

Projeto e Análise de Algoritmos*

Ordenação

Segundo Semestre de 2019

*Criado por C. de Souza, C. da Silva, O. Lee, F. Miyazawa et al.

A maior parte deste conjunto de slides foi inicialmente preparada por Cid Carvalho de Souza e Cândida Nunes da Silva para cursos de Análise de Algoritmos. Além desse material, diversos conteúdos foram adicionados ou incorporados por outros professores, em especial por Orlando Lee e por Flávio Keidi Miyazawa. Os slides usados nessa disciplina são uma junção dos materiais didáticos gentilmente cedidos por esses professores e contêm algumas modificações, que podem ter introduzido erros.

O conjunto de slides de cada unidade do curso será disponibilizado como guia de estudos e deve ser usado unicamente para revisar as aulas. Para estudar e praticar, leia o livro-texto indicado e resolva os exercícios sugeridos.

Lehilton

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
 - ▶ Flávio Keidi Miyazawa
 - ▶ José Coelho de Pina
 - ▶ Orlando Lee
 - ▶ Paulo Feofiloff
 - ▶ Pedro Rezende
 - ▶ Ricardo Dahab
 - ▶ Zanoni Dias

Ordenação

O problema da ordenação

Problema:

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

Ou simplesmente:

Problema:

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

O problema da ordenação

Problema:

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

Ou simplesmente:

Problema:

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

Veremos vários algoritmos de ordenação:

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

Insertion sort

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

Insertion sort

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

Insertion sort

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

Insertion Sort

Insertion sort – pseudocódigo

```
INSERTION-SORT( $A, n$ )
1  para  $j \leftarrow 2$  até  $n$  faça
2      chave  $\leftarrow A[j]$ 
3       $\triangleright$  Insere  $A[j]$  no subvetor ordenado  $A[1..j-1]$ 
4       $i \leftarrow j-1$ 
5      enquanto  $i \geq 1$  e  $A[i] >$  chave faça
6           $A[i+1] \leftarrow A[i]$ 
7           $i \leftarrow i-1$ 
8       $A[i+1] \leftarrow$  chave
```

Já analisamos antes a correção e complexidade.

Vamos analisar novamente a complexidade usando a notação assintótica.

Insertion sort – pseudocódigo

INSERTION-SORT(A, n)

1 para $j \leftarrow 2$ até n faça

2 $chave \leftarrow A[j]$

3 ▷ Insere $A[j]$ no subvetor ordenado $A[1..j-1]$

4 $i \leftarrow j - 1$

5 enquanto $i \geq 1$ e $A[i] > chave$ faça

6 $A[i+1] \leftarrow A[i]$

7 $i \leftarrow i - 1$

8 $A[i+1] \leftarrow chave$

Já analisamos antes a correção e complexidade.

Vamos analisar novamente a complexidade usando a notação assintótica.

Insertion sort – pseudocódigo

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

Já analisamos antes a correção e complexidade.

Vamos analisar novamente a complexidade usando a notação assintótica.

Complexidade de tempo de Insertion sort

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

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

Complexidade de tempo de Insertion sort

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

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

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)
 $O(n)$ comparações
zero movimentações
- ▶ Complexidade de espaço/consumo espaço: $\Theta(n)$

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)
 $O(n)$ comparações
zero movimentações
- ▶ **Complexidade de espaço/consumo espaço: $\Theta(n)$**

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)
 $O(n)$ comparações
zero movimentações
- ▶ **Complexidade de espaço/consumo espaço: $\Theta(n)$**

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)
 $O(n)$ comparações
zero movimentações
- ▶ **Complexidade de espaço/consumo espaço:** $\Theta(n)$

Um pouco de terminologia

- ▶ Um algoritmo A tem complexidade de tempo (no pior caso) $O(f(n))$ se para qualquer entrada de tamanho n ele gasta tempo no máximo $O(f(n))$.
- ▶ Um algoritmo A tem complexidade de tempo no pior caso $\Theta(f(n))$ se para qualquer entrada de tamanho n ele gasta tempo no máximo $O(f(n))$ e para alguma entrada de tamanho n ele gasta tempo pelo menos $\Omega(f(n))$.
- ▶ Por exemplo, INSERTION-SORT tem complexidade de tempo no pior caso $\Theta(n^2)$.

Um pouco de terminologia

- ▶ Um algoritmo A tem complexidade de tempo (no **pior caso**) $O(f(n))$ se para **qualquer** entrada de tamanho n ele gasta tempo no **máximo** $O(f(n))$.
- ▶ Um algoritmo A tem complexidade de tempo no pior caso $\Theta(f(n))$ se para qualquer entrada de tamanho n ele gasta tempo no máximo $O(f(n))$ e para alguma entrada de tamanho n ele gasta tempo pelo menos $\Omega(f(n))$.
- ▶ Por exemplo, **INSERTION-SORT** tem complexidade de tempo no **pior caso** $\Theta(n^2)$.

Um pouco de terminologia

- ▶ Um algoritmo A tem complexidade de tempo (no **pior caso**) $O(f(n))$ se para **qualquer** entrada de tamanho n ele gasta tempo no **máximo** $O(f(n))$.
- ▶ Um algoritmo A tem complexidade de tempo no pior caso $\Theta(f(n))$ se para qualquer entrada de tamanho n ele gasta tempo no **máximo** $O(f(n))$ e para alguma entrada de tamanho n ele gasta tempo pelo menos $\Omega(f(n))$.
- ▶ Por exemplo, **INSERTION-SORT** tem complexidade de tempo no pior caso $\Theta(n^2)$.

Um pouco de terminologia

- ▶ Um algoritmo A tem complexidade de tempo (no **pior caso**) $O(f(n))$ se para **qualquer** entrada de tamanho n ele gasta tempo no **máximo** $O(f(n))$.
- ▶ Um algoritmo A tem complexidade de tempo no pior caso $\Theta(f(n))$ se para qualquer entrada de tamanho n ele gasta tempo no **máximo** $O(f(n))$ e para alguma entrada de tamanho n ele gasta tempo pelo menos $\Omega(f(n))$.
- ▶ Por exemplo, **INSERTION-SORT** tem complexidade de tempo no **pior caso** $\Theta(n^2)$.

Um pouco de terminologia

- ▶ Faz sentido dizer que um algoritmo tem complexidade de tempo no **pior caso** pelo menos $O(f(n))$?
- ▶ Faz sentido dizer que um algoritmo tem complexidade de tempo no **pior caso** $\Omega(f(n))$?
- ▶ Faz sentido dizer que um algoritmo tem complexidade de tempo no **melhor caso** $\Omega(f(n))$?

Um pouco de terminologia

- ▶ Faz sentido dizer que um algoritmo tem complexidade de tempo no **pior caso** pelo menos $O(f(n))$?
- ▶ Faz sentido dizer que um algoritmo tem complexidade de tempo no **pior caso** $\Omega(f(n))$?
- ▶ Faz sentido dizer que um algoritmo tem complexidade de tempo no **melhor caso** $\Omega(f(n))$?

Um pouco de terminologia

- ▶ Faz sentido dizer que um algoritmo tem complexidade de tempo no **pioor caso** pelo menos $O(f(n))$?
- ▶ Faz sentido dizer que um algoritmo tem complexidade de tempo no **pioor caso** $\Omega(f(n))$?
- ▶ Faz sentido dizer que um algoritmo tem complexidade de tempo no **melhor caso** $\Omega(f(n))$?

Um pouco de terminologia

- ▶ Faz sentido dizer que um algoritmo tem complexidade de tempo no **pior caso** pelo menos $O(f(n))$?
- ▶ Faz sentido dizer que um algoritmo tem complexidade de tempo no **pior caso** $\Omega(f(n))$?
- ▶ Faz sentido dizer que um algoritmo tem complexidade de tempo no **melhor caso** $\Omega(f(n))$?

Selection Sort

Selection sort

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

A cada passo selecionamos o menor elemento em $A[i \dots n]$ e o colocamos em $A[i]$.

- ▶ Repetimos o processo para $i = 1, \dots, n - 1$ e ordenamos vetor.

Selection sort

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

A cada passo selecionamos o menor elemento em $A[i \dots n]$ e o colocamos em $A[i]$.

- ▶ Repetimos o processo para $i = 1, \dots, n - 1$ e ordenamos vetor.

Selection sort

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

A cada passo selecionamos o menor elemento em $A[i \dots n]$ e o colocamos em $A[i]$.

- ▶ Repetimos o processo para $i = 1, \dots, n - 1$ e ordenamos vetor.

Selection sort

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

A cada passo selecionamos o menor elemento em $A[i \dots n]$ e o colocamos em $A[i]$.

- ▶ Repetimos o processo para $i = 1, \dots, n - 1$ e ordenamos vetor.

Selection sort

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

A cada passo selecionamos o menor elemento em $A[i \dots n]$ e o colocamos em $A[i]$.

- ▶ Repetimos o processo para $i = 1, \dots, n - 1$ e ordenamos vetor.

Selection sort

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

A cada passo selecionamos o menor elemento em $A[i \dots n]$ e o colocamos em $A[i]$.

- ▶ Repetimos o processo para $i = 1, \dots, n - 1$ e ordenamos vetor.

Selection sort – pseudocódigo

SELECTION-SORT(A, n)

```
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       $A[i] \leftrightarrow A[min]$ 
```

Invariantes:

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

Selection sort – pseudocódigo

SELECTION-SORT(A, n)

```
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       $A[i] \leftrightarrow A[min]$ 
```

Invariantes:

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

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

- ▶ 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: $\Theta(n)$

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: $\Theta(n)$**

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: $\Theta(n)$**

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:** $\Theta(n)$

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**.

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.

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**.

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**.

Merge Sort

Vimos que 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.

Vimos que 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.

Vimos que 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.

Vimos que 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 – pseudocódigo

```
MERGE-SORT( $A, p, r$ )  
1   se  $p < r$   
2     então  $q \leftarrow \lfloor (p + r)/2 \rfloor$   
3         MERGE-SORT( $A, p, q$ )  
4         MERGE-SORT( $A, q + 1, r$ )  
5         INTERCALA( $A, p, q, r$ )
```

A complexidade de MERGE-SORT 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.

Mergesort – pseudocódigo

```
MERGE-SORT( $A, p, r$ )  
1   se  $p < r$   
2     então  $q \leftarrow \lfloor (p + r)/2 \rfloor$   
3         MERGE-SORT( $A, p, q$ )  
4         MERGE-SORT( $A, q + 1, r$ )  
5         INTERCALA( $A, p, q, r$ )
```

A complexidade de **MERGE-SORT** 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**.

Correção do Mergesort

```
MERGE-SORT( $A, p, r$ )
1  se  $p < r$ 
2    então  $q \leftarrow \lfloor (p + r)/2 \rfloor$ 
3          MERGE-SORT( $A, p, q$ )
4          MERGE-SORT( $A, q + 1, r$ )
5          INTERCALA( $A, p, q, r$ )
```

O algoritmo está correto?

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

Você consegue ver por quê?

Correção do Mergesort

```
MERGE-SORT( $A, p, r$ )
1  se  $p < r$ 
2    então  $q \leftarrow \lfloor (p + r)/2 \rfloor$ 
3          MERGE-SORT( $A, p, q$ )
4          MERGE-SORT( $A, q + 1, r$ )
5          INTERCALA( $A, p, q, r$ )
```

O algoritmo está correto?

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

Você consegue ver por quê?

Correção do Mergesort

```
MERGE-SORT( $A, p, r$ )
1  se  $p < r$ 
2    então  $q \leftarrow \lfloor (p + r)/2 \rfloor$ 
3          MERGE-SORT( $A, p, q$ )
4          MERGE-SORT( $A, q + 1, r$ )
5          INTERCALA( $A, p, q, r$ )
```

O algoritmo está correto?

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

Você consegue ver por quê?

Correção do Mergesort

```
MERGE-SORT( $A, p, r$ )
1  se  $p < r$ 
2    então  $q \leftarrow \lfloor (p + r)/2 \rfloor$ 
3          MERGE-SORT( $A, p, q$ )
4          MERGE-SORT( $A, q + 1, r$ )
5          INTERCALA( $A, p, q, r$ )
```

O algoritmo está correto?

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

Você consegue ver por quê?

Correção do Mergesort

Base: Mergesort ordena vetores de tamanho 0 ou 1.

Hipótese de indução: Mergesort ordena vetores com $< n$ 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 correção de Intercala, segue que o vetor resultante da intercalação é um vetor ordenado de n elementos.

Correção do Mergesort

Base: Mergesort ordena vetores de tamanho 0 ou 1.

Hipótese de indução: Mergesort ordena vetores com $< n$ 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 correção de Intercala, segue que o vetor resultante da intercalação é um vetor ordenado de n elementos.

Correção do Mergesort

Base: Mergesort ordena vetores de tamanho 0 ou 1.

Hipótese de indução: Mergesort ordena vetores com $< n$ 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 correção de Intercala, segue que o vetor resultante da intercalação é um vetor ordenado de n elementos.

Correção do Mergesort

Base: Mergesort ordena vetores de tamanho 0 ou 1.

Hipótese de indução: Mergesort ordena vetores com $< n$ 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 correção de Intercala, segue que o vetor resultante da intercalação é um vetor ordenado de n elementos.

Correção do Mergesort

Base: Mergesort ordena vetores de tamanho 0 ou 1.

Hipótese de indução: Mergesort ordena vetores com $< n$ 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 correção de Intercala, segue que o vetor resultante da intercalação é um vetor ordenado de n elementos.

Complexidade de Mergesort

```
MERGE-SORT( $A, p, r$ )  
1  se  $p < r$   
2    então  $q \leftarrow \lfloor (p + r)/2 \rfloor$   
3        MERGE-SORT( $A, p, q$ )  
4        MERGE-SORT( $A, q + 1, r$ )  
5        INTERCALA( $A, p, q, r$ )
```

$T(n)$: complexidade de pior caso de MERGE-SORT.
Então

$$T(n) = T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + \Theta(n).$$

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

Complexidade de Mergesort

```
MERGE-SORT( $A, p, r$ )  
1  se  $p < r$   
2    então  $q \leftarrow \lfloor (p + r)/2 \rfloor$   
3        MERGE-SORT( $A, p, q$ )  
4        MERGE-SORT( $A, q + 1, r$ )  
5        INTERCALA( $A, p, q, r$ )
```

$T(n)$: complexidade de pior caso de MERGE-SORT.

Então

$$T(n) = T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + \Theta(n).$$

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

Complexidade de Mergesort

```
MERGE-SORT( $A, p, r$ )
1  se  $p < r$ 
2    então  $q \leftarrow \lfloor (p + r)/2 \rfloor$ 
3          MERGE-SORT( $A, p, q$ )
4          MERGE-SORT( $A, q + 1, r$ )
5          INTERCALA( $A, p, q, r$ )
```

$T(n)$: complexidade de pior caso de MERGE-SORT.

Então

$$T(n) = T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + \Theta(n).$$

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

Complexidade de Mergesort

```
MERGE-SORT( $A, p, r$ )
1  se  $p < r$ 
2    então  $q \leftarrow \lfloor (p + r)/2 \rfloor$ 
3          MERGE-SORT( $A, p, q$ )
4          MERGE-SORT( $A, q + 1, r$ )
5          INTERCALA( $A, p, q, r$ )
```

$T(n)$: complexidade de pior caso de MERGE-SORT.

Então

$$T(n) = T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + \Theta(n).$$

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

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 útil para **ordenação externa**, quando não é possível armazenar todos os elementos na memória primária.

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 útil para **ordenação externa**, quando não é possível armazenar todos os elementos na memória primária.

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 útil para **ordenação externa**, quando não é possível armazenar todos os elementos na memória primária.

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 útil para **ordenação externa**, quando não é possível armazenar todos os elementos na memória primária.

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 útil para **ordenação externa**, quando não é possível armazenar todos os elementos na memória primária.

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 útil para **ordenação externa**, quando não é possível armazenar todos os elementos na memória primária.

Heap Sort

Heapsort

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

Heapsort

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

Heapsort

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

Heapsort

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

Heapsort

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

Heaps

- ▶ Considere um vetor $A[1 \dots n]$ representando um **heap**.
- ▶ Cada posição do vetor corresponde a um **nó** do **heap**.

Pais

- ▶ O **pai** de um nó i é $\lfloor i/2 \rfloor$.
- ▶ O nó 1 não tem pai.

- ▶ Considere um vetor $A[1 \dots n]$ representando um heap.
- ▶ Cada posição do vetor corresponde a um nó do heap.

Pais

- ▶ O pai de um nó i é $\lfloor i/2 \rfloor$.
- ▶ O nó 1 não tem pai.

Heaps

- ▶ Considere um vetor $A[1 \dots n]$ representando um heap.
- ▶ Cada posição do vetor corresponde a um nó do heap.

Pais

- ▶ O pai de um nó i é $\lfloor i/2 \rfloor$.
- ▶ O nó 1 não tem pai.

Heaps

- ▶ Considere um vetor $A[1 \dots n]$ representando um heap.
- ▶ Cada posição do vetor corresponde a um nó do heap.

Pais

- ▶ O pai de um nó i é $\lfloor i/2 \rfloor$.
- ▶ O nó 1 não tem pai.

Heaps

- ▶ Considere um vetor $A[1 \dots n]$ representando um heap.
- ▶ Cada posição do vetor corresponde a um nó do heap.

Pais

- ▶ O pai de um nó i é $\lfloor i/2 \rfloor$.
- ▶ O nó 1 não tem pai.

Heaps

- ▶ Considere um vetor $A[1 \dots n]$ representando um heap.
- ▶ Cada posição do vetor corresponde a um nó do heap.

Pais

- ▶ O pai de um nó i é $\lfloor i/2 \rfloor$.
- ▶ O nó 1 não tem pai.

Filhos

Um nó i tem

- ▶ $2i$ como filho esquerdo e
 - ▶ $2i + 1$ como filho direito.
-
- ▶ O nó i tem filho esquerdo apenas se $2i \leq n$ e
 - ▶ O nó i tem filho direito apenas se $2i + 1 \leq n$.

Folhas

- ▶ Um nó i é uma **folha** se não tem filhos, ou seja, se $2i > n$.
- ▶ As folhas são $\lfloor n/2 \rfloor + 1, \dots, n - 1, n$.

Filhos

Um nó i tem

- ▶ $2i$ como filho esquerdo e
 - ▶ $2i + 1$ como filho direito.
-
- ▶ O nó i tem filho esquerdo apenas se $2i \leq n$ e
 - ▶ O nó i tem filho direito apenas se $2i + 1 \leq n$.

Folhas

- ▶ Um nó i é uma **folha** se não tem filhos, ou seja, se $2i > n$.
- ▶ As folhas são $\lfloor n/2 \rfloor + 1, \dots, n - 1, n$.

Filhos

Um nó i tem

- ▶ $2i$ como filho esquerdo e
 - ▶ $2i + 1$ como filho direito.
-
- ▶ O nó i tem filho esquerdo apenas se $2i \leq n$ e
 - ▶ O nó i tem filho direito apenas se $2i + 1 \leq n$.

Folhas

- ▶ Um nó i é uma **folha** se não tem filhos, ou seja, se $2i > n$.
- ▶ As folhas são $\lfloor n/2 \rfloor + 1, \dots, n - 1, n$.

Filhos

Um nó i tem

- ▶ $2i$ como filho esquerdo e
 - ▶ $2i + 1$ como filho direito.
-
- ▶ O nó i tem filho esquerdo apenas se $2i \leq n$ e
 - ▶ O nó i tem filho direito apenas se $2i + 1 \leq n$.

Folhas

- ▶ Um nó i é uma **folha** se não tem filhos, ou seja, se $2i > n$.
- ▶ As folhas são $\lfloor n/2 \rfloor + 1, \dots, n - 1, n$.

Filhos

Um nó i tem

- ▶ $2i$ como filho esquerdo e
 - ▶ $2i + 1$ como filho direito.
-
- ▶ O nó i tem filho esquerdo apenas se $2i \leq n$ e
 - ▶ O nó i tem filho direito apenas se $2i + 1 \leq n$.

Folhas

- ▶ Um nó i é uma **folha** se não tem filhos, ou seja, se $2i > n$.
- ▶ As folhas são $\lfloor n/2 \rfloor + 1, \dots, n - 1, n$.

Filhos

Um nó i tem

- ▶ $2i$ como filho esquerdo e
 - ▶ $2i + 1$ como filho direito.
-
- ▶ O nó i tem filho esquerdo apenas se $2i \leq n$ e
 - ▶ O nó i tem filho direito apenas se $2i + 1 \leq n$.

Folhas

- ▶ Um nó i é uma **folha** se não tem filhos, ou seja, se $2i > n$.
- ▶ As folhas são $\lfloor n/2 \rfloor + 1, \dots, n - 1, n$.

Filhos

Um nó i tem

- ▶ $2i$ como filho esquerdo e
 - ▶ $2i + 1$ como filho direito.
-
- ▶ O nó i tem filho esquerdo apenas se $2i \leq n$ e
 - ▶ O nó i tem filho direito apenas se $2i + 1 \leq n$.

Folhas

- ▶ Um nó i é uma **folha** se não tem filhos, ou seja, se $2i > n$.
- ▶ As folhas são $\lfloor n/2 \rfloor + 1, \dots, n - 1, n$.

Filhos

Um nó i tem

- ▶ $2i$ como filho esquerdo e
 - ▶ $2i + 1$ como filho direito.
-
- ▶ O nó i tem filho esquerdo apenas se $2i \leq n$ e
 - ▶ O nó i tem filho direito apenas se $2i + 1 \leq n$.

Folhas

- ▶ Um nó i é uma **folha** se não tem filhos, ou seja, se $2i > n$.
- ▶ As folhas são $\lfloor n/2 \rfloor + 1, \dots, n - 1, n$.

Níveis

- ▶ Cada nível p , exceto talvez o último, tem exatamente 2^p nós
- ▶ Esses nós são

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

Nível do item i

- ▶ O nó i pertence ao nível $\lfloor \lg i \rfloor$.
- ▶ Prova: Se p é o nível do nó i , então

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

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

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

Níveis

- ▶ Cada nível p , exceto talvez o último, tem exatamente 2^p nós
- ▶ Esses nós são

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

Nível do item i

- ▶ O nó i pertence ao nível $\lfloor \lg i \rfloor$.
- ▶ *Prova:* Se p é o nível do nó i , então

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

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

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

Níveis

- ▶ Cada nível p , exceto talvez o último, tem exatamente 2^p nós
- ▶ Esses nós são

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

Nível do item i

- ▶ O nó i pertence ao nível $\lfloor \lg i \rfloor$.
- ▶ *Prova:* Se p é o nível do nó i , então

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

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

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

Níveis

- ▶ Cada nível p , exceto talvez o último, tem exatamente 2^p nós
- ▶ Esses nós são

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

Nível do item i

- ▶ O nó i pertence ao nível $\lfloor \lg i \rfloor$.
- ▶ *Prova:* Se p é o nível do nó i , então

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

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

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

Níveis

- ▶ Cada nível p , exceto talvez o último, tem exatamente 2^p nós
- ▶ Esses nós são

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

Nível do item i

- ▶ O nó i pertence ao nível $\lfloor \lg i \rfloor$.
- ▶ *Prova:* Se p é o nível do nó i , então

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

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

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

Níveis

- ▶ Cada nível p , exceto talvez o último, tem exatamente 2^p nós
- ▶ Esses nós são

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

Nível do item i

- ▶ O nó i pertence ao nível $\lfloor \lg i \rfloor$.
- ▶ *Prova:* Se p é o nível do nó i , então

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

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

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

Níveis

- ▶ Cada nível p , exceto talvez o último, tem exatamente 2^p nós
- ▶ Esses nós são

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

Nível do item i

- ▶ O nó i pertence ao nível $\lfloor \lg i \rfloor$.
- ▶ **Prova:** Se p é o nível do nó i , então

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

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

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

Níveis

- ▶ Cada nível p , exceto talvez o último, tem exatamente 2^p nós
- ▶ Esses nós são

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

Nível do item i

- ▶ O nó i pertence ao nível $\lfloor \lg i \rfloor$.
- ▶ **Prova:** Se p é o nível do nó i , então

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

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

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

Níveis

- ▶ Cada nível p , exceto talvez o último, tem exatamente 2^p nós
- ▶ Esses nós são

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

Nível do item i

- ▶ O nó i pertence ao nível $\lfloor \lg i \rfloor$.
- ▶ **Prova:** Se p é o nível do nó i , então

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

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

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

Níveis

- ▶ Cada nível p , exceto talvez o último, tem exatamente 2^p nós
- ▶ Esses nós são

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

Nível do item i

- ▶ O nó i pertence ao nível $\lfloor \lg i \rfloor$.
- ▶ **Prova:** Se p é o nível do nó i , então

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

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

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

Níveis

- ▶ Cada nível p , exceto talvez o último, tem exatamente 2^p nós
- ▶ Esses nós são

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

Nível do item i

- ▶ O nó i pertence ao nível $\lfloor \lg i \rfloor$.
- ▶ **Prova:** Se p é o nível do nó i , então

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

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

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

Níveis

- ▶ Cada nível p , exceto talvez o último, tem exatamente 2^p nós
- ▶ Esses nós são

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

Nível do item i

- ▶ O nó i pertence ao nível $\lfloor \lg i \rfloor$.
- ▶ **Prova:** Se p é o nível do nó i , então

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

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

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

- ▶ 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 ?

- ▶ 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 ?

- ▶ 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 ?

- ▶ 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 ?

A altura de um nó i é o comprimento da sequência

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

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

Assim,

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

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

A altura de um nó i é o comprimento da sequência

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

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

Assim,

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

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

A altura de um nó i é o comprimento da sequência

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

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

Assim,

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

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

Max-heaps

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

Max-heaps

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

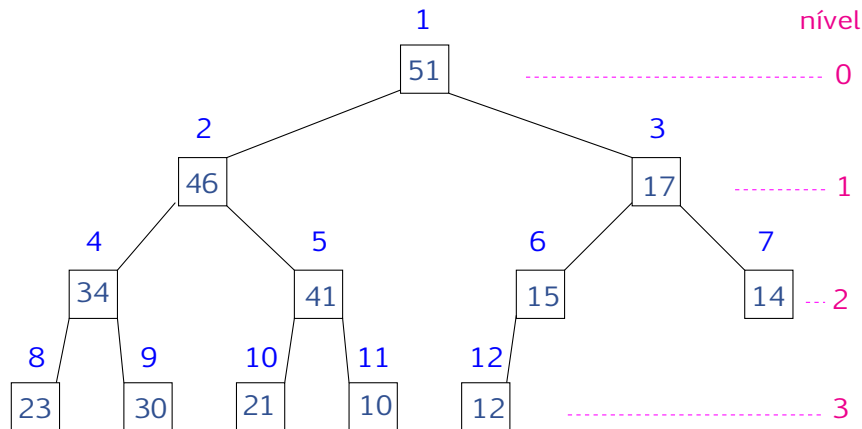
Max-heaps

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

Max-heaps

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

Max-heap



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

Min-heaps

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

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**.

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**.

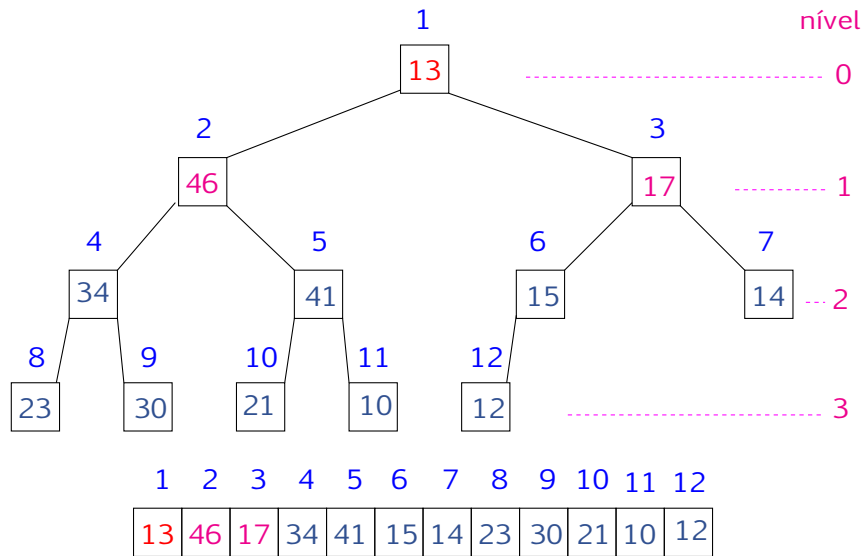
Min-heaps

- ▶ 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**.

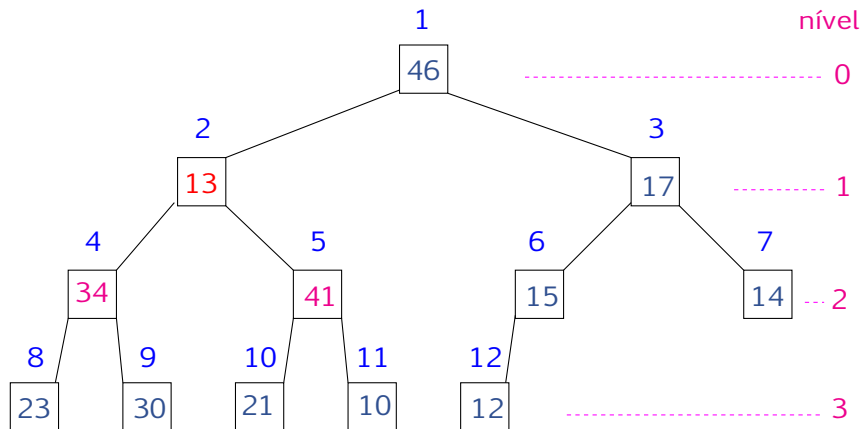
Min-heaps

- ▶ 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

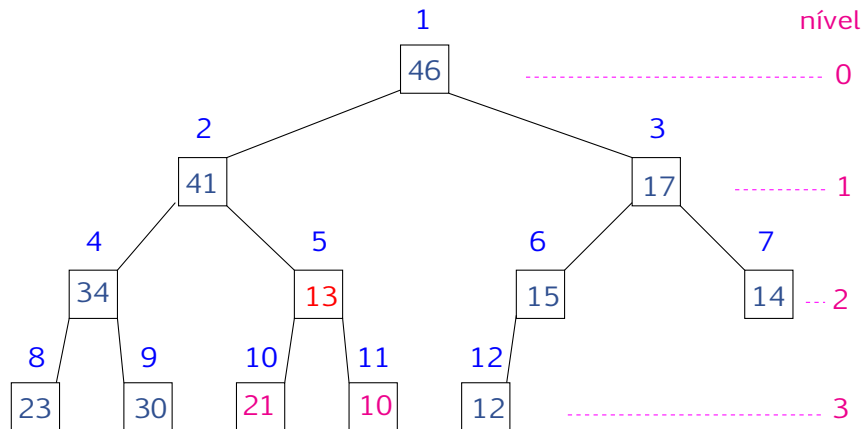


Manipulação de max-heap



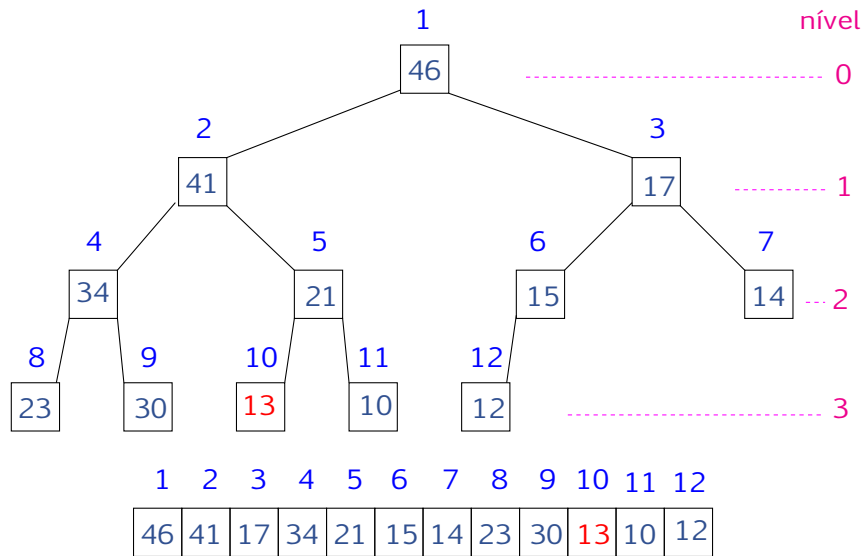
1	2	3	4	5	6	7	8	9	10	11	12
46	13	17	34	41	15	14	23	30	21	10	12

Manipulação de max-heap

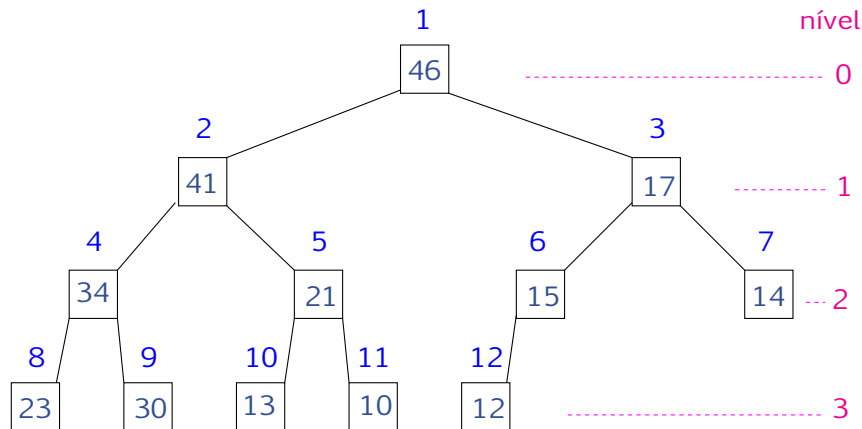


1	2	3	4	5	6	7	8	9	10	11	12
46	41	17	34	13	15	14	23	30	21	10	12

Manipulação de max-heap



Manipulação de max-heap



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

Manipulação de max-heap

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

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

Correção de MAX-HEAPIFY

A correção de MAX-HEAPIFY segue por indução na altura h do nó i .

Base: para $h = 0$, o algoritmo funciona.

Hipótese de indução: MAX-HEAPIFY funciona para heaps de altura $< h$.

Passo de indução:

A variável *maior* na linha 8 guarda o índice do maior elemento entre $A[i]$, $A[2i]$ e $A[2i + 1]$.

Após a troca na linha 9, temos $A[2i], A[2i + 1] \leq A[i]$.

O algoritmo MAX-HEAPIFY transforma a subárvore com raiz *maior* em um max-heap (hipótese de indução).

Correção de MAX-HEAPIFY

A correção de MAX-HEAPIFY segue por indução na altura h do nó i .

Base: para $h = 0$, o algoritmo funciona.

Hipótese de indução: MAX-HEAPIFY funciona para heaps de altura $< h$.

Passo de indução:

A variável *maior* na linha 8 guarda o índice do maior elemento entre $A[i]$, $A[2i]$ e $A[2i + 1]$.

Após a troca na linha 9, temos $A[2i], A[2i + 1] \leq A[i]$.

O algoritmo MAX-HEAPIFY transforma a subárvore com raiz *maior* em um max-heap (hipótese de indução).

Correção de MAX-HEAPIFY

A correção de MAX-HEAPIFY segue por indução na altura h do nó i .

Base: para $h = 0$, o algoritmo funciona.

Hipótese de indução: MAX-HEAPIFY funciona para heaps de altura $< h$.

Passo de indução:

A variável *maior* na linha 8 guarda o índice do maior elemento entre $A[i]$, $A[2i]$ e $A[2i + 1]$.

Após a troca na linha 9, temos $A[2i], A[2i + 1] \leq A[i]$.

O algoritmo MAX-HEAPIFY transforma a subárvore com raiz *maior* em um max-heap (hipótese de indução).

Correção de MAX-HEAPIFY

A correção de MAX-HEAPIFY segue por indução na altura h do nó i .

Base: para $h = 0$, o algoritmo funciona.

Hipótese de indução: MAX-HEAPIFY funciona para heaps de altura $< h$.

Passo de indução:

A variável *maior* na linha 8 guarda o índice do maior elemento entre $A[i]$, $A[2i]$ e $A[2i + 1]$.

Após a troca na linha 9, temos $A[2i], A[2i + 1] \leq A[i]$.

O algoritmo MAX-HEAPIFY transforma a subárvore com raiz *maior* em um max-heap (hipótese de indução).

Correção de MAX-HEAPIFY

A correção de MAX-HEAPIFY segue por indução na altura h do nó i .

Base: para $h = 0$, o algoritmo funciona.

Hipótese de indução: MAX-HEAPIFY funciona para heaps de altura $< h$.

Passo de indução:

A variável *maior* na linha 8 guarda o índice do maior elemento entre $A[i]$, $A[2i]$ e $A[2i + 1]$.

Após a troca na linha 9, temos $A[2i], A[2i + 1] \leq A[i]$.

O algoritmo MAX-HEAPIFY transforma a subárvore com raiz *maior* em um max-heap (hipótese de indução).

Correção de MAX-HEAPIFY

A correção de MAX-HEAPIFY segue por indução na altura h do nó i .

Base: para $h = 0$, o algoritmo funciona.

Hipótese de indução: MAX-HEAPIFY funciona para heaps de altura $< h$.

Passo de indução:

A variável *maior* na linha 8 guarda o índice do maior elemento entre $A[i]$, $A[2i]$ e $A[2i + 1]$.

Após a troca na linha 9, temos $A[2i], A[2i + 1] \leq A[i]$.

O algoritmo MAX-HEAPIFY transforma a subárvore com raiz *maior* em um max-heap (hipótese de indução).

Passo de indução:

A variável *maior* na linha 8 guarda o índice do maior elemento entre $A[i]$, $A[2i]$ e $A[2i + 1]$.

Após a troca na linha 9, temos $A[2i], A[2i + 1] \leq A[i]$.

O algoritmo MAX-HEAPIFY transforma a subárvore com raiz *maior* em um max-heap (hipótese de indução).

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

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

Passo de indução:

A variável *maior* na linha 8 guarda o índice do maior elemento entre $A[i]$, $A[2i]$ e $A[2i + 1]$.

Após a troca na linha 9, temos $A[2i], A[2i + 1] \leq A[i]$.

O algoritmo MAX-HEAPIFY transforma a subárvore com raiz *maior* em um max-heap (hipótese de indução).

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

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

Passo de indução:

A variável *maior* na linha 8 guarda o índice do maior elemento entre $A[i]$, $A[2i]$ e $A[2i + 1]$.

Após a troca na linha 9, temos $A[2i], A[2i + 1] \leq A[i]$.

O algoritmo MAX-HEAPIFY transforma a subárvore com raiz *maior* em um max-heap (hipótese de indução).

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

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

Complexidade de MAX-HEAPIFY

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

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 $A[d] > A[maior]$	$\Theta(1)$
7 então $maior \leftarrow d$	$O(1)$
8 se $maior \neq i$	$\Theta(1)$
9 então $A[i] \leftrightarrow A[maior]$	$O(1)$
10 MAX-HEAPIFY($A, n, maior$)	$T(h - 1)$

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

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

Complexidade de MAX-HEAPIFY

$h :=$ 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 ?

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

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

Complexidade de MAX-HEAPIFY

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 ?

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

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

Complexidade de MAX-HEAPIFY

$h :=$ 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, $O(\lg \frac{n}{i})$).

Complexidade de MAX-HEAPIFY

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

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

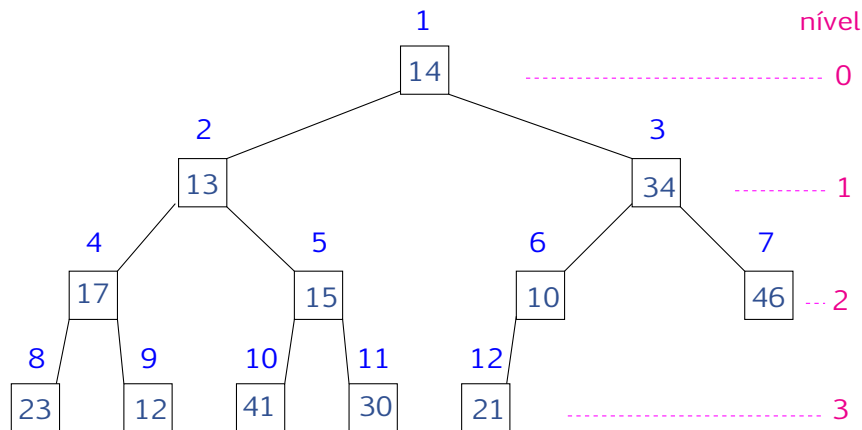
$$T(h) \leq T(h-1) + \Theta(1)$$

Solução assintótica: $T(n)$ é $O(h)$.

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

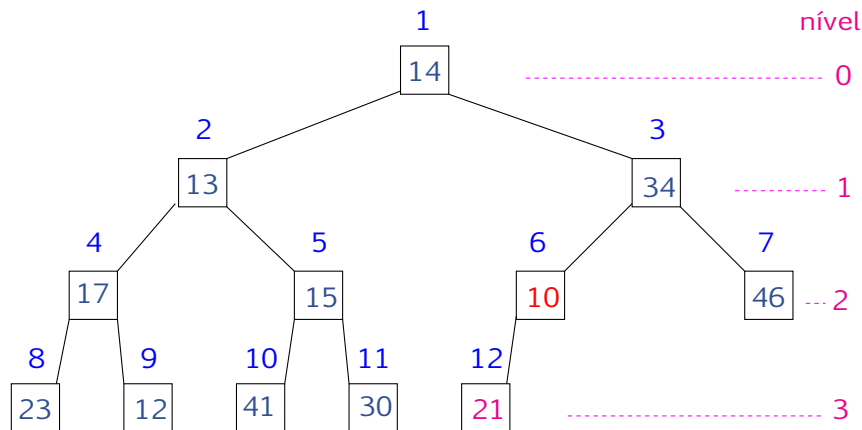
O consumo de tempo do algoritmo MAX-HEAPIFY é $O(\lg n)$
(ou melhor ainda, $O(\lg \frac{n}{i})$).

Construção de um max-heap



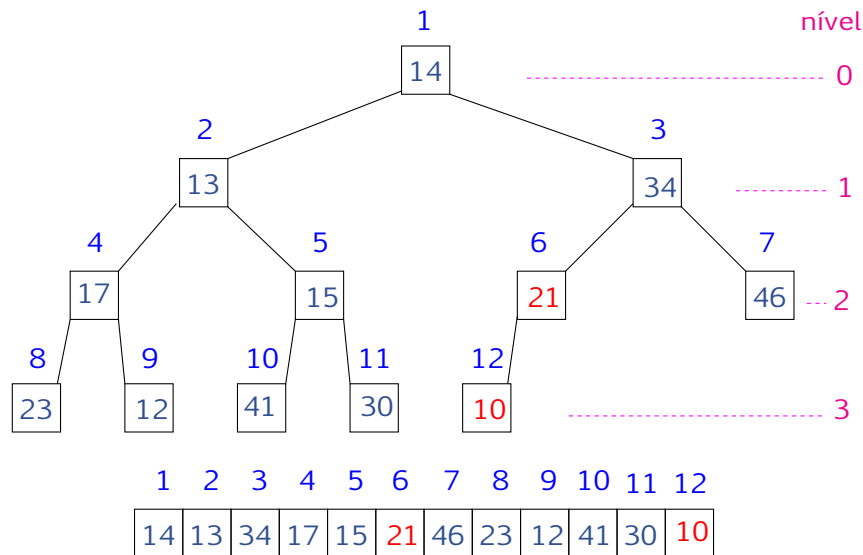
1	2	3	4	5	6	7	8	9	10	11	12
14	13	34	17	15	10	46	23	12	41	30	21

Construção de um max-heap

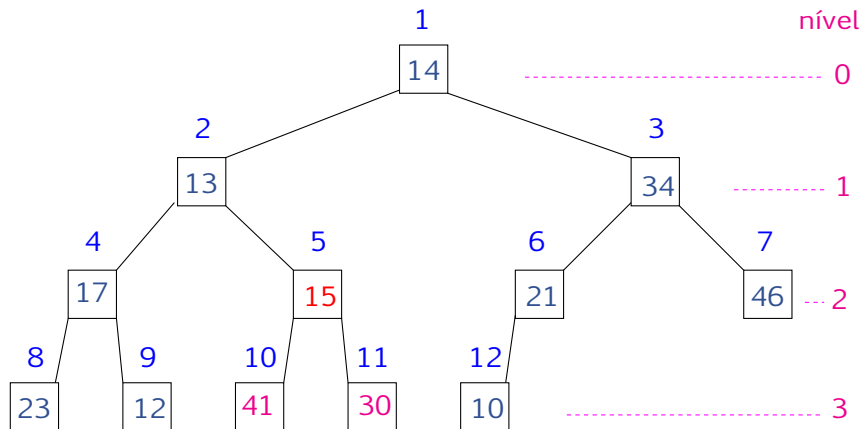


1	2	3	4	5	6	7	8	9	10	11	12
14	13	34	17	15	10	46	23	12	41	30	21

Construção de um max-heap

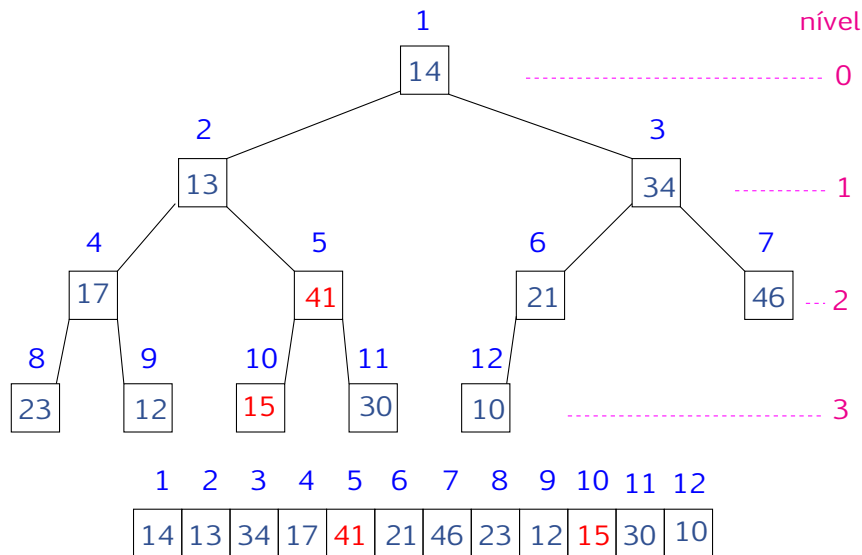


Construção de um max-heap

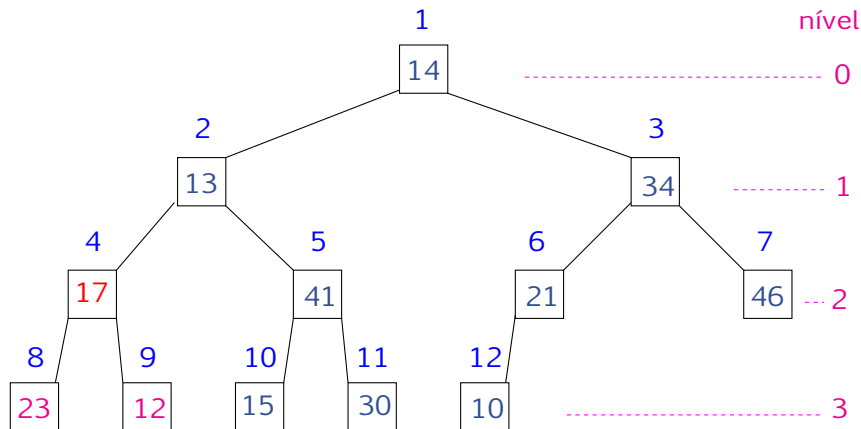


1	2	3	4	5	6	7	8	9	10	11	12
14	13	34	17	15	21	46	23	12	41	30	10

Construção de um max-heap

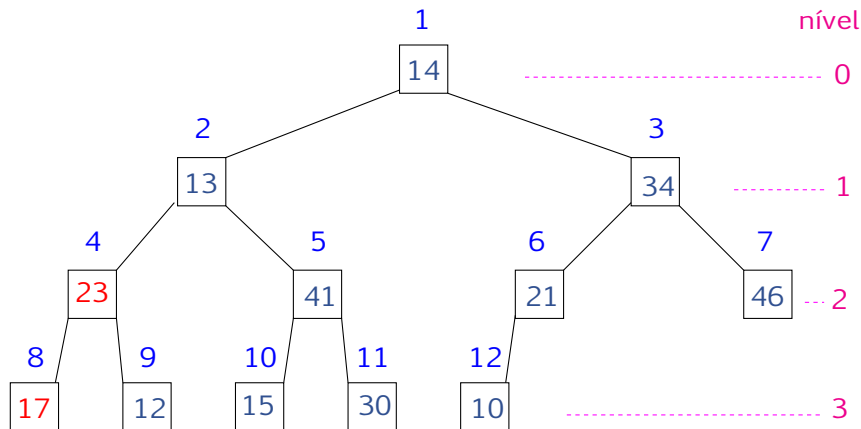


Construção de um max-heap



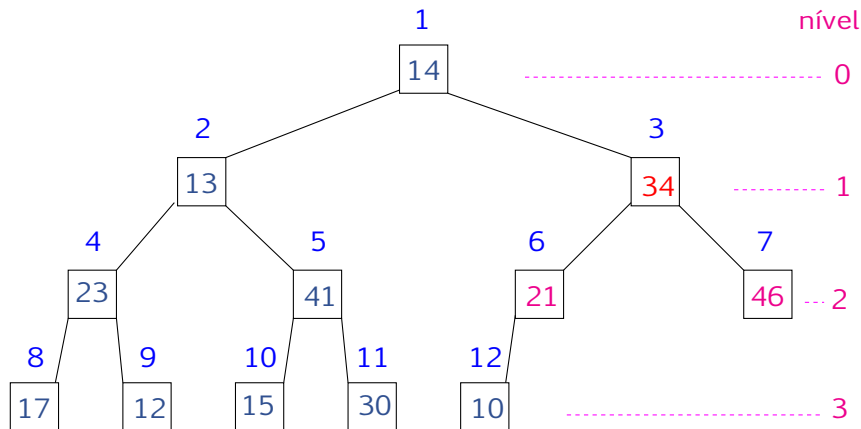
1	2	3	4	5	6	7	8	9	10	11	12
14	13	34	17	41	21	46	23	12	15	30	10

Construção de um max-heap



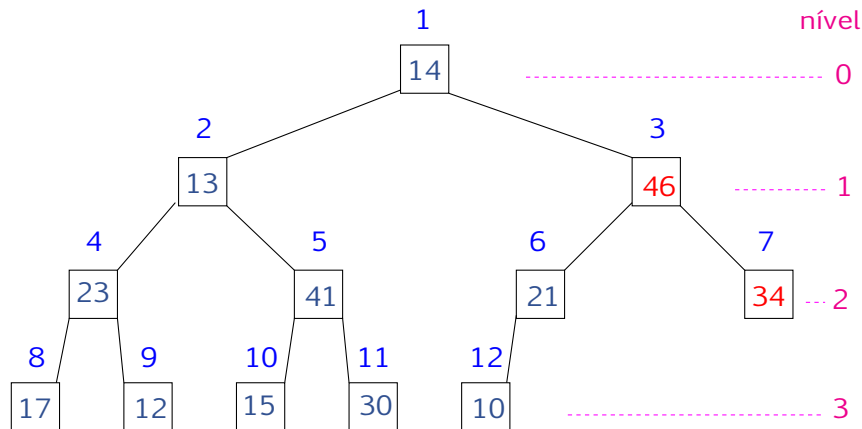
1	2	3	4	5	6	7	8	9	10	11	12
14	13	34	23	41	21	46	17	12	15	30	10

Construção de um max-heap



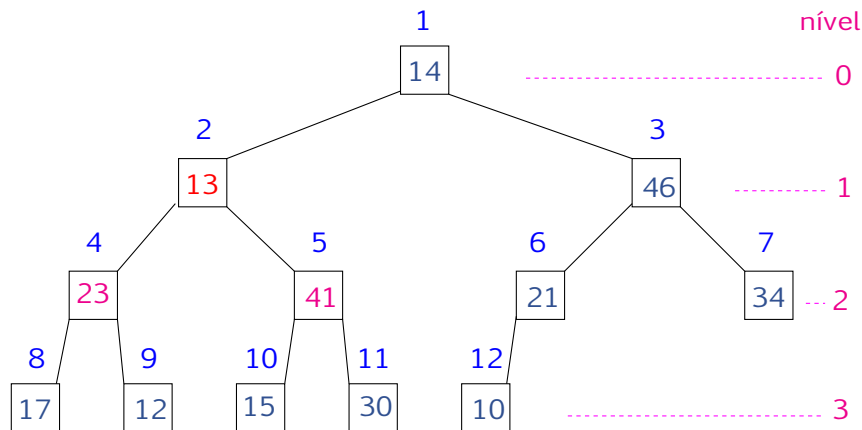
1	2	3	4	5	6	7	8	9	10	11	12
14	13	34	23	41	21	46	17	12	15	30	10

Construção de um max-heap



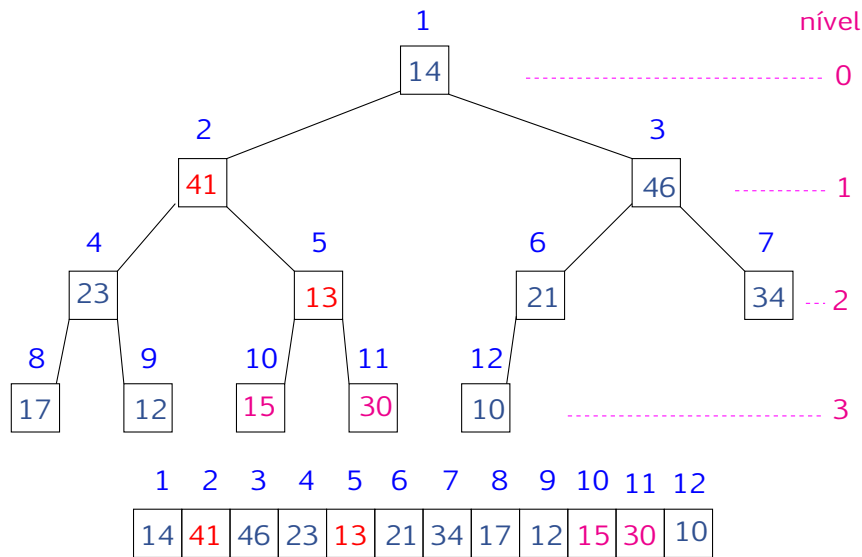
1	2	3	4	5	6	7	8	9	10	11	12
14	13	46	23	41	21	34	17	12	15	30	10

Construção de um max-heap

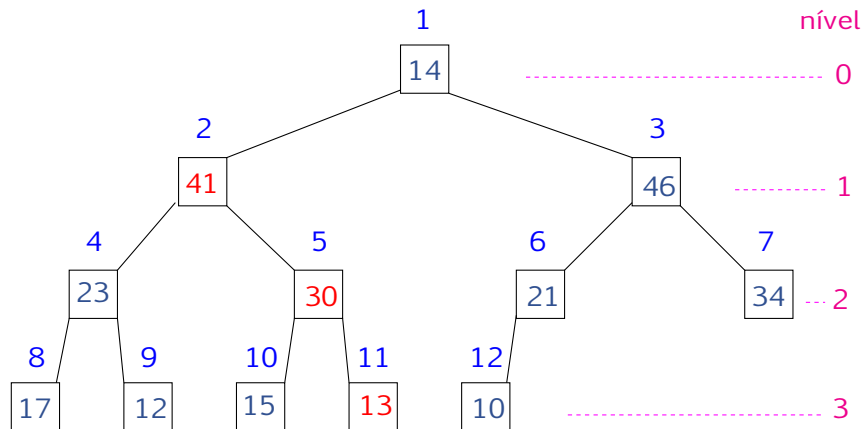


1	2	3	4	5	6	7	8	9	10	11	12
14	13	46	23	41	21	34	17	12	15	30	10

Construção de um max-heap

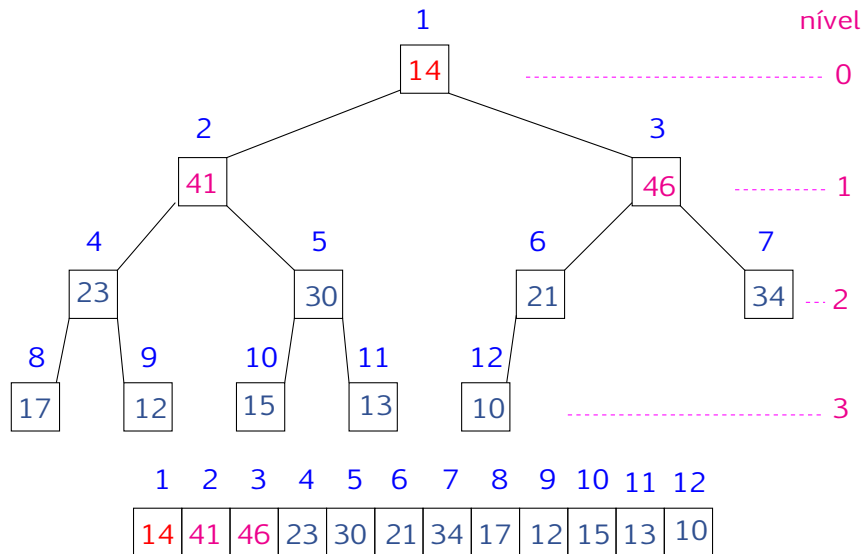


Construção de um max-heap

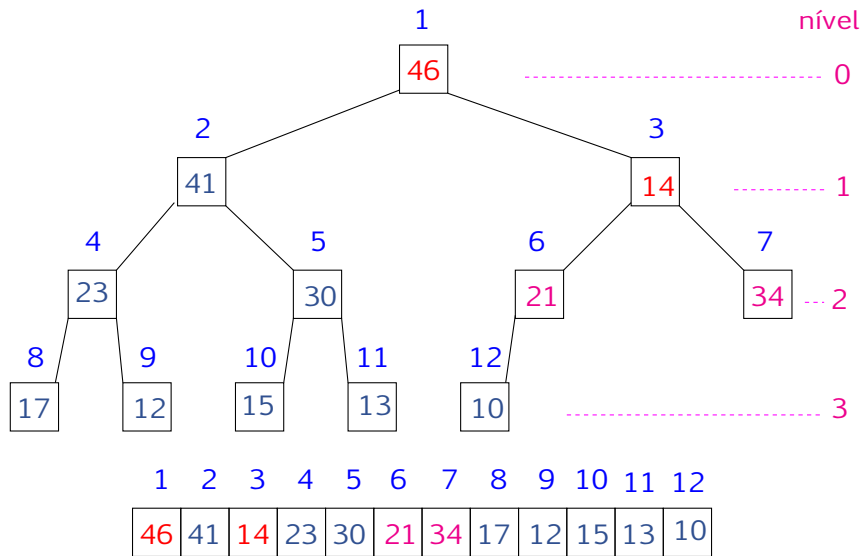


1	2	3	4	5	6	7	8	9	10	11	12
14	41	46	23	30	21	34	17	12	15	13	10

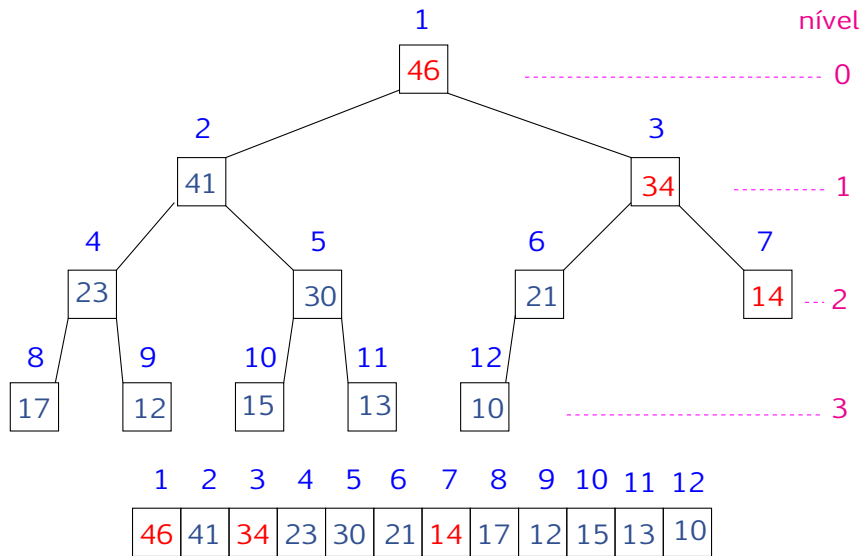
Construção de um max-heap



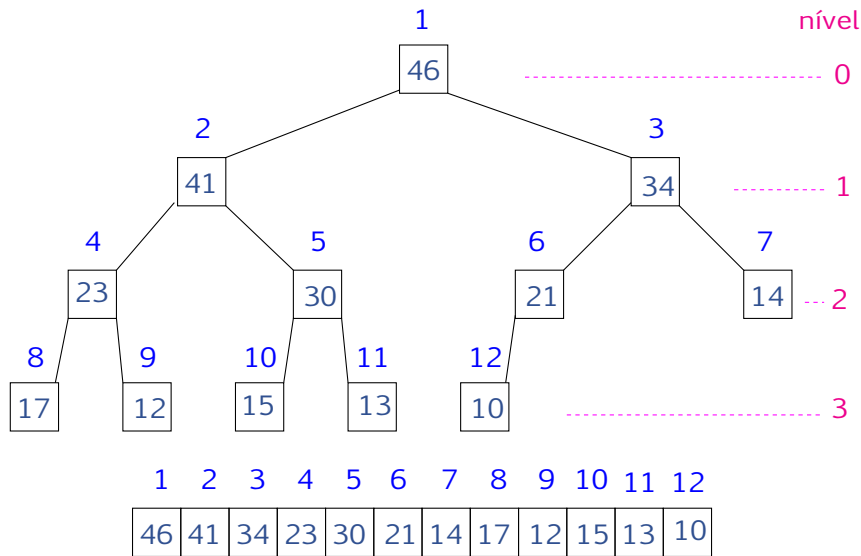
Construção de um max-heap



Construção de um max-heap



Construção de um max-heap



Construção de um max-heap

Recebe um vetor $A[1 \dots n]$ 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 $A[1 \dots n]$ 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, $i + 1, \dots, n$ são raízes de max-heaps.

Construção de um max-heap

Recebe um vetor $A[1 \dots n]$ 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, $i + 1, \dots, n$ são raízes de max-heaps.

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

Construção de um max-heap

Recebe um vetor $A[1 \dots n]$ 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, $i + 1, \dots, n$ são raízes de max-heaps.

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

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

Construção de um max-heap

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

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

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

Construção de um max-heap

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

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

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

Construção de um max-heap

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

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

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

Construção de um max-heap

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

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

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

Construção de um max-heap

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

Mais precisamente, $T(n) = \Theta(n)$. (Por quê?)

- ▶ Veja no CLRS uma prova diferente deste fato.

Construção de um max-heap

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

Construção de um max-heap

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

Mais precisamente, $T(n) = \Theta(n)$. (Por quê?)

- ▶ Veja no CLRS uma prova diferente deste fato.

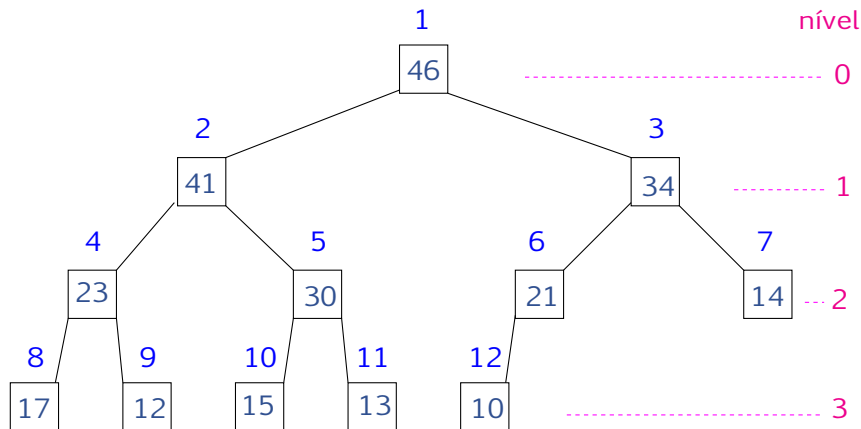
Construção de um max-heap

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

Construção de um max-heap

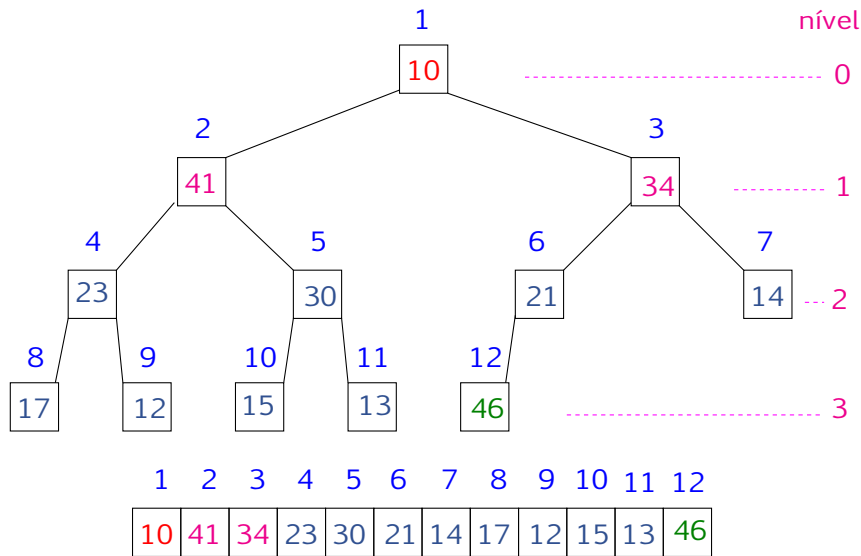
- ▶ Pode-se provar por indução que $S(h) = 2^{h+1} - h - 2$.
- ▶ Logo, a complexidade de BUILD-MAX-HEAP é $T(n) = O(S(\lg n)) = O(n)$.
Mais precisamente, $T(n) = \Theta(n)$. (Por quê?)
- ▶ Veja no CLRS uma prova diferente deste fato.

HeapSort

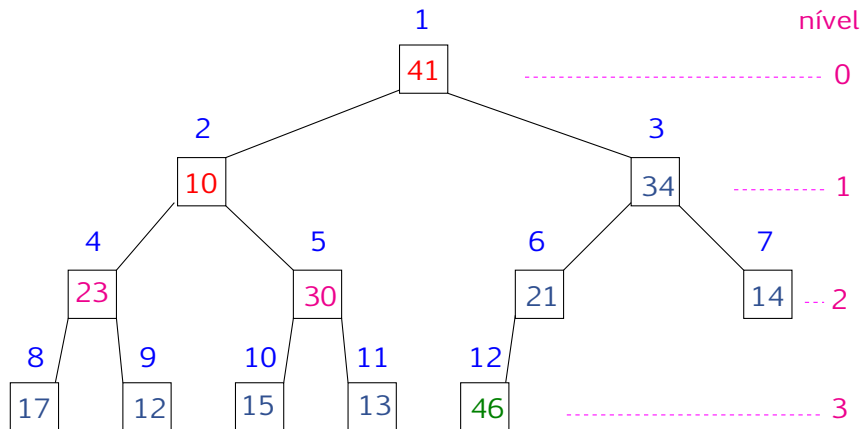


1	2	3	4	5	6	7	8	9	10	11	12
46	41	34	23	30	21	14	17	12	15	13	10

HeapSort

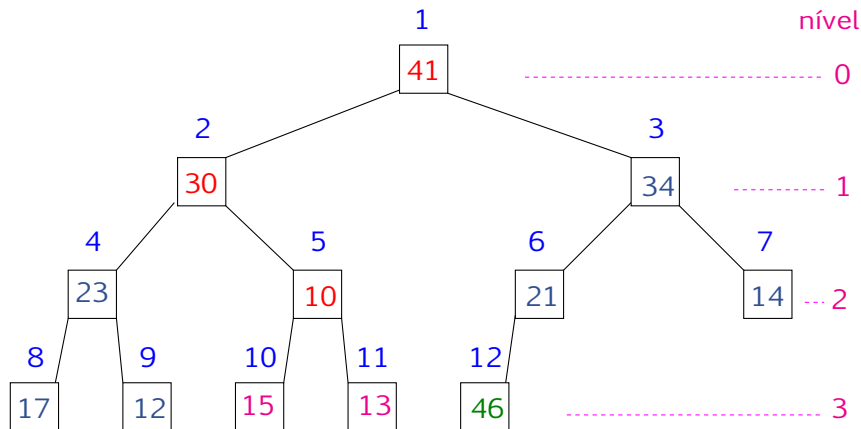


HeapSort



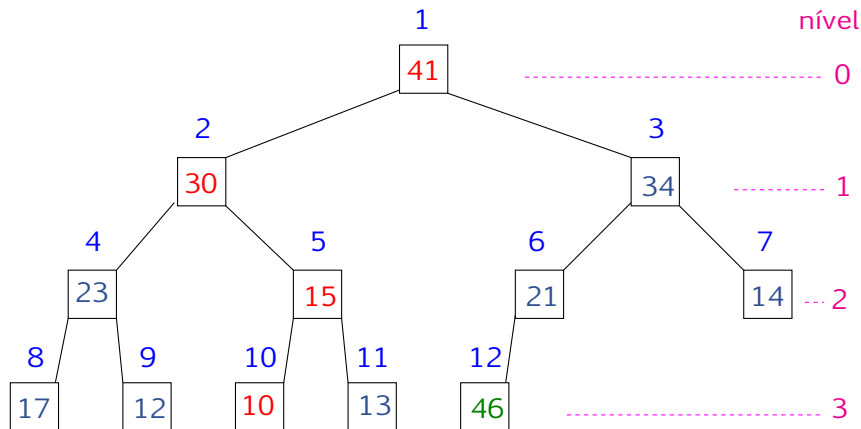
1	2	3	4	5	6	7	8	9	10	11	12
41	10	34	23	30	21	14	17	12	15	13	46

HeapSort



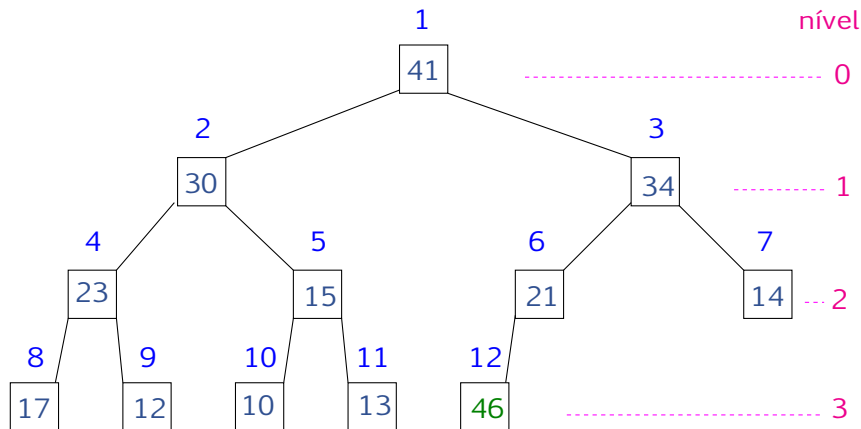
1	2	3	4	5	6	7	8	9	10	11	12
41	30	34	23	10	21	14	17	12	15	13	46

HeapSort



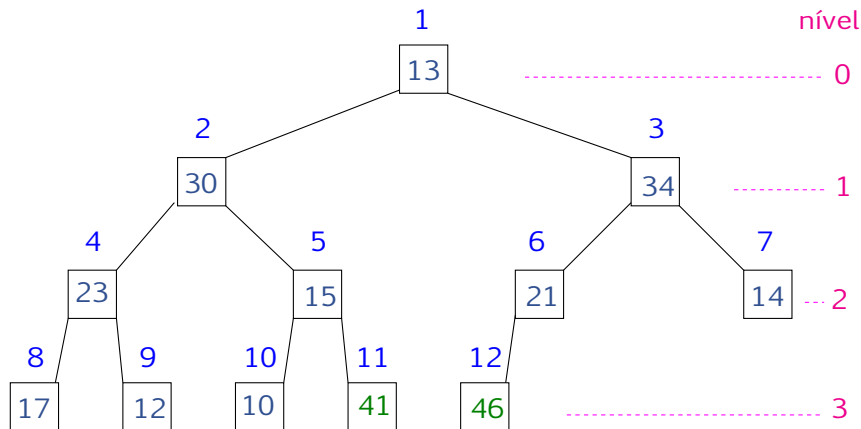
1	2	3	4	5	6	7	8	9	10	11	12
41	30	34	23	15	21	14	17	12	10	13	46

HeapSort



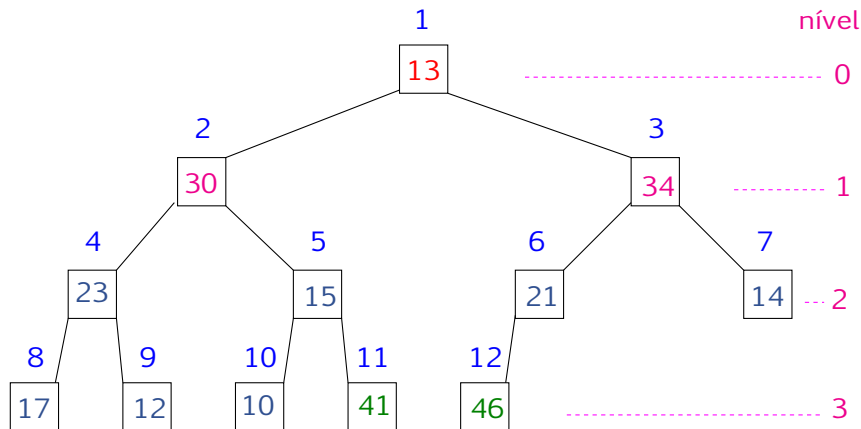
1	2	3	4	5	6	7	8	9	10	11	12
41	30	34	23	15	21	14	17	12	10	13	46

HeapSort



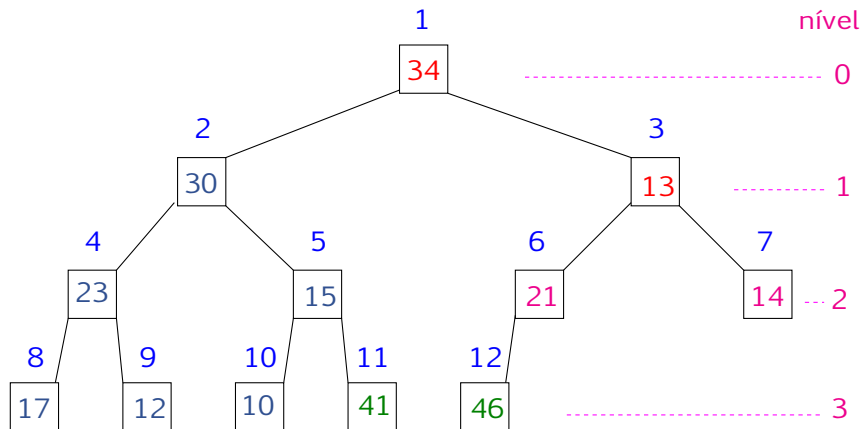
1	2	3	4	5	6	7	8	9	10	11	12
13	30	34	23	15	21	14	17	12	10	41	46

HeapSort



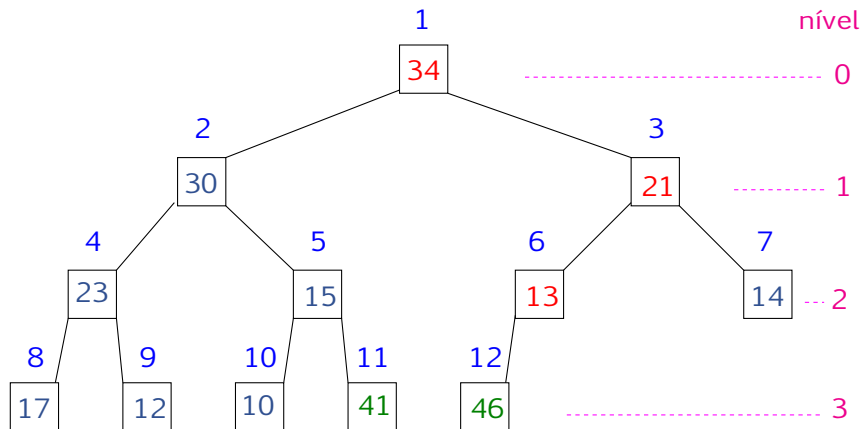
1	2	3	4	5	6	7	8	9	10	11	12
13	30	34	23	15	21	14	17	12	10	41	46

HeapSort



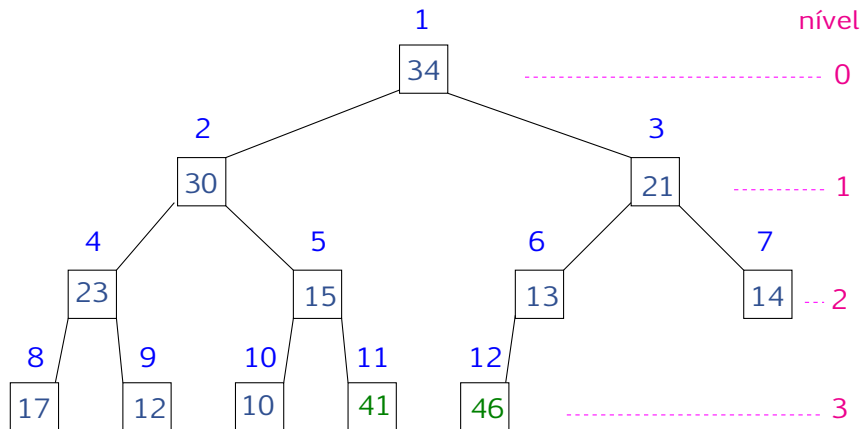
1	2	3	4	5	6	7	8	9	10	11	12
34	30	13	23	15	21	14	17	12	10	41	46

HeapSort



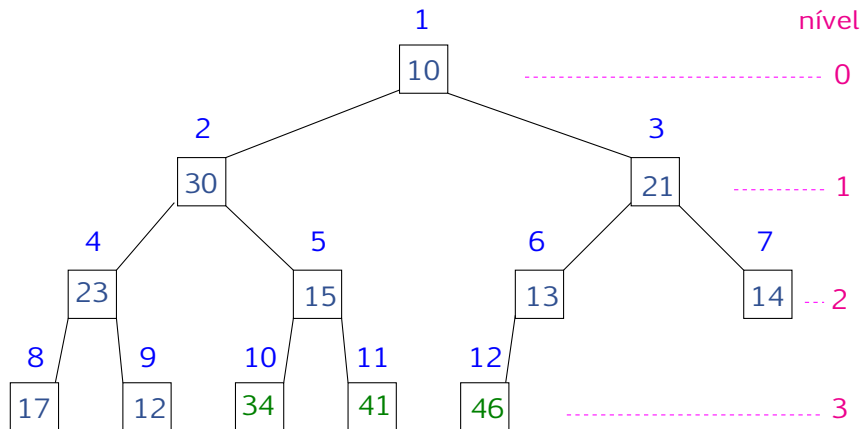
1	2	3	4	5	6	7	8	9	10	11	12
34	30	21	23	15	13	14	17	12	10	41	46

HeapSort



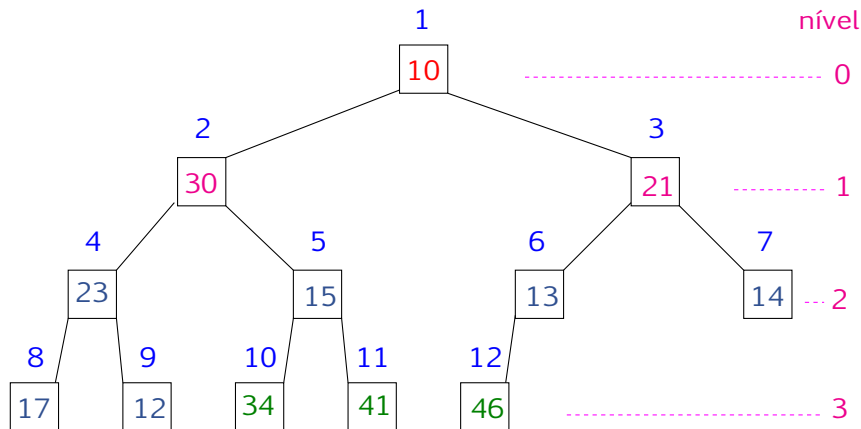
1	2	3	4	5	6	7	8	9	10	11	12
34	30	21	23	15	13	14	17	12	10	41	46

HeapSort



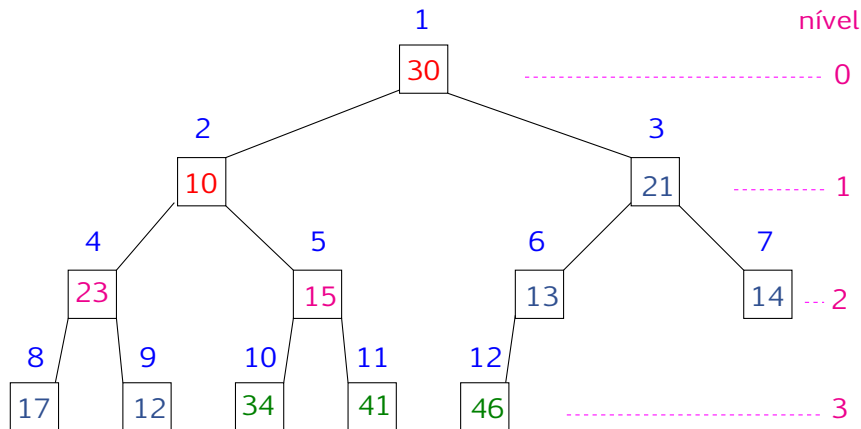
1	2	3	4	5	6	7	8	9	10	11	12
10	30	21	23	15	13	14	17	12	34	41	46

HeapSort



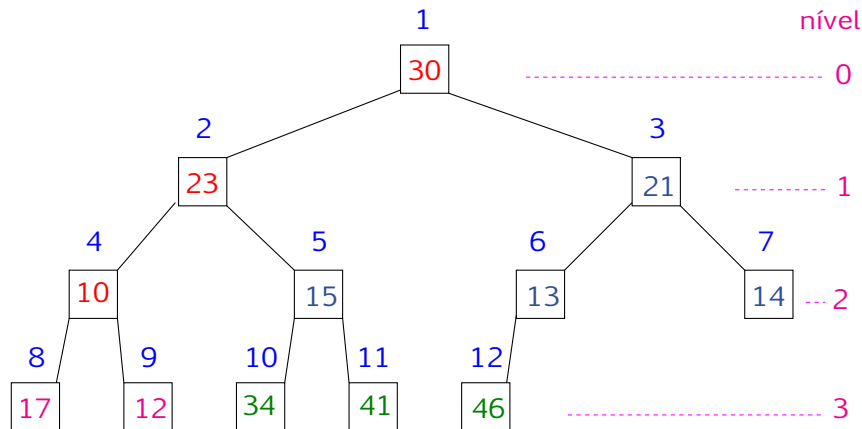
1	2	3	4	5	6	7	8	9	10	11	12
10	30	21	23	15	13	14	17	12	34	41	46

HeapSort



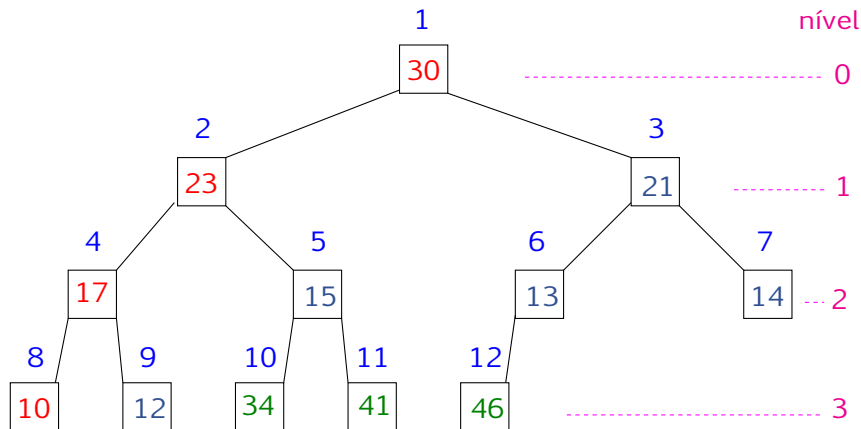
1	2	3	4	5	6	7	8	9	10	11	12
30	10	21	23	15	13	14	17	12	34	41	46

HeapSort



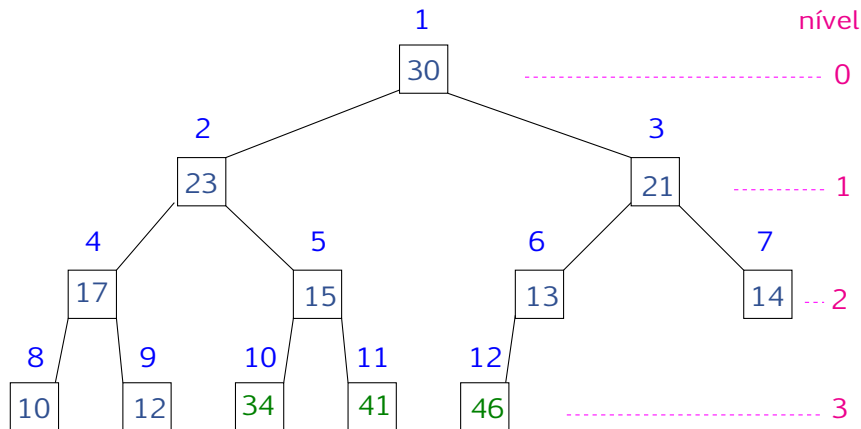
1	2	3	4	5	6	7	8	9	10	11	12
30	23	21	10	15	13	14	17	12	34	41	46

HeapSort



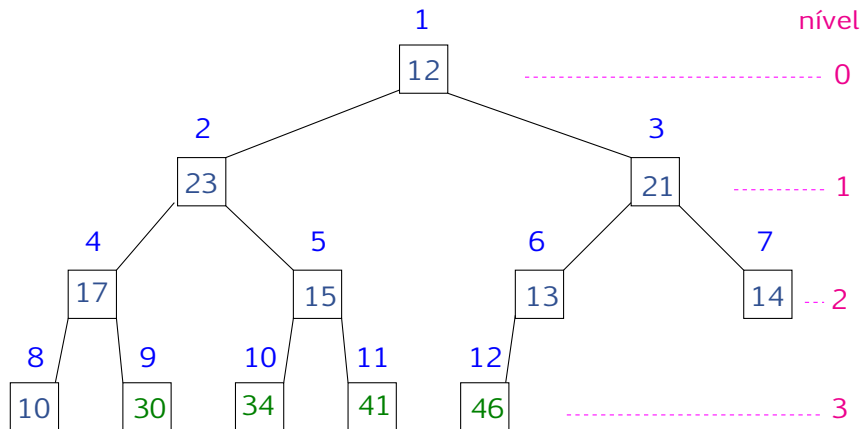
1	2	3	4	5	6	7	8	9	10	11	12
30	23	21	17	15	13	14	10	12	34	41	46

HeapSort



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

HeapSort



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

HeapSort

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

HEAP-SORT(A, n)

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

Invariantes:

No início de cada iteração na linha 3 vale que:

1. $A[m + 1 \dots n]$ é crescente e contém os $n - m$ maiores elementos de $A[1 \dots n]$;
2. $A[1 \dots m] \leq A[m + 1]$;
3. $A[1 \dots m]$ é um max-heap.

HeapSort

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

HEAP-SORT(A, n)

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

Invariantes:

No início de cada iteração na linha 3 vale que:

1. $A[m + 1 \dots n]$ é crescente e contém os $n - m$ maiores elementos de $A[1 \dots n]$;
2. $A[1 \dots m] \leq A[m + 1]$;
3. $A[1 \dots m]$ é um max-heap.

HeapSort

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

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

A complexidade de HEAP-SORT no pior caso é $O(n \lg n)$.

Como seria a complexidade de tempo no melhor caso?

HeapSort

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

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

A complexidade de HEAP-SORT no pior caso é $O(n \lg n)$.

Como seria a complexidade de tempo no melhor caso?

HeapSort

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

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

A complexidade de HEAP-SORT no pior caso é $O(n \lg n)$.

Como seria a complexidade de tempo no melhor caso?

HeapSort

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

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

A complexidade de HEAP-SORT no pior caso é $O(n \lg n)$.

Como seria a complexidade de tempo no melhor caso?

HeapSort

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

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

A complexidade de HEAP-SORT no pior caso é $O(n \lg n)$.

Como seria a complexidade de tempo no melhor caso?

Filas com prioridades

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

Algumas operações típicas em uma fila com prioridades são:

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

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

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

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

Implementação com max-heap

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

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

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

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

Implementação com max-heap

HEAP-INCREASE-KEY($A, i, chave$)

- 1 \triangleright Supõe que $chave \geq A[i]$
- 2 $A[i] \leftarrow chave$
- 3 enquanto $i > 1$ e $A[\lfloor i/2 \rfloor] < A[i]$ faça
- 4 $A[i] \leftrightarrow A[\lfloor i/2 \rfloor]$
- 5 $i \leftarrow \lfloor i/2 \rfloor$

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

MAX-HEAP-INSERT($A, n, chave$)

- 1 $n \leftarrow n + 1$
- 2 $A[n] \leftarrow -\infty$
- 3 **HEAP-INCREASE-KEY**($A, n, chave$)

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