Mergesort

```
MergeSort(A, p, r)
1   se p < r
2      então q ← ⌊(p + r)/2⌋
3            MergeSort(A, p, q)
4            MergeSort(A, q + 1, r)
5            Intercala(A, p, q, r)
```

$T(n)$: complexidade de pior caso de MergeSort

Então

$$T(n) = \begin{cases} \Theta(1), & n = 0, \\ T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + \Theta(n), & n > 0. \end{cases}$$

Complexidade de Mergesort

```
MergeSort(A, p, r)
1   se p < r
2      então q ← ⌊(p + r)/2⌋
3            MergeSort(A, p, q)
4            MergeSort(A, q + 1, r)
5            Intercala(A, p, q, r)
```

$T(n)$: complexidade de pior caso de MergeSort

Então

$$T(n) = \begin{cases} \Theta(1), & n = 0, \\ T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + \Theta(n), & n > 0. \end{cases}$$

A solução da recorrência é $T(n) = \Theta(n \lg n)$.

Mergesort

Mergesort

- Complexidade de tempo: $\Theta(n \lg n)$
  $\Theta(n \lg n)$ comparações
  $\Theta(n \lg n)$ movimentações

Mergesort

- Complexidade de tempo: $\Theta(n \lg n)$
  $\Theta(n \lg n)$ comparações
  $\Theta(n \lg n)$ movimentações

  O pior caso e o melhor caso têm a mesma complexidade.

Mergesort

- Complexidade de tempo: $\Theta(n \lg n)$
  $\Theta(n \lg n)$ comparações
  $\Theta(n \lg n)$ movimentações

  O pior caso e o melhor caso têm a mesma complexidade.
- Complexidade de espaço/consumo espaço: $\Theta(n)$

Mergesort

- Complexidade de tempo: $\Theta(n \lg n)$
  $\Theta(n \lg n)$ comparações
  $\Theta(n \lg n)$ movimentações

  O pior caso e o melhor caso têm a mesma complexidade.
- Complexidade de espaço/consumo espaço: $\Theta(n)$

  O *Mergesort* usa um vetor auxiliar de tamanho n para fazer a intercalação, mas o espaço ainda é $\Theta(n)$.

Mergesort

- Complexidade de tempo: $\Theta(n \lg n)$
  $\Theta(n \lg n)$ comparações
  $\Theta(n \lg n)$ movimentações

  O pior caso e o melhor caso têm a mesma complexidade.
- Complexidade de espaço/consumo espaço: $\Theta(n)$

  O *Mergesort* usa um vetor auxiliar de tamanho n para fazer a intercalação, mas o espaço ainda é $\Theta(n)$.
- O *Mergesort* é util para ordenação externa, quando não é possível armazenar todos os elementos na memória primária.

Heapsort

Heapsort

- O *Heapsort* é um algoritmo de ordenação que usa uma estrutura de dados sofisticada chamada *heap*.

Heapsort

- O *Heapsort* é um algoritmo de ordenação que usa uma estrutura de dados sofisticada chamada *heap*.
- A complexidade de pior caso é $\Theta(n \lg n)$.

Heapsort

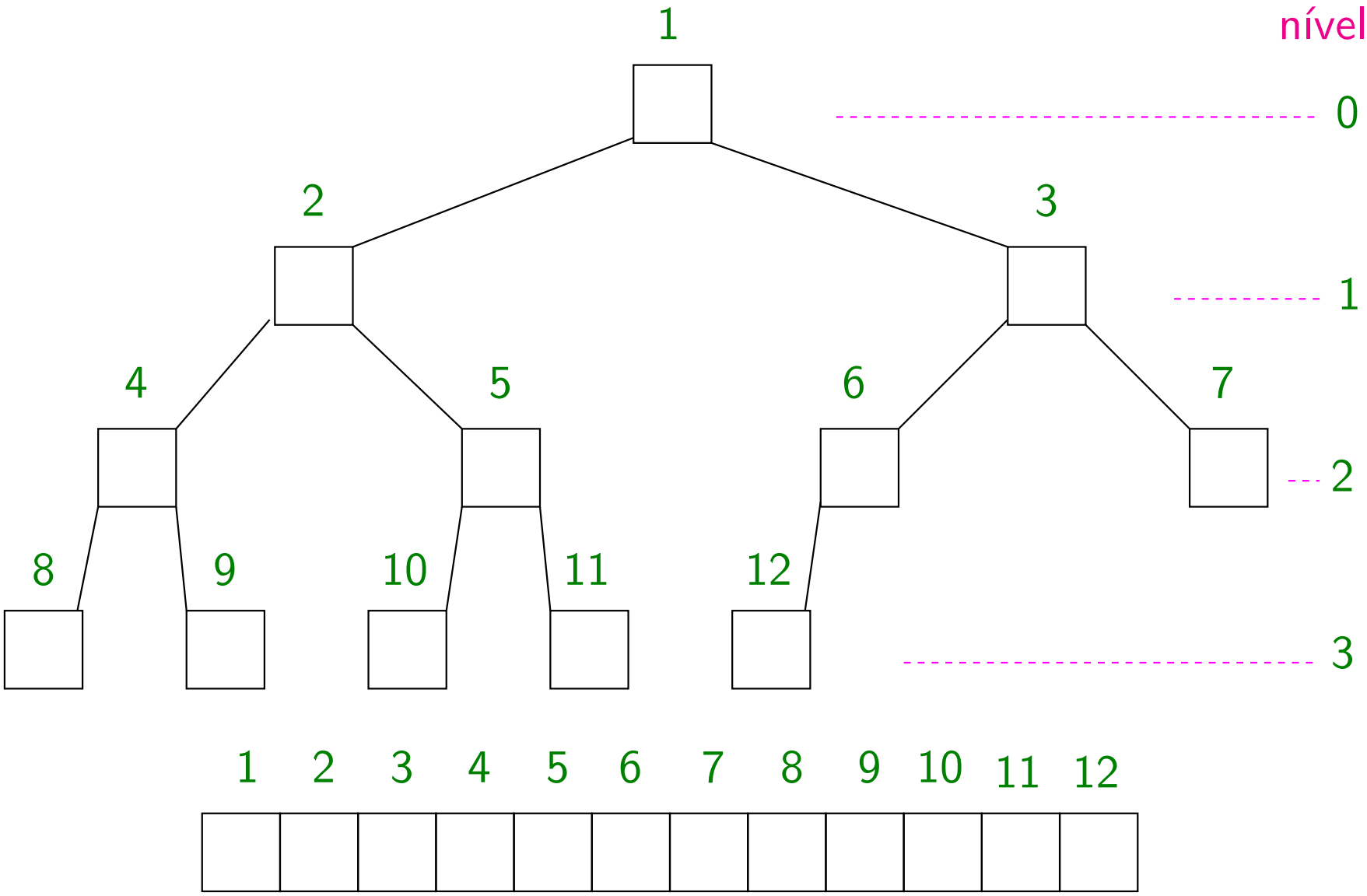
- O *Heapsort* é um algoritmo de ordenação que usa uma estrutura de dados sofisticada chamada *heap*.
- A complexidade de pior caso é $\Theta(n \lg n)$.
- *Heaps* podem ser utilizados para implementar filas de prioridade que são extremamente úteis em outros algoritmos.

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 quase completa: todos os níveis estão completamente preenchidos, com a possível exceção do último que é preenchido da esquerda até um certo ponto.

  Uma árvore binária completa tem todos os seus níveis completamente preenchidos.

Heaps



Heaps

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

Heaps

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

- Cada posição do vetor corresponde a um nó do heap.

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

Heaps

- Um nó i tem
  $2i$ como filho esquerdo e
  $2i + 1$ como filho direito.

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 $2i + 1 \leq n$.

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 $2i > n$.

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

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

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 ???.

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 $\lfloor \lg i \rfloor$.

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

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 $\lfloor \lg i \rfloor$.

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 ???.

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 $\lfloor \lg i \rfloor$.

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 $1 + \lfloor \lg n \rfloor$.

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.

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

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

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).

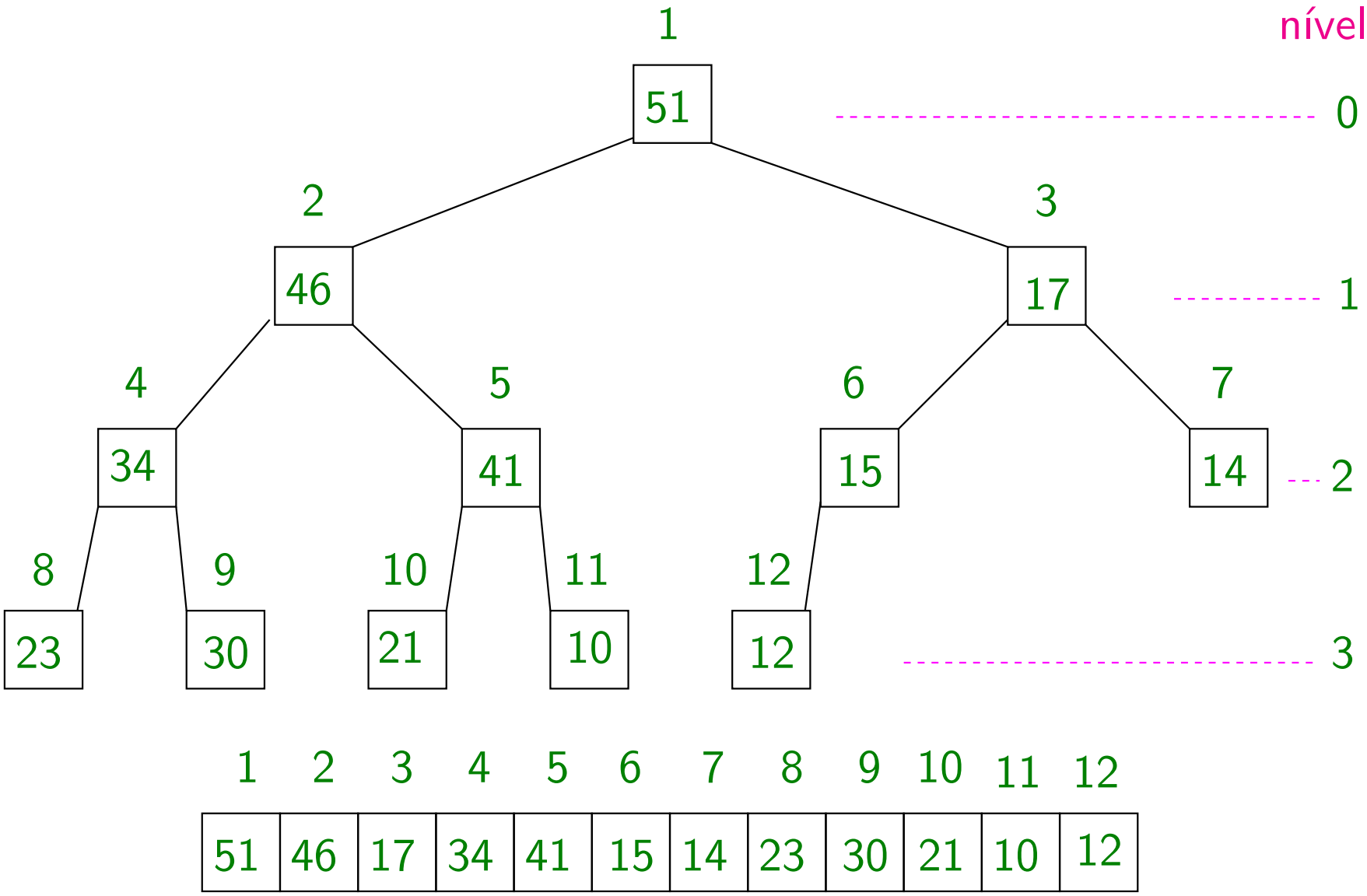
Max-heaps

- 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 quase completa é um *max-heap* se todo nó distinto da raiz satisfaz a propriedade de heap.

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 quase 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


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

Min-heaps

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).

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.

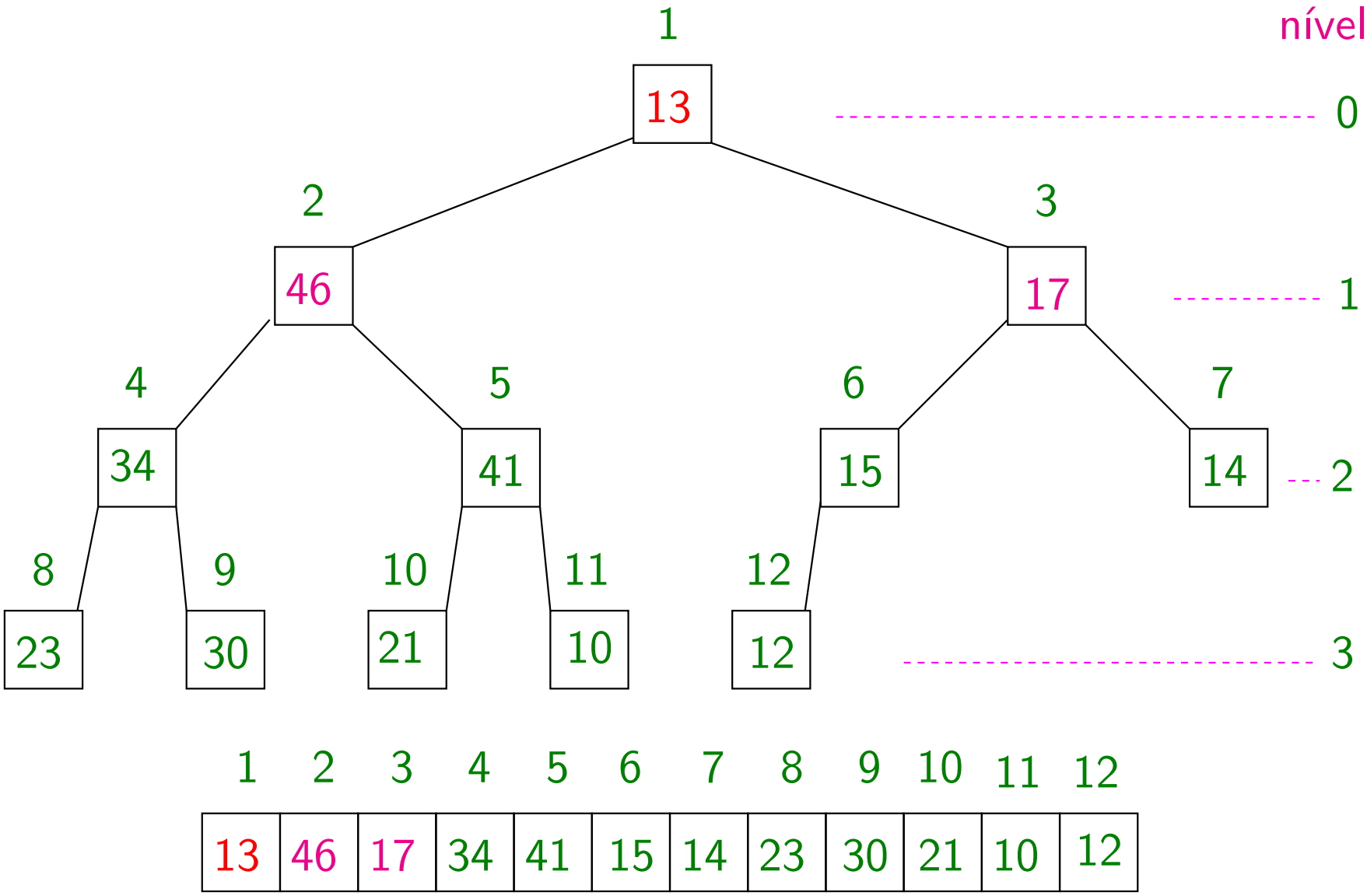
Min-heaps

- 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.

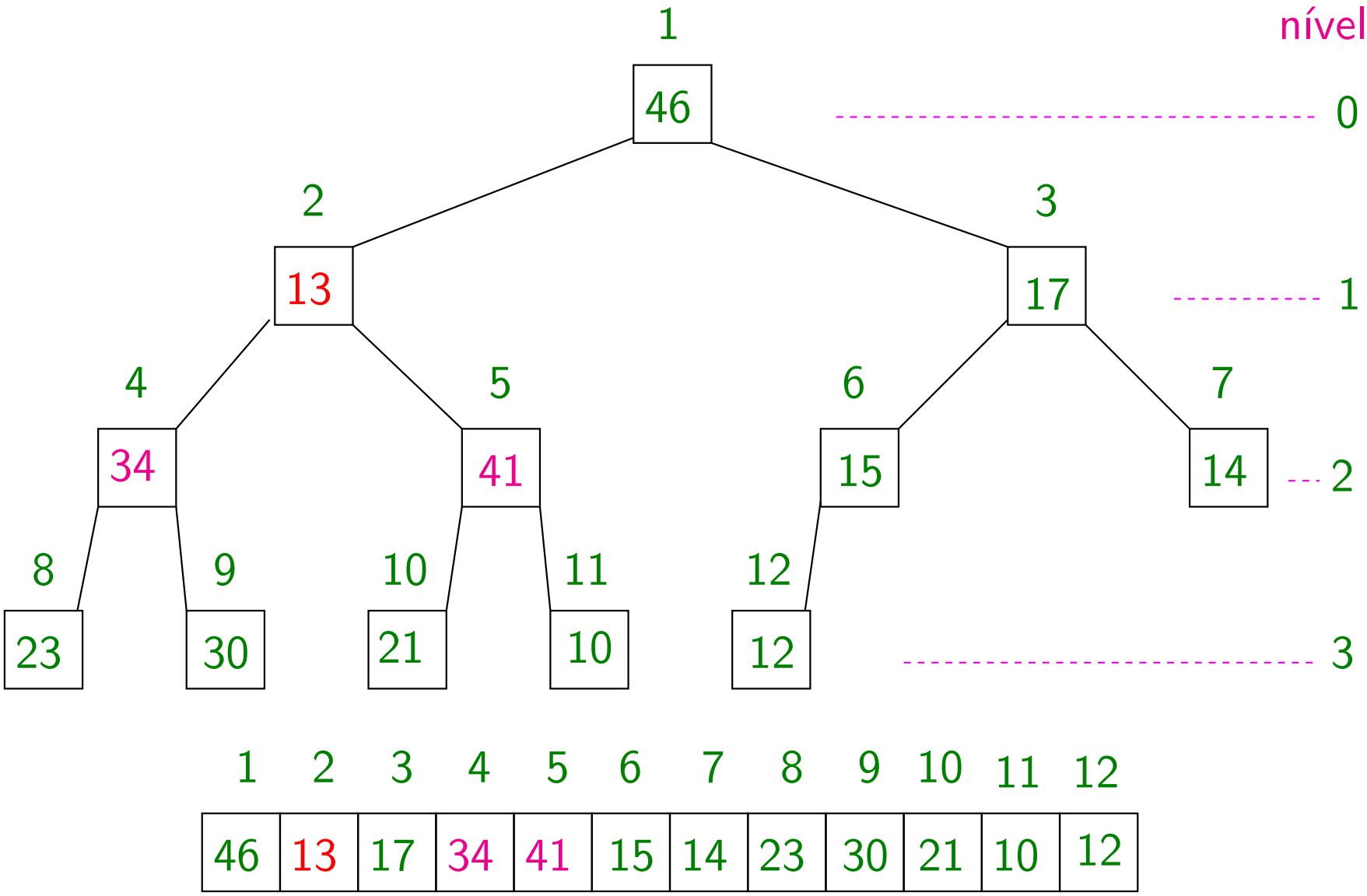
Min-heaps

- 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.

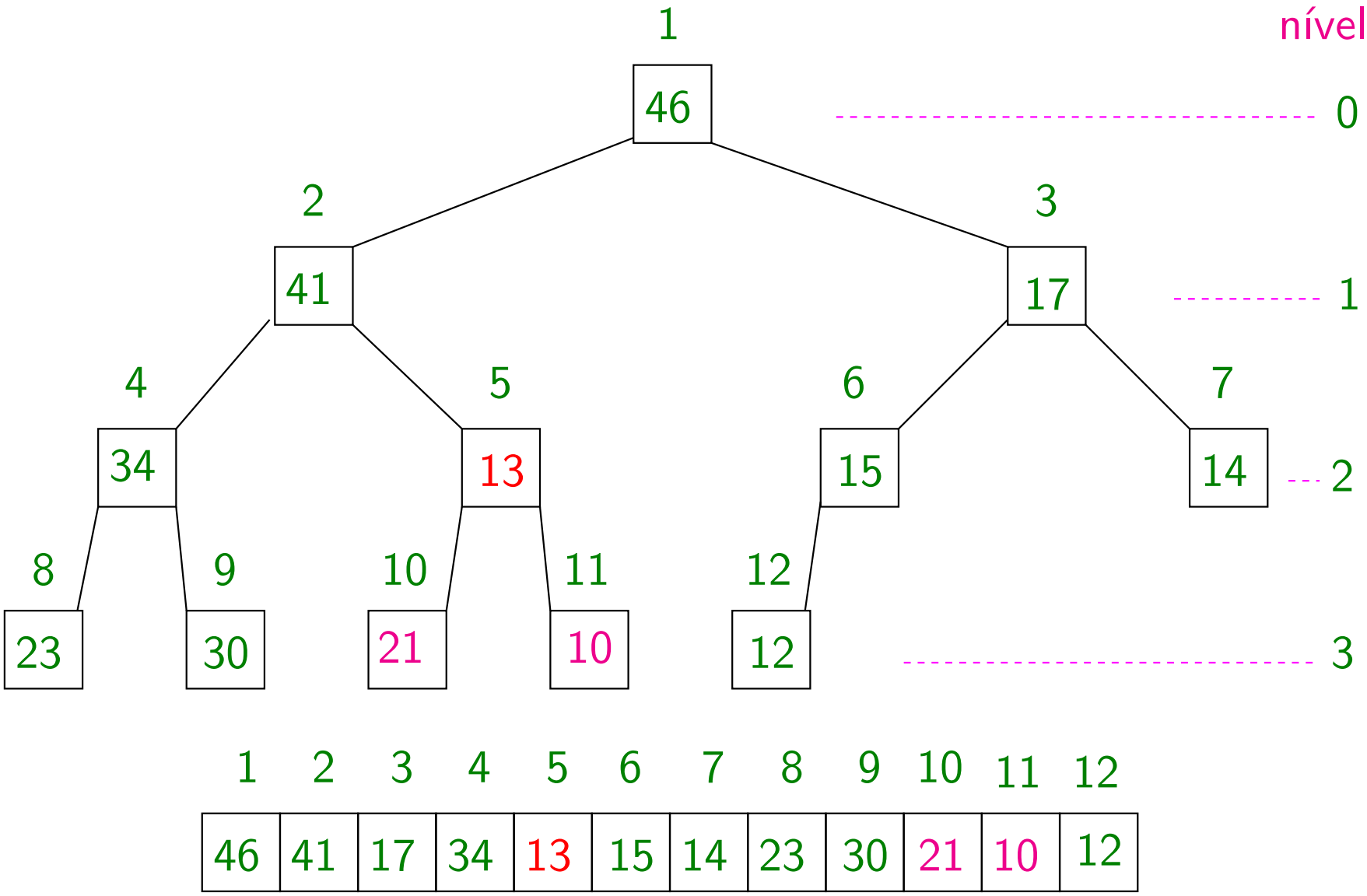
Manipulação de max-heap



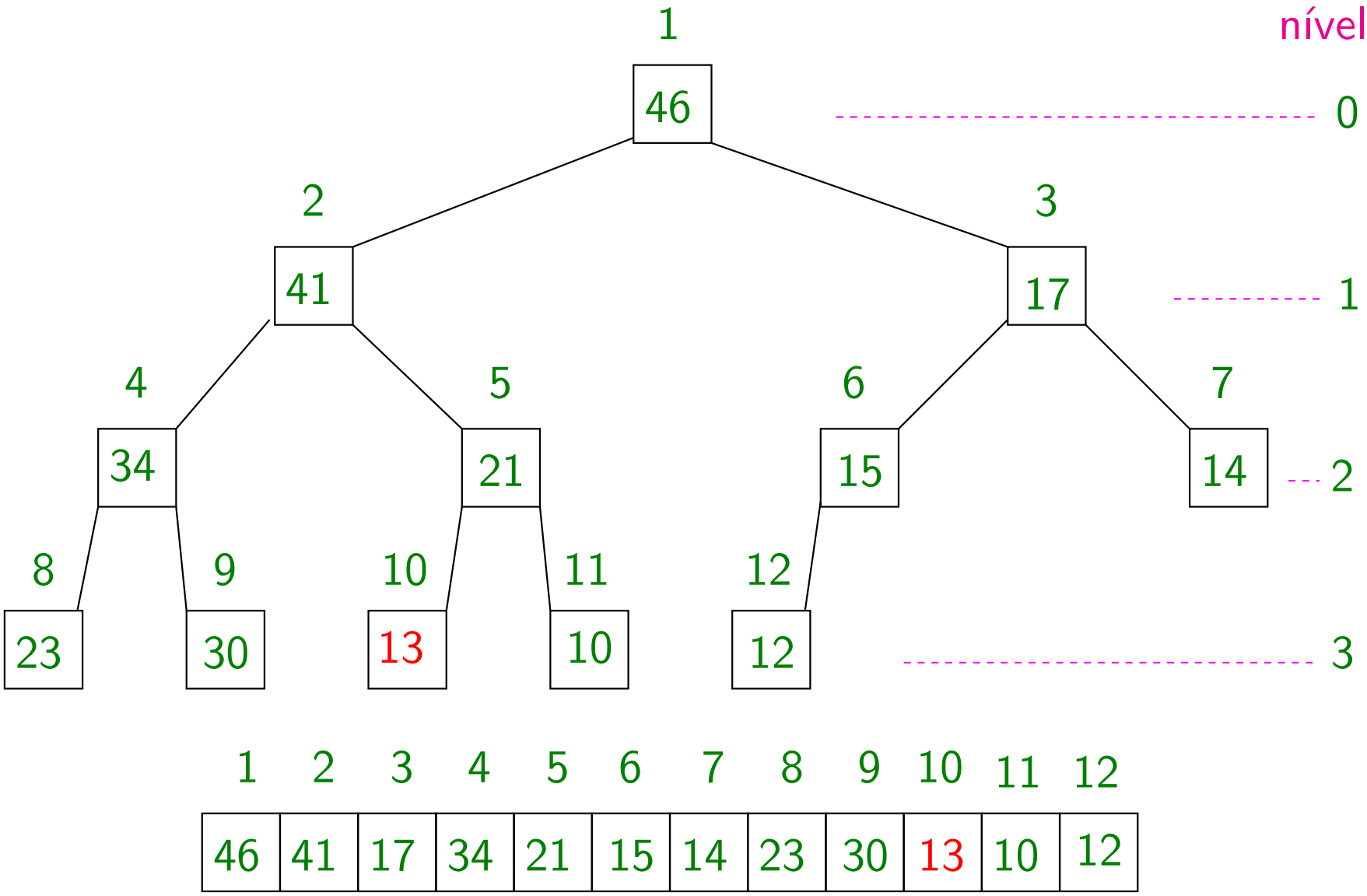
Manipulação de max-heap



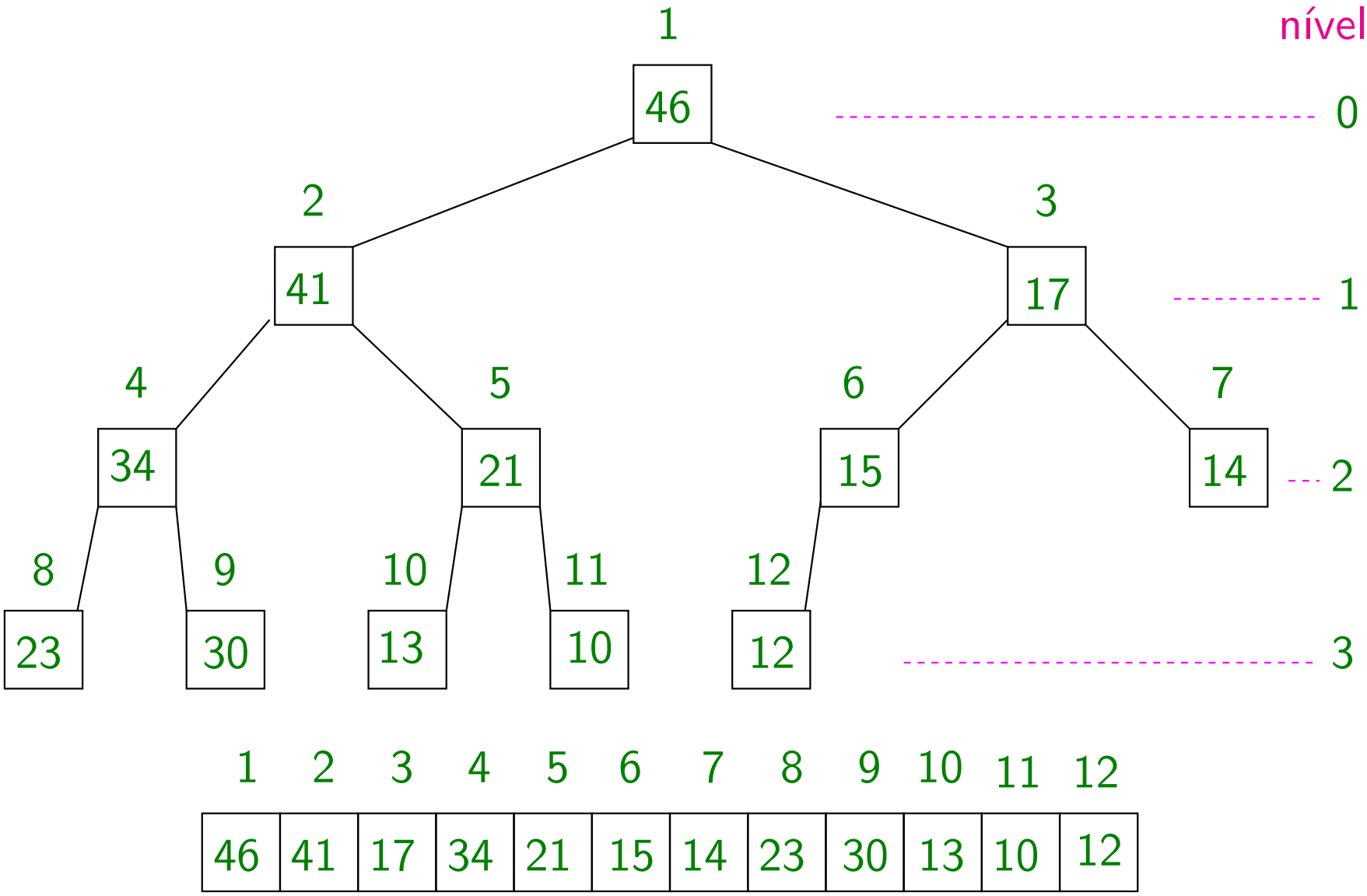
Manipulação de max-heap



Manipulação de max-heap



Manipulação de max-heap



Manipulação de max-heap

Recebe $A[1\,..\,n]$ e $i \geq 1$ tais que as subárvores com raízes $2i$ e $2i+1$ são max-heaps e rearranja A de modo que a 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.

Corretude de MaxHeapify

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

Base: para $h = 0$ (a raiz é folha), o algoritmo funciona.

Corretude de MaxHeapify

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

Base: para $h = 0$ (a raiz é folha), o algoritmo funciona.

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

Corretude de MaxHeapify

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

Base: para $h = 0$ (a raiz é folha), 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 $A[2i + 1]$.

Corretude de MaxHeapify

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

Base: para $h = 0$ (a raiz é folha), 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 $A[2i + 1]$.
- Se maior $= i$ na linha 8, então claramente a subárvore com raiz i é um max-hep.

Corretude de MaxHeapify

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

Base: para $h = 0$ (a raiz é folha), 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 $A[2i + 1]$.
- Se maior $= i$ na linha 8, então claramente a subárvore com raiz i é um max-hep.
- Se maior $\neq i$, então após a troca na linha 9, temos que $A[i]$ é maior ou igual a qualquer elemento das árvores com raízes $A[2i]$ e $A[2i + 1]$, respectivamente.

Corretude de MaxHeapify

Passo de indução:

- A variável maior na linha 8 guarda o índice do maior elemento entre $A[i]$, $A[2i]$ e $A[2i+1]$.
- Se $\mathrm{maior} \neq i$, então após a troca na linha 9, temos que $A[i]$ é maior ou igual a qualquer elemento das árvores com raízes $A[2i]$ e $A[2i+1]$, respectivamente.

Corretude de MaxHeapify

Passo de indução:

- A variável maior na linha 8 guarda o índice do maior elemento entre $A[i]$, $A[2i]$ e $A[2i+1]$.
- Se $\mathrm{maior} \neq i$, então após a troca na linha 9, temos que $A[i]$ é maior ou igual a qualquer elemento das árvores com raízes $A[2i]$ e $A[2i+1]$, respectivamente.

Além disso, as subárvores com raízes $2\mathrm{maior}$ e $2\mathrm{maior}+1$ são max-heaps. Por HI, o algoritmo Max-Heapify transforma a subárvore com raiz maior em um max-heap.

Corretude de MaxHeapify

Passo de indução:

- A variável maior na linha 8 guarda o índice do maior elemento entre $A[i]$, $A[2i]$ e $A[2i+1]$.
- Se $\mathrm{maior} \neq i$, então após a troca na linha 9, temos que $A[i]$ é maior ou igual a qualquer elemento das árvores com raízes $A[2i]$ e $A[2i+1]$, respectivamente.

Além disso, as subárvores com raízes $2\mathrm{maior}$ e $2\mathrm{maior}+1$ são max-heaps. Por HI, o algoritmo Max-Heapify transforma a subárvore com raiz maior em um max-heap.

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

Corretude de MaxHeapify

Passo de indução:

- A variável maior na linha 8 guarda o índice do maior elemento entre $A[i]$, $A[2i]$ e $A[2i+1]$.
- Se $\text{maior} \neq i$, então após a troca na linha 9, temos que $A[i]$ é maior ou igual a qualquer elemento das árvores com raízes $A[2i]$ e $A[2i+1]$, respectivamente.

Além disso, as subárvores com raízes 2maior e $2\text{maior}+1$ são max-heaps. Por HI, o algoritmo Max-Heapify transforma a subárvore com raiz maior em um max-heap.

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** $\text{maior} \leftarrow e$	?
5	**senão** $\text{maior} \leftarrow i$	?
6	**se** $d \leq n$ e $A[d] > A[\text{maior}]$	?
7	**então** $\text{maior} \leftarrow d$	?
8	**se** $\text{maior} \neq i$	?
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 $i = \lfloor \lg \frac{n}{i} \rfloor$

$T(h) \leq T(h-1) + \Theta(5) + O(4)$.

Complexidade de MaxHeapify

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

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

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

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 ???.

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 $O(h)$.

Complexidade de MaxHeapify

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

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

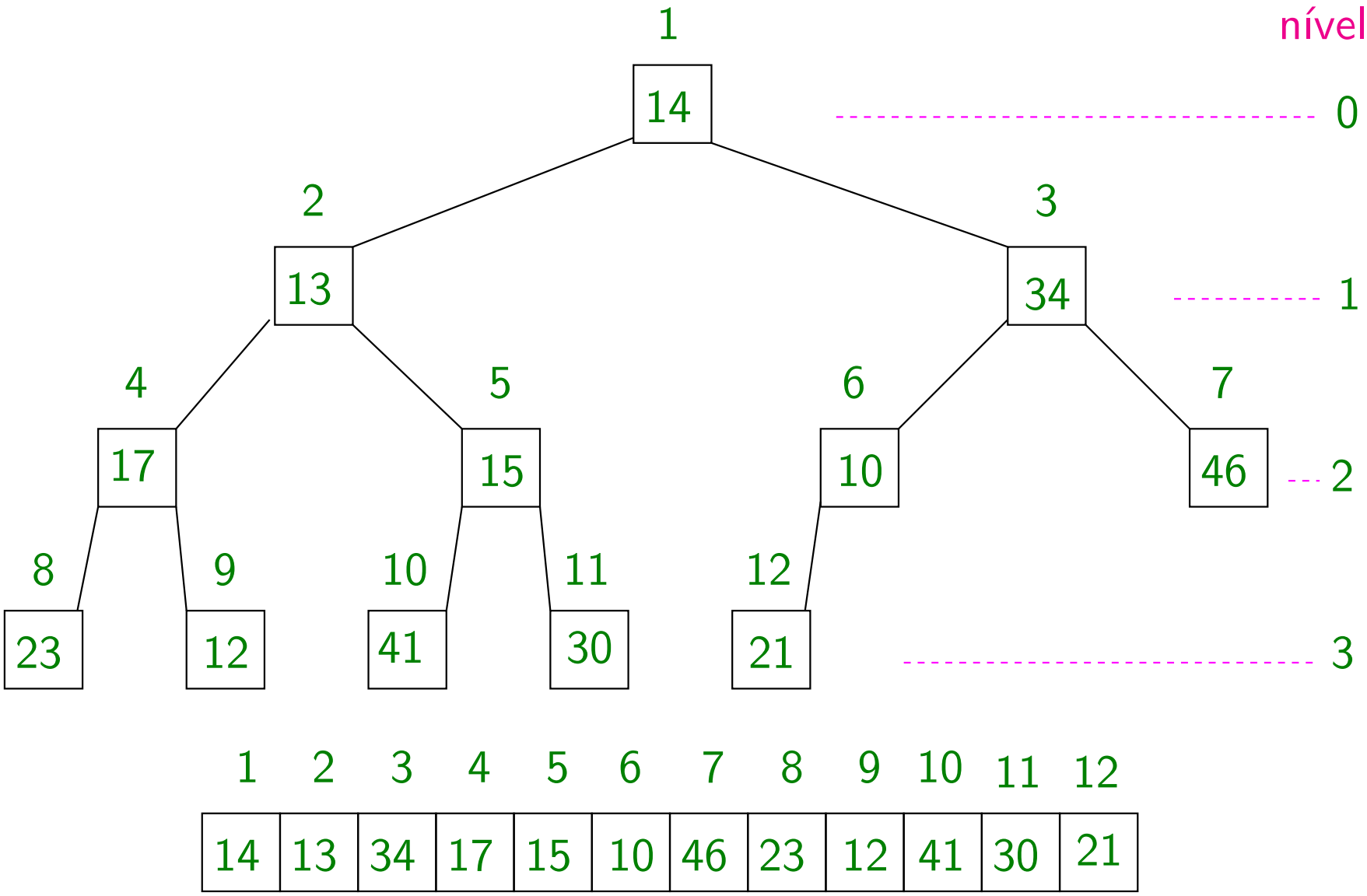
$T(h) \leq T(h-1) + \Theta(1)$

Solução assintótica: $T(n)$ é $O(h)$.

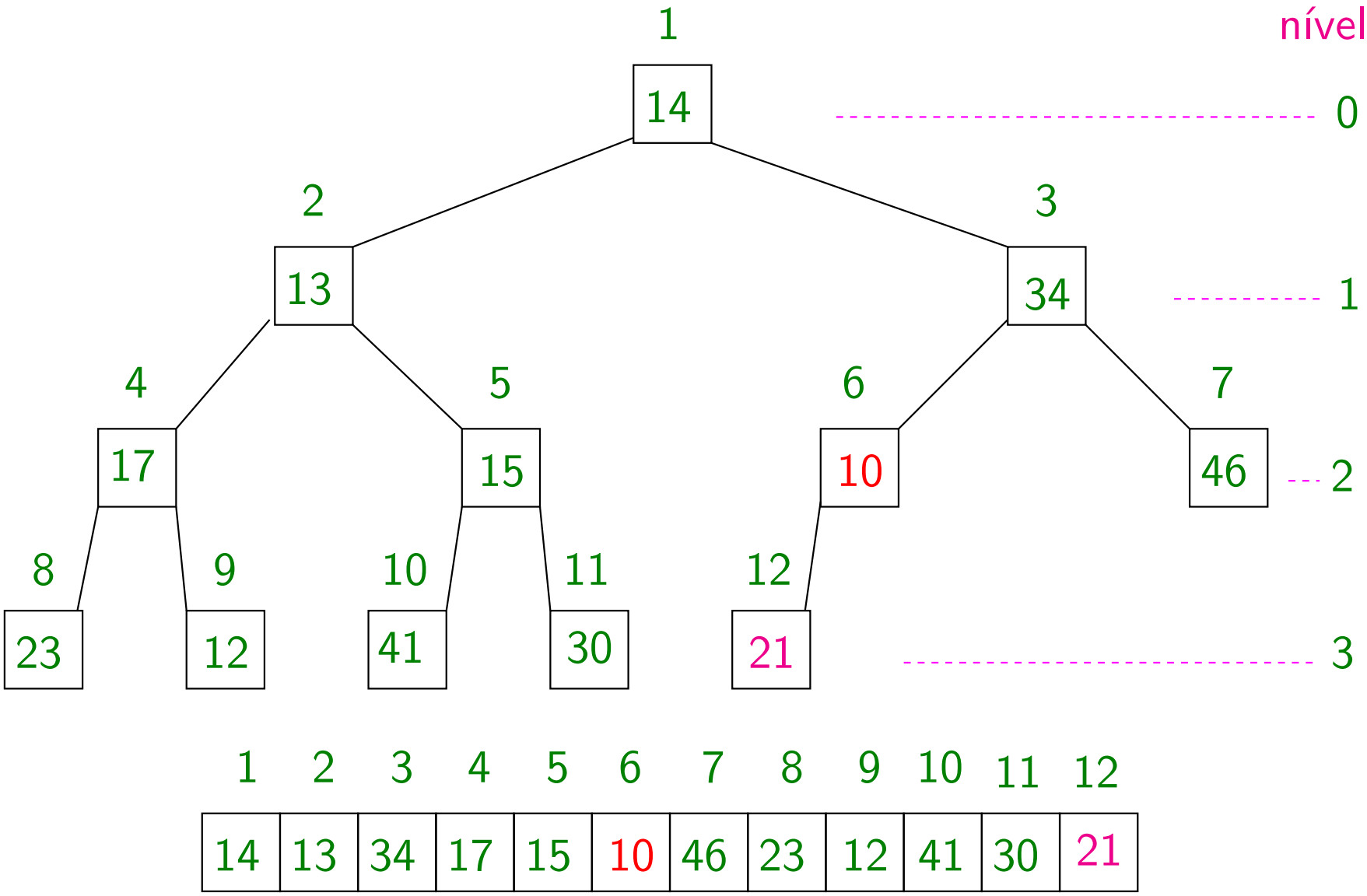
Como $h \leq \lg n$, podemos dizer que:

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

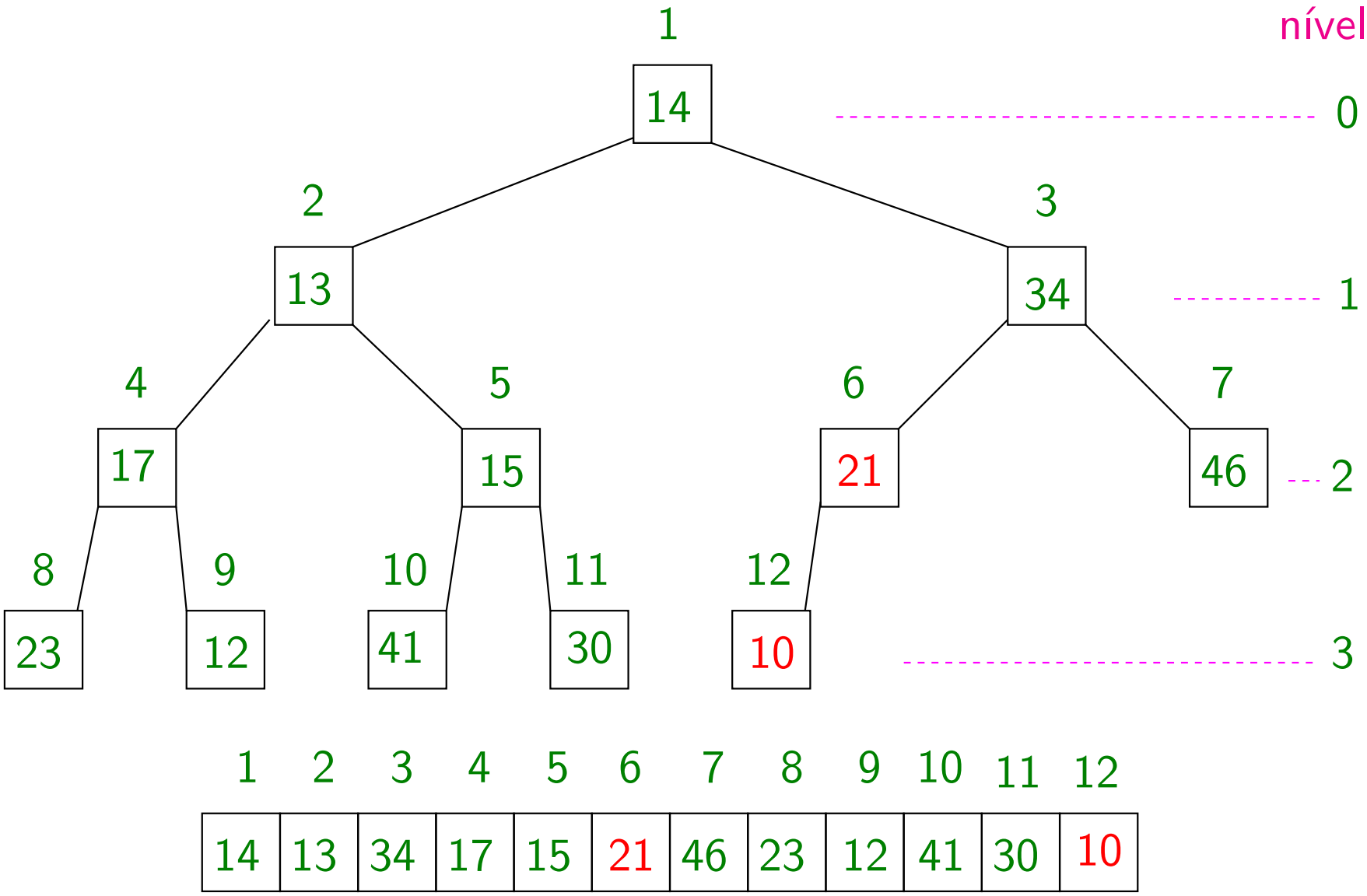
Construção de um max-heap



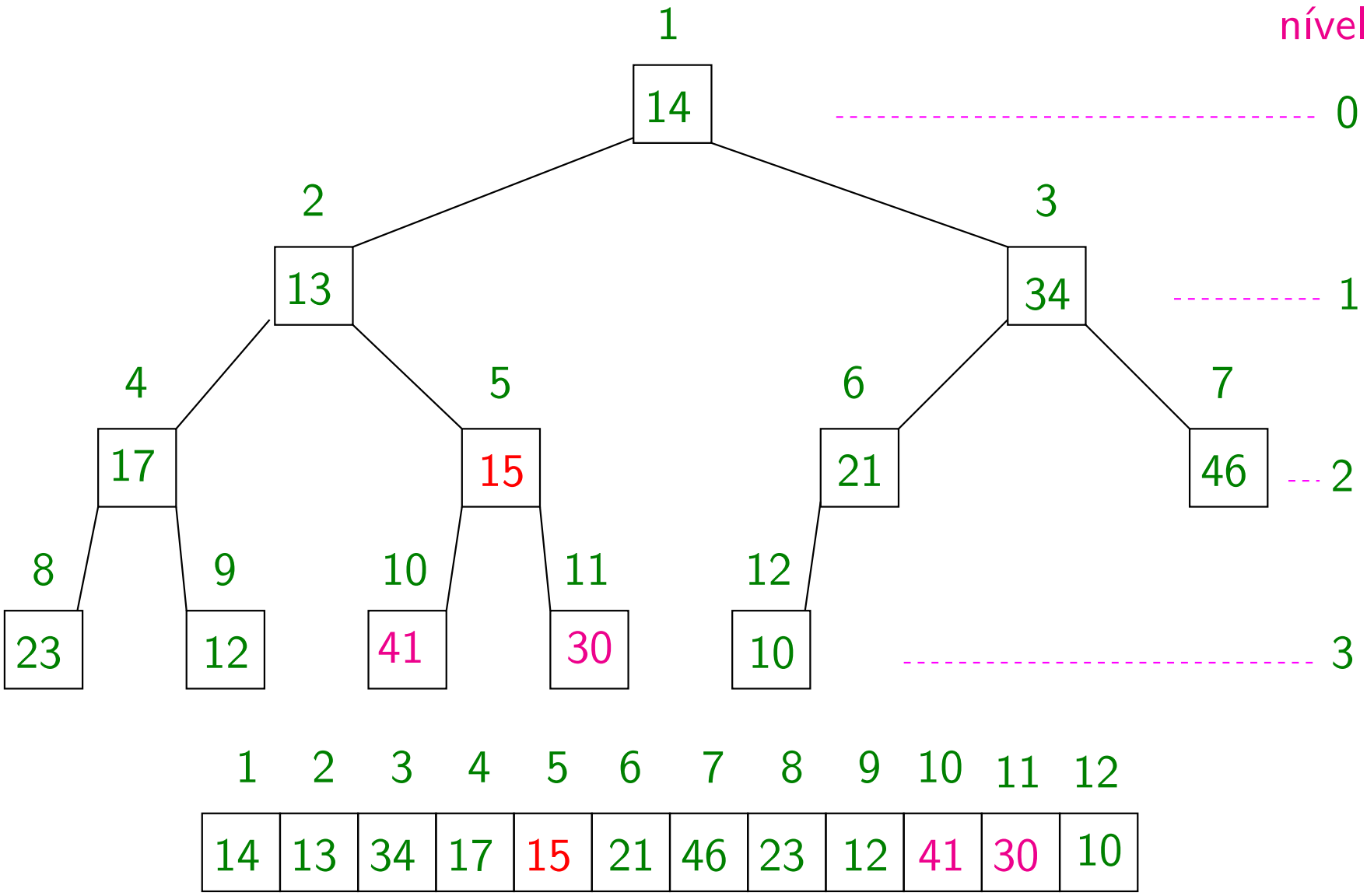
Construção de um max-heap



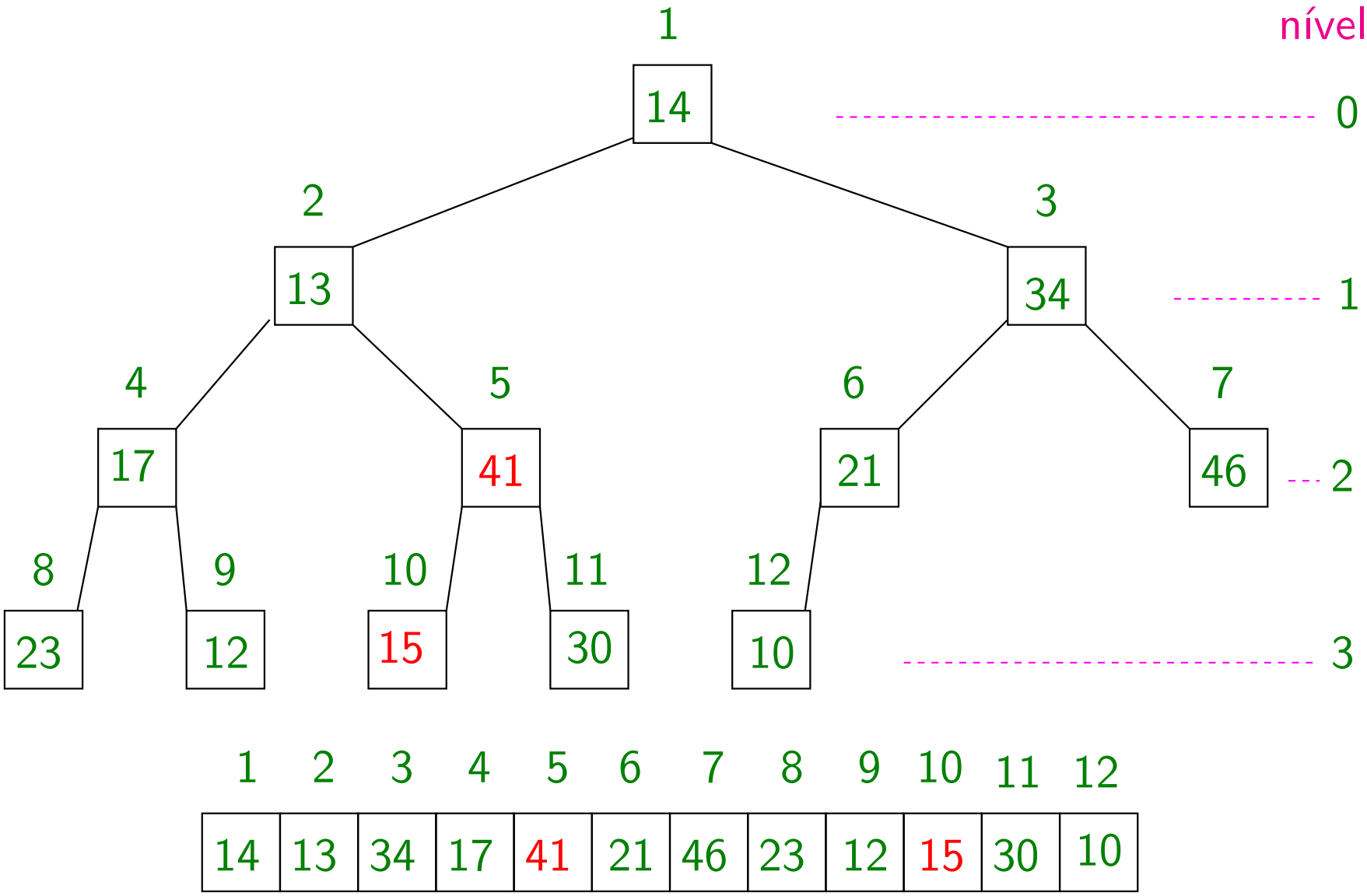
Construção de um max-heap



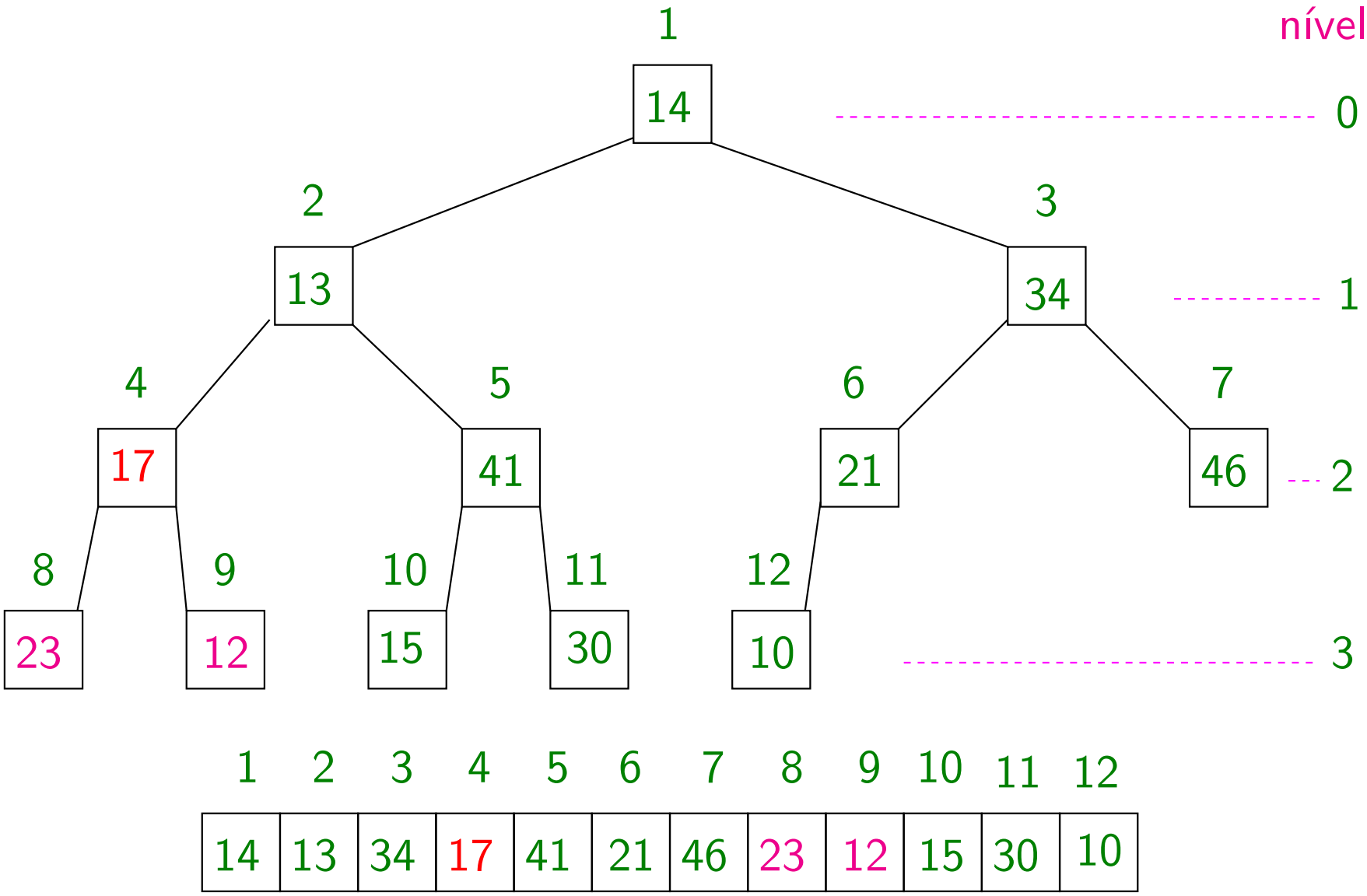
Construção de um max-heap



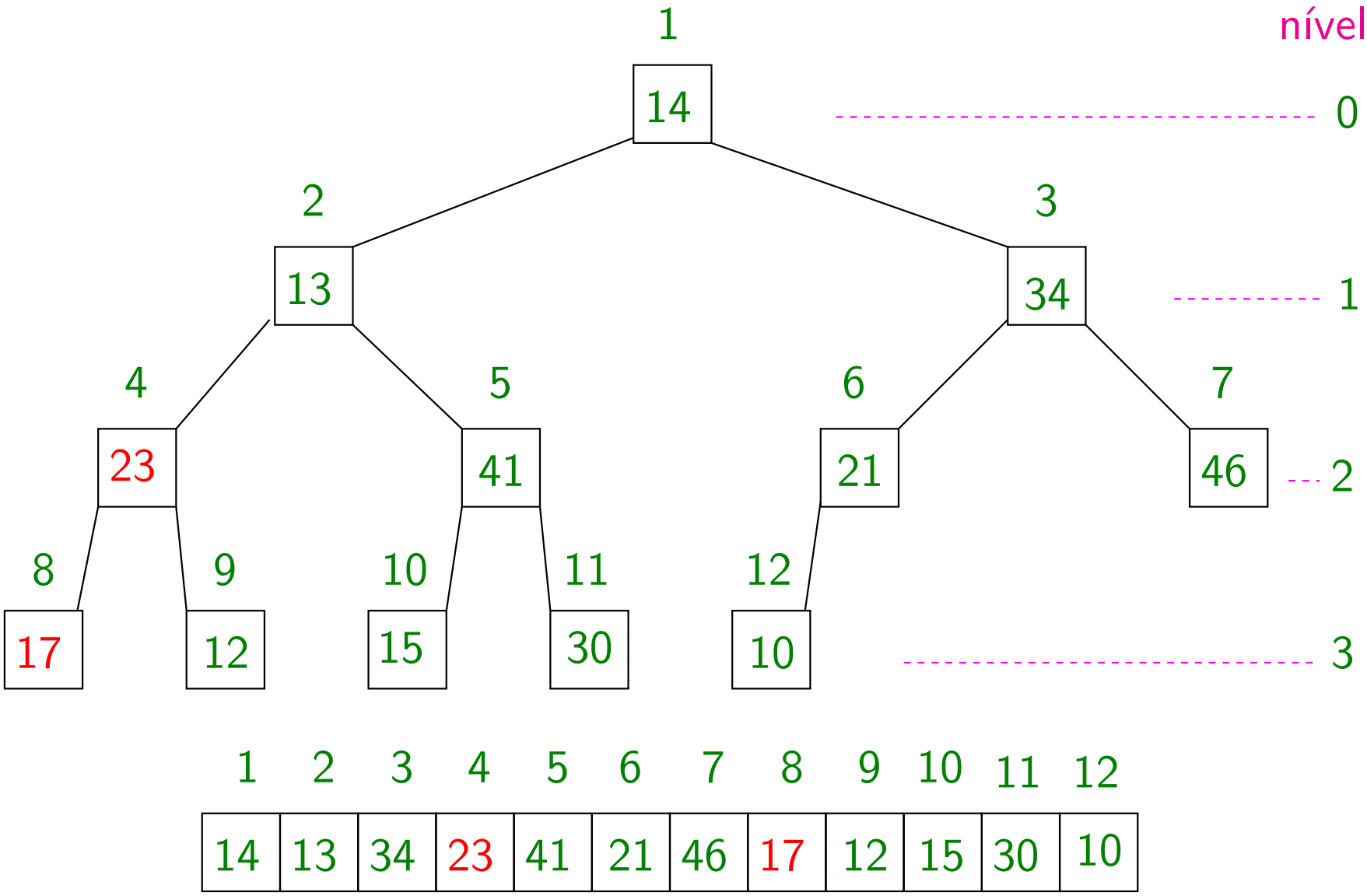
Construção de um max-heap



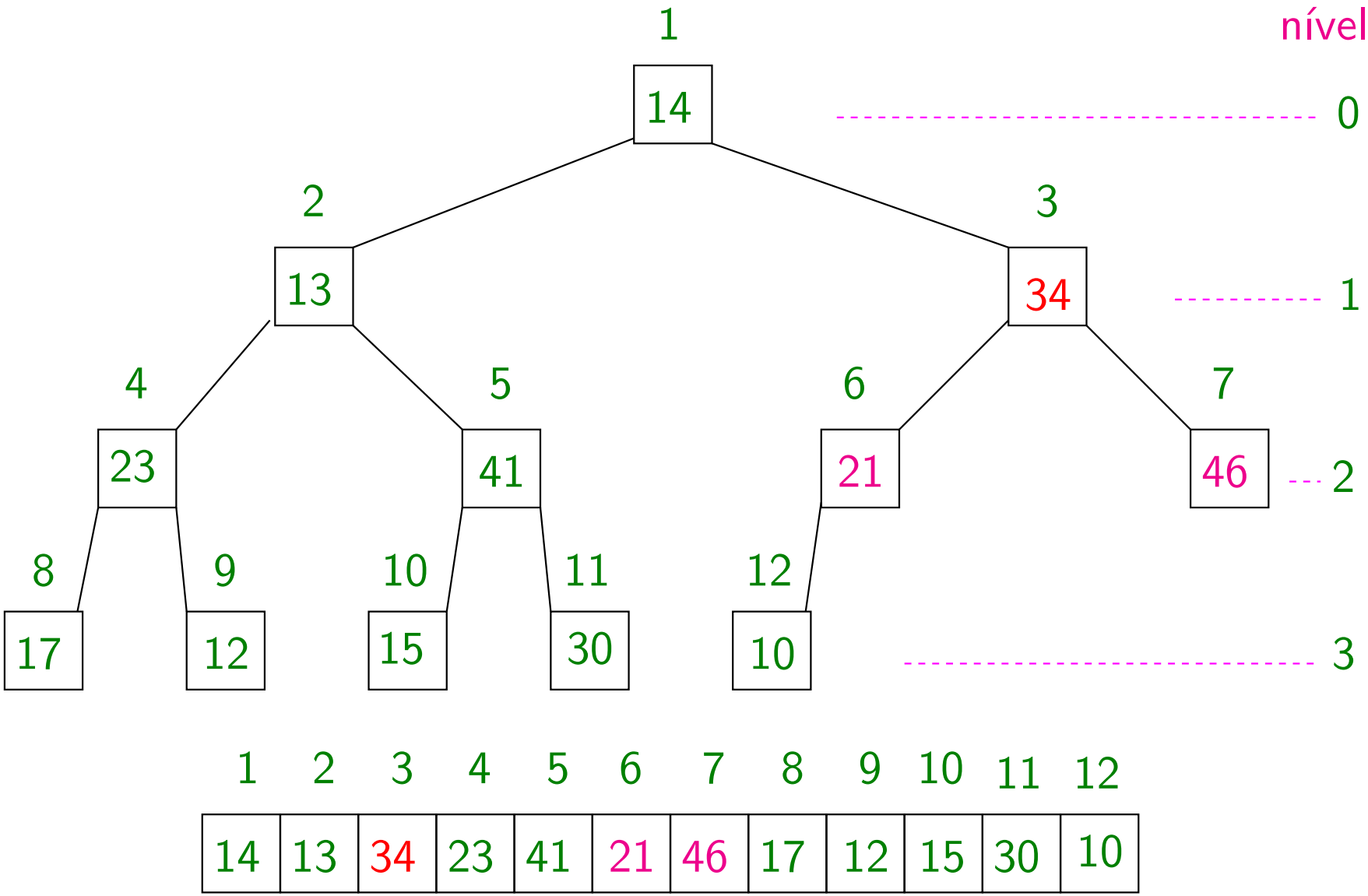
Construção de um max-heap



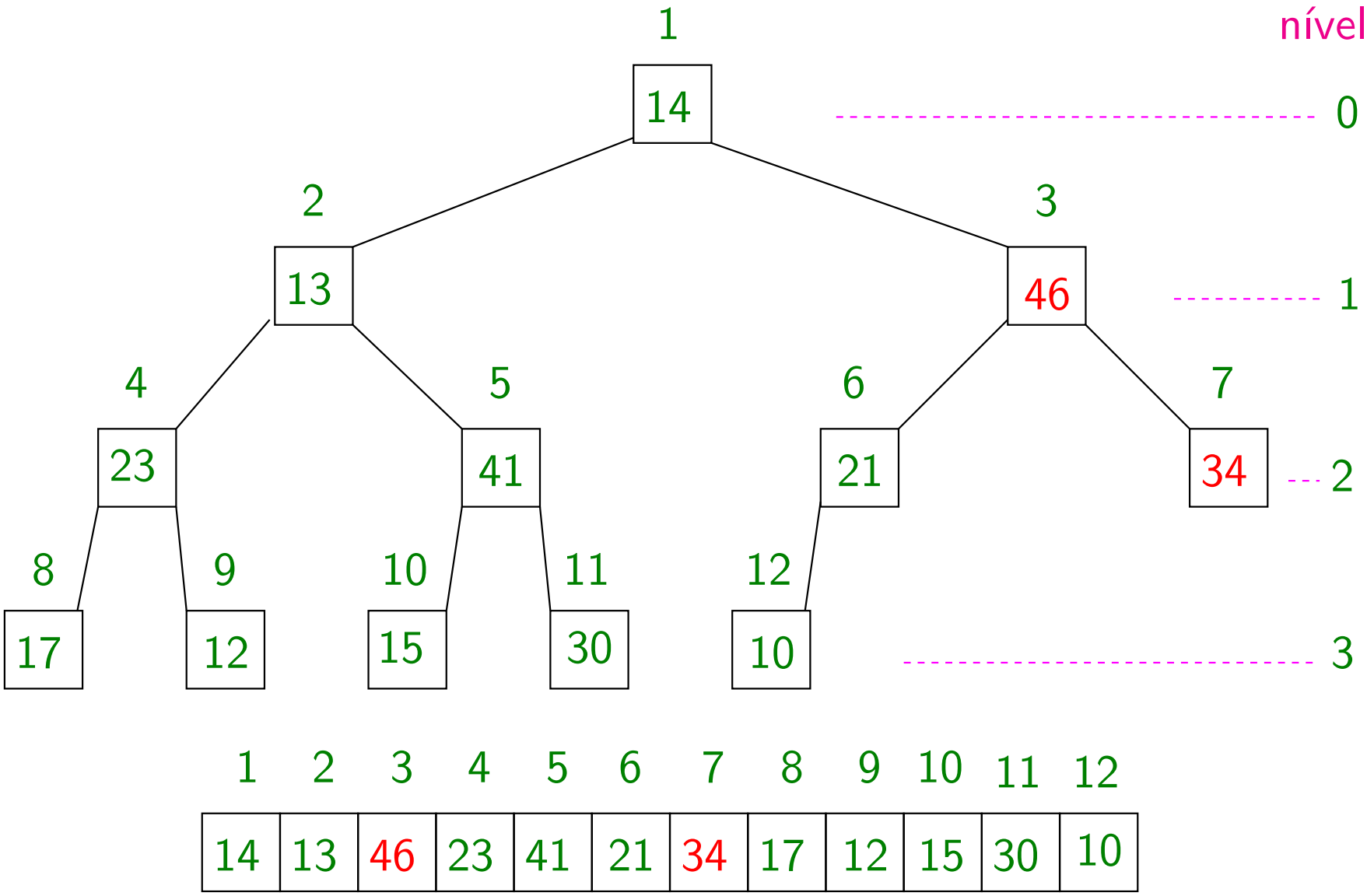
Construção de um max-heap



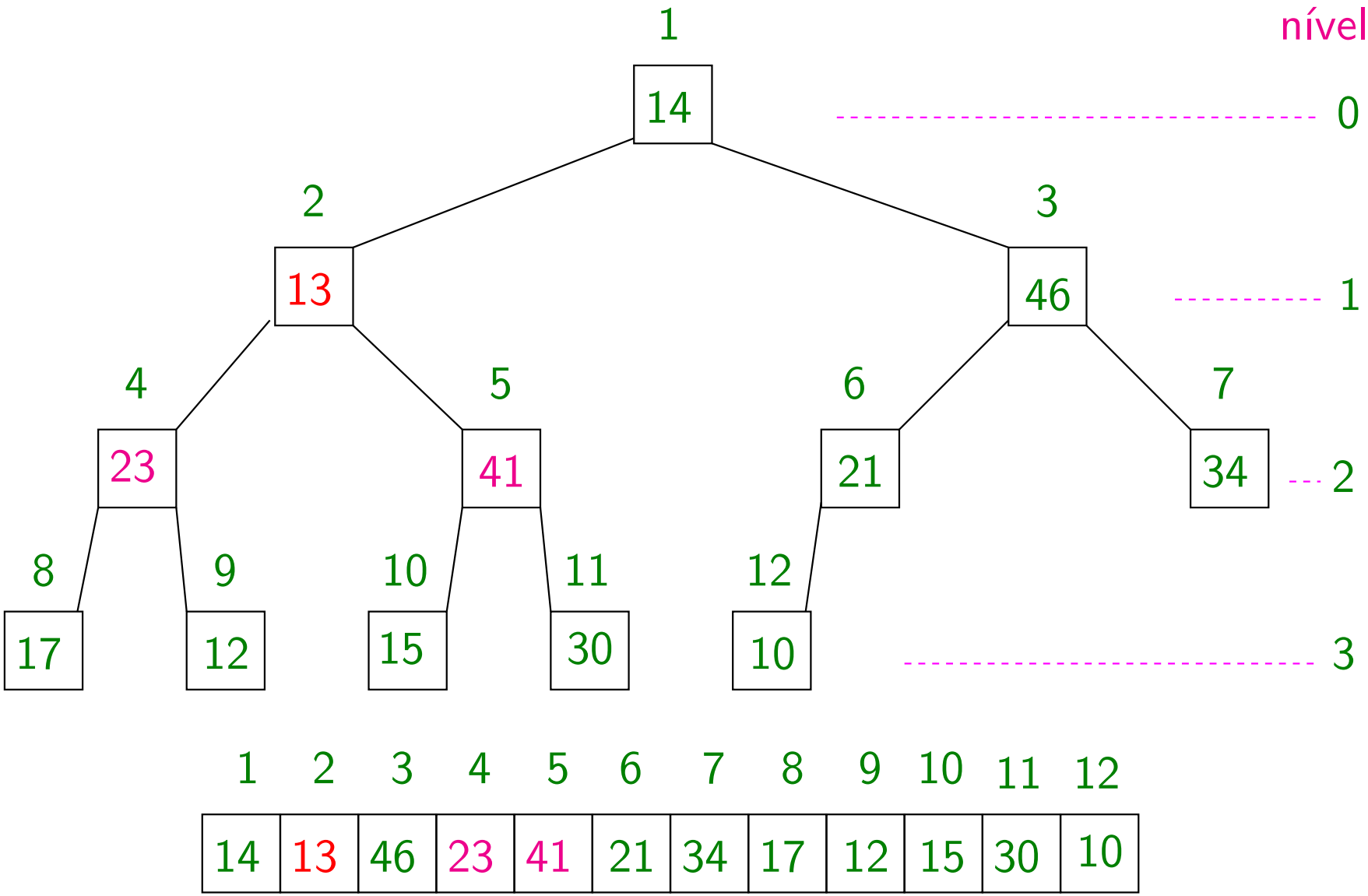
Construção de um max-heap



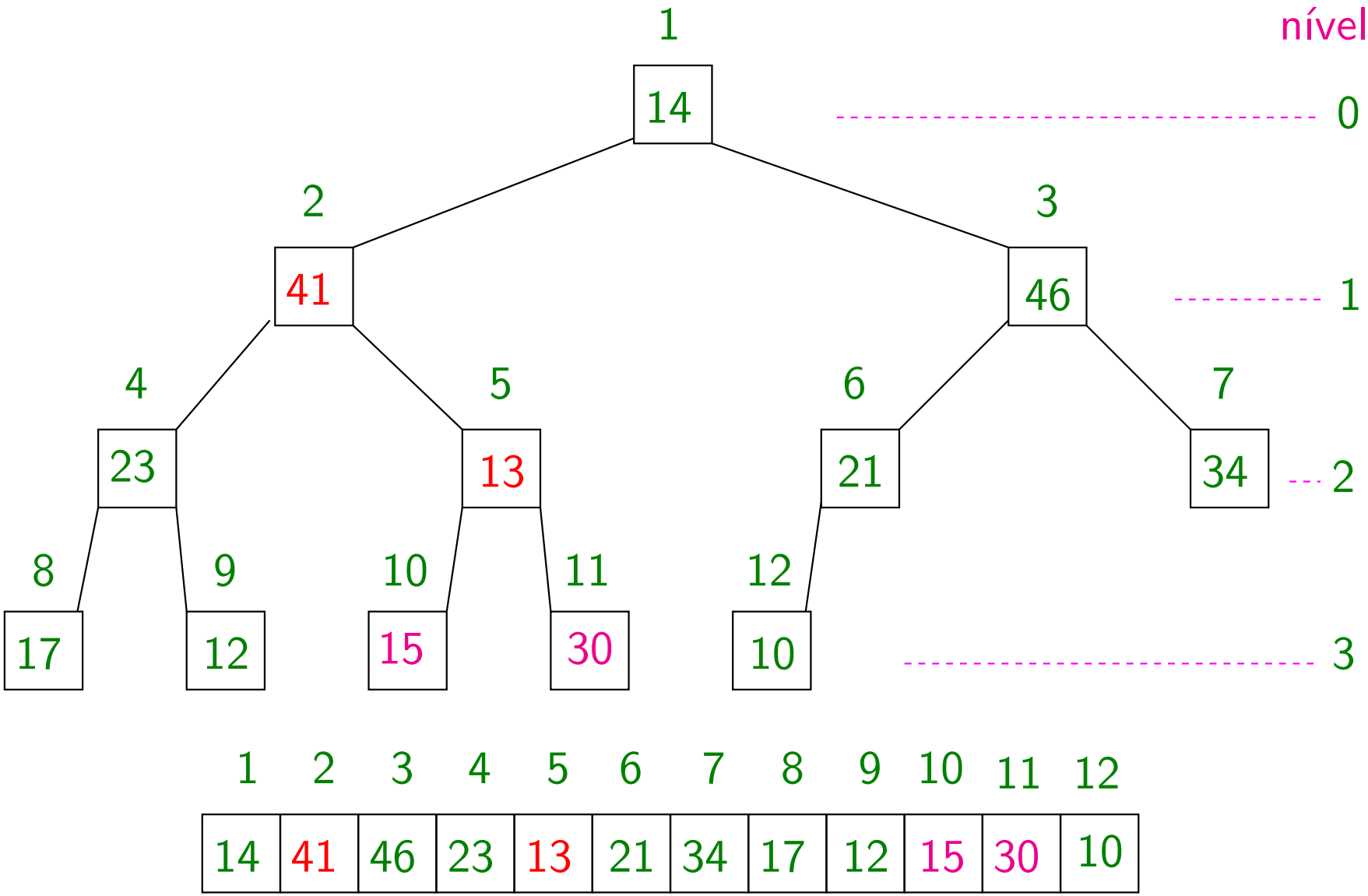
Construção de um max-heap



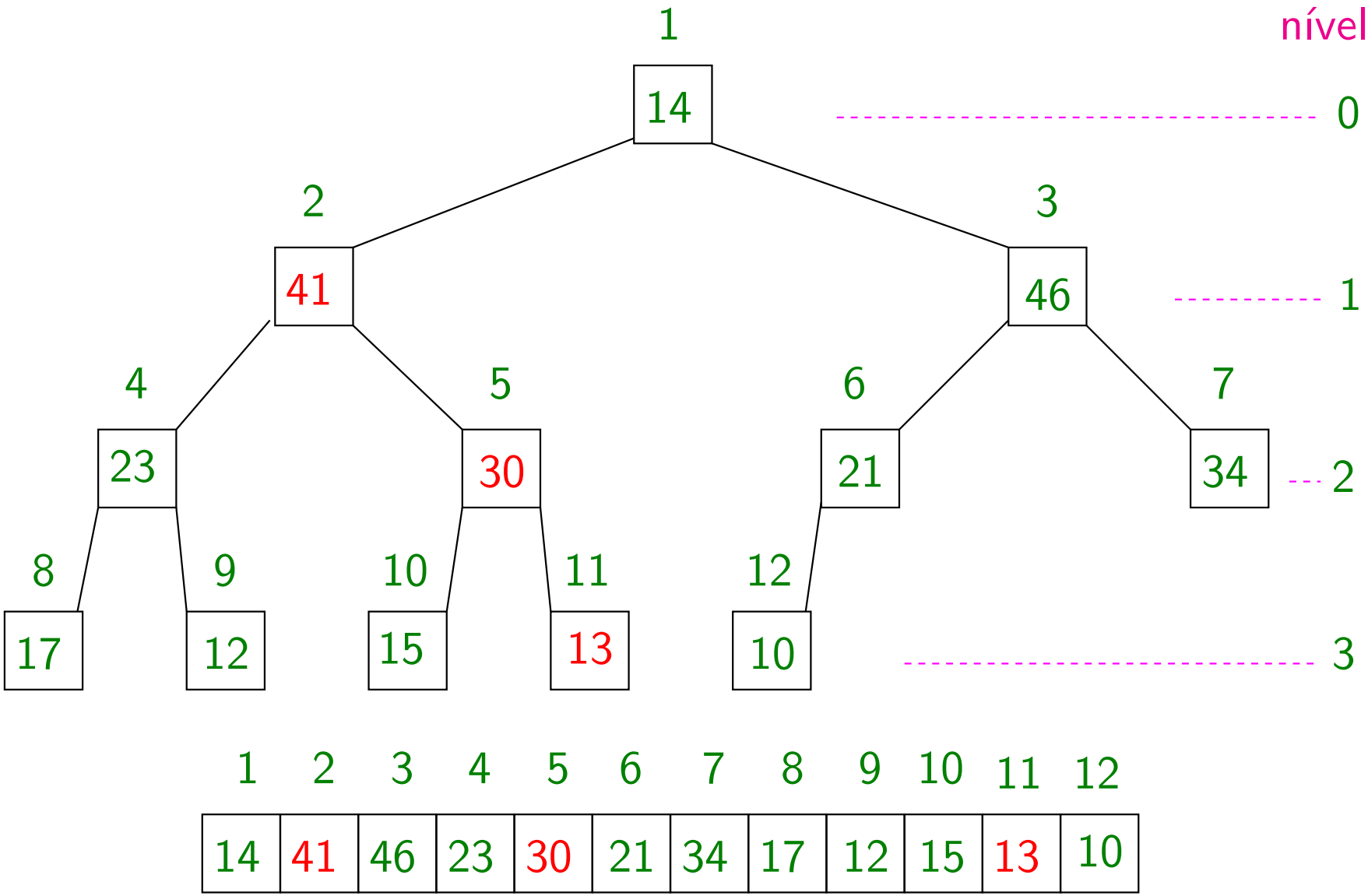
Construção de um max-heap



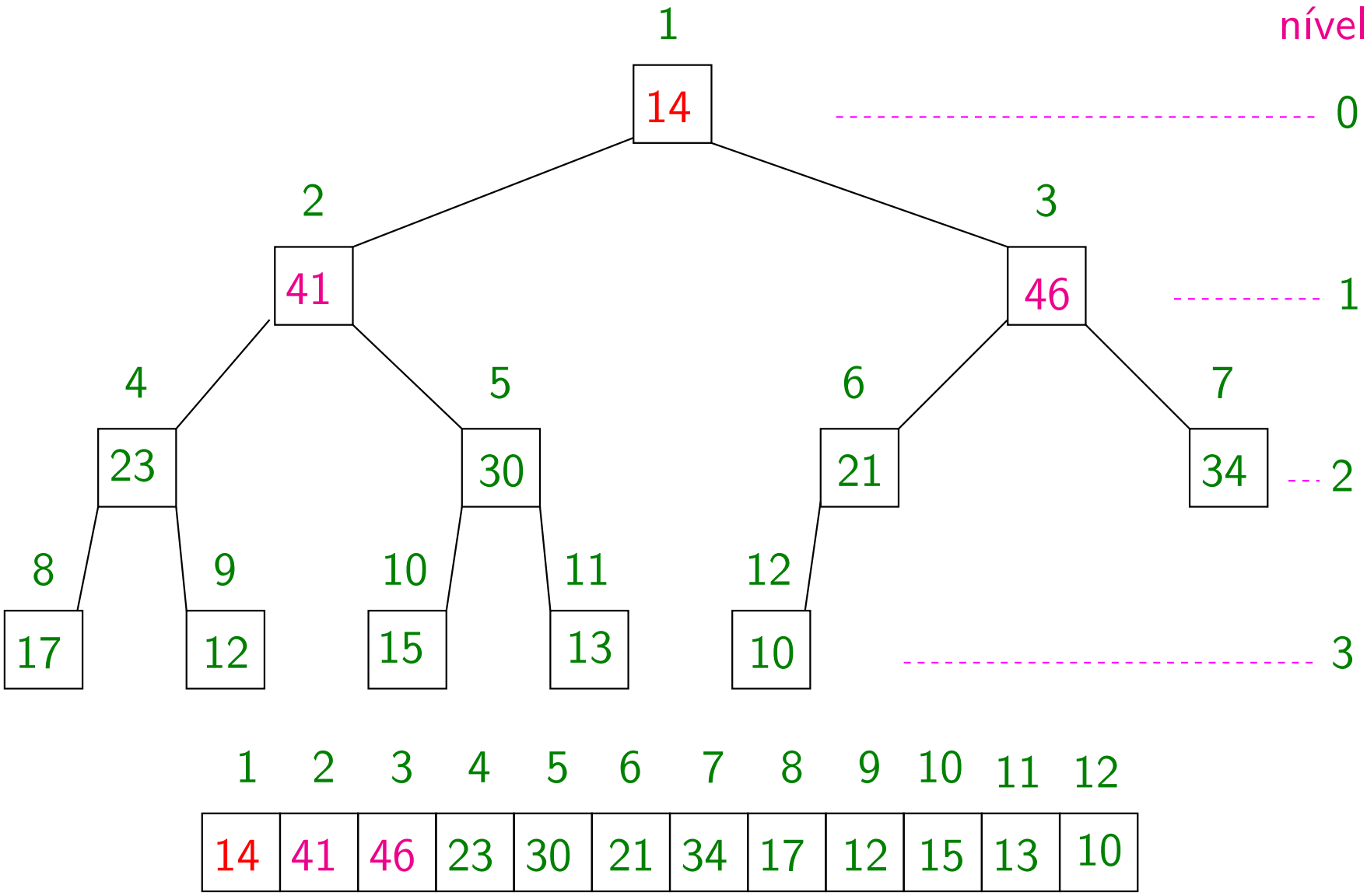
Construção de um max-heap



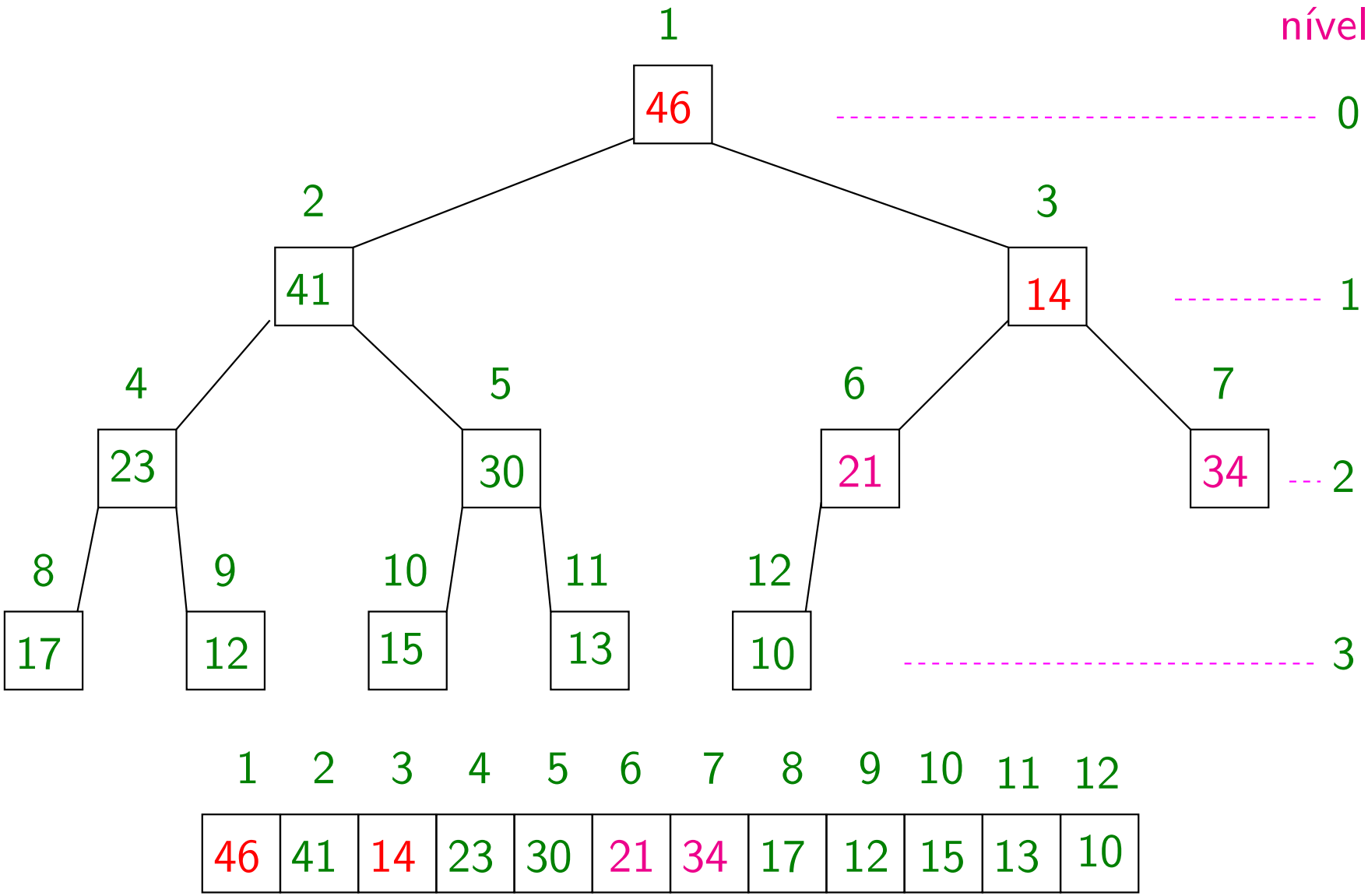
Construção de um max-heap



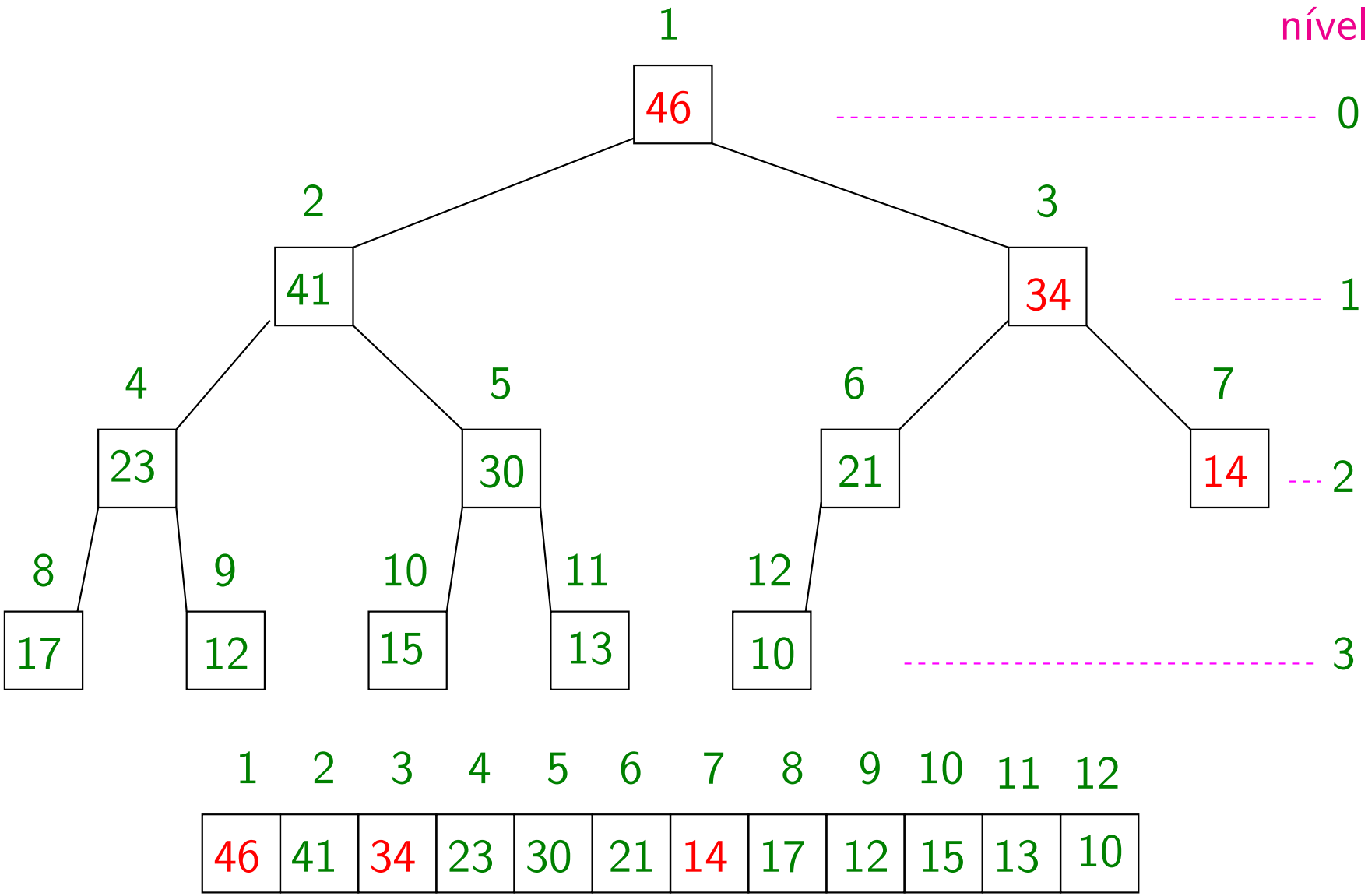
Construção de um max-heap



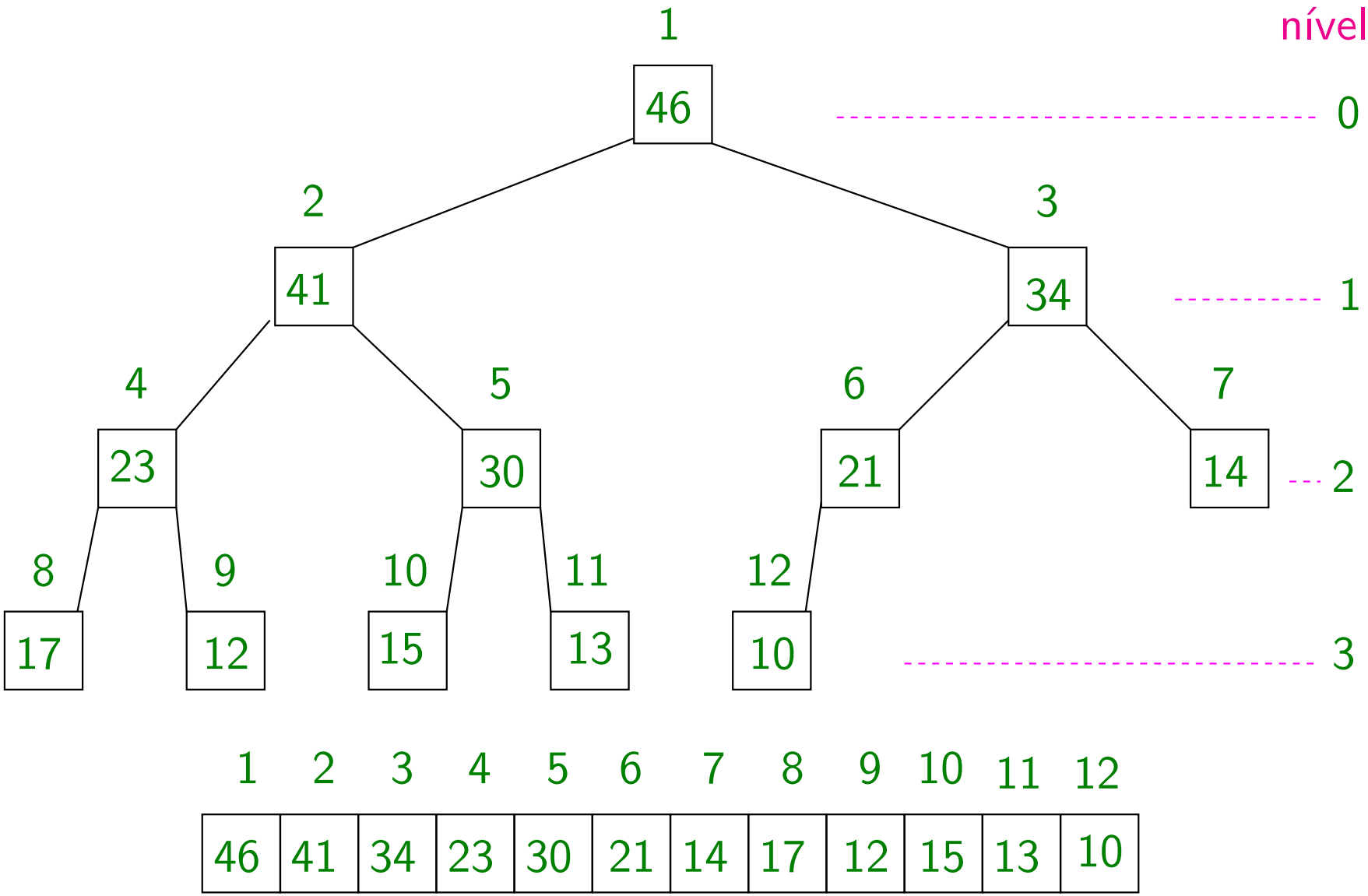
Construção de um max-heap



Construção de um max-heap



Construção de um max-heap



Construção de um max-heap

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

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

Construção de um max-heap

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

BUILD-MAX-HEAP(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.

Construção de um max-heap

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

BUILD-MAX-HEAP(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

Construção de um max-heap

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

BUILD-MAX-HEAP(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 $O(n)$.

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.

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. Assim, a complexidade é $\sum_{i=1}^{n} O(h_i) = O(\sum_{i=1}^{n} h_i)$.

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. Assim, a complexidade é $\sum_{i=1}^{n} O(h_i) = O(\sum_{i=1}^{n} h_i)$.
- Seja $S(h)$ a soma das alturas de todos os nós de uma **árvore binária completa** de altura h.

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. Assim, a complexidade é $\sum_{i=1}^{n} O(h_i) = O(\sum_{i=1}^{n} h_i)$.
- Seja $S(h)$ a soma das alturas de todos os nós de uma **árvore binária completa** de altura h.
- A altura de um heap com n elementos é $\lfloor \lg n \rfloor + 1$. Logo, $\sum_{i=1}^{n} h_i \leq S(\lfloor \lg n \rfloor + 1)$.

  A complexidade de BUILD-MAX-HEAP é $T(n) = O(S(\lg n))$.

Construção de um max-heap

Construção de um max-heap

- Pode-se provar por indução que $S(h) = 2^{h+1} - h - 2$. (Exercício!)

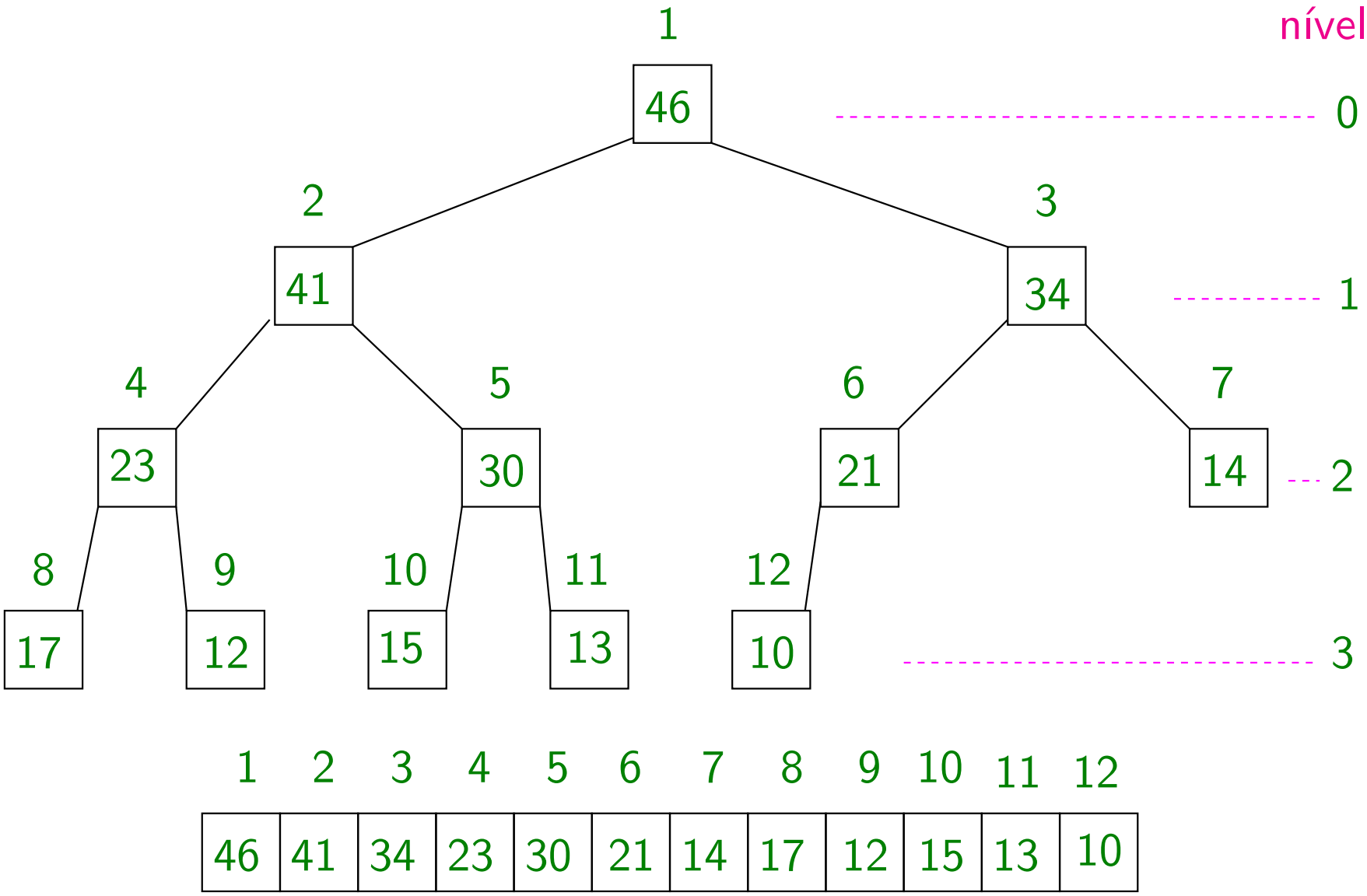
Construção de um max-heap

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

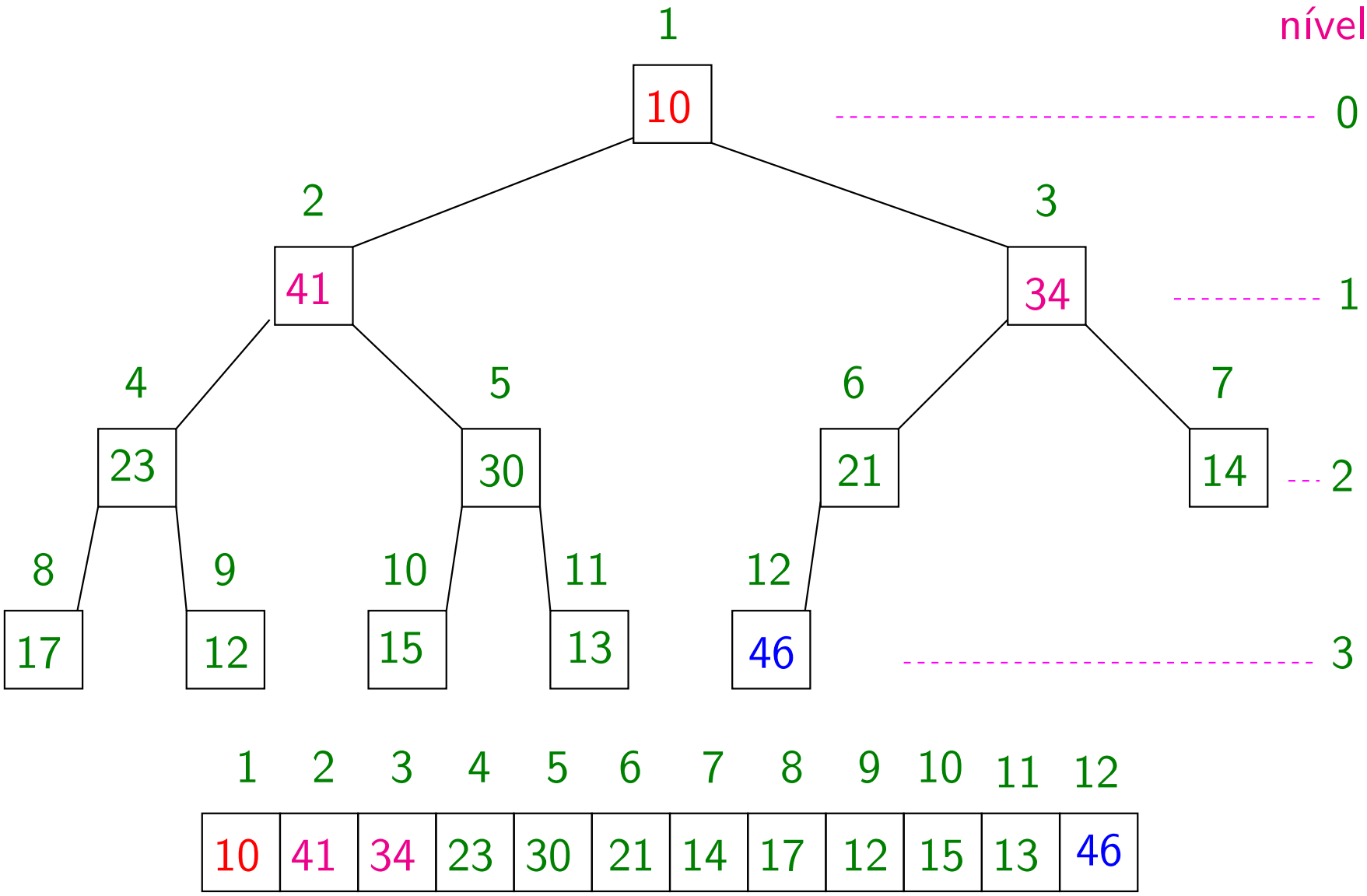
Construção de um max-heap

- Pode-se provar por indução que $S(h) = 2^{h+1} - h - 2$. (Exercício!)
- Logo, a complexidade de BUILD-MAX-HEAP é $T(n) = O(S(\lg n)) = O(n)$.
- Veja no CLRS uma prova diferente deste fato.

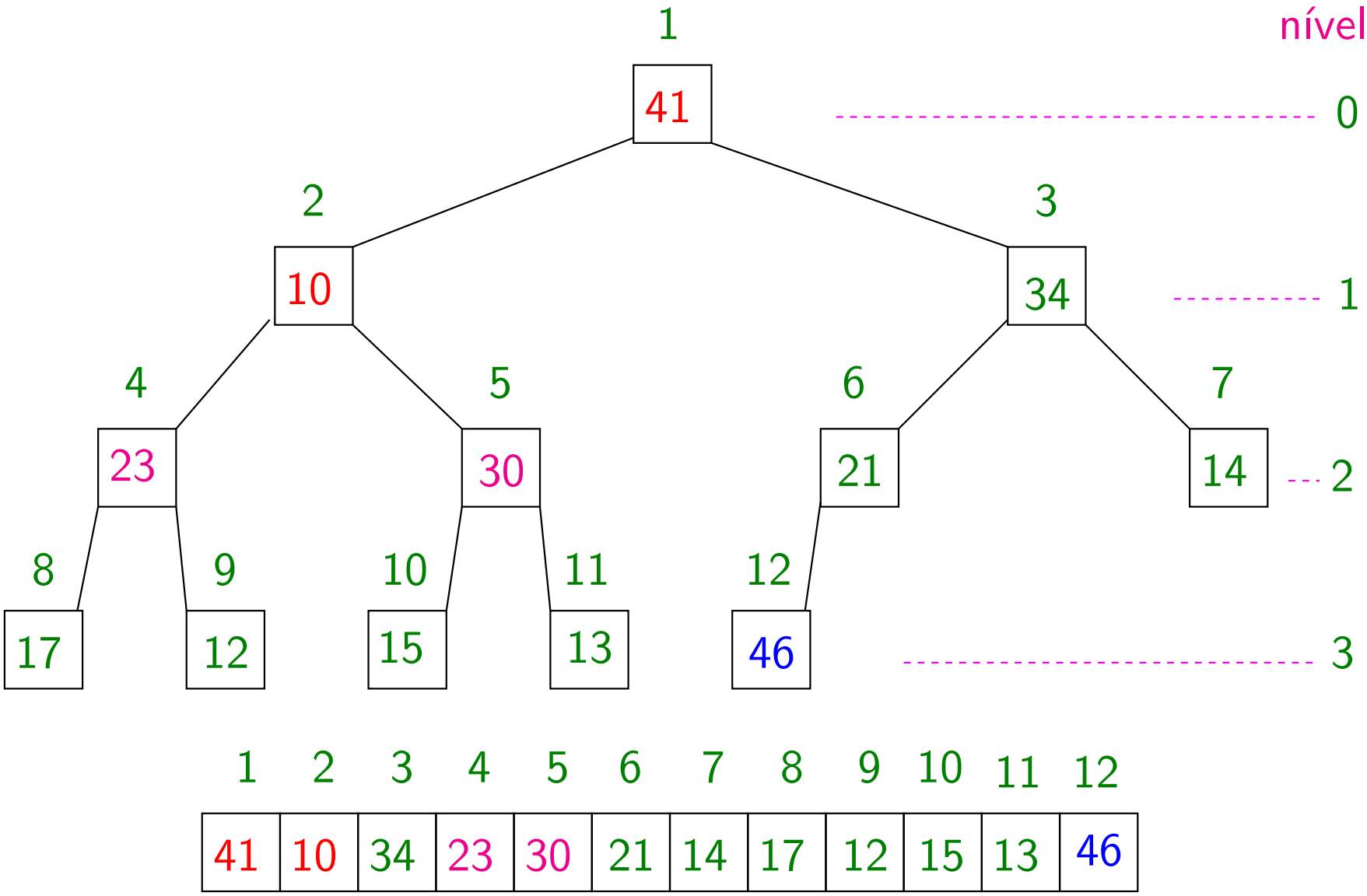
HeapSort


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

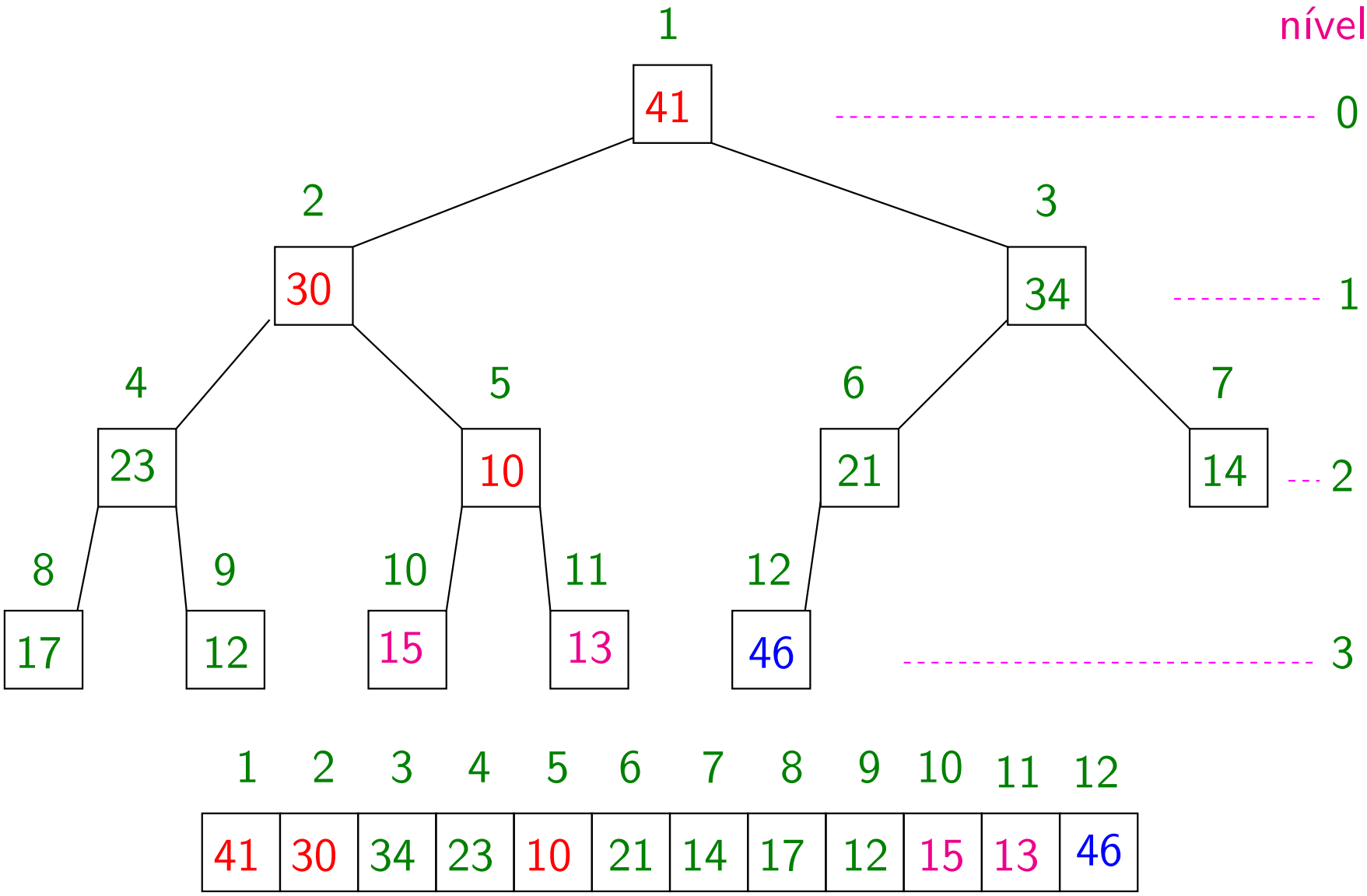
HeapSort


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

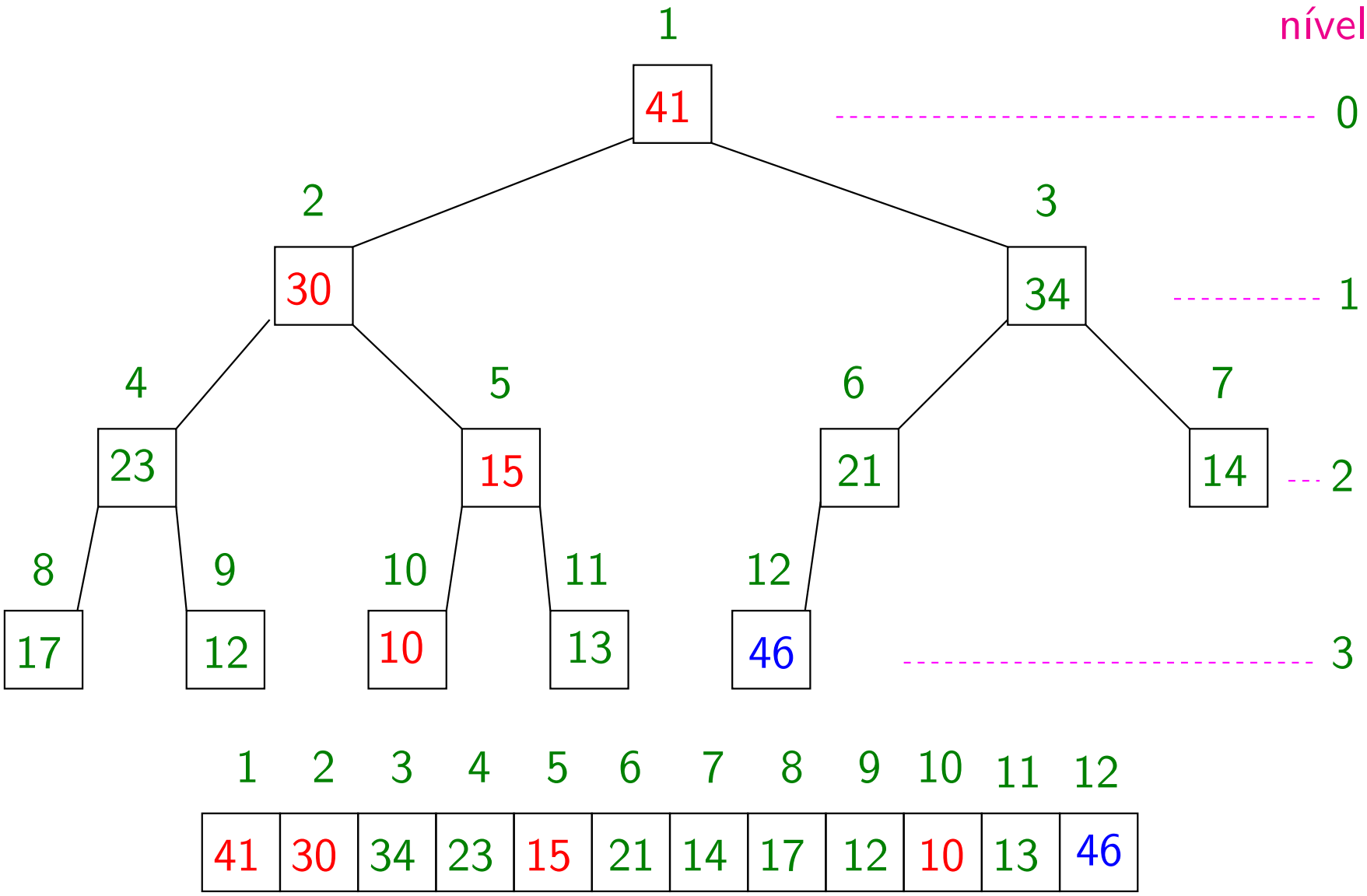
HeapSort


1
nível
41
0
2
3
10
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
41 10 34 23 30 21 14 17 12 15 13 46

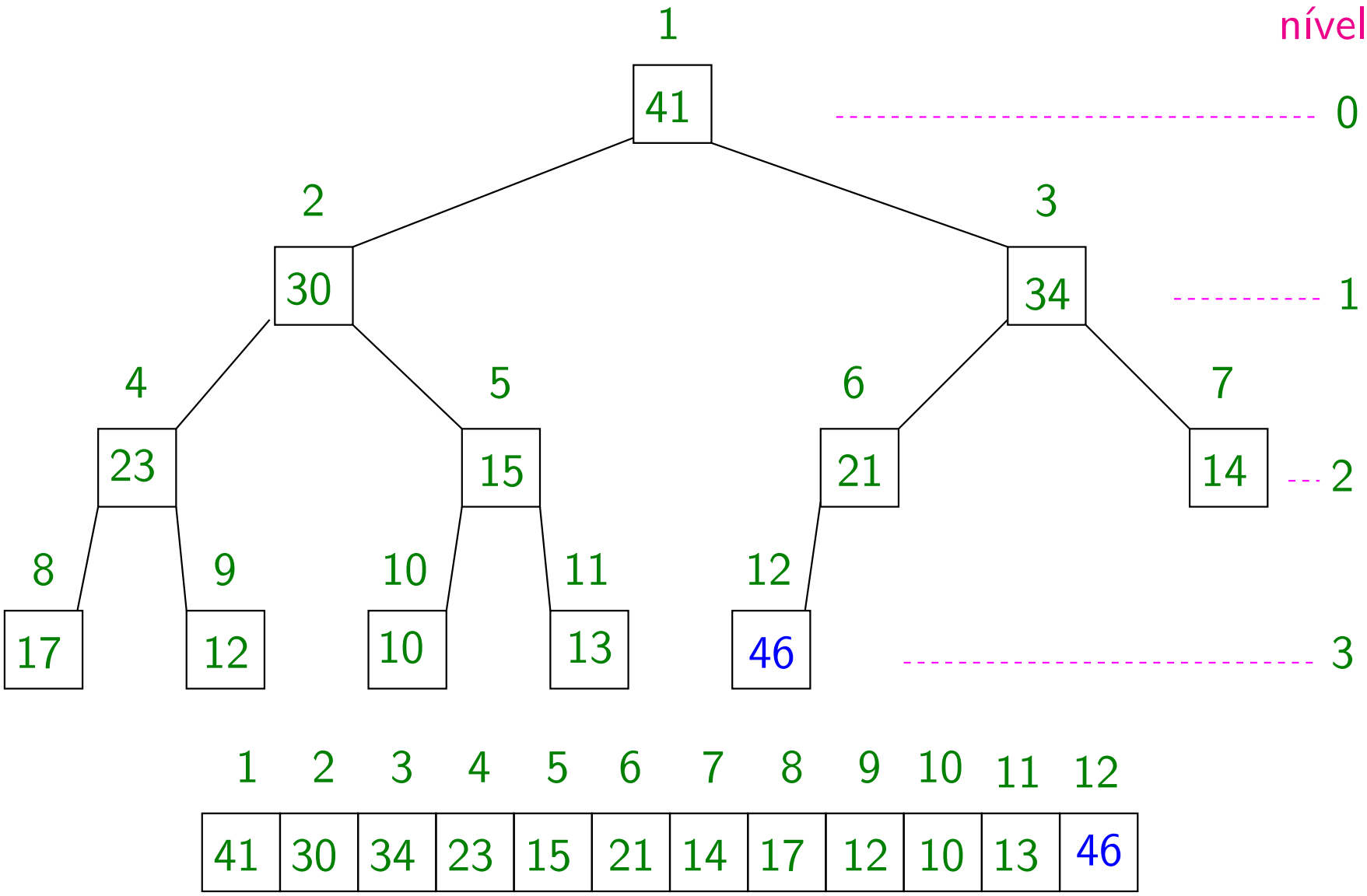
HeapSort


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

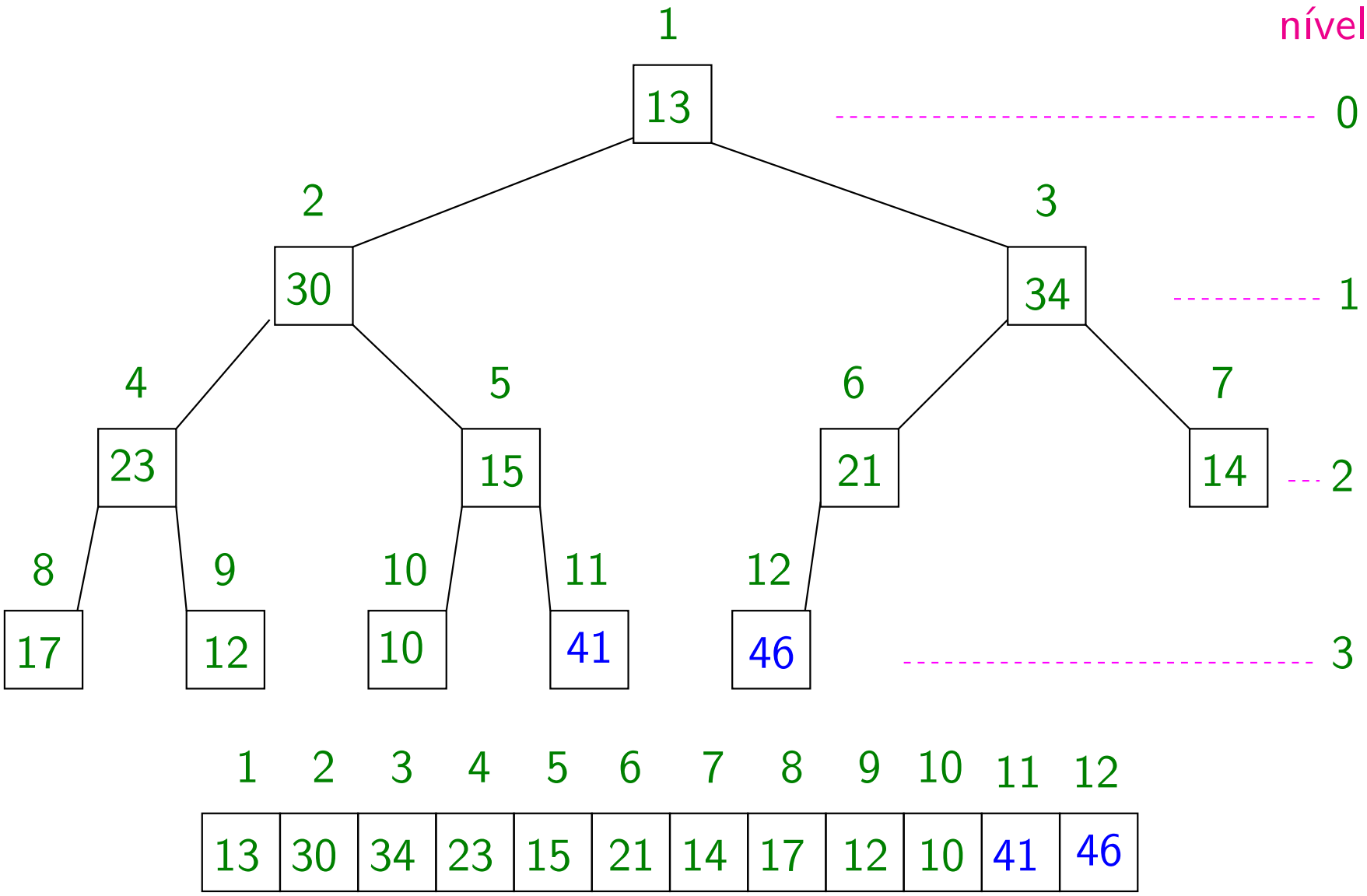
HeapSort


1
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

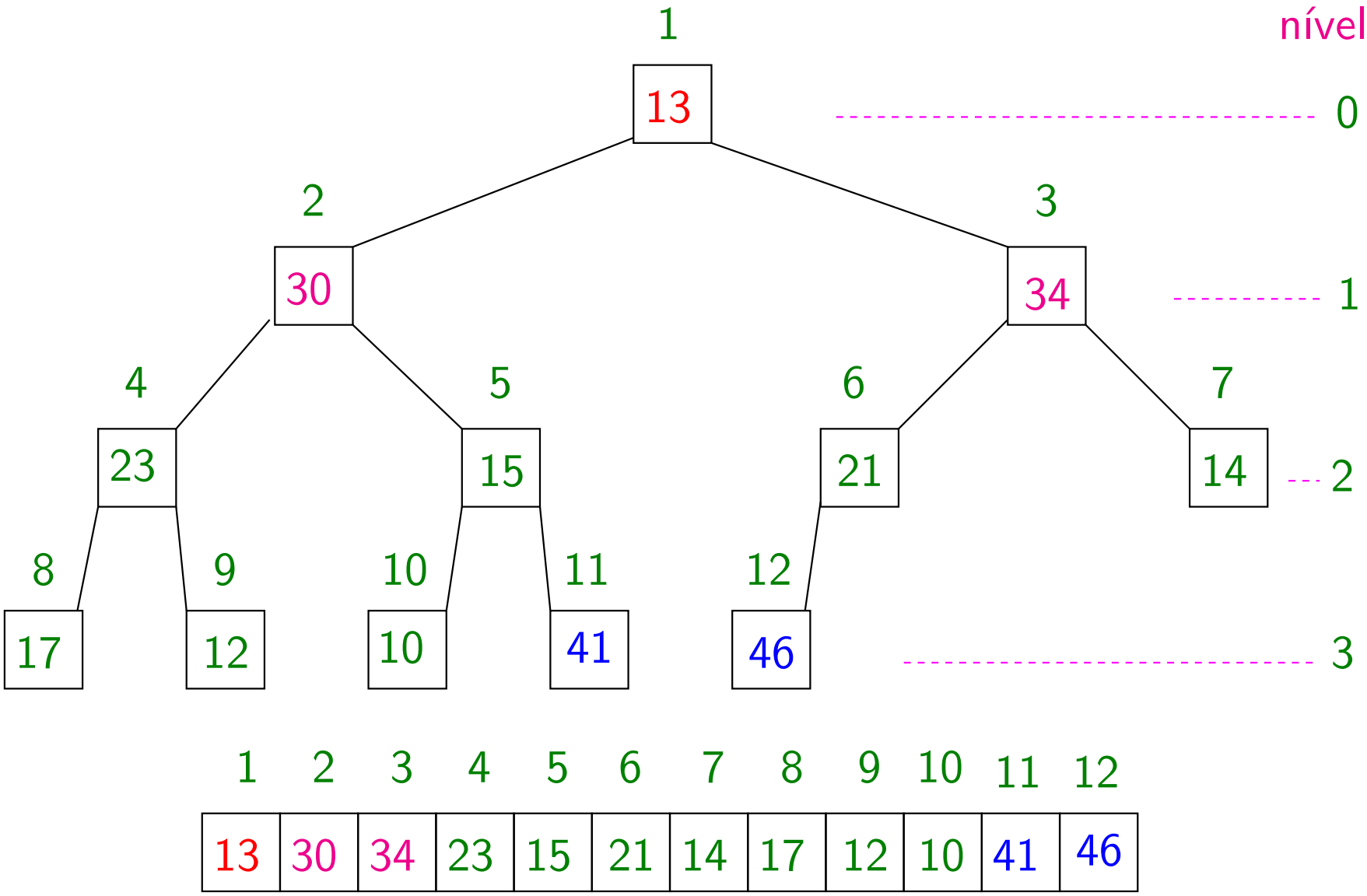
HeapSort


1
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

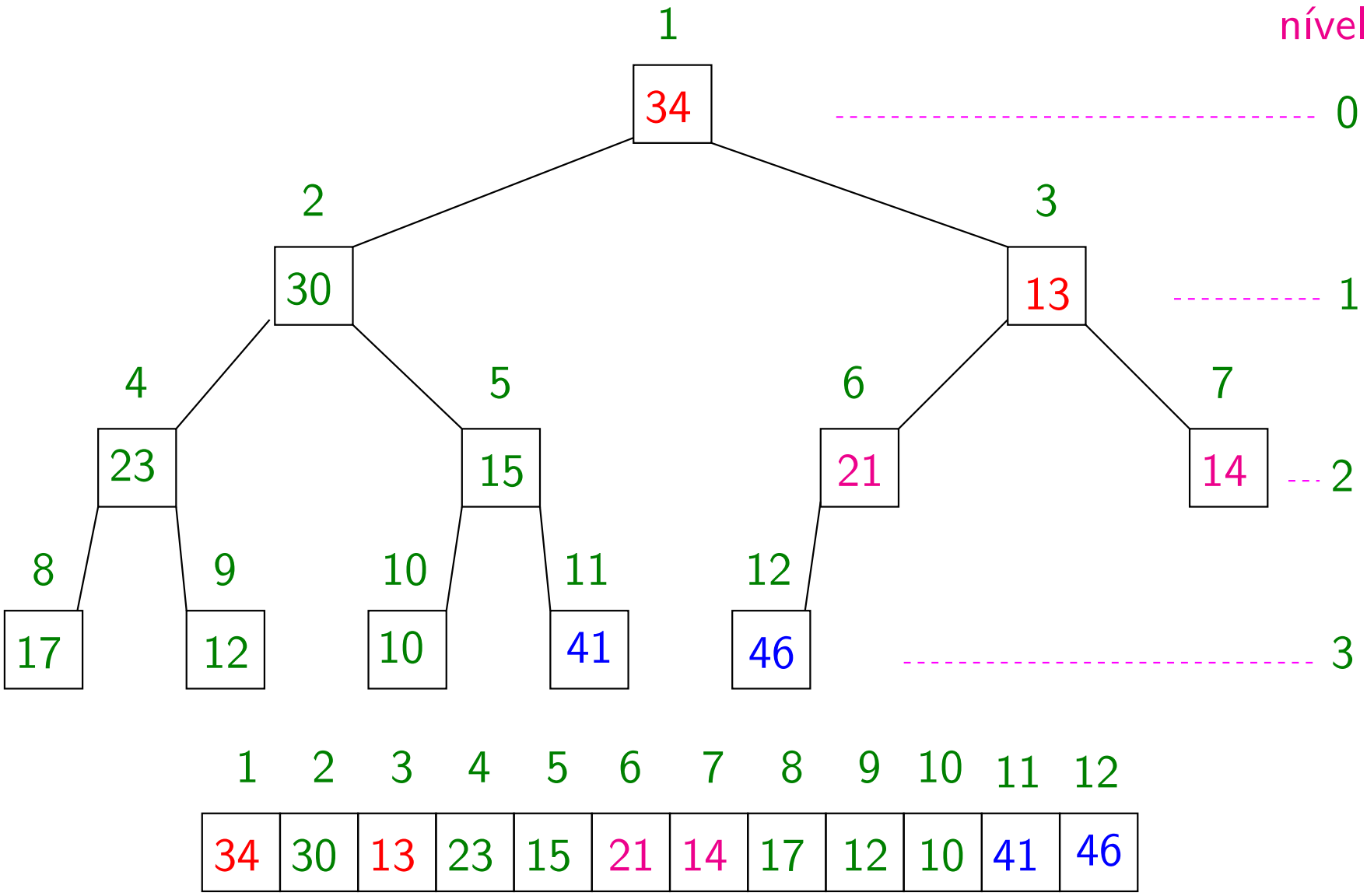
HeapSort


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

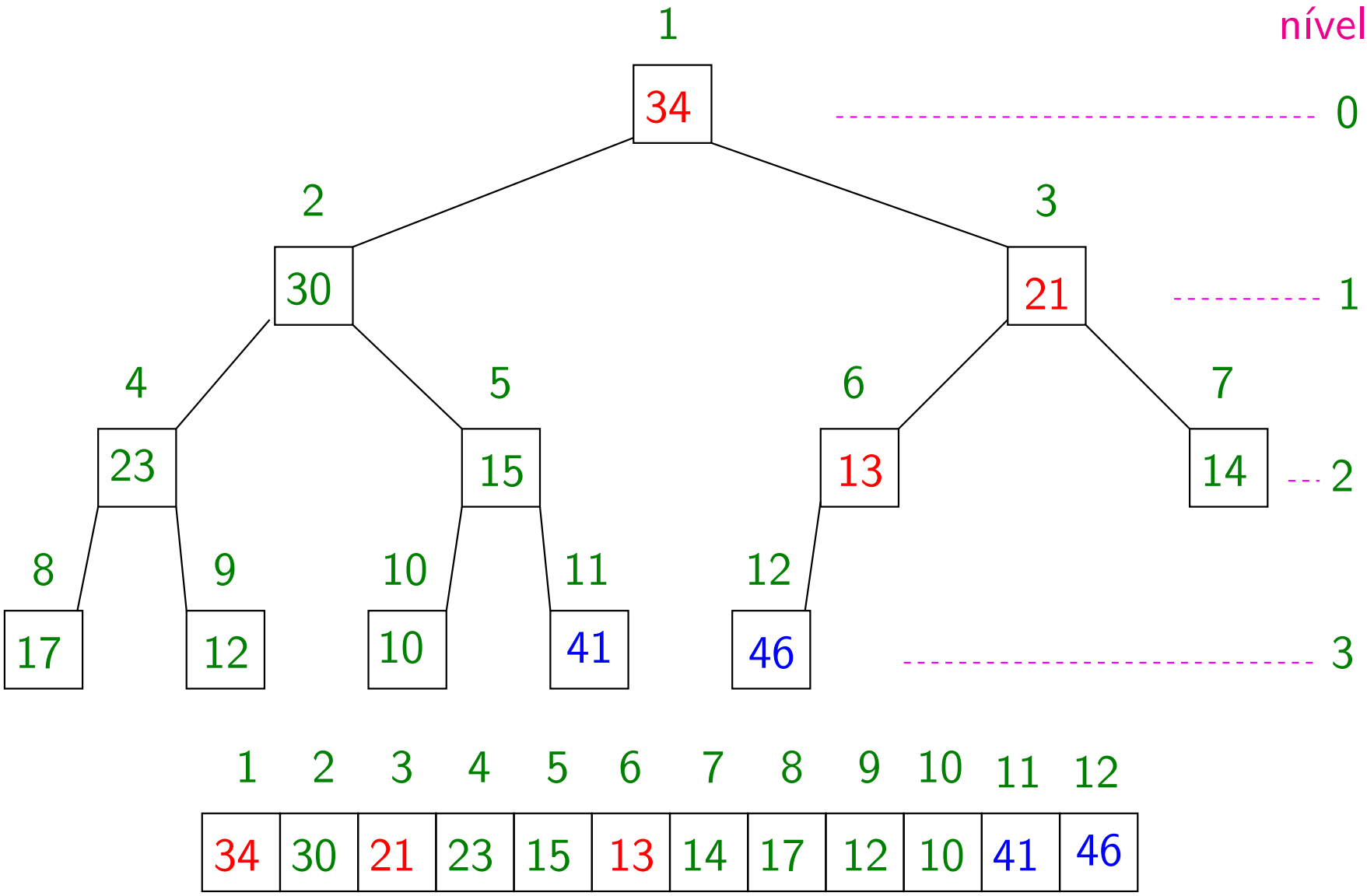
HeapSort


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

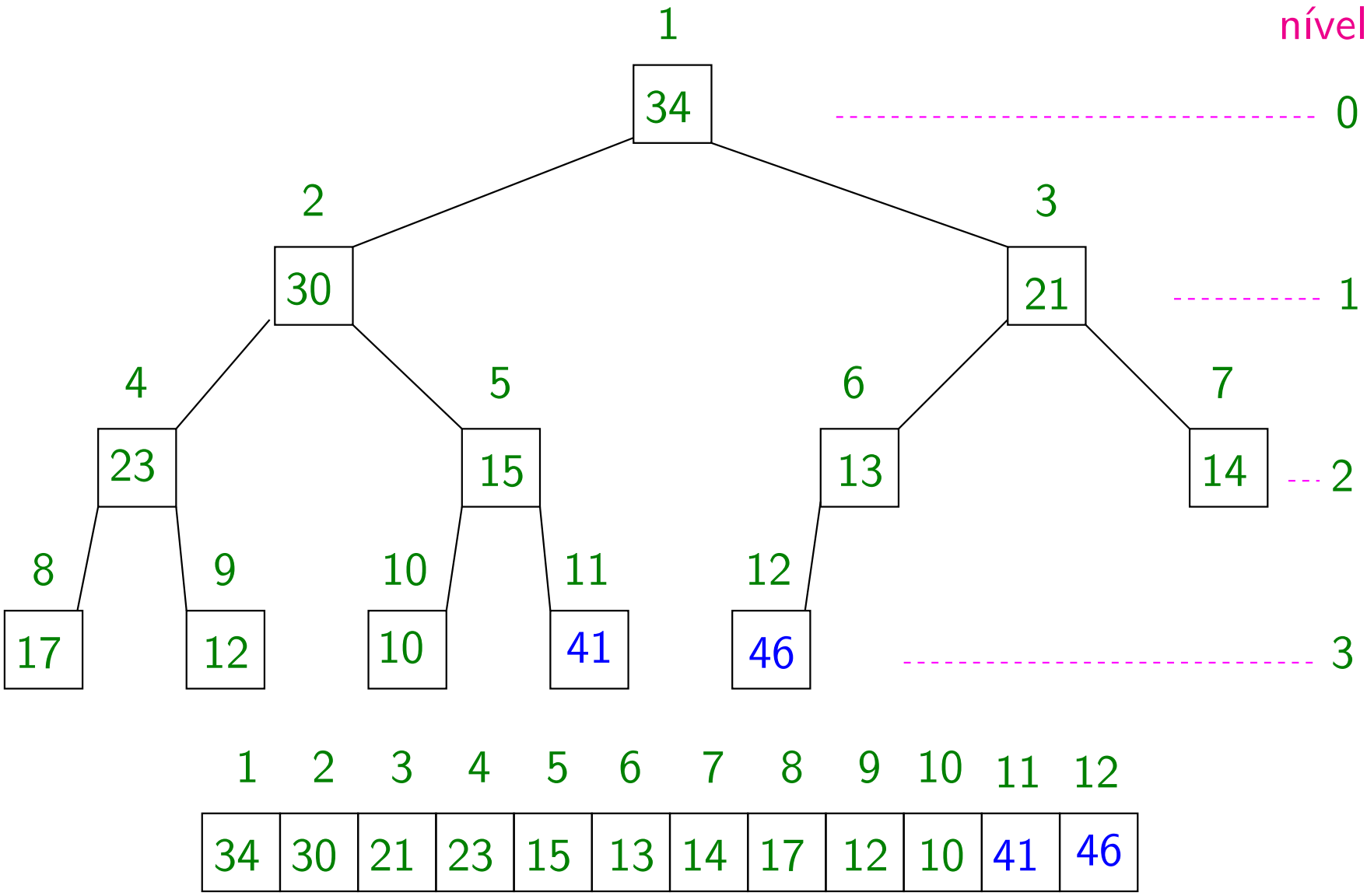
HeapSort


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

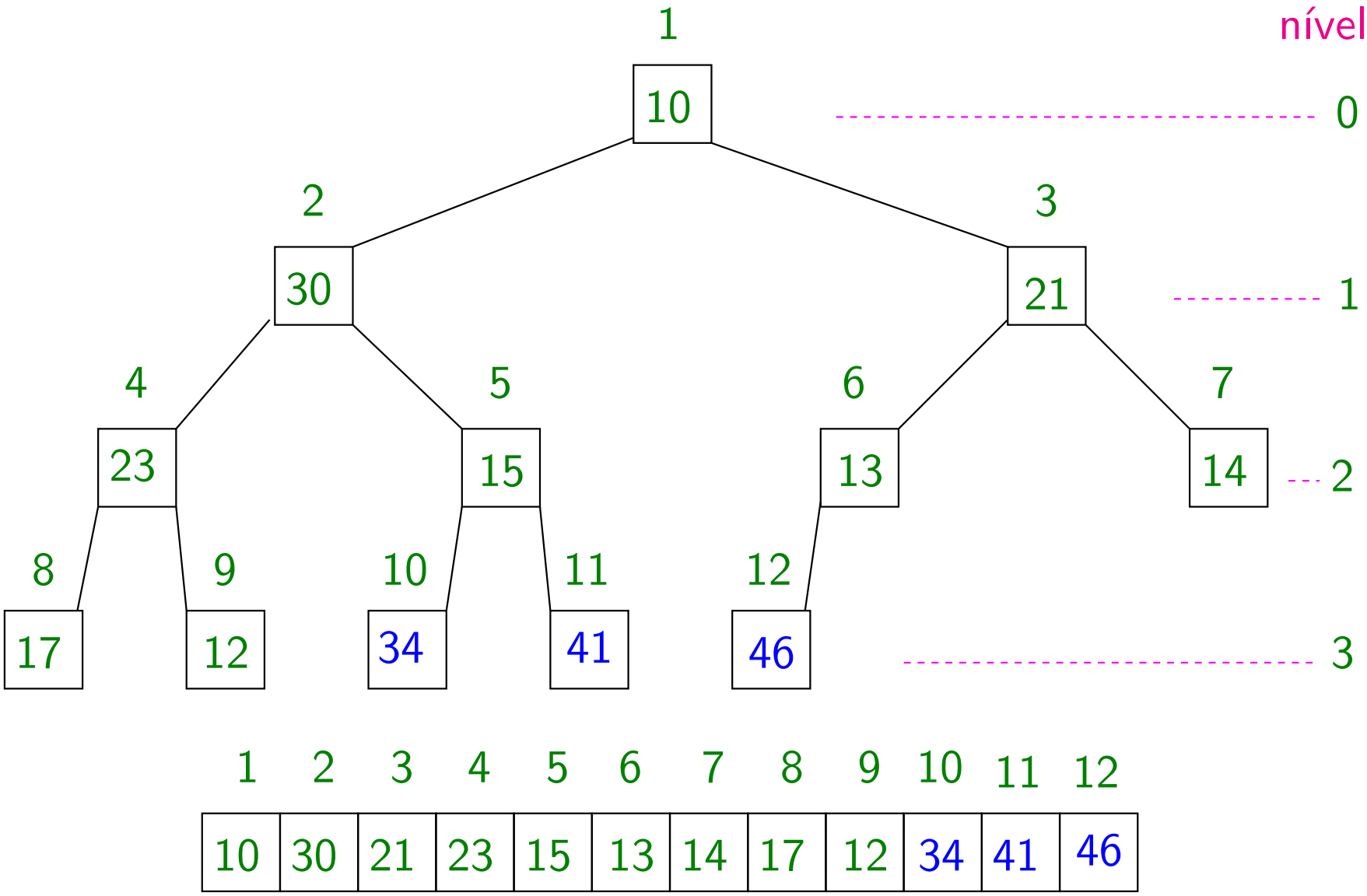
HeapSort


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

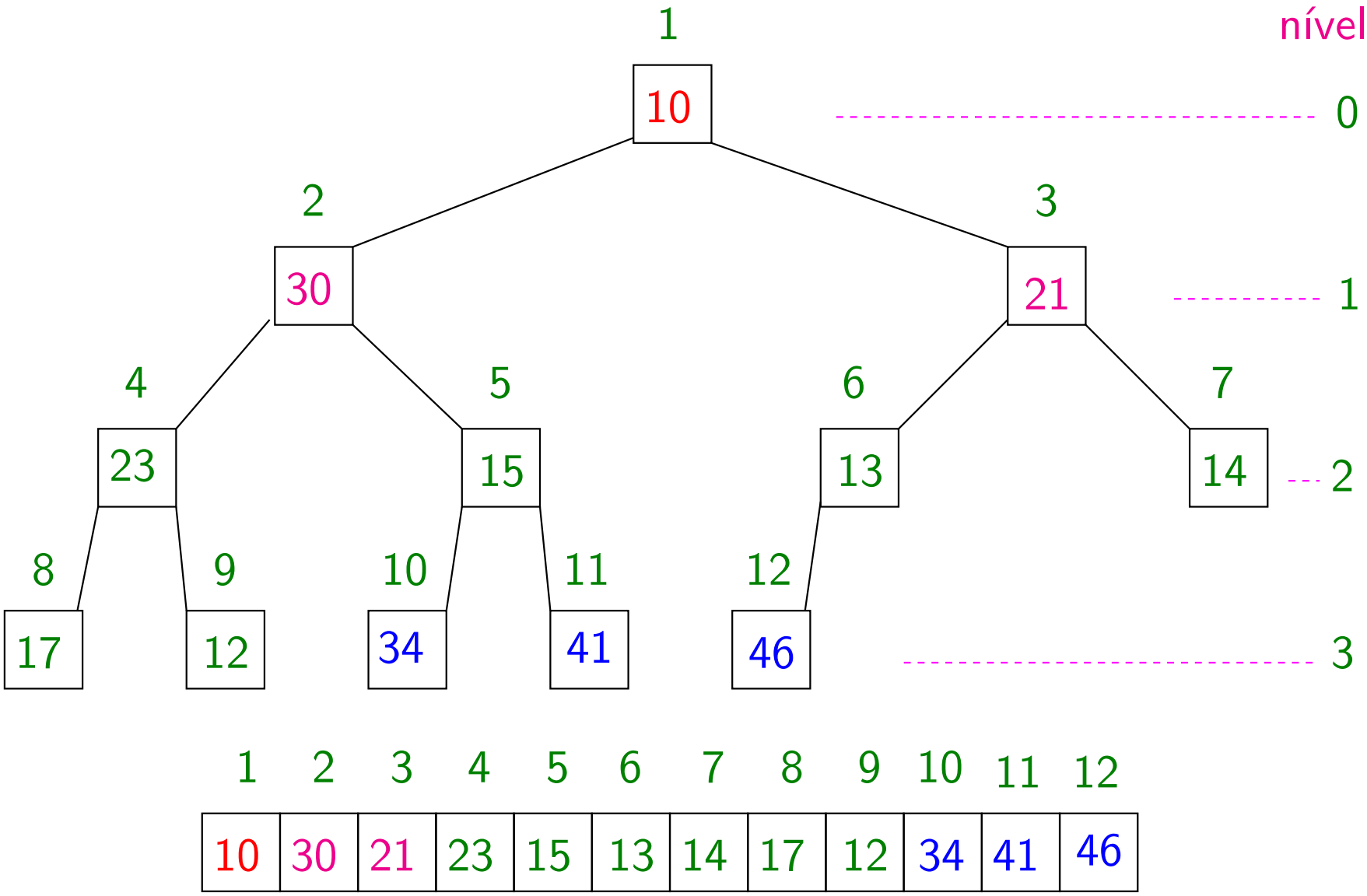
HeapSort


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

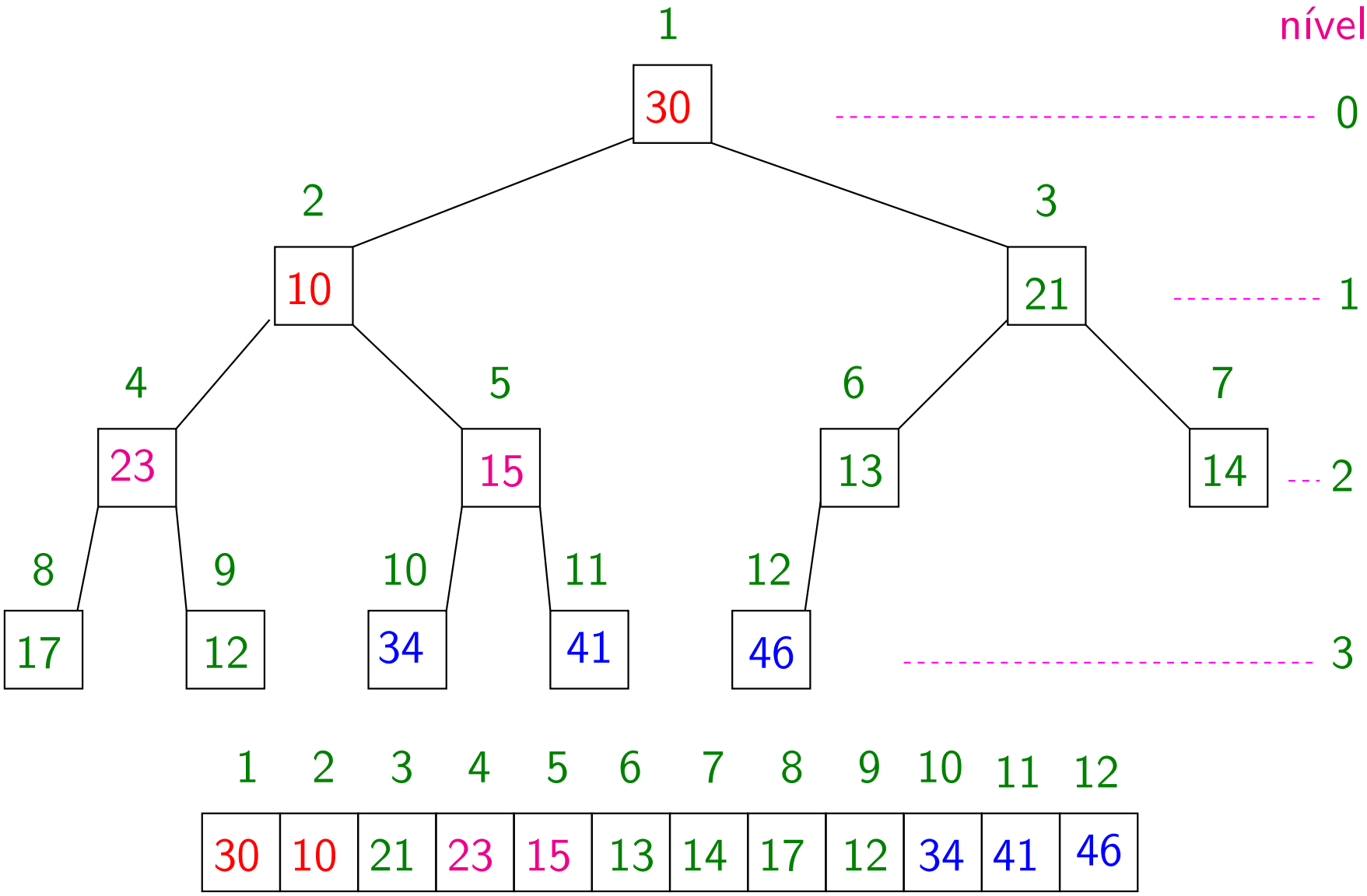
HeapSort


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

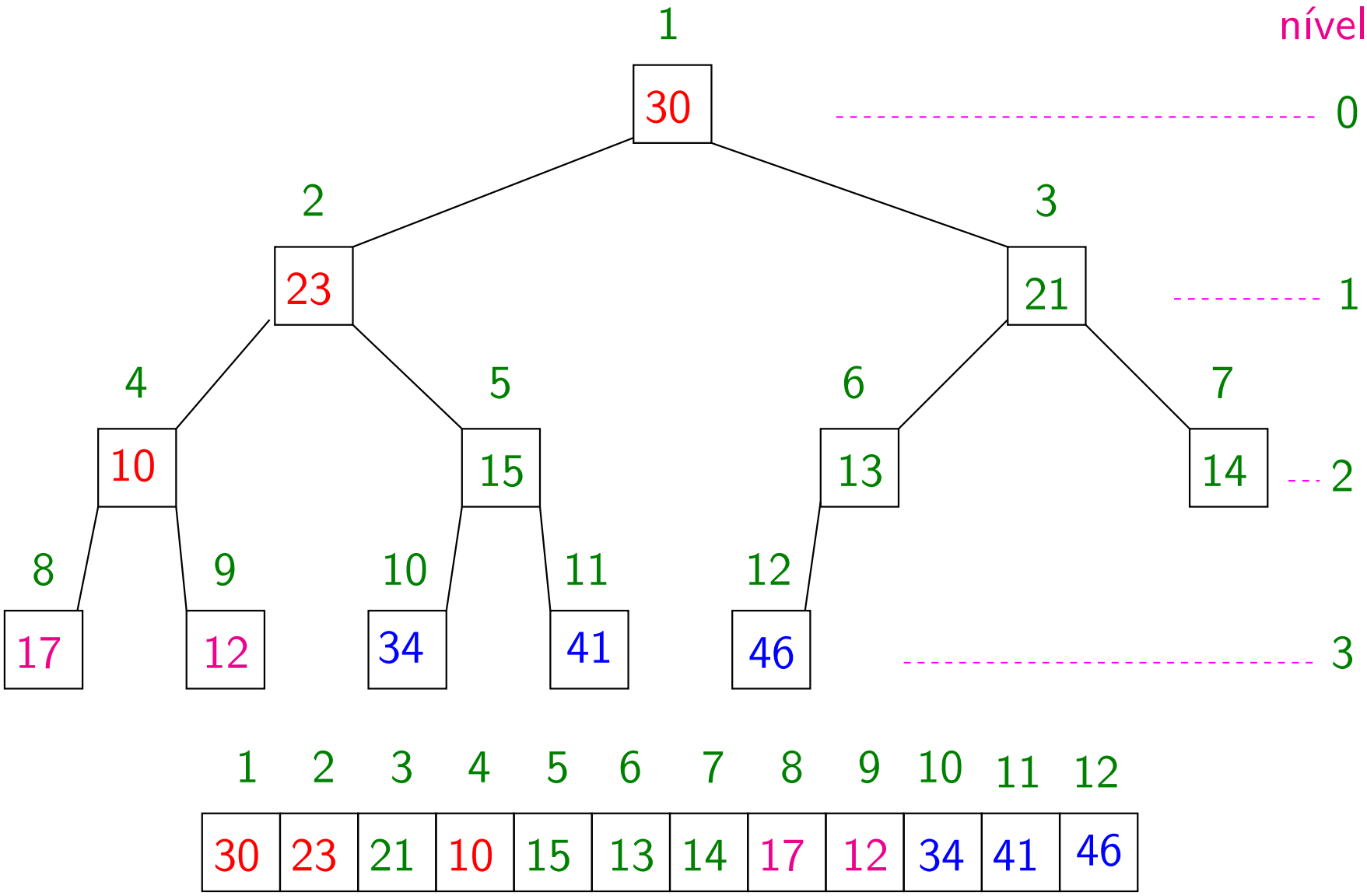
HeapSort


1
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

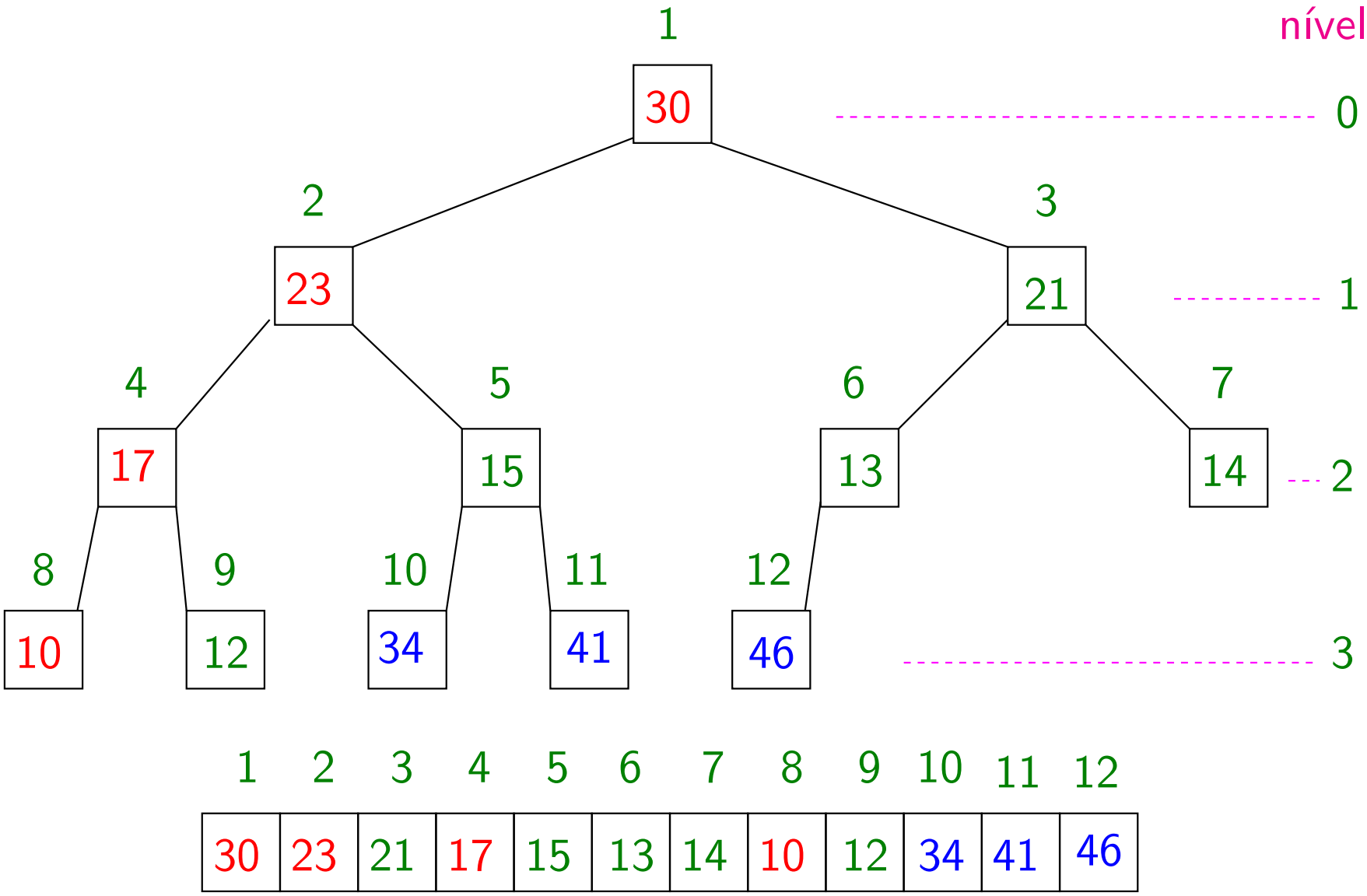
HeapSort


1
nível
30
0
2
3
10
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
30 10 21 23 15 13 14 17 12 34 41 46

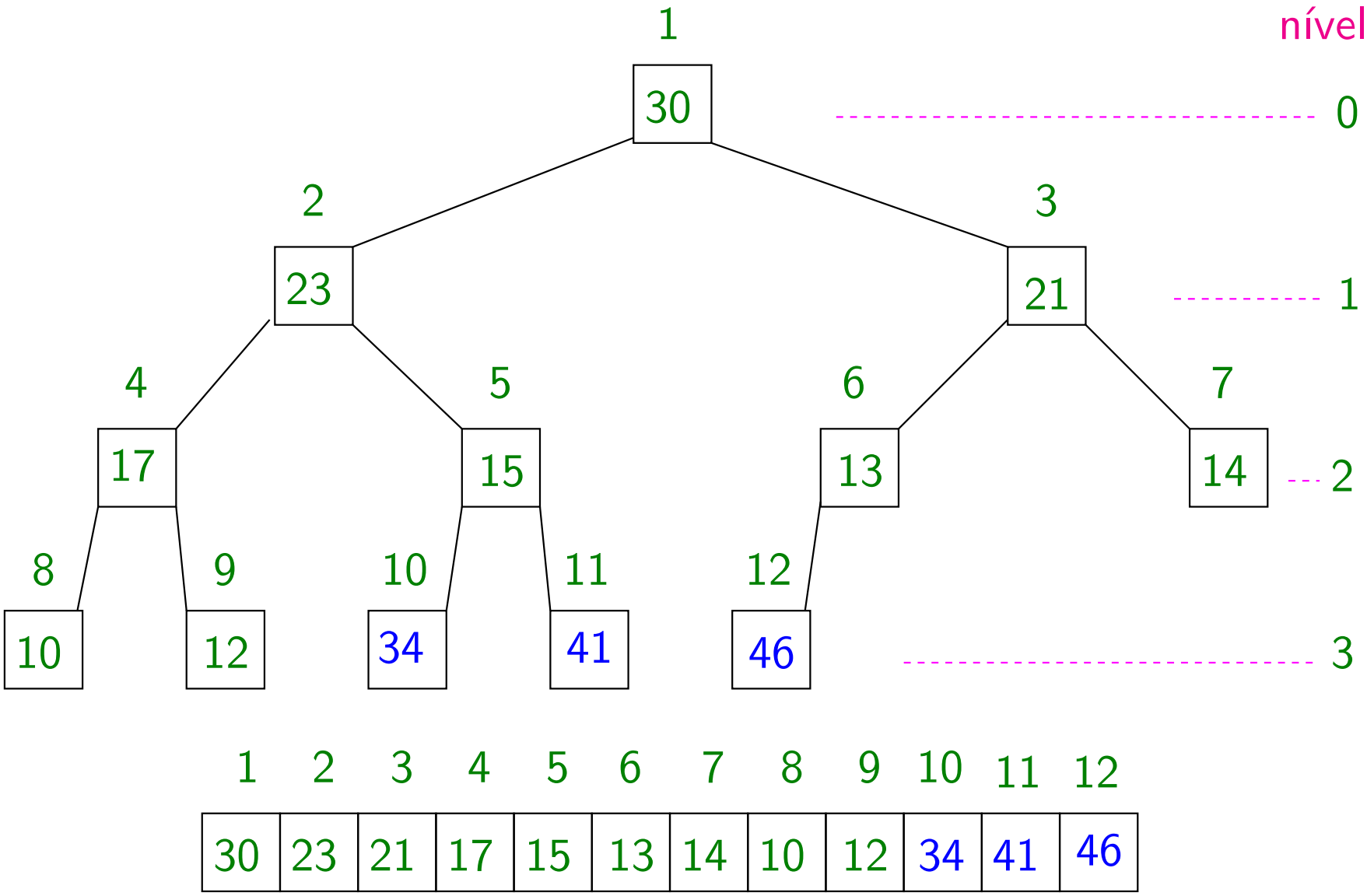
HeapSort


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

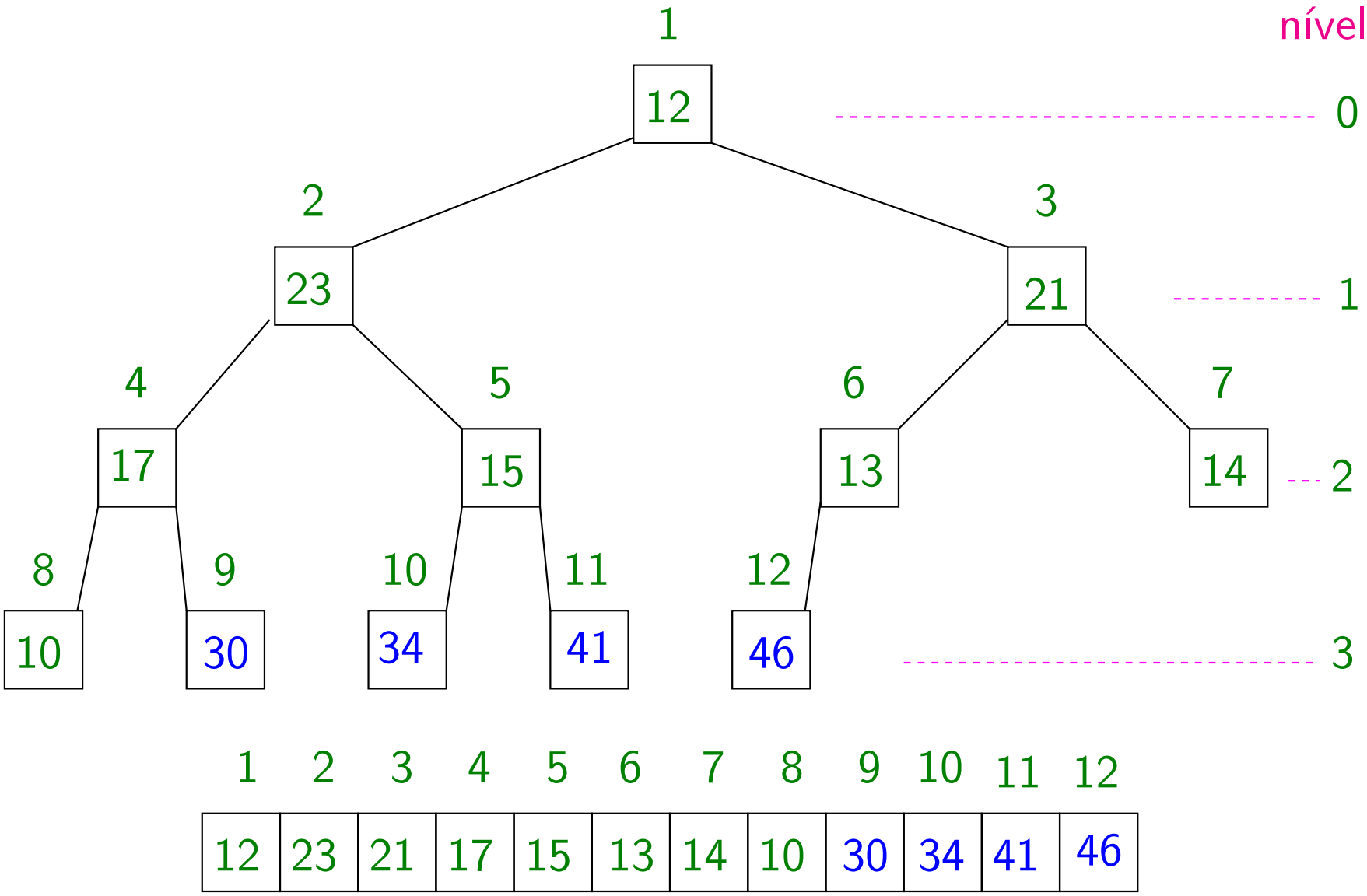
HeapSort


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

```
HeapSort(A, n)
1  Build-Max-Heap(A, n)
2  para i ← n decrescendo até 2 faça
3      A[1] ↔ A[i]
4      Max-Heapify(A, i − 1, 1)
```

HeapSort

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

```
HeapSort(A, n)
1  Build-Max-Heap(A, n)
2  para i ← n decrescendo até 2 faça
3      A[1] ↔ A[i]
4      Max-Heapify(A, i − 1, 1)
```

Invariantes:

No início de cada iteração na linha 2 vale que:

1. $A[1\,..\,i]$ é um max-heap;
2. $A[1\,..\,i] \leq A[i+1\,..\,n]$;
3. $A[i+1\,..\,n]$ está ordenado.

HeapSort

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

HeapSort(A, n)	Tempo
1 Build-Max-Heap(A, n)	?
2 **para** $i \leftarrow n$ **decrescendo até** 2 **faça**	?
3 $\quad A[1] \leftrightarrow A[i]$	?
4 $\quad$ Max-Heapify$(A, i-1, 1)$	?

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

HeapSort

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

HeapSort(A, n)	Tempo
1 Build-Max-Heap(A, n)	$\Theta(n)$
2 **para** $i \leftarrow n$ **decrescendo até** 2 **faça**	$\Theta(n)$
3 $\quad A[1] \leftrightarrow A[i]$	$\Theta(n)$
4 $\quad$ Max-Heapify$(A, i-1, 1)$	$nO(\lg n)$

$T(n) = ??$

HeapSort

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

HeapSort(A, n)	Tempo
1 Build-Max-Heap(A, n)	$\Theta(n)$
2 **para** $i \leftarrow n$ **decrescendo até** 2 **faça**	$\Theta(n)$
3 $\quad A[1] \leftrightarrow A[i]$	$\Theta(n)$
4 $\quad$ Max-Heapify$(A, i-1, 1)$	$nO(\lg n)$

$T(n) = nO(\lg n) + 3\Theta(n) = O(n \lg n)$

HeapSort

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

HeapSort(A, n)	Tempo
1 Build-Max-Heap(A, n)	$\Theta(n)$
2 **para** $i \leftarrow n$ **decrescendo até** 2 **faça**	$\Theta(n)$
3 $\quad A[1] \leftrightarrow A[i]$	$\Theta(n)$
4 $\quad$ Max-Heapify$(A, i-1, 1)$	$nO(\lg n)$

$$T(n) = nO(\lg n) + 3\Theta(n) = O(n \lg n)$$

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

HeapSort

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

HeapSort(A, n)	Tempo
1 Build-Max-Heap(A, n)	$\Theta(n)$
2 **para** $i \leftarrow n$ **decrescendo até** 2 **faça**	$\Theta(n)$
3 $\quad A[1] \leftrightarrow A[i]$	$\Theta(n)$
4 $\quad$ Max-Heapify$(A, i-1, 1)$	$nO(\lg n)$

$T(n) = nO(\lg n) + 3\Theta(n) = O(n \lg n)$

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

Como seria a complexidade de tempo no melhor caso?

HeapSort

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

HeapSort(A, n)	Tempo
1 Build-Max-Heap(A, n)	$\Theta(n)$
2 **para** $i \leftarrow n$ **decrescendo até** 2 **faça**	$\Theta(n)$
3 $\quad A[1] \leftrightarrow A[i]$	$\Theta(n)$
4 $\quad$ Max-Heapify$(A, i-1, 1)$	$nO(\lg n)$

$$T(n) = nO(\lg n) + 3\Theta(n) = O(n \lg n)$$

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

Como seria a complexidade de tempo no melhor caso?

Pode-se mostrar que no melhor caso, a complexidade é $\Theta(n \lg n)$, supondo que os elementos são distintos.

Filas com prioridades

Uma fila de 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) $\triangleright$ $n \geq 1$
1 max $\leftarrow A[1]$
2 $A[1] \leftarrow A[n]$
3 $n \leftarrow n - 1$
4 MAX-HEAPIFY $(A, n, 1)$
5 **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 $A[\text{pai}(i)] < A[i]$ **faça**
4 $A[i] \leftrightarrow A[\text{pai}(i)]$
5 $i \leftarrow \text{pai}(i)$

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: $O(\lg n)$.

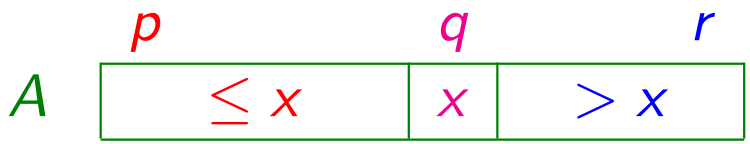
QuickSort

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

QuickSort

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

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

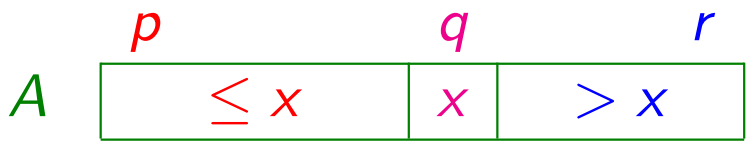


$$A[p\,..\,q-1] \leq A[q] < A[q+1\,..\,r]$$

QuickSort

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

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



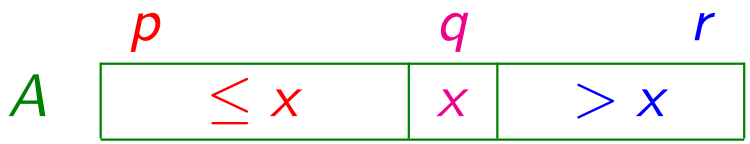
$$A[p\,..\,q-1] \leq A[q] < A[q+1\,..\,r]$$

Conquista: ordene os dois subvetores recursivamente usando o QuickSort;

QuickSort

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

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



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

Conquista: ordene os dois subvetores recursivamente usando o QuickSort;

Combinação: nada a fazer, o vetor está ordenado.

Partição

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

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

Entrada:

Partição

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

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

Entrada:

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

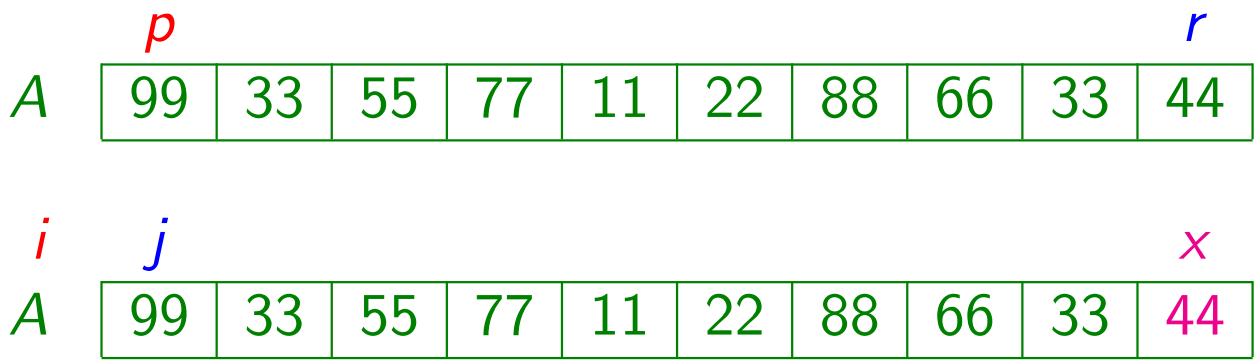
Saída:

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

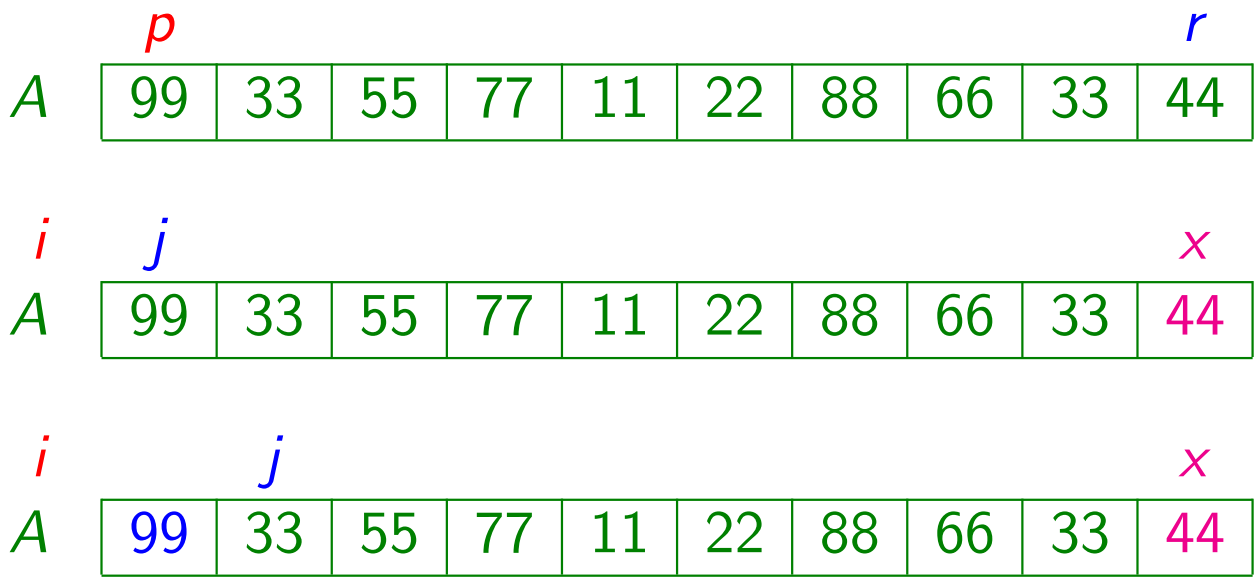
Particione

Particione

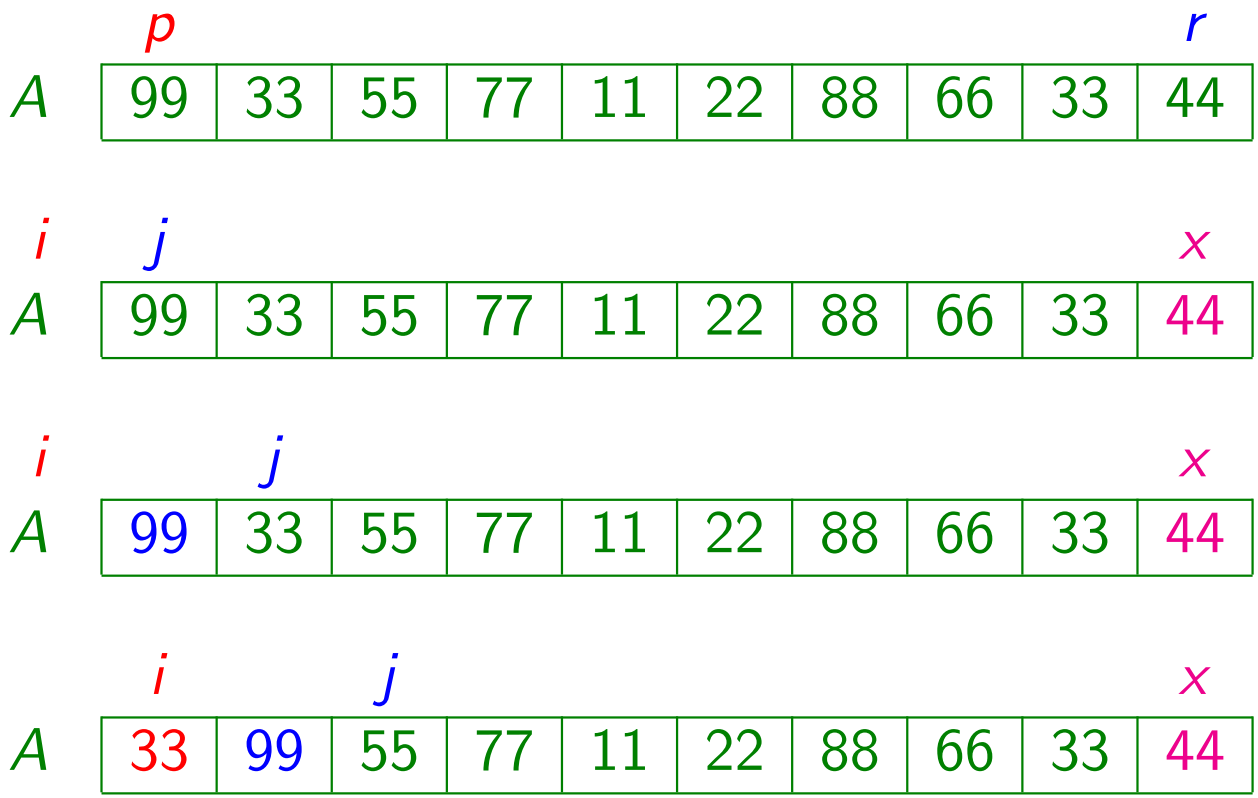
Particione



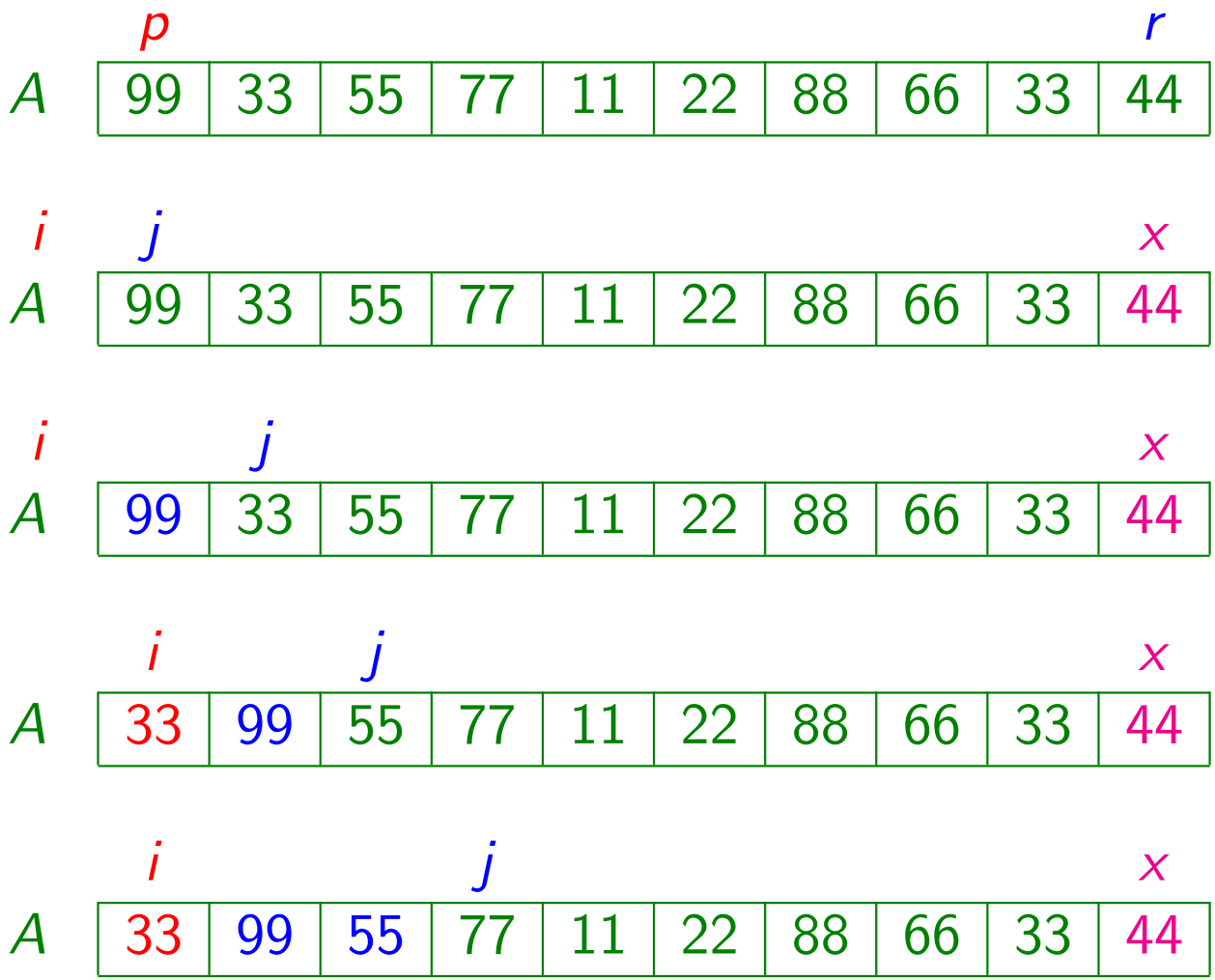
Particione


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

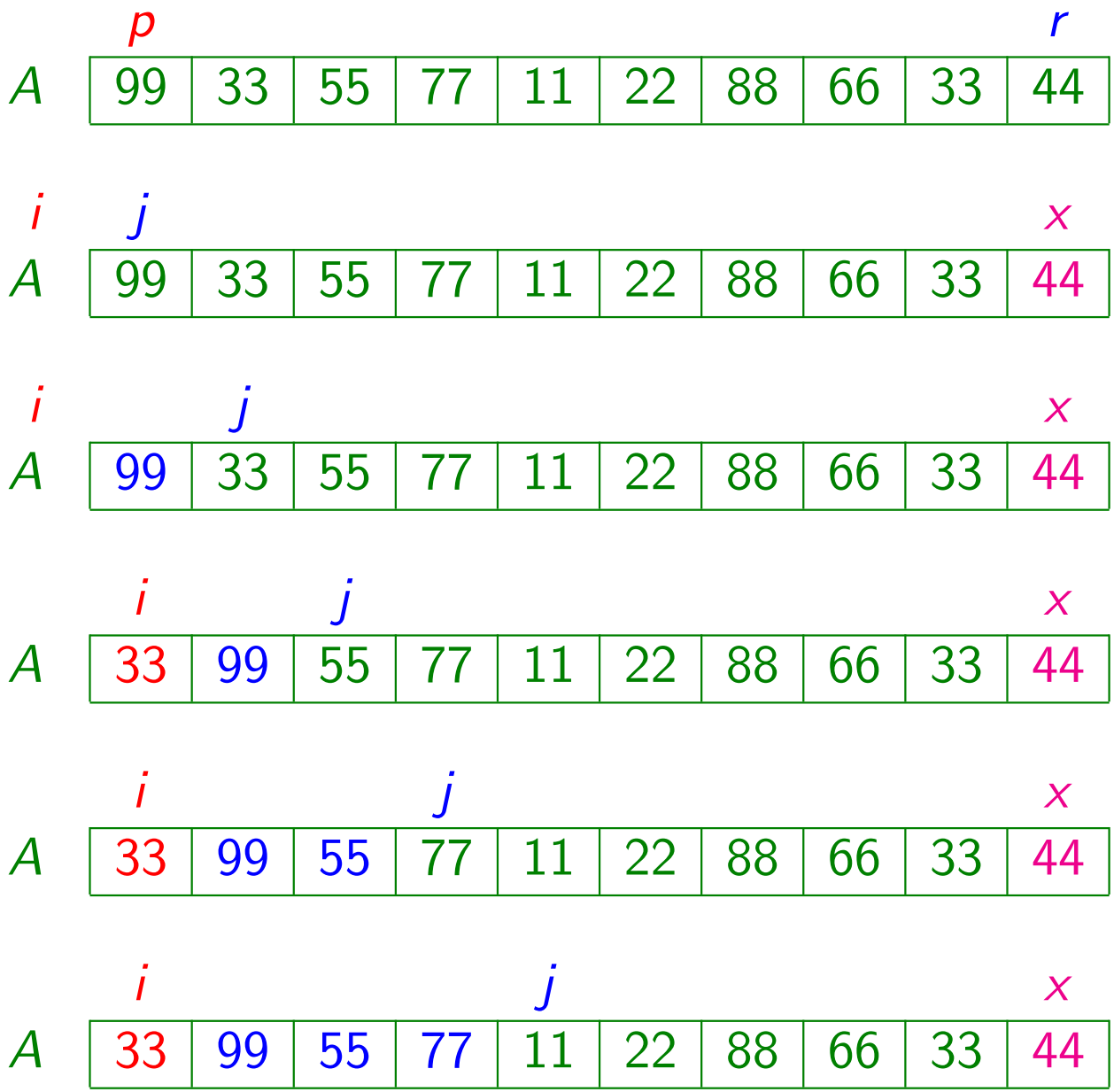
Particione


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

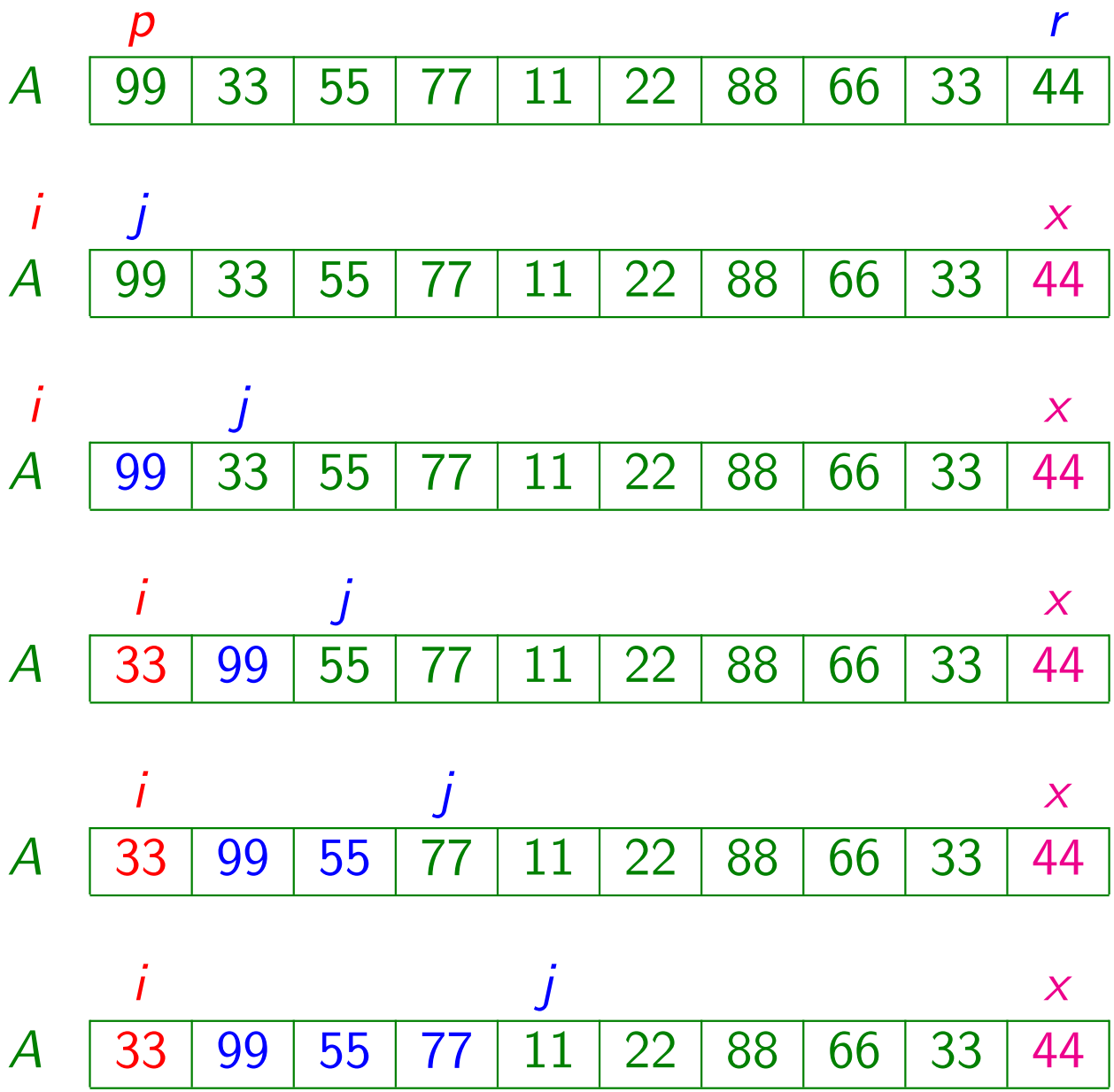
Particione


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

Particione

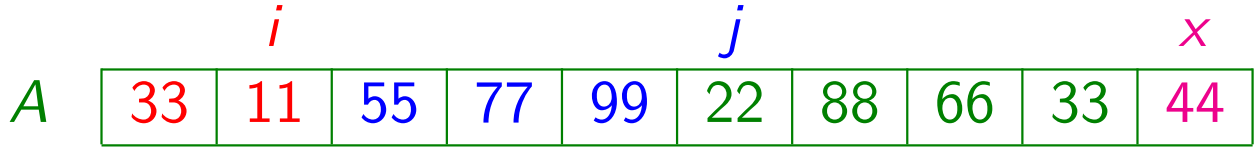

p
r
A
99 33 55 77 11 22 88 66 33 44
i
j
x
A
99 33 55 77 11 22 88 66 33 44
i
j
x
A
99 33 55 77 11 22 88 66 33 44
i
j
x
A
33 99 55 77 11 22 88 66 33 44
i
j
x
A
33 99 55 77 11 22 88 66 33 44
i
j
x
A
33 99 55 77 11 22 88 66 33 44

Particione

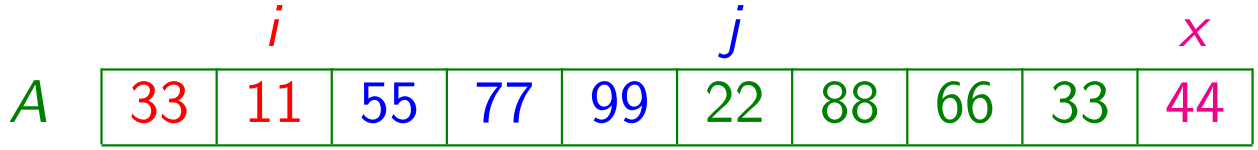

p
r
A
99 33 55 77 11 22 88 66 33 44
i
j
x
A
99 33 55 77 11 22 88 66 33 44
i
j
x
A
99 33 55 77 11 22 88 66 33 44
i
j
x
A
33 99 55 77 11 22 88 66 33 44
i
j
x
A
33 99 55 77 11 22 88 66 33 44
i
j
x
A
33 99 55 77 11 22 88 66 33 44

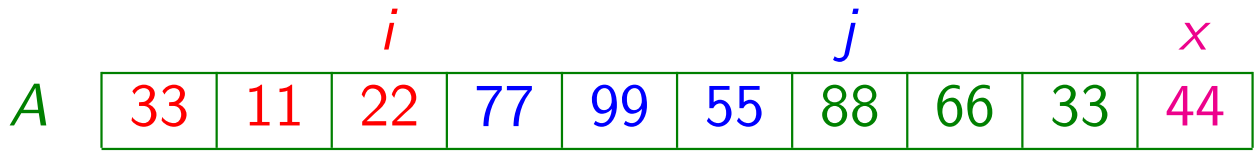
Particione

Particione

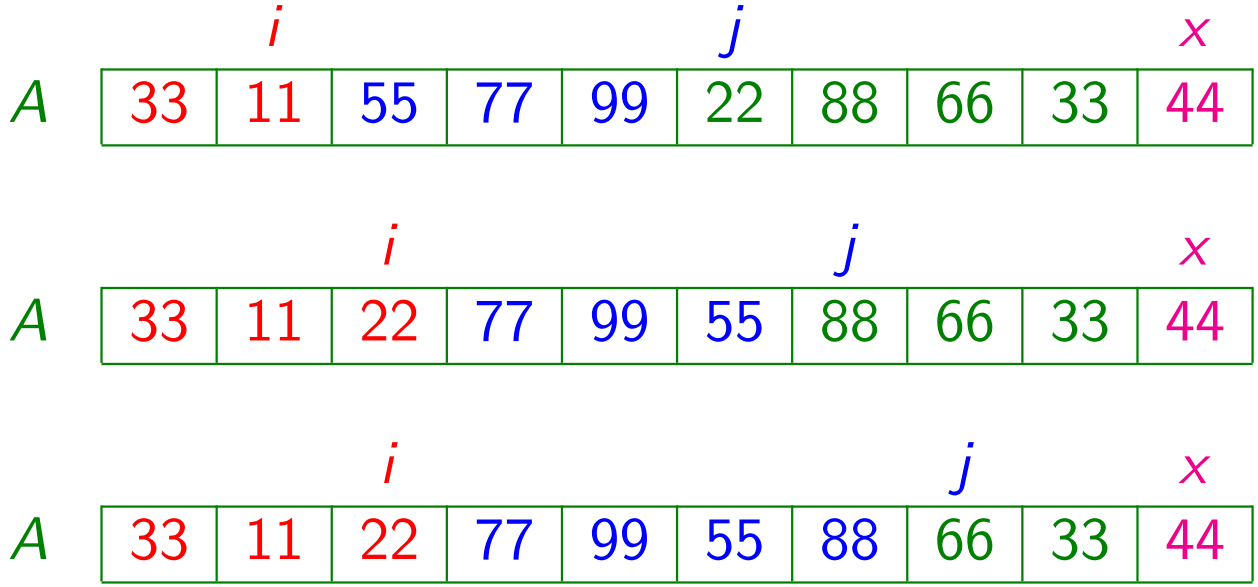


Particione

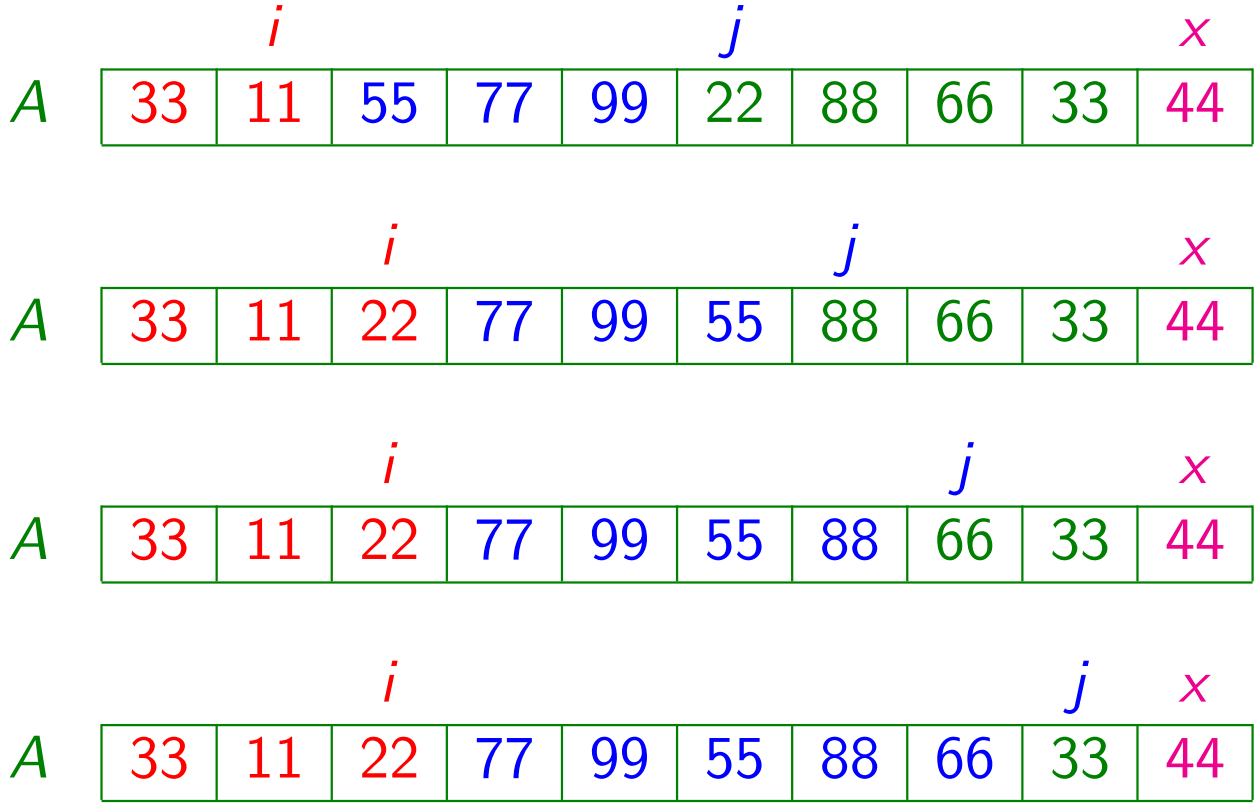




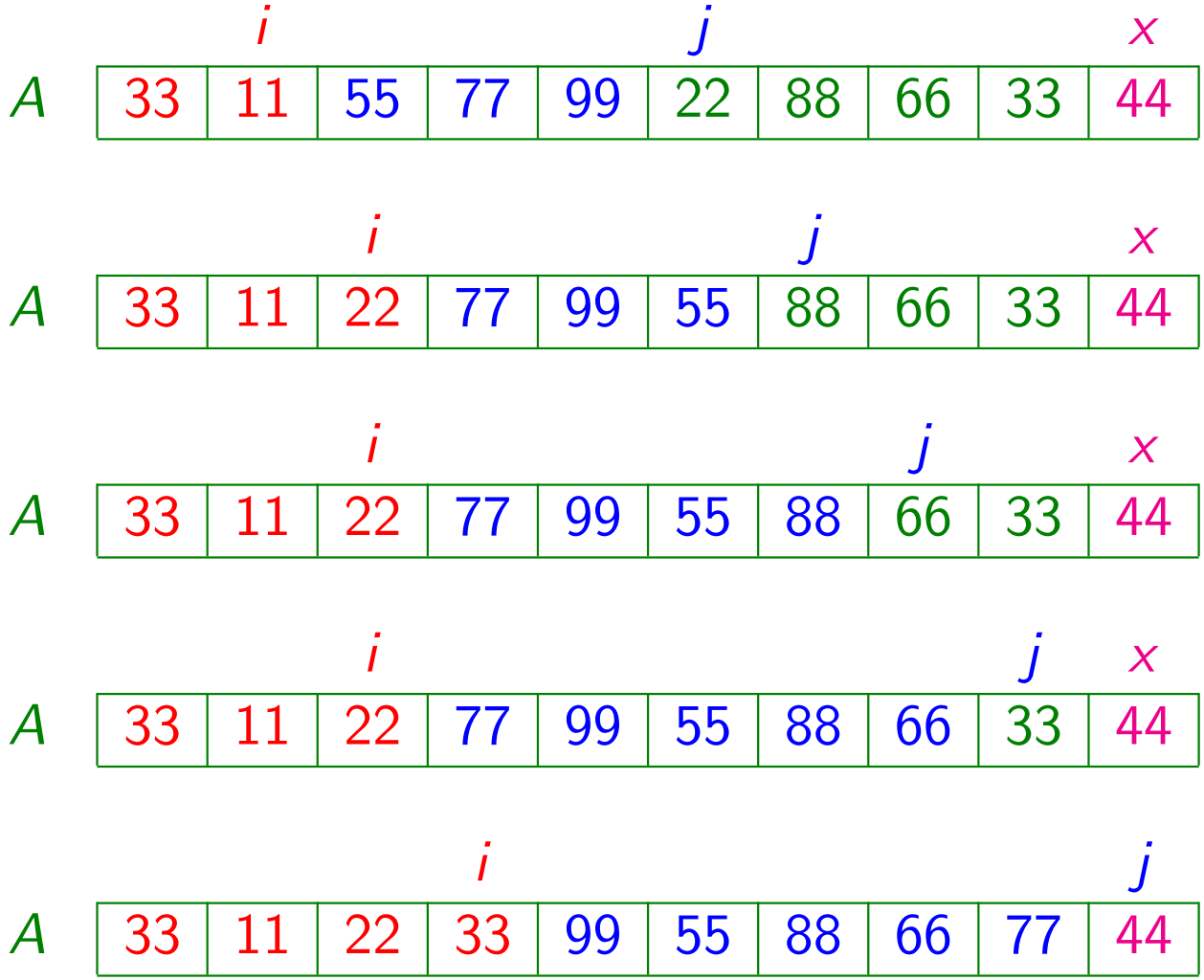
Particione


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

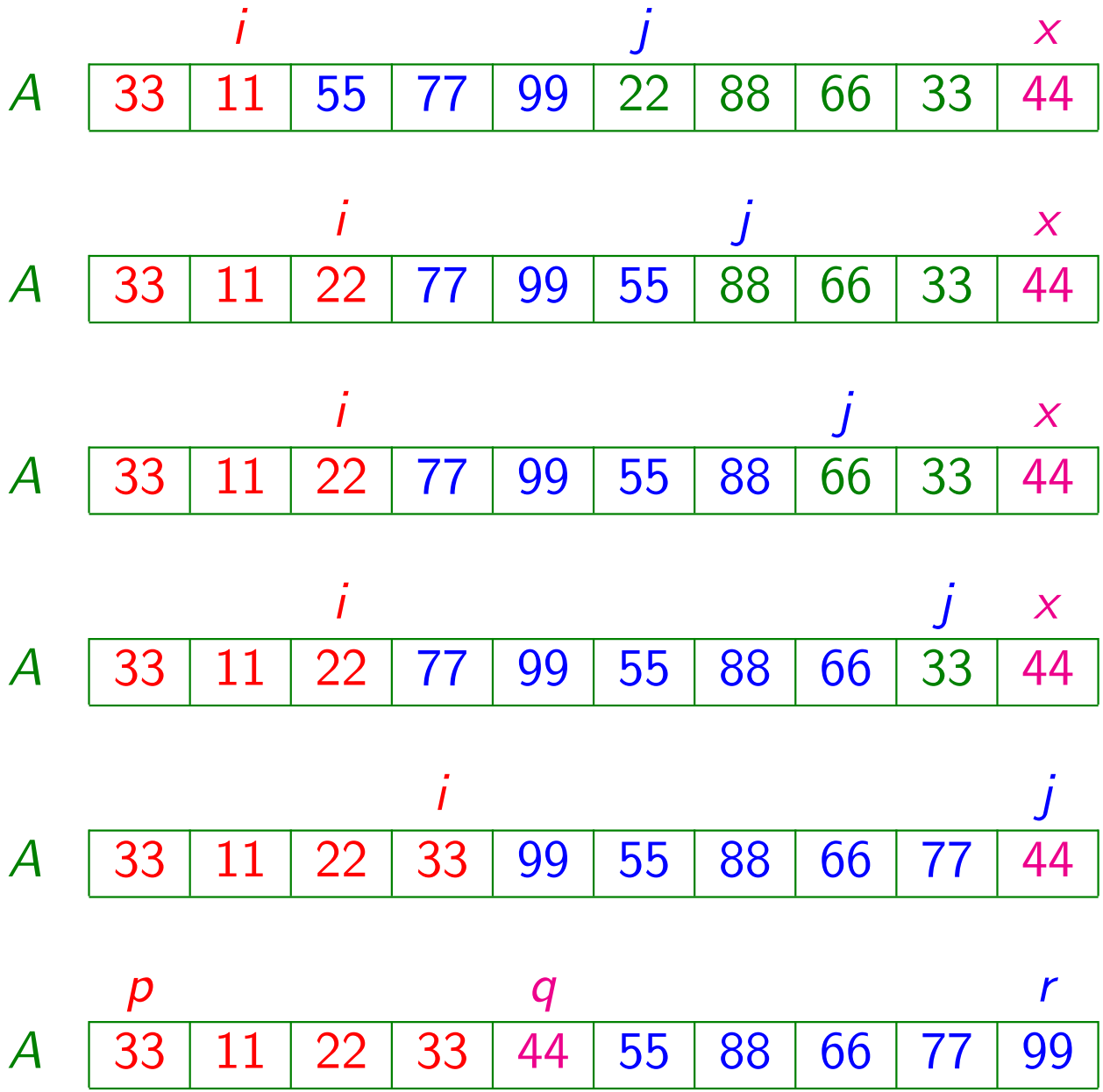
Particione


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

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

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 $A[p\,..\,r]$ de modo que $p \leq q \leq r$ e
$A[p\,..\,q{-}1] \leq A[q] < A[q{+}1\,..\,r]$

```
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
```

Particione

Rearranja $A[p\,..\,r]$ de modo que $p \le q \le r$ e
$A[p\,..\,q{-}1] \le A[q] < A[q{+}1\,..\,r]$

```
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) $A[p\,..\,i] \le x$ (2) $A[i{+}1\,..\,j{-}1] > x$ (3) $A[r] = x$

Complexidade de PARTICIONE

PARTICIONE(A, p, r)		Tempo
1	$x \leftarrow A[r]$ $\quad \triangleright\, x$ 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** $i+1$	$\Theta(1)$

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

Conclusão:

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

QuickSort

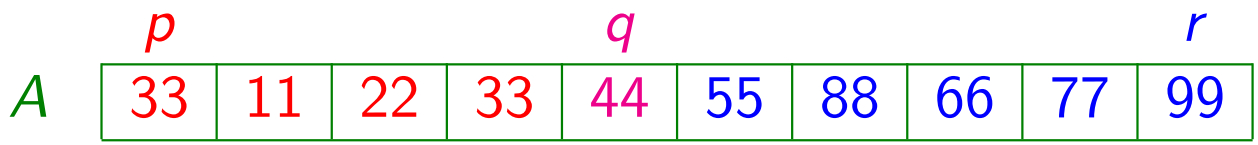
Rearranja um vetor $A[p\,..\,r]$ em ordem crescente.

```
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 $A[p\,..\,r]$ 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)
```



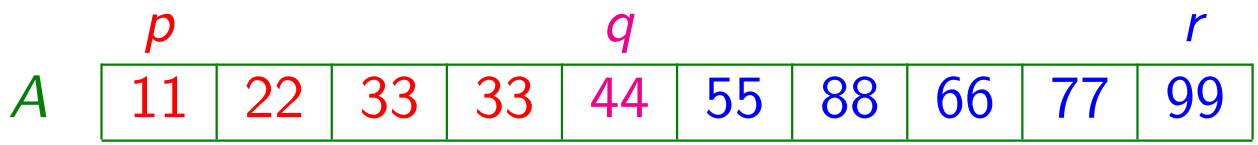
No começo da linha 3,

$$A[p\,..\,q{-}1] \leq A[q] < A[q{+}1\,..\,r]$$

QuickSort

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

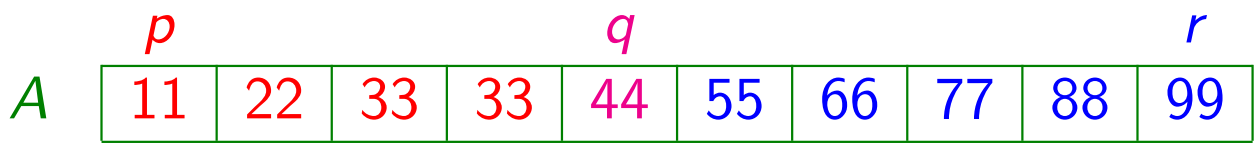
```
QuickSort(A, p, r)
1  se p < r
2     então q ← Particione(A, p, r)
3           QuickSort(A, p, q − 1)
4           QuickSort(A, q + 1, r)
```



QuickSort

Rearranja um vetor $A[p\,..\,r]$ 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

$T(n) :=$ 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

$T(n) :=$ 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 $\Theta(n^2)$.

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.

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

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

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

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 de ordenação mais eficiente.

Caso médio

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

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

Caso médio

Apesar da complexidade de tempo do QuickSort no pior caso ser $\Theta(n^2)$, na prática ele é o algoritmo de ordenação 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??

Caso médio

Apesar da complexidade de tempo do QuickSort no pior caso ser $\Theta(n^2)$, na prática ele é o algoritmo de ordenação 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 $\frac{1}{10}$ para $\frac{9}{10}$. Então

Caso médio

Apesar da complexidade de tempo do QuickSort no pior caso ser $\Theta(n^2)$, na prática ele é o algoritmo de ordenação 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 $\frac{1}{10}$ para $\frac{9}{10}$. Então

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

Caso médio

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

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

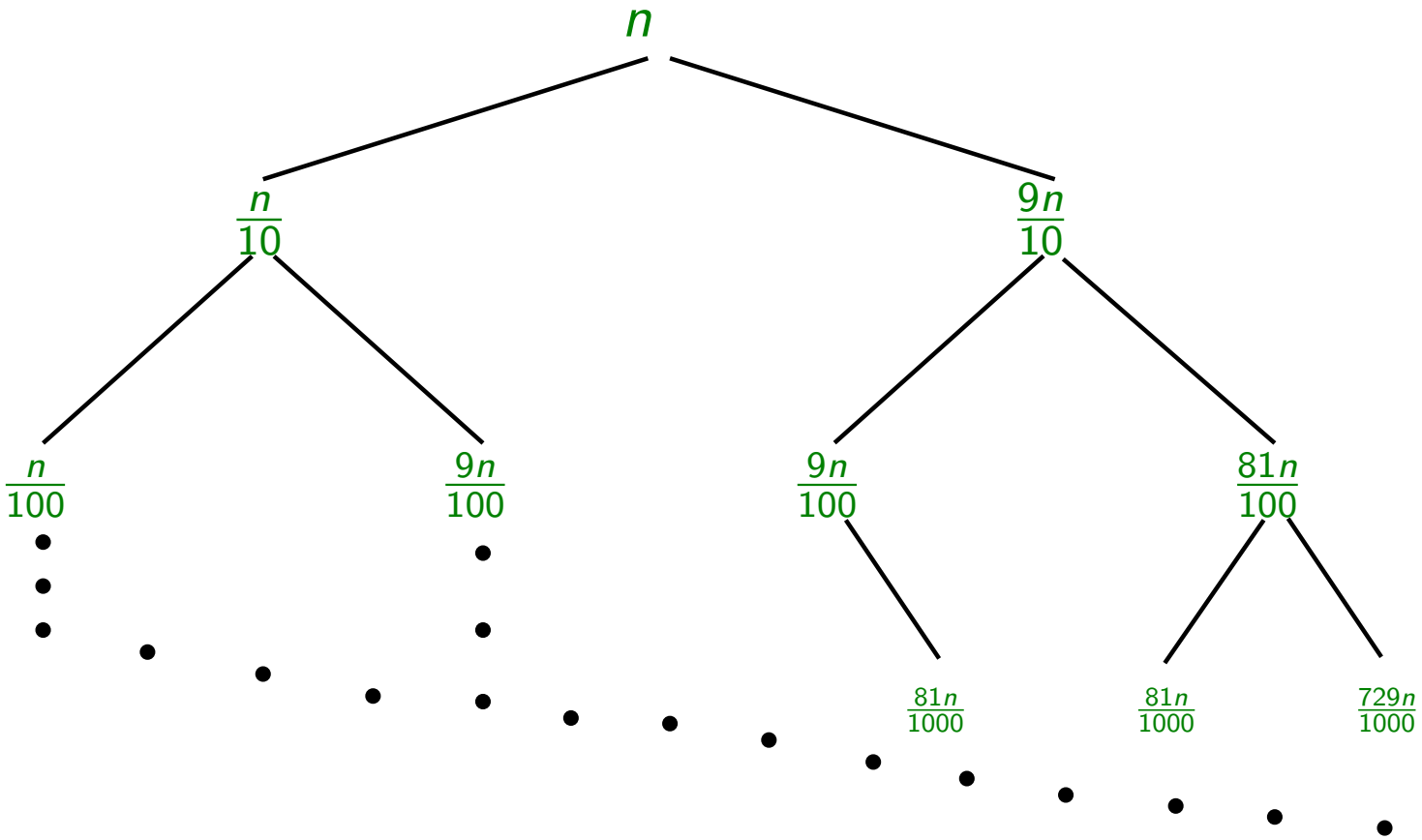
Por quê??

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

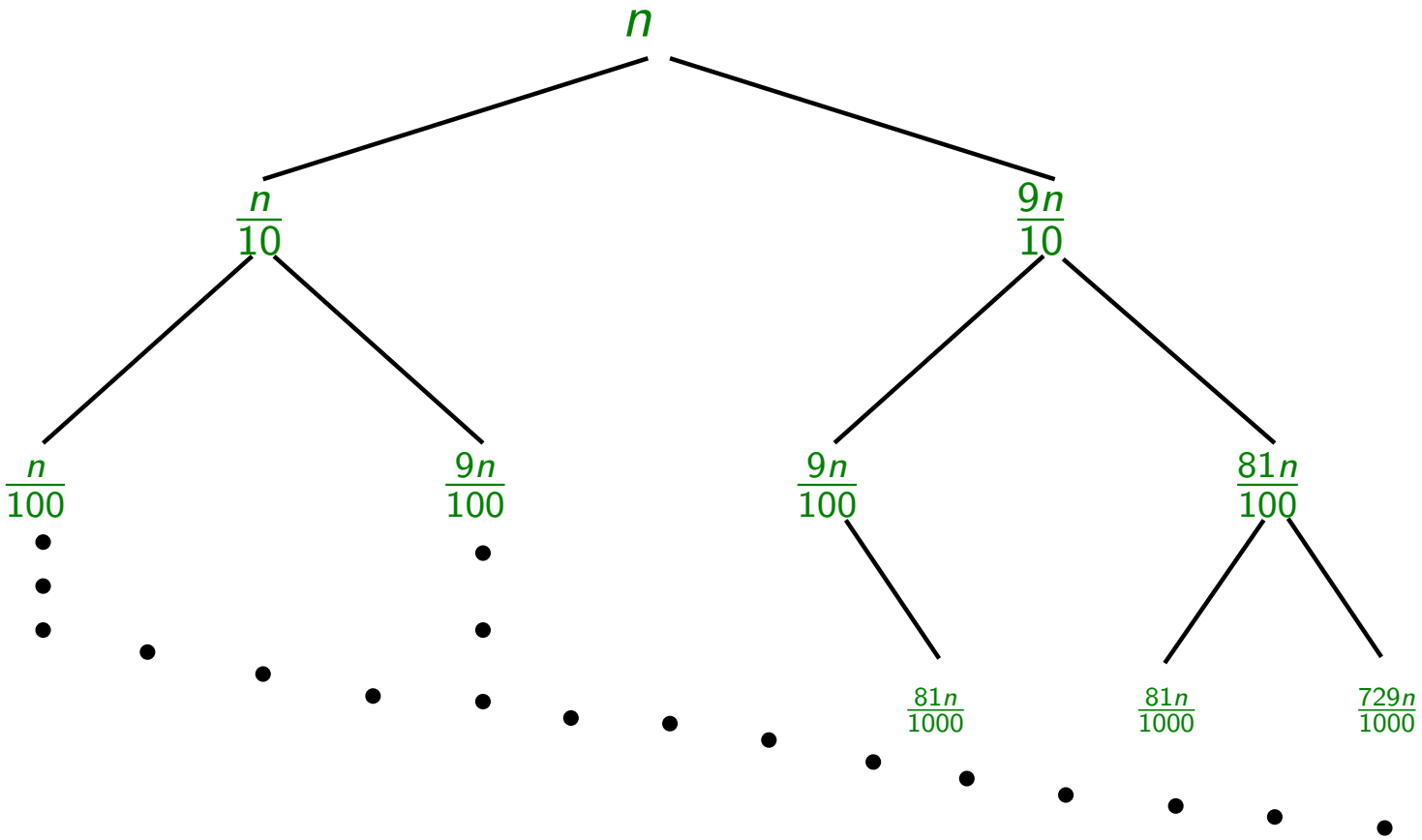
$$T(n) = T\left(\left\lfloor \frac{n-1}{10} \right\rfloor\right) + T\left(\left\lceil \frac{9(n-1)}{10} \right\rceil\right) + \Theta(n)$$

Solução: $T(n)$ é $\Theta(n \lg n)$.

Árvore de recorrência

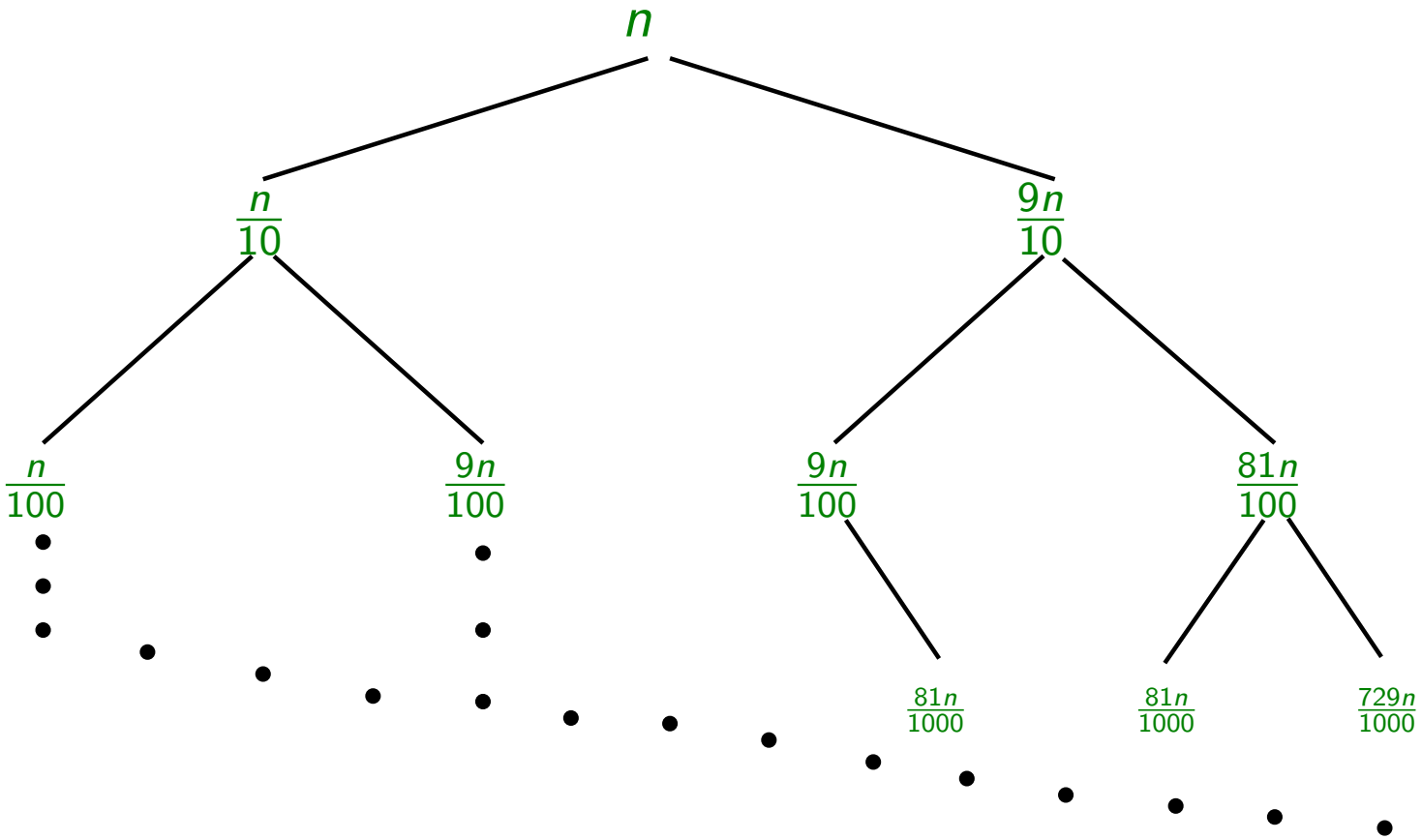


Árvore de recorrência



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

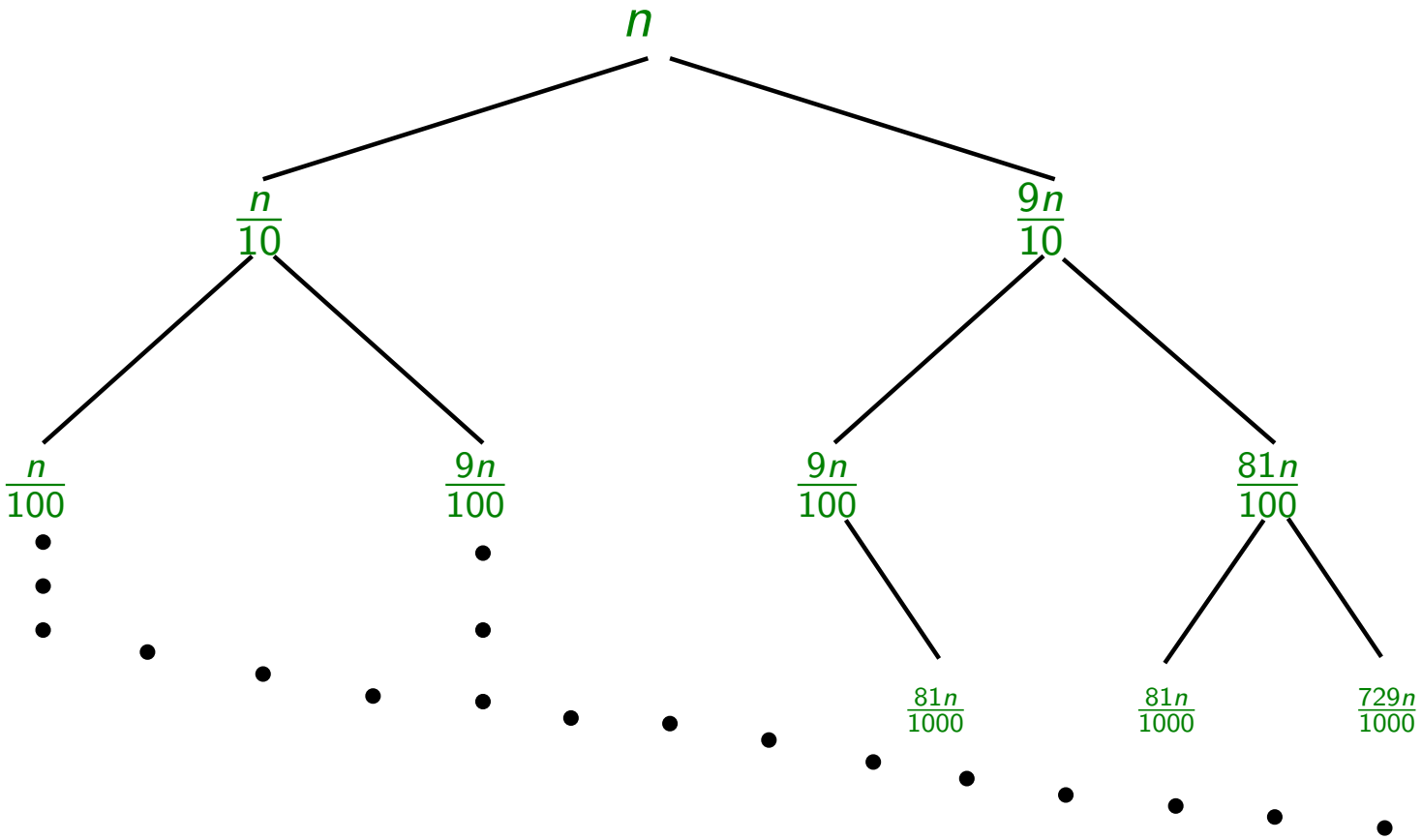
Árvore de recorrência



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

Em cada nível o custo é $\leq n$.

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

QuickSort Aleatório

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

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 $\quad i \leftarrow$ RANDOM(p, r)
2 $\quad A[i] \leftrightarrow A[r]$
3 $\quad$ **devolva** PARTICIONE(A, p, r)

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.

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

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

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

$T(n)$ é $\Theta(???)$.

Análise do caso médio

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

Análise do caso médio

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

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

Análise do caso médio

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

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

Vou mostrar que $T(n) \leq cn \lg n$ para $n \geq 1$ onde $c > 0$ é uma constante.

Demonstração

$$
\begin{aligned}
T(n) &\leq \frac{2}{n}\sum_{k=0}^{n-1} T(k) + an \\
&\leq \frac{2}{n}\sum_{k=0}^{n-1} (ck \lg k) + an \\
&= \frac{2c}{n}\sum_{k=1}^{n-1} k \lg k + an
\end{aligned}
$$

Demonstração

$$
\begin{aligned}
T(n) &\leq \frac{2}{n}\sum_{k=0}^{n-1} T(k) + an \\
&\leq \frac{2}{n}\sum_{k=0}^{n-1} (ck \lg k) + an \\
&= \frac{2c}{n}\sum_{k=1}^{n-1} k \lg k + an
\end{aligned}
$$

Lema.

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

Demonstração

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

escolhendo c de modo que $an - \frac{cn}{4} \leq 0$, i.e., $c \geq 4a$.

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 $\Theta(n \lg n)$.