Projeto e Análise de Algoritmos

Fila de prioridade

Lehilton Pedrosa

Primeiro Semestre de 2020

Selection Sort

Selecionando o próximo menor

Vamos ordenar usando ideia diferente

- suponha que o subvetor $A[1 \dots i-1]$ está ordenado
- também, suponha que $\max A[1 \dots i-1] \leq \min A[i \dots n]$.
- substituímos a posição $A[i]$ pelo mínimo em $A[i \dots n]$

Antes de substituir:

1						i				n
20	25	35	40	44	55	70	80	99	65	85

Após substituir:

1						i				n
20	25	35	40	44	55	65	80	99	70	85

Selecionando o próximo menor

Vamos ordenar usando ideia diferente

- suponha que o subvetor $A[1 \ldots i-1]$ está ordenado
- também, suponha que $\max A[1 \ldots i-1] \leq \min A[i \ldots n]$.
- substituímos a posição $A[i]$ pelo mínimo em $A[i \ldots n]$

Antes de substituir:

1						i				n
20	25	35	40	44	55	70	80	99	65	85

Após substituir:

1						i				n
20	25	35	40	44	55	65	80	99	70	85

Selecionando o próximo menor

Vamos ordenar usando ideia diferente

- suponha que o subvetor $A[1 \dots i-1]$ está ordenado
- também, suponha que $\max A[1 \dots i-1] \leq \min A[i \dots n]$.
- substituímos a posição $A[i]$ pelo mínimo em $A[i \dots n]$

Antes de substituir:

1						i				n
20	25	35	40	44	55	70	80	99	65	85

Após substituir:

1						i				n
20	25	35	40	44	55	65	80	99	70	85

Selecionando o próximo menor

Vamos ordenar usando ideia diferente

- suponha que o subvetor $A[1 \dots i-1]$ está ordenado
- também, suponha que $\max A[1 \dots i-1] \leq \min A[i \dots n]$.
- substituímos a posição $A[i]$ pelo mínimo em $A[i \dots n]$

Antes de substituir:

1						i				n
20	25	35	40	44	55	70	80	99	65	85

Após substituir:

1						i				n
20	25	35	40	44	55	65	80	99	70	85

Selecionando o próximo menor

Vamos ordenar usando ideia diferente

- suponha que o subvetor $A[1 \ldots i-1]$ está ordenado
- também, suponha que $\max A[1 \ldots i-1] \leq \min A[i \ldots n]$.
- substituímos a posição $A[i]$ pelo mínimo em $A[i \ldots n]$

Antes de substituir:

1						i				n
20	25	35	40	44	55	70	80	99	65	85

Após substituir:

1						i				n
20	25	35	40	44	55	65	80	99	70	85

Selecionando o próximo menor

Vamos ordenar usando ideia diferente

- suponha que o subvetor $A[1 \dots i-1]$ está ordenado
- também, suponha que $\max A[1 \dots i-1] \leq \min A[i \dots n]$.
- substituímos a posição $A[i]$ pelo mínimo em $A[i \dots n]$

Antes de substituir:

1						i				n
20	25	35	40	44	55	70	80	99	65	85

Após substituir:

1						i				n
20	25	35	40	44	55	65	80	99	70	85

Pseudocódigo de Selection-Sort

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

Invariante

Ao início de cada iteração:

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

Pseudocódigo de Selection-Sort

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

Invariante

Ao início de cada iteração:

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

Pseudocódigo de SELECTION-SORT

SELECTION-SORT(A, n)

```
para i ← 1 até n − 1 faça
    min ← i
    para j ← i + 1 até n faça
        se A[j] < A[min] então min ← j
    A[i] ↔ A[min]
```

Invariante

Ao início de cada iteração:

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

Pseudocódigo de SELECTION-SORT

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

Invariante

Ao início de cada iteração:

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
para $i \leftarrow 1$ **até** $n-1$ **faça**	?
$\quad min \leftarrow i$	?
$\quad$ **para** $j \leftarrow i+1$ **até** n **faça**	?
$\quad\quad$ **se** $A[j] < A[min]$ **então** $min \leftarrow j$	?
$\quad A[i] \leftrightarrow A[min]$	?

Consumo de tempo no pior caso? $\Theta(n^2)$
E no melhor caso?

Complexidade de SELECTION-SORT

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

Consumo de tempo no pior caso? $\Theta(n^2)$
E no melhor caso?

Complexidade de SELECTION-SORT

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

Consumo de tempo no pior caso? $\Theta(n^2)$

E no melhor caso?

Complexidade de SELECTION-SORT

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

Consumo de tempo no pior caso? $\Theta(n^2)$

E no melhor caso?

Complexidade de SELECTION-SORT

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

Consumo de tempo no pior caso? $\Theta(n^2)$

E no melhor caso?

Uma versão alternativa

Podemos reescrever esse algoritmo

- ordenamos a partir do final
- selecionamos o maior remanescente
- refatoramos com uma sub-rotina MAXIMUM

Uma versão alternativa

Podemos reescrever esse algoritmo

- ordenamos a partir do final
- selecionamos o maior remanescente
- refatoramos com uma sub-rotina Maximum

Uma versão alternativa

Podemos reescrever esse algoritmo

- ordenamos a partir do final
- selecionamos o maior remanescente
- refatoramos com uma sub-rotina Maximum

Uma versão alternativa

Podemos reescrever esse algoritmo

- ordenamos a partir do final
- selecionamos o maior remanescente
- refatoramos com uma sub-rotina MAXIMUM

Uma versão alternativa

Podemos reescrever esse algoritmo

- ordenamos a partir do final
- selecionamos o maior remanescente
- refatoramos com uma sub-rotina MAXIMUM

SELECTION-SORT(A, n)

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

  $max \leftarrow$ MAXIMUM(A, i)

  $A[i] \leftrightarrow A[max]$

Revendo a complexidade

```
SELECTION-SORT(A, n)
    para i ← n decrescendo até 2 faça
        max ← MAXIMUM(A, i)
        A[i] ↔ A[max]
```

- suponha que MAXIMUM(A, i) leva tempo $O(t(i))$
- então o tempo total é

$$T(n) = \sum_{i=2}^{n} O(t(i)) \leq \sum_{i=2}^{n} O(t(n)) = O(n \cdot t(n))$$

- vamos tentar otimizar a rotina MAXIMUM(A, i)

Revendo a complexidade

```
SELECTION-SORT(A, n)
    para i ← n decrescendo até 2 faça
        max ← MAXIMUM(A, i)
        A[i] ↔ A[max]
```

- suponha que MAXIMUM(A, i) leva tempo $O(t(i))$
- então o tempo total é

$$T(n) = \sum_{i=2}^{n} O(t(i)) \leq \sum_{i=2}^{n} O(t(n)) = O(n \cdot t(n))$$

- vamos tentar otimizar a rotina MAXIMUM(A, i)

Revendo a complexidade

```
SELECTION-SORT(A, n)
    para i ← n decrescendo até 2 faça
        max ← MAXIMUM(A, i)
        A[i] ↔ A[max]
```

- suponha que MAXIMUM(A, i) leva tempo $O(t(i))$
- então o tempo total é

$$T(n) = \sum_{i=2}^{n} O(t(i)) \leq \sum_{i=2}^{n} O(t(n)) = O(n \cdot t(n))$$

- vamos tentar otimizar a rotina MAXIMUM(A, i)

Revendo a complexidade

```
SELECTION-SORT(A, n)
    para i ← n decrescendo até 2 faça
        max ← MAXIMUM(A, i)
        A[i] ↔ A[max]
```

- suponha que MAXIMUM(A, i) leva tempo $O(t(i))$
- então o tempo total é

$$T(n) = \sum_{i=2}^{n} O(t(i)) \leq \sum_{i=2}^{n} O(t(n)) = O(n \cdot t(n))$$

- vamos tentar otimizar a rotina MAXIMUM(A, i)

Revendo a complexidade

$\textsc{Selection-Sort}(A, n)$

para $i \leftarrow n$ **decrescendo até** 2 **faça**
  $max \leftarrow \textsc{Maximum}(A, i)$
  $A[i] \leftrightarrow A[max]$

- suponha que $\textsc{Maximum}(A, i)$ leva tempo $O(t(i))$
- então o tempo total é

$$T(n) = \sum_{i=2}^{n} O(t(i)) \leq \sum_{i=2}^{n} O(t(n)) = O(n \cdot t(n))$$

- vamos tentar otimizar a rotina $\textsc{Maximum}(A, i)$

Heapsort

O algoritmo HEAP-SORT

Vamos estudar a fila de prioridade

- é uma estrutura de dados também chamada de max-heap
- implementa MAXIMUM com tempo $O(\log n)$
- usando um heap, podemos ordenar em $O(n \log n)$
- esse algoritmo de ordenação é chamado de heapsort

Representando um heap

Um heap é uma árvore binária armazenada em vetor $A[1 \dots n]$

- Filhos de um nó i
  - o filho esquerdo é $2i$
  - o filho direito é $2i + 1$
- Pais
  - o pai de um nó i é $\lfloor i/2 \rfloor$
  - o nó 1 não tem pai.
- Folhas
  - um nó i é folha se não tiver filhos, i.e., se $2i > n$.
  - as folhas são $\lfloor n/2 \rfloor + 1, \dots, n - 1, n$.

Representando um heap

Um heap é uma árvore binária armazenada em vetor $A[1 \dots n]$

- Filhos de um nó i
  - o filho esquerdo é $2i$
  - o filho direito é $2i + 1$
- Pais
  - o pai de um nó i é $\lfloor i/2 \rfloor$
  - o nó 1 não tem pai.
- Folhas
  - um nó i é folha se não tiver filhos, i.e., se $2i > n$.
  - as folhas são $\lfloor n/2 \rfloor + 1, \dots, n - 1, n$.

Representando um heap

Um heap é uma árvore binária armazenada em vetor $A[1 \ldots n]$

- Filhos de um nó i
  - o filho esquerdo é $2i$
  - o filho direito é $2i + 1$
- Pais
  - o pai de um nó i é $\lfloor i/2 \rfloor$
  - o nó 1 não tem pai.
- Folhas
  - um nó i é folha se não tiver filhos, i.e., se $2i > n$.
  - as folhas são $\lfloor n/2 \rfloor + 1, \ldots, n-1, n$.

Representando um heap

Um heap é uma árvore binária armazenada em vetor $A[1 \dots n]$

- Filhos de um nó i
  - o filho esquerdo é $2i$
  - o filho direito é $2i + 1$
- Pais
  - o pai de um nó i é $\lfloor i/2 \rfloor$
  - o nó 1 não tem pai.
- Folhas
  - um nó i é folha se não tiver filhos, i.e., se $2i > n$.
  - as folhas são $\lfloor n/2 \rfloor + 1, \dots, n-1, n$.

Representando um heap

Um heap é uma árvore binária armazenada em vetor $A[1 \ldots n]$

- Filhos de um nó i
  - o filho esquerdo é $2i$
  - o filho direito é $2i + 1$

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

- Folhas
  - um nó i é folha se não tiver filhos, i.e., se $2i > n$.
  - as folhas são $\lfloor n/2 \rfloor + 1, \ldots, n - 1, n$.

Representando um heap

Um heap é uma árvore binária armazenada em vetor $A[1 \dots n]$

- Filhos de um nó i
  - o filho esquerdo é $2i$
  - o filho direito é $2i + 1$
- Pais
  - o pai de um nó i é $\lfloor i/2 \rfloor$
  - o nó 1 não tem pai.
- Folhas
  - um nó i é folha se não tiver filhos, i.e., se $2i > n$.
  - as folhas são $\lfloor n/2 \rfloor + 1, \dots, n-1, n$.

Representando um heap

Um heap é uma árvore binária armazenada em vetor $A[1 \dots n]$

- Filhos de um nó i
  - o filho esquerdo é $2i$
  - o filho direito é $2i + 1$
- Pais
  - o pai de um nó i é $\lfloor i/2 \rfloor$
  - o nó 1 não tem pai.
- Folhas
  - um nó i é folha se não tiver filhos, i.e., se $2i > n$.
  - as folhas são $\lfloor n/2 \rfloor + 1, \dots, n-1, n$.

Representando um heap

Um heap é uma árvore binária armazenada em vetor $A[1 \dots n]$

- Filhos de um nó i
  - o filho esquerdo é $2i$
  - o filho direito é $2i + 1$
- Pais
  - o pai de um nó i é $\lfloor i/2 \rfloor$
  - o nó 1 não tem pai.
- Folhas
  - um nó i é folha se não tiver filhos, i.e., se $2i > n$.
  - as folhas são $\lfloor n/2 \rfloor + 1, \dots, n-1, n$.

Representando um heap

Um heap é uma árvore binária armazenada em vetor $A[1 \ldots n]$

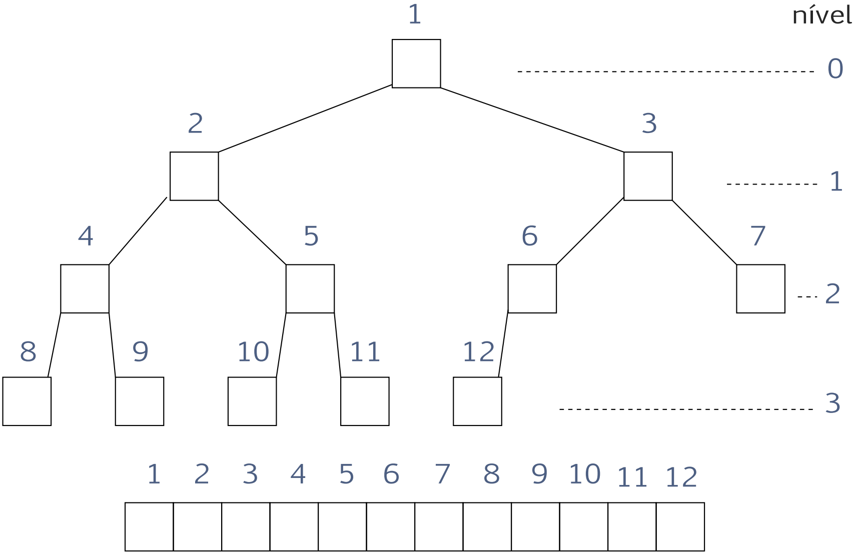
- Filhos de um nó i
  - o filho esquerdo é $2i$
  - o filho direito é $2i + 1$
- Pais
  - o pai de um nó i é $\lfloor i/2 \rfloor$
  - o nó 1 não tem pai.
- Folhas
  - um nó i é folha se não tiver filhos, i.e., se $2i > n$.
  - as folhas são $\lfloor n/2 \rfloor + 1, \ldots, n-1, n$.

Representando um heap

Um heap é uma árvore binária armazenada em vetor $A[1 \dots n]$

- Filhos de um nó i
  - o filho esquerdo é $2i$
  - o filho direito é $2i + 1$
- Pais
  - o pai de um nó i é $\lfloor i/2 \rfloor$
  - o nó 1 não tem pai.
- Folhas
  - um nó i é folha se não tiver filhos, i.e., se $2i > n$.
  - as folhas são $\lfloor n/2 \rfloor + 1, \dots, n - 1, n$.

Exemplo de um heap



Árvore completa

Um heap é uma árvore completa

- cada nível $\ell = 0, 1, 2, \ldots$ tem 2^ℓ nós (a não ser o último)
- os nós desse nível ℓ são $2^\ell, 2^\ell + 1, 2^\ell + 2, \ldots, 2^{\ell+1} - 1$

Nível de um nó

O nível de um nó i é $\lfloor \lg i \rfloor$.

- se i está no nível ℓ, então

$$2^\ell \leq i < 2^{\ell+1} \Rightarrow$$

- assim, $\ell = \lfloor \lg i \rfloor$.

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

Árvore completa

Um heap é uma árvore completa

- cada nível $\ell = 0, 1, 2, \ldots$ tem 2^ℓ nós (a não ser o último)
- os nós desse nível ℓ são $2^\ell, 2^\ell + 1, 2^\ell + 2, \ldots, 2^{\ell+1} - 1$

Nível de um nó

O nível de um nó i é $\lfloor \lg i \rfloor$.

- se i está no nível ℓ, então

$$2^\ell \leq i < 2^{\ell+1} \Rightarrow$$

- assim, $\ell = \lfloor \lg i \rfloor$.

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

Árvore completa

Um heap é uma árvore completa

- cada nível $\ell = 0, 1, 2, \ldots$ tem 2^ℓ nós (a não ser o último)
- os nós desse nível ℓ são $2^\ell, 2^\ell + 1, 2^\ell + 2, \ldots, 2^{\ell+1} - 1$

Nível de um nó

O nível de um nó i é $\lfloor \lg i \rfloor$.

- se i está no nível ℓ, então

$$2^\ell \leq i < 2^{\ell+1} \Rightarrow$$

- assim, $\ell = \lfloor \lg i \rfloor$.

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

Árvore completa

Um heap é uma árvore completa

- cada nível $\ell = 0, 1, 2, \ldots$ tem 2^ℓ nós (a não ser o último)
- os nós desse nível ℓ são $2^\ell, 2^\ell + 1, 2^\ell + 2, \ldots, 2^{\ell+1} - 1$

Nível de um nó

O nível de um nó i é $\lfloor \lg i \rfloor$.

- se i está no nível ℓ, então

$$2^\ell \leq i < 2^{\ell+1} \Rightarrow$$

- assim, $\ell = \lfloor \lg i \rfloor$.

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

Árvore completa

Um heap é uma árvore completa

- cada nível $\ell = 0, 1, 2, \ldots$ tem 2^ℓ nós (a não ser o último)
- os nós desse nível ℓ são $2^\ell, 2^\ell + 1, 2^\ell + 2, \ldots, 2^{\ell+1} - 1$

Nível de um nó

O nível de um nó i é $\lfloor \lg i \rfloor$.

- se i está no nível ℓ, então

$$2^\ell \leq i < 2^{\ell+1} \Rightarrow$$

- assim, $\ell = \lfloor \lg i \rfloor$.

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

Árvore completa

Um heap é uma árvore completa

- cada nível $\ell = 0, 1, 2, \ldots$ tem 2^ℓ nós (a não ser o último)
- os nós desse nível ℓ são $2^\ell, 2^\ell + 1, 2^\ell + 2, \ldots, 2^{\ell+1} - 1$

Nível de um nó

O nível de um nó i é $\lfloor \lg i \rfloor$.

- se i está no nível ℓ, então

$$\begin{array}{rcccl} 2^\ell & \le & i & < & 2^{\ell+1} & \Rightarrow \\ \lg 2^\ell & \le & \lg i & < & \lg 2^{\ell+1} & \Rightarrow \\ \ell & \le & \lg i & < & \ell + 1 & \end{array}$$

- assim, $\ell = \lfloor \lg i \rfloor$.

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

Árvore completa

Um heap é uma árvore completa

- cada nível $\ell = 0, 1, 2, \ldots$ tem 2^ℓ nós (a não ser o último)
- os nós desse nível ℓ são $2^\ell, 2^\ell + 1, 2^\ell + 2, \ldots, 2^{\ell+1} - 1$

Nível de um nó

O nível de um nó i é $\lfloor \lg i \rfloor$.

- se i está no nível ℓ, então

$$
\begin{array}{ccccccc}
2^\ell & \leq & i & < & 2^{\ell+1} & \Rightarrow \\
\lg 2^\ell & \leq & \lg i & < & \lg 2^{\ell+1} & \Rightarrow \\
\ell & \leq & \lg i & < & \ell + 1
\end{array}
$$

- assim, $\ell = \lfloor \lg i \rfloor$.

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

Árvore completa

Um heap é uma árvore completa

- cada nível $\ell = 0, 1, 2, \dots$ tem 2^ℓ nós (a não ser o último)
- os nós desse nível ℓ são $2^\ell, 2^\ell + 1, 2^\ell + 2, \dots, 2^{\ell+1} - 1$

Nível de um nó

O nível de um nó i é $\lfloor \lg i \rfloor$.

- se i está no nível ℓ, então

$$
\begin{array}{ccccccc}
2^\ell & \le & i & < & 2^{\ell+1} & \Rightarrow \\
\lg 2^\ell & \le & \lg i & < & \lg 2^{\ell+1} & \Rightarrow \\
\ell & \le & \lg i & < & \ell + 1
\end{array}
$$

- assim, $\ell = \lfloor \lg i \rfloor$.

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

Árvore completa

Um heap é uma árvore completa

- cada nível $\ell = 0, 1, 2, \ldots$ tem 2^ℓ nós (a não ser o último)
- os nós desse nível ℓ são $2^\ell, 2^\ell + 1, 2^\ell + 2, \ldots, 2^{\ell+1} - 1$

Nível de um nó

O nível de um nó i é $\lfloor \lg i \rfloor$.

- se i está no nível ℓ, então

$$\begin{array}{ccccccc} 2^\ell & \leq & i & < & 2^{\ell+1} & \Rightarrow \\ \lg 2^\ell & \leq & \lg i & < & \lg 2^{\ell+1} & \Rightarrow \\ \ell & \leq & \lg i & < & \ell + 1 & \end{array}$$

- assim, $\ell = \lfloor \lg i \rfloor$.

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

Árvore completa

Um heap é uma árvore completa

- cada nível $\ell = 0, 1, 2, \ldots$ tem 2^ℓ nós (a não ser o último)
- os nós desse nível ℓ são $2^\ell, 2^\ell + 1, 2^\ell + 2, \ldots, 2^{\ell+1} - 1$

Nível de um nó

O nível de um nó i é $\lfloor \lg i \rfloor$.

- se i está no nível ℓ, então

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

- assim, $\ell = \lfloor \lg i \rfloor$.

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

Árvore completa

Um heap é uma árvore completa

- cada nível $\ell = 0, 1, 2, \ldots$ tem 2^ℓ nós (a não ser o último)
- os nós desse nível ℓ são $2^\ell, 2^\ell + 1, 2^\ell + 2, \ldots, 2^{\ell+1} - 1$

Nível de um nó

O nível de um nó i é $\lfloor \lg i \rfloor$.

- se i está no nível ℓ, então

$$
\begin{array}{ccccccc}
2^\ell & \leq & i & < & 2^{\ell+1} & \Rightarrow \\
\lg 2^\ell & \leq & \lg i & < & \lg 2^{\ell+1} & \Rightarrow \\
\ell & \leq & \lg i & < & \ell + 1 &
\end{array}
$$

- assim, $\ell = \lfloor \lg i \rfloor$.

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

Árvore completa

Um heap é uma árvore completa

- cada nível $\ell = 0, 1, 2, \ldots$ tem 2^ℓ nós (a não ser o último)
- os nós desse nível ℓ são $2^\ell, 2^\ell + 1, 2^\ell + 2, \ldots, 2^{\ell+1} - 1$

Nível de um nó

O nível de um nó i é $\lfloor \lg i \rfloor$.

- se i está no nível ℓ, então

$$\begin{array}{ccccccc} 2^\ell & \leq & i & < & 2^{\ell+1} & \Rightarrow \\ \lg 2^\ell & \leq & \lg i & < & \lg 2^{\ell+1} & \Rightarrow \\ \ell & \leq & \lg i & < & \ell + 1 & \end{array}$$

- assim, $\ell = \lfloor \lg i \rfloor$.

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

Max-heaps

Definição

Um heap $A[1 \ldots n]$ é chamado de max-heap se cada nó tiver valor maior que seus filhos, i.e., para cada $i \geq n/2$,

- $A[i] \geq A[2i]$ e $A[i] \geq A[2i+1]$

- essa restrição é a propriedade de max-heap
- o valor da raiz é um máximo do heap
- cada subárvore também é um max-heap

Max-heaps

Definição

Um heap $A[1 \dots n]$ é chamado de max-heap se cada nó tiver valor maior que seus filhos, i.e., para cada $i \geq n/2$,

- $A[i] \geq A[2i]$ e $A[i] \geq A[2i+1]$

- essa restrição é a propriedade de max-heap
- o valor da raiz é um máximo do heap
- cada subárvore também é um max-heap

Max-heaps

Definição

Um heap $A[1 \ldots n]$ é chamado de max-heap se cada nó tiver valor maior que seus filhos, i.e., para cada $i \geq n/2$,

- $A[i] \geq A[2i]$ e $A[i] \geq A[2i+1]$

- essa restrição é a propriedade de max-heap
- o valor da raiz é um máximo do heap
- cada subárvore também é um max-heap

Max-heaps

Definição

Um heap $A[1 \dots n]$ é chamado de max-heap se cada nó tiver valor maior que seus filhos, i.e., para cada $i \geq n/2$,

- $A[i] \geq A[2i]$ e $A[i] \geq A[2i+1]$

- essa restrição é a propriedade de max-heap
- o valor da raiz é um máximo do heap
- cada subárvore também é um max-heap

Max-heaps

Definição

Um heap $A[1 \dots n]$ é chamado de max-heap se cada nó tiver valor maior que seus filhos, i.e., para cada $i \geq n/2$,

- $A[i] \geq A[2i]$ e $A[i] \geq A[2i+1]$

- essa restrição é a propriedade de max-heap
- o valor da raiz é um máximo do heap
- cada subárvore também é um max-heap

Max-heaps

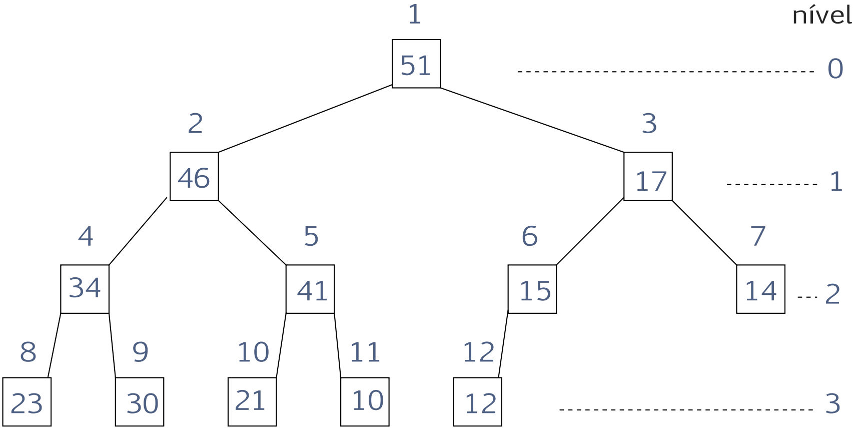
Definição

Um heap $A[1 \ldots n]$ é chamado de max-heap se cada nó tiver valor maior que seus filhos, i.e., para cada $i \geq n/2$,

- $A[i] \geq A[2i]$ e $A[i] \geq A[2i+1]$

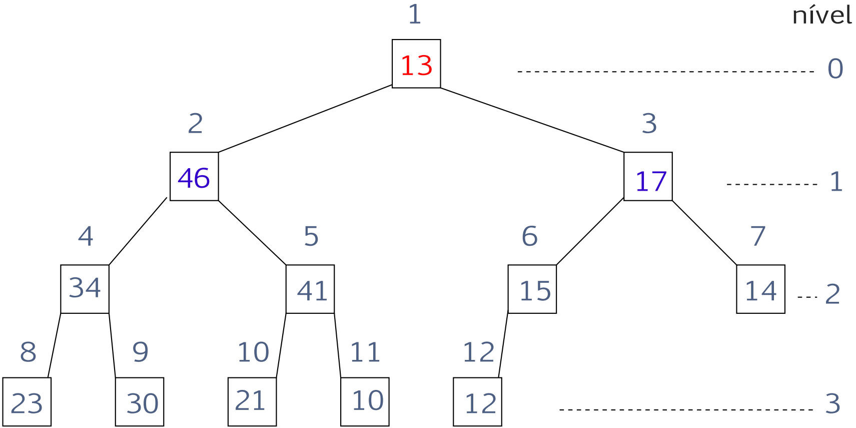
- essa restrição é a propriedade de max-heap
- o valor da raiz é um máximo do heap
- cada subárvore também é um max-heap

Exemplo de max-heap



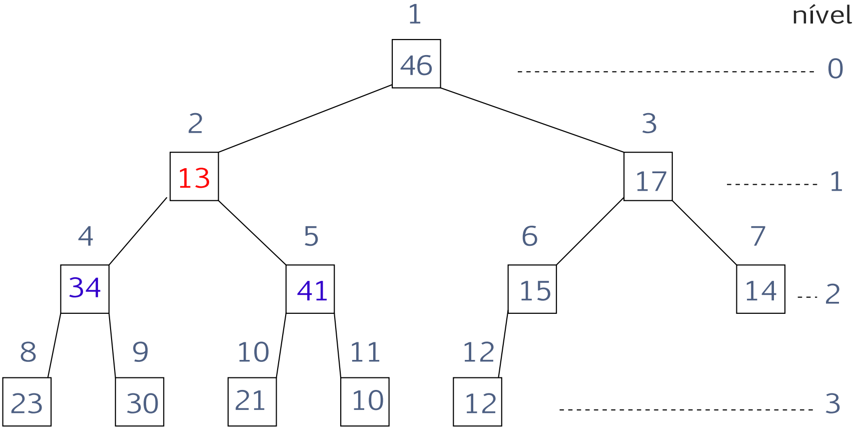
1	2	3	4	5	6	7	8	9	10	11	12
51	46	17	34	41	15	14	23	30	21	10	12

Manipulação de max-heap



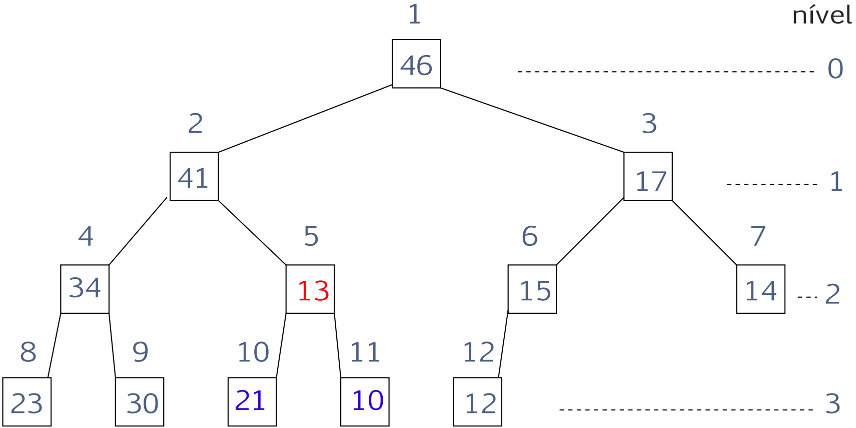
1	2	3	4	5	6	7	8	9	10	11	12
13	46	17	34	41	15	14	23	30	21	10	12

Manipulação de max-heap



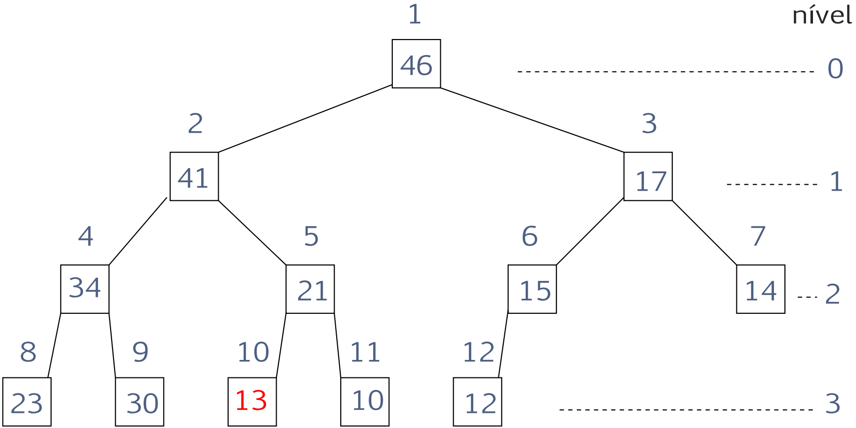
1	2	3	4	5	6	7	8	9	10	11	12
46	13	17	34	41	15	14	23	30	21	10	12

Manipulação de max-heap



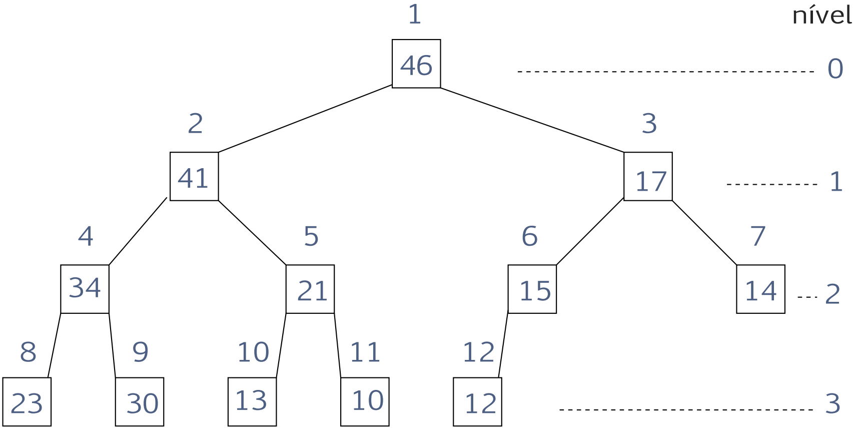
1	2	3	4	5	6	7	8	9	10	11	12
46	41	17	34	13	15	14	23	30	21	10	12

Manipulação de max-heap



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

Manipulação de max-heap



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

Consertando um max-heap

- suponha que as subárvores $2i$ e $2i+1$ são max-heaps
- como transformar a subárvore i em um max-heap?

MAX-HEAPIFY(A, n, i)

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

Consertando um max-heap

- suponha que as subárvores $2i$ e $2i+1$ são max-heaps
- como transformar a subárvore i em um max-heap?

MAX-HEAPIFY(A, n, i)

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

Consertando um max-heap

- suponha que as subárvores $2i$ e $2i+1$ são max-heaps
- como transformar a subárvore i em um max-heap?

```
MAX-HEAPIFY(A, n, i)
   e ← 2i
   d ← 2i + 1
   maior ← i
   se e ≤ n e A[e] > A[i]
      então maior ← e
   se d ≤ n e A[d] > A[maior]
      então maior ← d
   se maior ≠ i
      então A[i] ↔ A[maior]
            MAX-HEAPIFY(A, n, maior)
```

Correção de Max-Heapify

Lema

Max-Heapify transforma a subárvore i em max-heap.

Ideia para demonstração: indução na altura h do nó i.

- se $h = 0$, então i é folha e o algoritmo está correto
- considere um nó i com altura $h > 0$
- suponha que o algoritmo funciona para árvores menores
  - antes da linha 8, temos $A[maior] \geq A[i], A[2i], A[2i+1]$
  - após a linha 9, temos $A[i] \geq A[2i], A[2i+1]$
  - segue que $A[i]$ é máximo no vetor
  - pela hipótese de indução, Max-Heapify transforma a subárvore com raiz $maior$ em max-heap
  - segue que i é max-heap

Correção de MAX-HEAPIFY

Lema

MAX-HEAPIFY transforma a subárvore i em max-heap.

Ideia para demonstração: indução na altura h do nó i.

- se $h = 0$, então i é folha e o algoritmo está correto
- considere um nó i com altura $h > 0$
- suponha que o algoritmo funciona para árvores menores
    - antes da linha 8, temos $A[maior] \geq A[i], A[2i], A[2i+1]$
    - após a linha 9, temos $A[i] \geq A[2i], A[2i+1]$
    - segue que $A[i]$ é máximo no vetor
    - pela hipótese de indução, MAX-HEAPIFY transforma a subárvore com raiz $maior$ em max-heap
    - segue que i é max-heap

Correção de Max-Heapify

Lema

Max-Heapify transforma a subárvore i em max-heap.

Ideia para demonstração: indução na altura h do nó i.

- se $h = 0$, então i é folha e o algoritmo está correto
- considere um nó i com altura $h > 0$
- suponha que o algoritmo funciona para árvores menores
  - antes da linha 8, temos $A[maior] \geq A[i], A[2i], A[2i+1]$
  - após a linha 9, temos $A[i] \geq A[2i], A[2i+1]$
  - segue que $A[i]$ é máximo no vetor
  - pela hipótese de indução, Max-Heapify transforma a subárvore com raiz $maior$ em max-heap
  - segue que i é max-heap

Correção de MAX-HEAPIFY

Lema

MAX-HEAPIFY transforma a subárvore i em max-heap.

Ideia para demonstração: indução na altura h do nó i.

- se $h = 0$, então i é folha e o algoritmo está correto
- considere um nó i com altura $h > 0$
- suponha que o algoritmo funciona para árvores menores
  - antes da linha 8, temos $A[maior] \geq A[i], A[2i], A[2i+1]$
  - após a linha 9, temos $A[i] \geq A[2i], A[2i+1]$
  - segue que $A[i]$ é máximo no vetor
  - pela hipótese de indução, MAX-HEAPIFY transforma a subárvore com raiz $maior$ em max-heap
  - segue que i é max-heap

Correção de MAX-HEAPIFY

Lema

MAX-HEAPIFY transforma a subárvore i em max-heap.

Ideia para demonstração: indução na altura h do nó i.

- se $h = 0$, então i é folha e o algoritmo está correto
- considere um nó i com altura $h > 0$
- suponha que o algoritmo funciona para árvores menores
  - antes da linha 8, temos $A[maior] \geq A[i], A[2i], A[2i+1]$
  - após a linha 9, temos $A[i] \geq A[2i], A[2i+1]$
  - segue que $A[i]$ é máximo no vetor
  - pela hipótese de indução, MAX-HEAPIFY transforma a subárvore com raiz *maior* em max-heap
  - segue que i é max-heap

Correção de Max-Heapify

Lema

Max-Heapify transforma a subárvore i em max-heap.

Ideia para demonstração: indução na altura h do nó i.

- se $h = 0$, então i é folha e o algoritmo está correto
- considere um nó i com altura $h > 0$
- suponha que o algoritmo funciona para árvores menores
  - antes da linha 8, temos $A[maior] \geq A[i], A[2i], A[2i+1]$
  - após a linha 9, temos $A[i] \geq A[2i], A[2i+1]$
  - segue que $A[i]$ é máximo no vetor
  - pela hipótese de indução, Max-Heapify transforma a subárvore com raiz $maior$ em max-heap
  - segue que i é max-heap

Correção de MAX-HEAPIFY

Lema

MAX-HEAPIFY transforma a subárvore i em max-heap.

Ideia para demonstração: indução na altura h do nó i.

- se $h = 0$, então i é folha e o algoritmo está correto
- considere um nó i com altura $h > 0$
- suponha que o algoritmo funciona para árvores menores
  - antes da linha 8, temos $A[maior] \geq A[i], A[2i], A[2i+1]$
  - após a linha 9, temos $A[i] \geq A[2i], A[2i+1]$
  - segue que $A[i]$ é máximo no vetor
  - pela hipótese de indução, MAX-HEAPIFY transforma a subárvore com raiz $maior$ em max-heap
  - segue que i é max-heap

Correção de Max-Heapify

Lema

Max-Heapify transforma a subárvore i em max-heap.

Ideia para demonstração: indução na altura h do nó i.

- se $h = 0$, então i é folha e o algoritmo está correto
- considere um nó i com altura $h > 0$
- suponha que o algoritmo funciona para árvores menores
  - antes da linha 8, temos $A[maior] \geq A[i], A[2i], A[2i+1]$
  - após a linha 9, temos $A[i] \geq A[2i], A[2i+1]$
  - segue que $A[i]$ é máximo no vetor
  - pela hipótese de indução, Max-Heapify transforma a subárvore com raiz $maior$ em max-heap
  - segue que i é max-heap

Correção de Max-Heapify

Lema

Max-Heapify transforma a subárvore i em max-heap.

Ideia para demonstração: indução na altura h do nó i.

- se $h = 0$, então i é folha e o algoritmo está correto
- considere um nó i com altura $h > 0$
- suponha que o algoritmo funciona para árvores menores
  - antes da linha 8, temos $A[maior] \geq A[i], A[2i], A[2i+1]$
  - após a linha 9, temos $A[i] \geq A[2i], A[2i+1]$
  - segue que $A[i]$ é máximo no vetor
  - pela hipótese de indução, Max-Heapify transforma a subárvore com raiz *maior* em max-heap
  - segue que i é max-heap

Correção de Max-Heapify

Lema

Max-Heapify transforma a subárvore i em max-heap.

Ideia para demonstração: indução na altura h do nó i.

- se $h = 0$, então i é folha e o algoritmo está correto
- considere um nó i com altura $h > 0$
- suponha que o algoritmo funciona para árvores menores
  - antes da linha 8, temos $A[maior] \geq A[i], A[2i], A[2i+1]$
  - após a linha 9, temos $A[i] \geq A[2i], A[2i+1]$
  - segue que $A[i]$ é máximo no vetor
  - pela hipótese de indução, Max-Heapify transforma a subárvore com raiz $maior$ em max-heap
  - segue que i é max-heap

Complexidade de Max-Heapify

Max-Heapify(A, n, i)	Tempo
$e \leftarrow 2i$	?
$d \leftarrow 2i + 1$	?
$maior \leftarrow i$	?
se $e \leq n$ **e** $A[e] > A[i]$	?
então $maior \leftarrow e$	?
se $d \leq n$ e $A[d] > A[maior]$	?
então $maior \leftarrow d$	?
se $maior \neq i$	?
então $A[i] \leftrightarrow A[maior]$	?
Max-Heapify$(A, n, maior)$	?

O tempo de execução é $T(h) = T(h-1) + \Theta(1) = O(h)$

Complexidade de Max-Heapify

Max-Heapify(A, n, i)	Tempo
$e \leftarrow 2i$	$\Theta(1)$
$d \leftarrow 2i + 1$	$\Theta(1)$
$maior \leftarrow i$	$\Theta(1)$
se $e \leq n$ **e** $A[e] > A[i]$	$\Theta(1)$
então $maior \leftarrow e$	$O(1)$
se $d \leq n$ e $A[d] > A[maior]$	$\Theta(1)$
então $maior \leftarrow d$	$O(1)$
se $maior \neq i$	$\Theta(1)$
então $A[i] \leftrightarrow A[maior]$	$O(1)$
Max-Heapify$(A, n, maior)$	$T(h-1)$

O tempo de execução é $T(h) = T(h-1) + \Theta(1) = O(h)$

Complexidade de MAX-HEAPIFY

MAX-HEAPIFY(A, n, i)	Tempo
$e \leftarrow 2i$	$\Theta(1)$
$d \leftarrow 2i + 1$	$\Theta(1)$
$maior \leftarrow i$	$\Theta(1)$
se $e \leq n$ **e** $A[e] > A[i]$	$\Theta(1)$
então $maior \leftarrow e$	$O(1)$
se $d \leq n$ e $A[d] > A[maior]$	$\Theta(1)$
então $maior \leftarrow d$	$O(1)$
se $maior \neq i$	$\Theta(1)$
então $A[i] \leftrightarrow A[maior]$	$O(1)$
MAX-HEAPIFY$(A, n, maior)$	$T(h-1)$

O tempo de execução é $T(h) = T(h-1) + \Theta(1) = O(h)$

Complexidade de Max-Heapify

	Max-Heapify(A, n, i)	Tempo
1	$e \leftarrow 2i$	$\Theta(1)$
2	$d \leftarrow 2i + 1$	$\Theta(1)$
3	$maior \leftarrow i$	$\Theta(1)$
4	**se** $e \leq n$ **e** $A[e] > A[i]$	$\Theta(1)$
4	**então** $maior \leftarrow e$	$O(1)$
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$(A, n, maior)$	$T(h-1)$

O tempo de execução é $T(h) = T(h-1) + \Theta(1) = O(h)$

Construindo um max-heap

Podemos consertar um vetor inteiro

- recebemos um vetor $A[1 \dots n]$ desorganizado
- mas as folhas já são heap
- consertamos o penúltimo nível
- depois o antepenúltimo
- e assim por diante

Construindo um max-heap

Podemos consertar um vetor inteiro

- recebemos um vetor $A[1 \ldots n]$ desorganizado
- mas as folhas já são heap
- consertamos o penúltimo nível
- depois o antepenúltimo
- e assim por diante

Construindo um max-heap

Podemos consertar um vetor inteiro

- recebemos um vetor $A[1 \ldots n]$ desorganizado
- mas as folhas já são heap
- consertamos o penúltimo nível
- depois o antepenúltimo
- e assim por diante

Construindo um max-heap

Podemos consertar um vetor inteiro

- recebemos um vetor $A[1 \ldots n]$ desorganizado
- mas as folhas já são heap
- consertamos o penúltimo nível
- depois o antepenúltimo
- e assim por diante

Construindo um max-heap

Podemos consertar um vetor inteiro

- recebemos um vetor $A[1 \ldots n]$ desorganizado
- mas as folhas já são heap
- consertamos o penúltimo nível
- depois o antepenúltimo
- e assim por diante

Construindo um max-heap

Podemos consertar um vetor inteiro

- recebemos um vetor $A[1 \dots n]$ desorganizado
- mas as folhas já são heap
- consertamos o penúltimo nível
- depois o antepenúltimo
- e assim por diante

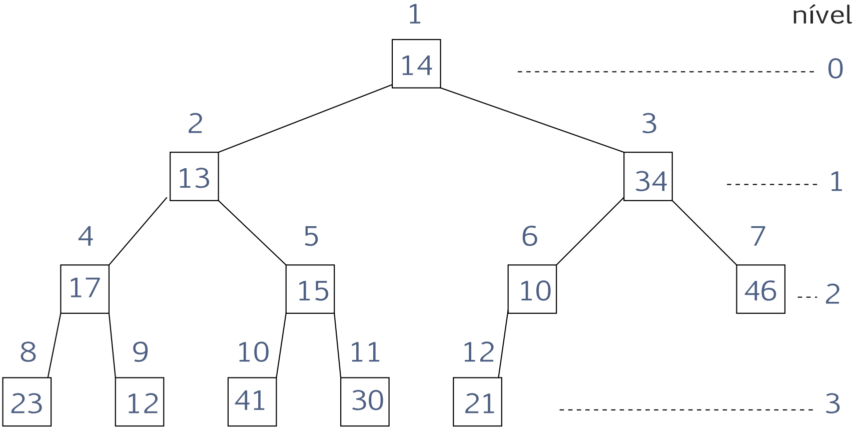
Construindo um max-heap

Podemos consertar um vetor inteiro

- recebemos um vetor $A[1 \ldots n]$ desorganizado
- mas as folhas já são heap
- consertamos o penúltimo nível
- depois o antepenúltimo
- e assim por diante

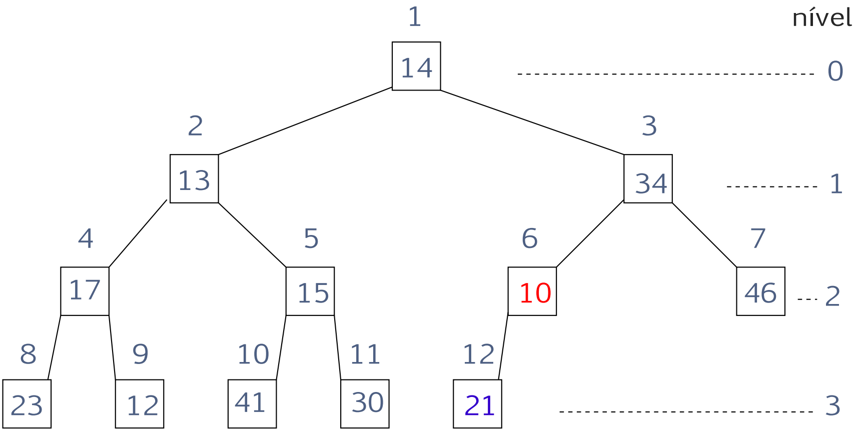
```
BUILD-MAX-HEAP(A, n)
  para i ← ⌊n/2⌋ decrescendo até 1 faça
      MAX-HEAPIFY(A, n, i)
```

Construção de um max-heap



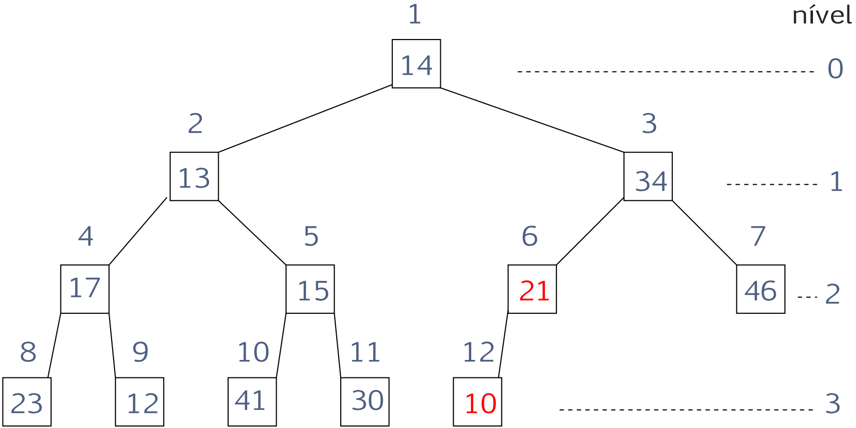
1	2	3	4	5	6	7	8	9	10	11	12
14	13	34	17	15	10	46	23	12	41	30	21

Construção de um max-heap



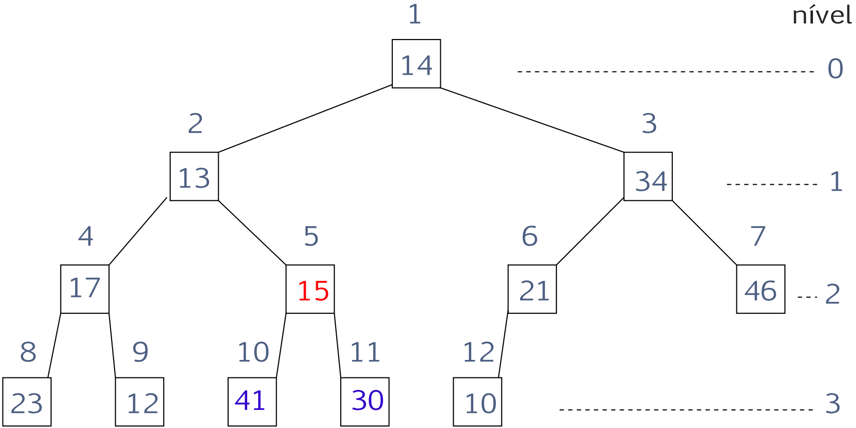
1	2	3	4	5	6	7	8	9	10	11	12
14	13	34	17	15	10	46	23	12	41	30	21

Construção de um max-heap



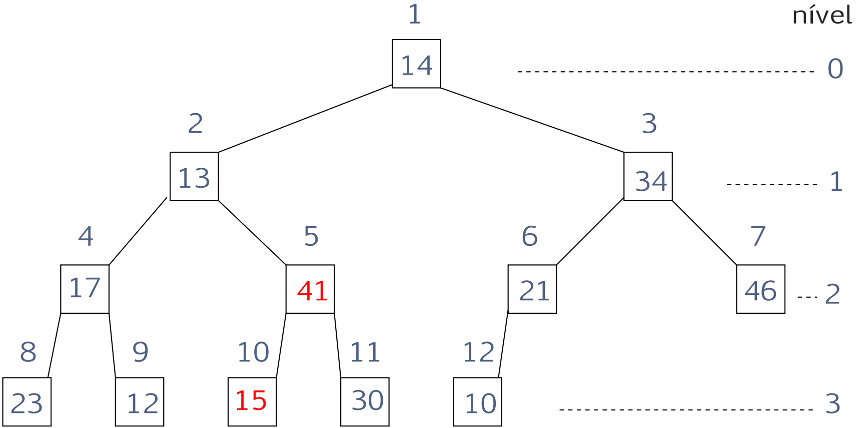
1	2	3	4	5	6	7	8	9	10	11	12
14	13	34	17	15	21	46	23	12	41	30	10

Construção de um max-heap



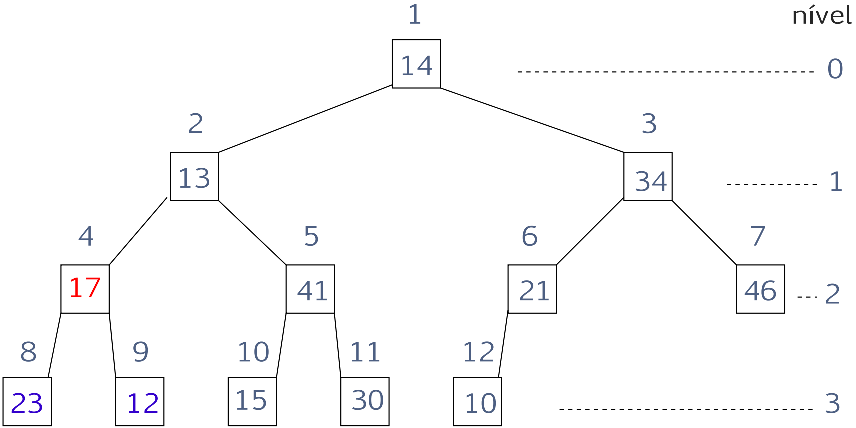
1	2	3	4	5	6	7	8	9	10	11	12
14	13	34	17	15	21	46	23	12	41	30	10

Construção de um max-heap



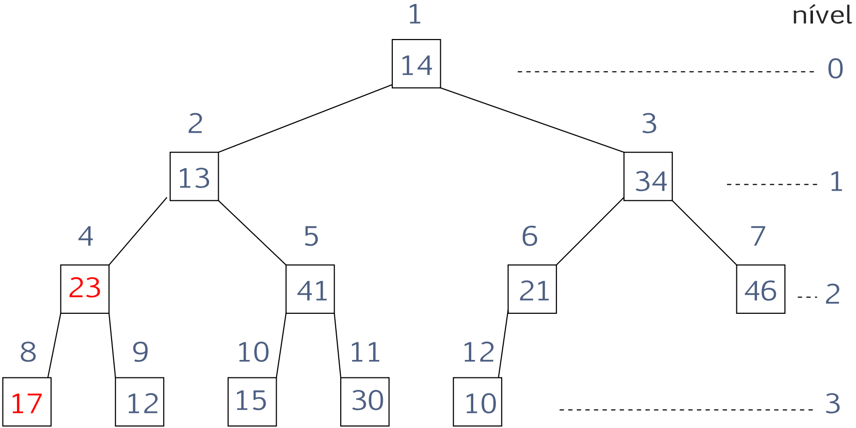
1	2	3	4	5	6	7	8	9	10	11	12
14	13	34	17	41	21	46	23	12	15	30	10

Construção de um max-heap



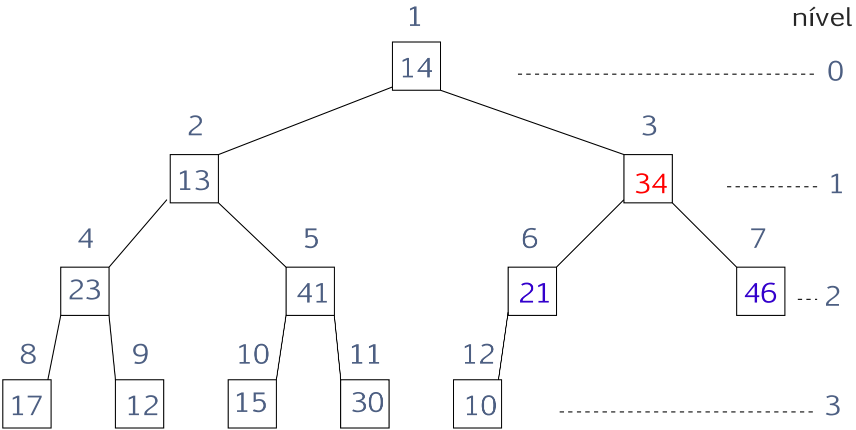
1	2	3	4	5	6	7	8	9	10	11	12
14	13	34	17	41	21	46	23	12	15	30	10

Construção de um max-heap



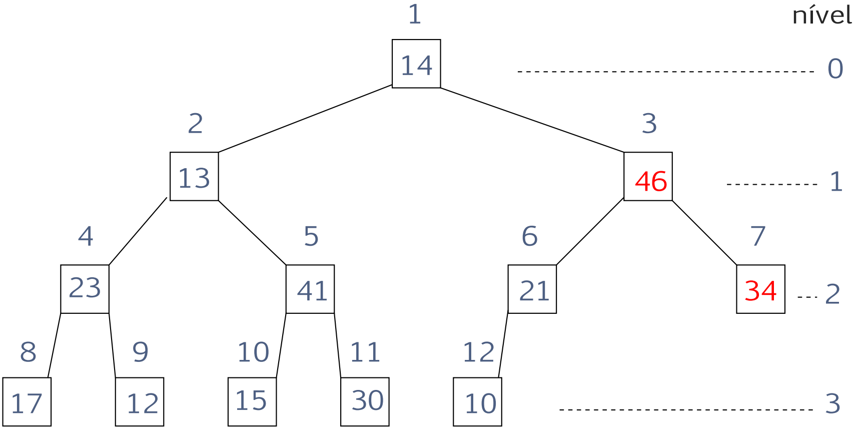
1	2	3	4	5	6	7	8	9	10	11	12
14	13	34	23	41	21	46	17	12	15	30	10

Construção de um max-heap



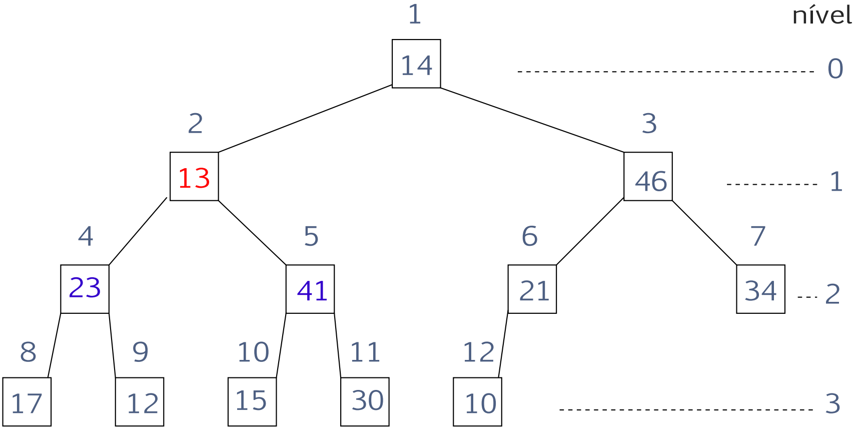
1	2	3	4	5	6	7	8	9	10	11	12
14	13	34	23	41	21	46	17	12	15	30	10

Construção de um max-heap



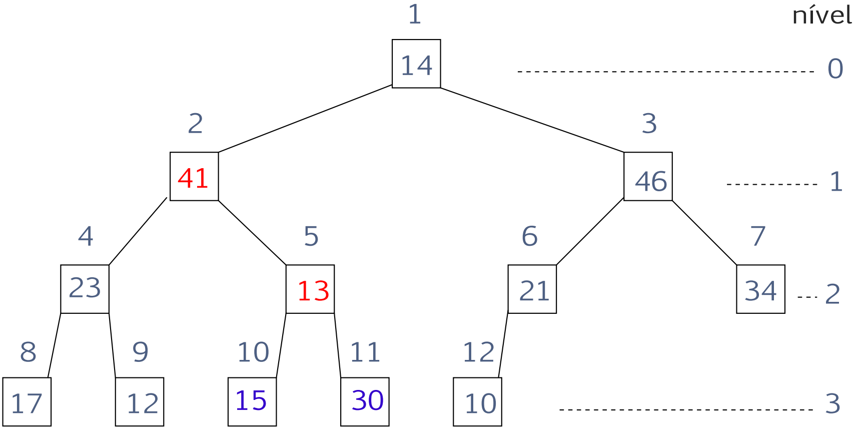
1	2	3	4	5	6	7	8	9	10	11	12
14	13	46	23	41	21	34	17	12	15	30	10

Construção de um max-heap



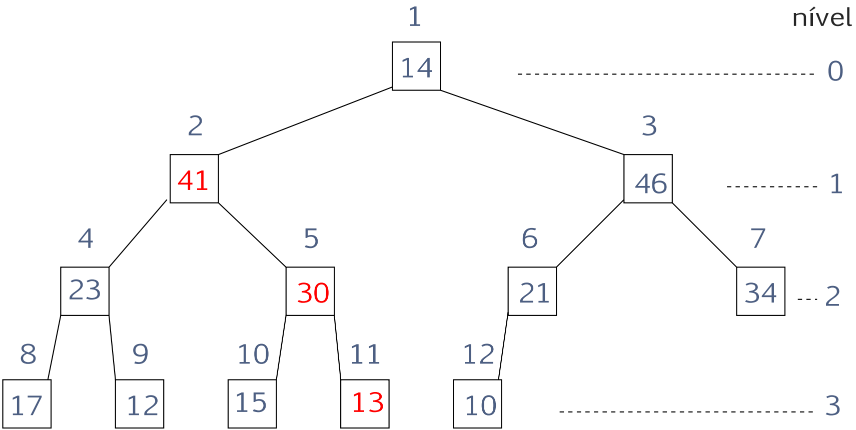
1	2	3	4	5	6	7	8	9	10	11	12
14	13	46	23	41	21	34	17	12	15	30	10

Construção de um max-heap



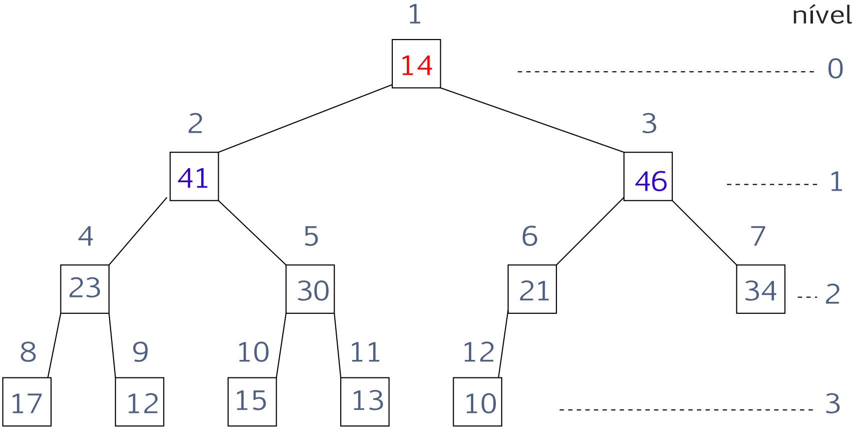
1	2	3	4	5	6	7	8	9	10	11	12
14	41	46	23	13	21	34	17	12	15	30	10

Construção de um max-heap



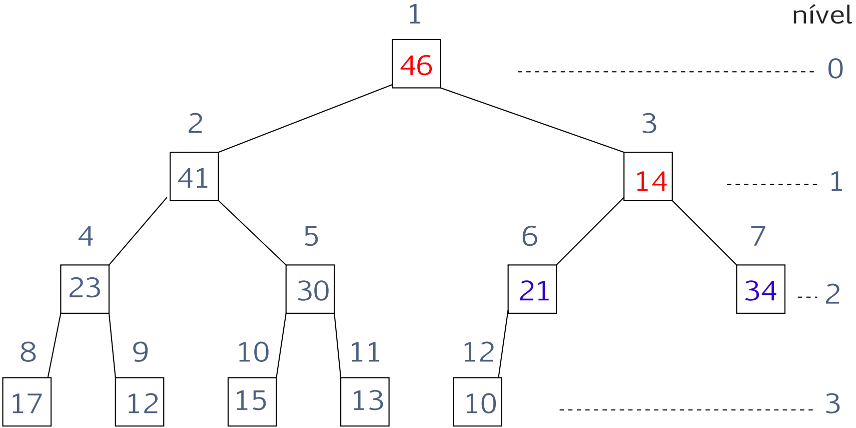
1	2	3	4	5	6	7	8	9	10	11	12
14	41	46	23	30	21	34	17	12	15	13	10

Construção de um max-heap



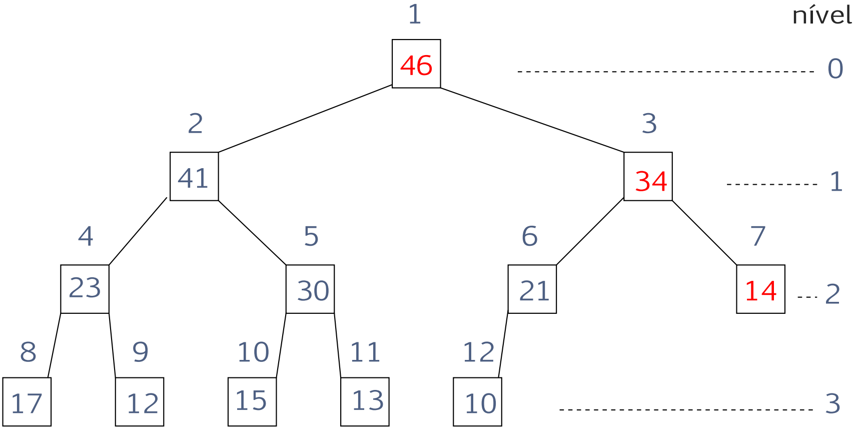
1	2	3	4	5	6	7	8	9	10	11	12
14	41	46	23	30	21	34	17	12	15	13	10

Construção de um max-heap



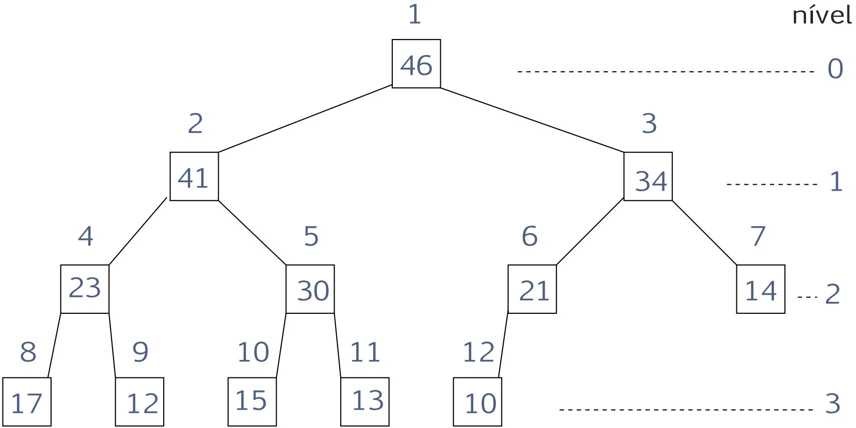
1	2	3	4	5	6	7	8	9	10	11	12
46	41	14	23	30	21	34	17	12	15	13	10

Construção de um max-heap



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

Construção de um max-heap



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

Análise de BUILD-MAX-HEAP

```
BUILD-MAX-HEAP(A, n)
    para i ← ⌊n/2⌋ decrescendo até 1 faça
        MAX-HEAPIFY(A, n, i)
```

Invariante

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

Complexidade

- uma análise rápida leva a $T(n) = n \cdot O(\lg n) = O(n \lg n)$
- mas na verdade mostraremos que $T(n)$ é linear!

Análise de BUILD-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.

Complexidade

- uma análise rápida leva a $T(n) = n \cdot O(\lg n) = O(n \lg n)$
- mas na verdade mostraremos que $T(n)$ é linear!

Análise de BUILD-MAX-HEAP

```
BUILD-MAX-HEAP(A, n)
   para i ← ⌊n/2⌋ decrescendo até 1 faça
       MAX-HEAPIFY(A, n, i)
```

Invariante

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

Complexidade

- uma análise rápida leva a $T(n) = n \cdot O(\lg n) = O(n \lg n)$
- mas na verdade mostraremos que $T(n)$ é linear!

Análise de BUILD-MAX-HEAP

```
BUILD-MAX-HEAP(A, n)
    para i ← ⌊n/2⌋ decrescendo até 1 faça
        MAX-HEAPIFY(A, n, i)
```

Invariante

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

Complexidade

- uma análise rápida leva a $T(n) = n \cdot O(\lg n) = O(n \lg n)$
- mas na verdade mostraremos que $T(n)$ é linear!

Complexidade de Build-Max-Heap

Análise mais cuidadosa:

- para um nó de altura h, Max-Heapify leva tempo $O(h)$
  - temos 1 nó de altura h
  - temos 2 nós de altura $h/2$
  - temos 4 nós de altura $h/4$
  - e assim por diante
- vamos somar os tempos de todas as chamadas

Complexidade de Build-Max-Heap

Análise mais cuidadosa:

- para um nó de altura h, Max-Heapify leva tempo $O(h)$
  - temos 1 nó de altura h
  - temos 2 nós de altura $h/2$
  - temos 4 nós de altura $h/4$
  - e assim por diante
- vamos somar os tempos de todas as chamadas

Complexidade de Build-Max-Heap

Análise mais cuidadosa:

- para um nó de altura h, Max-Heapify leva tempo $O(h)$
  - temos 1 nó de altura h
  - temos 2 nós de altura $h/2$
  - temos 4 nós de altura $h/4$
  - e assim por diante
- vamos somar os tempos de todas as chamadas

Complexidade de Build-Max-Heap

Análise mais cuidadosa:

- para um nó de altura h, Max-Heapify leva tempo $O(h)$
  - temos 1 nó de altura h
  - temos 2 nós de altura $h/2$
  - temos 4 nós de altura $h/4$
  - e assim por diante
- vamos somar os tempos de todas as chamadas

Complexidade de Build-Max-Heap

Análise mais cuidadosa:

- para um nó de altura h, Max-Heapify leva tempo $O(h)$
  - temos 1 nó de altura h
  - temos 2 nós de altura $h/2$
  - temos 4 nós de altura $h/4$
  - e assim por diante
- vamos somar os tempos de todas as chamadas

Complexidade de BUILD-MAX-HEAP

Análise mais cuidadosa:

- para um nó de altura h, MAX-HEAPIFY leva tempo $O(h)$
  - temos 1 nó de altura h
  - temos 2 nós de altura $h/2$
  - temos 4 nós de altura $h/4$
  - e assim por diante
- vamos somar os tempos de todas as chamadas

Complexidade de BUILD-MAX-HEAP

Análise mais cuidadosa:

- para um nó de altura h, MAX-HEAPIFY leva tempo $O(h)$
  - temos 1 nó de altura h
  - temos 2 nós de altura $h/2$
  - temos 4 nós de altura $h/4$
  - e assim por diante
- vamos somar os tempos de todas as chamadas

Complexidade de Build-Max-Heap (cont)

Seja $k = 1 + \lfloor \lg n \rfloor$ a altura da árvore inteira, então

$$T(n) = \sum_{h=1}^{k-1} 2^{k-h} \cdot O(h) = O(2^k) \sum_{h=1}^{k-1} \frac{h}{2^h}$$

Note que

$$\begin{aligned}
\sum_{h=1}^{k-1} \frac{h}{2^h} = & \left.\frac{1}{2} + \frac{1}{2^2} + \frac{1}{2^3} + \cdots + \frac{1}{2^{k-1}}\right\} \; 1 - \frac{1}{2^{k-1}} < 1 \\
& \left.+ \frac{1}{2^2} + \frac{1}{2^3} + \cdots + \frac{1}{2^{k-1}}\right\} \; \frac{1}{2} - \frac{1}{2^{k-1}} < \frac{1}{2} \\
& \left.+ \frac{1}{2^3} + \cdots + \frac{1}{2^{k-1}}\right\} \; \frac{1}{4} - \frac{1}{2^{k-1}} < \frac{1}{4}
\end{aligned}$$

Ou seja,

$$T(n) = O(2^k) \sum_{h=0}^{k} \frac{1}{2^h} \leq O(2^k) \cdot \sum_{h=0}^{\infty} \frac{1}{2^h} = O(2^k) \cdot 2 = O(n)$$

Complexidade de Build-Max-Heap (cont)

Seja $k = 1 + \lfloor \lg n \rfloor$ a altura da árvore inteira, então

$$T(n) = \sum_{h=1}^{k-1} 2^{k-h} \cdot O(h) = O(2^k) \sum_{h=1}^{k-1} \frac{h}{2^h}$$

Note que

$$\begin{aligned}
\sum_{h=1}^{k-1} \frac{h}{2^h} = \; & \left.\frac{1}{2} + \frac{1}{2^2} + \frac{1}{2^3} + \cdots + \frac{1}{2^{k-1}} \right\} \; 1 - \frac{1}{2^{k-1}} < 1 \\
& \left. + \frac{1}{2^2} + \frac{1}{2^3} + \cdots + \frac{1}{2^{k-1}} \right\} \; \frac{1}{2} - \frac{1}{2^{k-1}} < \frac{1}{2} \\
& \left. + \frac{1}{2^3} + \cdots + \frac{1}{2^{k-1}} \right\} \; \frac{1}{4} - \frac{1}{2^{k-1}} < \frac{1}{4}
\end{aligned}$$

Ou seja,

$$T(n) = O(2^k) \sum_{h=0}^{k} \frac{1}{2^h} \le O(2^k) \cdot \sum_{h=0}^{\infty} \frac{1}{2^h} = O(2^k) \cdot 2 = O(n)$$

Complexidade de Build-Max-Heap (cont)

Seja $k = 1 + \lfloor \lg n \rfloor$ a altura da árvore inteira, então

$$T(n) = \sum_{h=1}^{k-1} 2^{k-h} \cdot O(h) = O(2^k) \sum_{h=1}^{k-1} \frac{h}{2^h}$$

Note que

$$\begin{array}{rll}
\displaystyle\sum_{h=1}^{k-1} \frac{h}{2^h} = & \left. \frac{1}{2} + \frac{1}{2^2} + \frac{1}{2^3} + \cdots + \frac{1}{2^{k-1}} \right\} & 1 - \frac{1}{2^{k-1}} < 1 \\
& \left. + \frac{1}{2^2} + \frac{1}{2^3} + \cdots + \frac{1}{2^{k-1}} \right\} & \frac{1}{2} - \frac{1}{2^{k-1}} < \frac{1}{2} \\
& \left. + \frac{1}{2^3} + \cdots + \frac{1}{2^{k-1}} \right\} & \frac{1}{4} - \frac{1}{2^{k-1}} < \frac{1}{4}
\end{array}$$

Ou seja,

$$T(n) = O(2^k) \sum_{h=0}^{k} \frac{1}{2^h} \leq O(2^k) \cdot \sum_{h=0}^{\infty} \frac{1}{2^h} = O(2^k) \cdot 2 = O(n)$$

Complexidade de Build-Max-Heap (cont)

Seja $k = 1 + \lfloor \lg n \rfloor$ a altura da árvore inteira, então

$$T(n) = \sum_{h=1}^{k-1} 2^{k-h} \cdot O(h) = O(2^k) \sum_{h=1}^{k-1} \frac{h}{2^h}$$

Note que

$$\begin{array}{rl}
\displaystyle\sum_{h=1}^{k-1} \frac{h}{2^h} = \frac{1}{2} + \frac{1}{2^2} + \frac{1}{2^3} + \cdots + \frac{1}{2^{k-1}} & \Big\} \; 1 - \dfrac{1}{2^{k-1}} < 1 \\
+ \dfrac{1}{2^2} + \dfrac{1}{2^3} + \cdots + \dfrac{1}{2^{k-1}} & \Big\} \; \dfrac{1}{2} - \dfrac{1}{2^{k-1}} < \dfrac{1}{2} \\
+ \dfrac{1}{2^3} + \cdots + \dfrac{1}{2^{k-1}} & \Big\} \; \dfrac{1}{4} - \dfrac{1}{2^{k-1}} < \dfrac{1}{4}
\end{array}$$

Ou seja,

$$T(n) = O(2^k) \sum_{h=0}^{k} \frac{1}{2^h} \leq O(2^k) \cdot \sum_{h=0}^{\infty} \frac{1}{2^h} = O(2^k) \cdot 2 = O(n)$$

Complexidade de Build-Max-Heap (cont)

Seja $k = 1 + \lfloor \lg n \rfloor$ a altura da árvore inteira, então

$$T(n) = \sum_{h=1}^{k-1} 2^{k-h} \cdot O(h) = O(2^k) \sum_{h=1}^{k-1} \frac{h}{2^h}$$

Note que

$$\begin{aligned}
\sum_{h=1}^{k-1} \frac{h}{2^h} = \;& \left.\frac{1}{2} + \frac{1}{2^2} + \frac{1}{2^3} + \cdots + \frac{1}{2^{k-1}}\right\} \; 1 - \frac{1}{2^{k-1}} < 1 \\
& \left.+ \frac{1}{2^2} + \frac{1}{2^3} + \cdots + \frac{1}{2^{k-1}}\right\} \; \frac{1}{2} - \frac{1}{2^{k-1}} < \frac{1}{2} \\
& \left.+ \frac{1}{2^3} + \cdots + \frac{1}{2^{k-1}}\right\} \; \frac{1}{4} - \frac{1}{2^{k-1}} < \frac{1}{4}
\end{aligned}$$

Ou seja,

$$T(n) = O(2^k) \sum_{h=0}^{k} \frac{1}{2^h} \leq O(2^k) \cdot \sum_{h=0}^{\infty} \frac{1}{2^h} = O(2^k) \cdot 2 = O(n)$$

Complexidade de Build-Max-Heap (cont)

Seja $k = 1 + \lfloor \lg n \rfloor$ a altura da árvore inteira, então

$$T(n) = \sum_{h=1}^{k-1} 2^{k-h} \cdot O(h) = O(2^k) \sum_{h=1}^{k-1} \frac{h}{2^h}$$

Note que

$$\begin{aligned}
\sum_{h=1}^{k-1} \frac{h}{2^h} = {} & \left.\frac{1}{2} + \frac{1}{2^2} + \frac{1}{2^3} + \cdots + \frac{1}{2^{k-1}}\right\} \quad 1 - \frac{1}{2^{k-1}} < 1 \\
& \left.+ \frac{1}{2^2} + \frac{1}{2^3} + \cdots + \frac{1}{2^{k-1}}\right\} \quad \frac{1}{2} - \frac{1}{2^{k-1}} < \frac{1}{2} \\
& \left.+ \frac{1}{2^3} + \cdots + \frac{1}{2^{k-1}}\right\} \quad \frac{1}{4} - \frac{1}{2^{k-1}} < \frac{1}{4}
\end{aligned}$$

Ou seja,

$$T(n) = O(2^k) \sum_{h=0}^{k} \frac{1}{2^h} \leq O(2^k) \cdot \sum_{h=0}^{\infty} \frac{1}{2^h} = O(2^k) \cdot 2 = O(n)$$

Complexidade de Build-Max-Heap (cont)

Seja $k = 1 + \lfloor \lg n \rfloor$ a altura da árvore inteira, então

$$T(n) = \sum_{h=1}^{k-1} 2^{k-h} \cdot O(h) = O(2^k) \sum_{h=1}^{k-1} \frac{h}{2^h}$$

Note que

$$\begin{aligned}
\sum_{h=1}^{k-1} \frac{h}{2^h} = & \left.\frac{1}{2} + \frac{1}{2^2} + \frac{1}{2^3} + \cdots + \frac{1}{2^{k-1}} \right\} \quad 1 - \frac{1}{2^{k-1}} < 1 \\
& \left.+ \frac{1}{2^2} + \frac{1}{2^3} + \cdots + \frac{1}{2^{k-1}} \right\} \quad \frac{1}{2} - \frac{1}{2^{k-1}} < \frac{1}{2} \\
& \left.+ \frac{1}{2^3} + \cdots + \frac{1}{2^{k-1}} \right\} \quad \frac{1}{4} - \frac{1}{2^{k-1}} < \frac{1}{4}
\end{aligned}$$

Ou seja,

$$T(n) = O(2^k) \sum_{h=0}^{k} \frac{1}{2^h} \leq O(2^k) \cdot \sum_{h=0}^{\infty} \frac{1}{2^h} = O(2^k) \cdot 2 = O(n)$$

Complexidade de Build-Max-Heap (cont)

Seja $k = 1 + \lfloor \lg n \rfloor$ a altura da árvore inteira, então

$$T(n) = \sum_{h=1}^{k-1} 2^{k-h} \cdot O(h) = O(2^k) \sum_{h=1}^{k-1} \frac{h}{2^h}$$

Note que

$$\begin{aligned}
\sum_{h=1}^{k-1} \frac{h}{2^h} = {} & \left.\frac{1}{2} + \frac{1}{2^2} + \frac{1}{2^3} + \cdots + \frac{1}{2^{k-1}}\right\} \quad 1 - \frac{1}{2^{k-1}} < 1 \\
& \left.+ \frac{1}{2^2} + \frac{1}{2^3} + \cdots + \frac{1}{2^{k-1}}\right\} \quad \frac{1}{2} - \frac{1}{2^{k-1}} < \frac{1}{2} \\
& \left.+ \frac{1}{2^3} + \cdots + \frac{1}{2^{k-1}}\right\} \quad \frac{1}{4} - \frac{1}{2^{k-1}} < \frac{1}{4}
\end{aligned}$$

Ou seja,

$$T(n) = O(2^k) \sum_{h=0}^{k} \frac{1}{2^h} \leq O(2^k) \cdot \sum_{h=0}^{\infty} \frac{1}{2^h} = O(2^k) \cdot 2 = O(n)$$

Complexidade de Build-Max-Heap (cont)

Seja $k = 1 + \lfloor \lg n \rfloor$ a altura da árvore inteira, então

$$T(n) = \sum_{h=1}^{k-1} 2^{k-h} \cdot O(h) = O(2^k) \sum_{h=1}^{k-1} \frac{h}{2^h}$$

Note que

$$\begin{aligned}
\sum_{h=1}^{k-1} \frac{h}{2^h} = \frac{1}{2} + \frac{1}{2^2} + \frac{1}{2^3} + \cdots + \frac{1}{2^{k-1}} &\quad \Big\} \quad 1 - \frac{1}{2^{k-1}} < 1 \\
+ \frac{1}{2^2} + \frac{1}{2^3} + \cdots + \frac{1}{2^{k-1}} &\quad \Big\} \quad \frac{1}{2} - \frac{1}{2^{k-1}} < \frac{1}{2} \\
+ \frac{1}{2^3} + \cdots + \frac{1}{2^{k-1}} &\quad \Big\} \quad \frac{1}{4} - \frac{1}{2^{k-1}} < \frac{1}{4}
\end{aligned}$$

Ou seja,

$$T(n) = O(2^k) \sum_{h=0}^{k} \frac{1}{2^h} \leq O(2^k) \cdot \sum_{h=0}^{\infty} \frac{1}{2^h} = O(2^k) \cdot 2 = O(n)$$

Complexidade de Build-Max-Heap (cont)

Seja $k = 1 + \lfloor \lg n \rfloor$ a altura da árvore inteira, então

$$T(n) = \sum_{h=1}^{k-1} 2^{k-h} \cdot O(h) = O(2^k) \sum_{h=1}^{k-1} \frac{h}{2^h}$$

Note que

$$\begin{aligned}
\sum_{h=1}^{k-1} \frac{h}{2^h} = {} & \left.\frac{1}{2} + \frac{1}{2^2} + \frac{1}{2^3} + \cdots + \frac{1}{2^{k-1}}\right\} \quad 1 - \frac{1}{2^{k-1}} < 1 \\
& \left.+ \frac{1}{2^2} + \frac{1}{2^3} + \cdots + \frac{1}{2^{k-1}}\right\} \quad \frac{1}{2} - \frac{1}{2^{k-1}} < \frac{1}{2} \\
& \left.+ \frac{1}{2^3} + \cdots + \frac{1}{2^{k-1}}\right\} \quad \frac{1}{4} - \frac{1}{2^{k-1}} < \frac{1}{4}
\end{aligned}$$

Ou seja,

$$T(n) = O(2^k) \sum_{h=0}^{k} \frac{1}{2^h} \leq O(2^k) \cdot \sum_{h=0}^{\infty} \frac{1}{2^h} = O(2^k) \cdot 2 = O(n)$$

Complexidade de Build-Max-Heap (cont)

Seja $k = 1 + \lfloor \lg n \rfloor$ a altura da árvore inteira, então

$$T(n) = \sum_{h=1}^{k-1} 2^{k-h} \cdot O(h) = O(2^k) \sum_{h=1}^{k-1} \frac{h}{2^h}$$

Note que

$$\begin{aligned}
\sum_{h=1}^{k-1} \frac{h}{2^h} = & \left. \frac{1}{2} + \frac{1}{2^2} + \frac{1}{2^3} + \cdots + \frac{1}{2^{k-1}} \right\} \quad 1 - \frac{1}{2^{k-1}} < 1 \\
& \left. + \frac{1}{2^2} + \frac{1}{2^3} + \cdots + \frac{1}{2^{k-1}} \right\} \quad \frac{1}{2} - \frac{1}{2^{k-1}} < \frac{1}{2} \\
& \left. + \frac{1}{2^3} + \cdots + \frac{1}{2^{k-1}} \right\} \quad \frac{1}{4} - \frac{1}{2^{k-1}} < \frac{1}{4}
\end{aligned}$$

Ou seja,

$$T(n) = O(2^k) \sum_{h=0}^{k} \frac{1}{2^h} \leq O(2^k) \cdot \sum_{h=0}^{\infty} \frac{1}{2^h} = O(2^k) \cdot 2 = O(n)$$

Complexidade de Build-Max-Heap (cont)

Seja $k = 1 + \lfloor \lg n \rfloor$ a altura da árvore inteira, então

$$T(n) = \sum_{h=1}^{k-1} 2^{k-h} \cdot O(h) = O(2^k) \sum_{h=1}^{k-1} \frac{h}{2^h}$$

Note que

$$\begin{aligned}
\sum_{h=1}^{k-1} \frac{h}{2^h} = {} & \left.\frac{1}{2} + \frac{1}{2^2} + \frac{1}{2^3} + \cdots + \frac{1}{2^{k-1}}\right\} \quad 1 - \frac{1}{2^{k-1}} < 1 \\
& \left.+ \frac{1}{2^2} + \frac{1}{2^3} + \cdots + \frac{1}{2^{k-1}}\right\} \quad \frac{1}{2} - \frac{1}{2^{k-1}} < \frac{1}{2} \\
& \left.+ \frac{1}{2^3} + \cdots + \frac{1}{2^{k-1}}\right\} \quad \frac{1}{4} - \frac{1}{2^{k-1}} < \frac{1}{4}
\end{aligned}$$

Ou seja,

$$T(n) = O(2^k) \sum_{h=0}^{k} \frac{1}{2^h} \le O(2^k) \cdot \sum_{h=0}^{\infty} \frac{1}{2^h} = O(2^k) \cdot 2 = O(n)$$

Complexidade de BUILD-MAX-HEAP (cont)

Seja $k = 1 + \lfloor \lg n \rfloor$ a altura da árvore inteira, então

$$T(n) = \sum_{h=1}^{k-1} 2^{k-h} \cdot O(h) = O(2^k) \sum_{h=1}^{k-1} \frac{h}{2^h}$$

Note que

$$\begin{aligned}
\sum_{h=1}^{k-1} \frac{h}{2^h} = \;& \left.\frac{1}{2} + \frac{1}{2^2} + \frac{1}{2^3} + \cdots + \frac{1}{2^{k-1}}\right\} \quad 1 - \frac{1}{2^{k-1}} < 1 \\
& \left.+ \frac{1}{2^2} + \frac{1}{2^3} + \cdots + \frac{1}{2^{k-1}}\right\} \quad \frac{1}{2} - \frac{1}{2^{k-1}} < \frac{1}{2} \\
& \left.+ \frac{1}{2^3} + \cdots + \frac{1}{2^{k-1}}\right\} \quad \frac{1}{4} - \frac{1}{2^{k-1}} < \frac{1}{4}
\end{aligned}$$

Ou seja,

$$T(n) = O(2^k) \sum_{h=0}^{k} \frac{1}{2^h} \leq O(2^k) \cdot \sum_{h=0}^{\infty} \frac{1}{2^h} = O(2^k) \cdot 2 = O(n)$$

Complexidade de BUILD-MAX-HEAP (cont)

Seja $k = 1 + \lfloor \lg n \rfloor$ a altura da árvore inteira, então

$$T(n) = \sum_{h=1}^{k-1} 2^{k-h} \cdot O(h) = O(2^k) \sum_{h=1}^{k-1} \frac{h}{2^h}$$

Note que

$$\begin{array}{rl}
\displaystyle\sum_{h=1}^{k-1} \frac{h}{2^h} = & \left.\dfrac{1}{2} + \dfrac{1}{2^2} + \dfrac{1}{2^3} + \cdots + \dfrac{1}{2^{k-1}} \right\} \quad 1 - \dfrac{1}{2^{k-1}} < 1 \\
& \left.\quad\;\; + \dfrac{1}{2^2} + \dfrac{1}{2^3} + \cdots + \dfrac{1}{2^{k-1}} \right\} \quad \dfrac{1}{2} - \dfrac{1}{2^{k-1}} < \dfrac{1}{2} \\
& \left.\quad\quad\quad\quad\;\; + \dfrac{1}{2^3} + \cdots + \dfrac{1}{2^{k-1}} \right\} \quad \dfrac{1}{4} - \dfrac{1}{2^{k-1}} < \dfrac{1}{4}
\end{array}$$

Ou seja,

$$T(n) = O(2^k) \sum_{h=0}^{k} \frac{1}{2^h} \leq O(2^k) \cdot \sum_{h=0}^{\infty} \frac{1}{2^h} = O(2^k) \cdot 2 = O(n)$$

Complexidade de BUILD-MAX-HEAP (cont)

Seja $k = 1 + \lfloor \lg n \rfloor$ a altura da árvore inteira, então

$$T(n) = \sum_{h=1}^{k-1} 2^{k-h} \cdot O(h) = O(2^k) \sum_{h=1}^{k-1} \frac{h}{2^h}$$

Note que

$$\begin{array}{rll}
\displaystyle\sum_{h=1}^{k-1} \frac{h}{2^h} = & \displaystyle\frac{1}{2} + \frac{1}{2^2} + \frac{1}{2^3} + \cdots + \frac{1}{2^{k-1}} & \Big\} \quad \displaystyle 1 - \frac{1}{2^{k-1}} < 1 \\
& \displaystyle + \frac{1}{2^2} + \frac{1}{2^3} + \cdots + \frac{1}{2^{k-1}} & \Big\} \quad \displaystyle \frac{1}{2} - \frac{1}{2^{k-1}} < \frac{1}{2} \\
& \displaystyle + \frac{1}{2^3} + \cdots + \frac{1}{2^{k-1}} & \Big\} \quad \displaystyle \frac{1}{4} - \frac{1}{2^{k-1}} < \frac{1}{4}
\end{array}$$

Ou seja,

$$T(n) = O(2^k) \sum_{h=0}^{k} \frac{1}{2^h} \leq O(2^k) \cdot \sum_{h=0}^{\infty} \frac{1}{2^h} = O(2^k) \cdot 2 = O(n)$$

Complexidade de Build-Max-Heap (cont)

Seja $k = 1 + \lfloor \lg n \rfloor$ a altura da árvore inteira, então

$$T(n) = \sum_{h=1}^{k-1} 2^{k-h} \cdot O(h) = O(2^k) \sum_{h=1}^{k-1} \frac{h}{2^h}$$

Note que

$$\begin{array}{rl}
\displaystyle\sum_{h=1}^{k-1} \frac{h}{2^h} = & \left.\displaystyle \frac{1}{2} + \frac{1}{2^2} + \frac{1}{2^3} + \cdots + \frac{1}{2^{k-1}} \right\} \quad 1 - \dfrac{1}{2^{k-1}} < 1 \\
& \left.\displaystyle + \frac{1}{2^2} + \frac{1}{2^3} + \cdots + \frac{1}{2^{k-1}} \right\} \quad \dfrac{1}{2} - \dfrac{1}{2^{k-1}} < \dfrac{1}{2} \\
& \left.\displaystyle + \frac{1}{2^3} + \cdots + \frac{1}{2^{k-1}} \right\} \quad \dfrac{1}{4} - \dfrac{1}{2^{k-1}} < \dfrac{1}{4}
\end{array}$$

Ou seja,

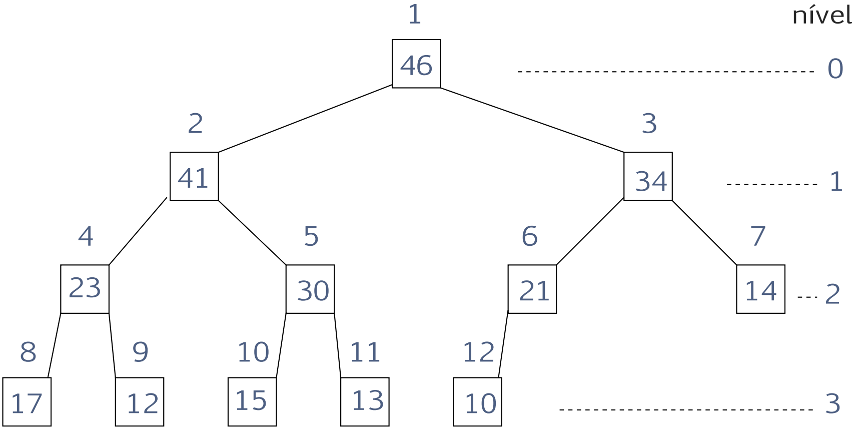
$$T(n) = O(2^k) \sum_{h=0}^{k} \frac{1}{2^h} \leq O(2^k) \cdot \sum_{h=0}^{\infty} \frac{1}{2^h} = O(2^k) \cdot 2 = O(n)$$

O algoritmo HEAP-SORT

HEAP-SORT(A, n)

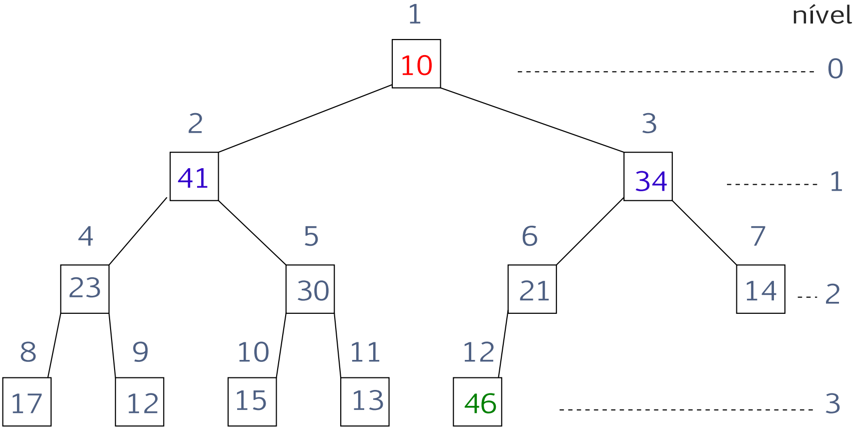
BUILD-MAX-HEAP(A, n)
para $m \leftarrow n$ **decrescendo até** 2 **faça**
  $A[1] \leftrightarrow A[m]$
  MAX-HEAPIFY$(A, m, 1)$

Execução de HEAP-SORT



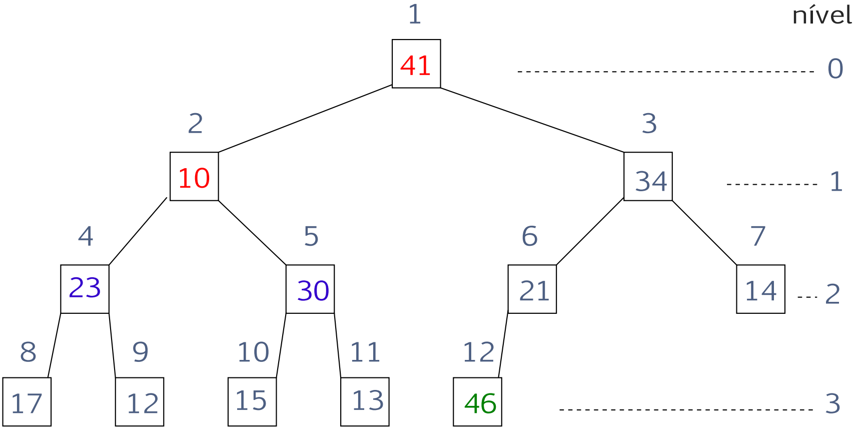
1	2	3	4	5	6	7	8	9	10	11	12
46	41	34	23	30	21	14	17	12	15	13	10

Execução de Heap-Sort



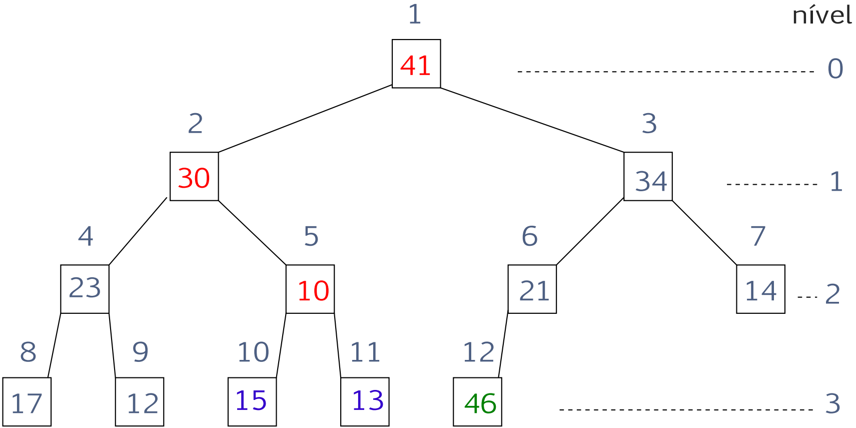
1	2	3	4	5	6	7	8	9	10	11	12
10	41	34	23	30	21	14	17	12	15	13	46

Execução de HEAP-SORT



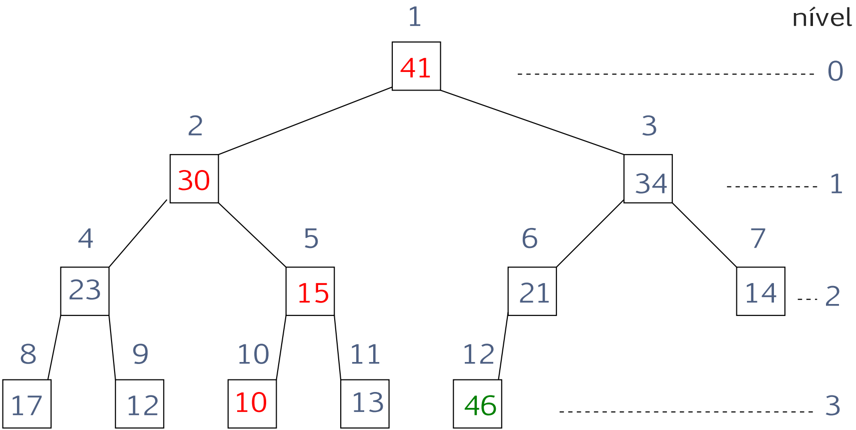
1	2	3	4	5	6	7	8	9	10	11	12
41	10	34	23	30	21	14	17	12	15	13	46

Execução de HEAP-SORT



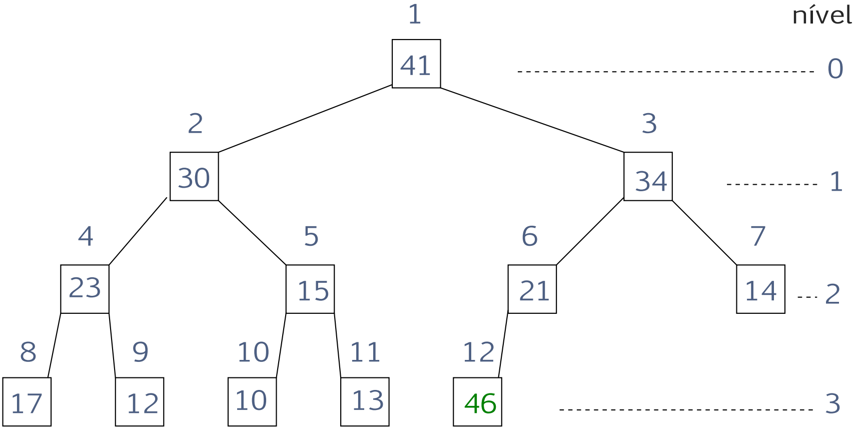
1	2	3	4	5	6	7	8	9	10	11	12
41	30	34	23	10	21	14	17	12	15	13	46

Execução de Heap-Sort



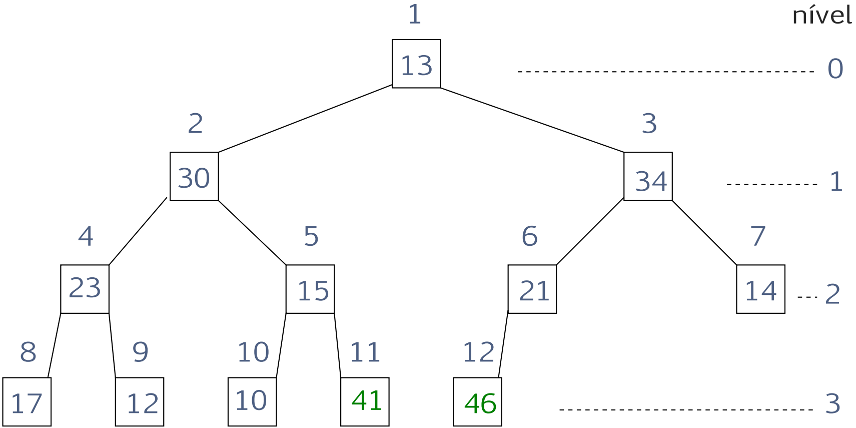
1	2	3	4	5	6	7	8	9	10	11	12
41	30	34	23	15	21	14	17	12	10	13	46

Execução de HEAP-SORT



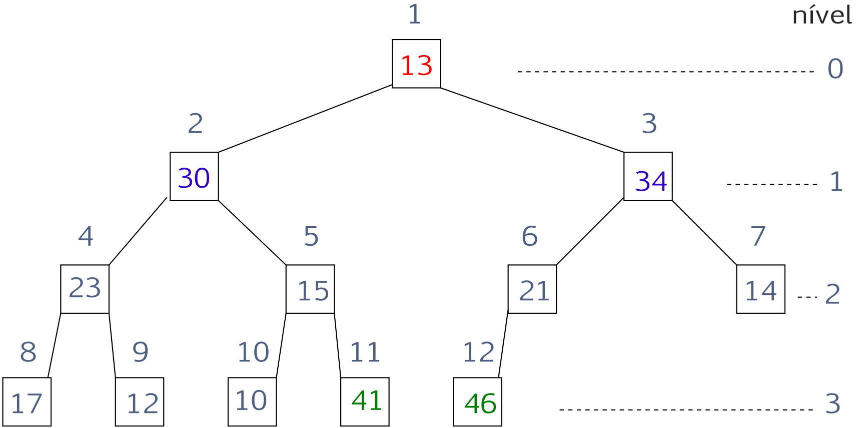
1	2	3	4	5	6	7	8	9	10	11	12
41	30	34	23	15	21	14	17	12	10	13	46

Execução de HEAP-SORT



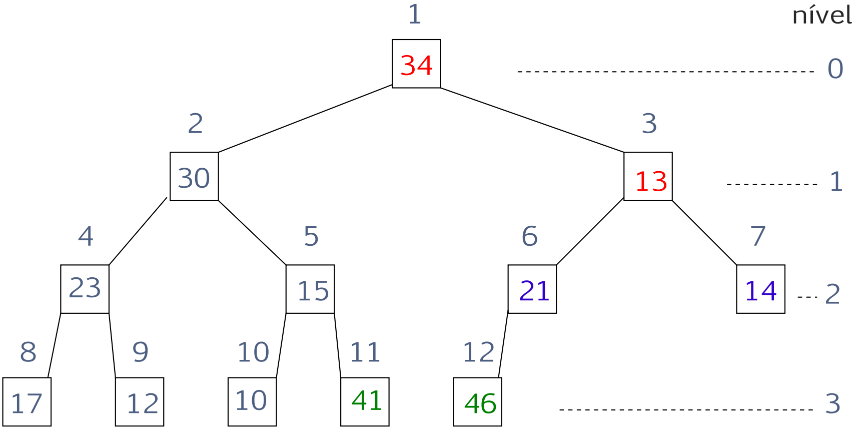
1	2	3	4	5	6	7	8	9	10	11	12
13	30	34	23	15	21	14	17	12	10	41	46

Execução de Heap-Sort



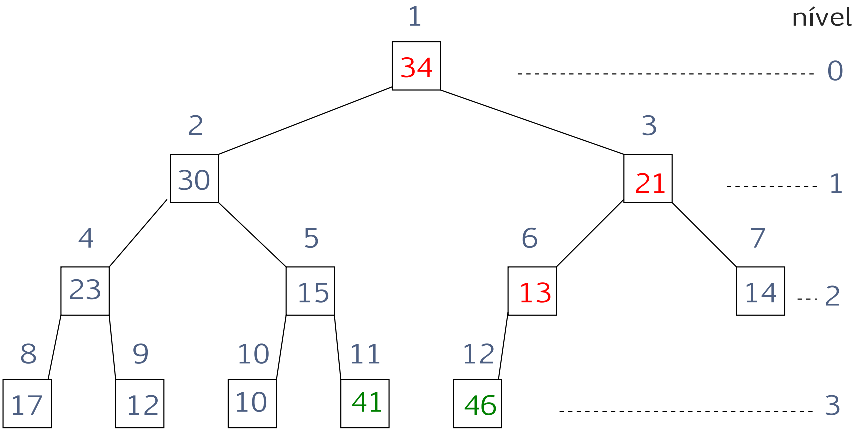
1	2	3	4	5	6	7	8	9	10	11	12
13	30	34	23	15	21	14	17	12	10	41	46

Execução de Heap-Sort



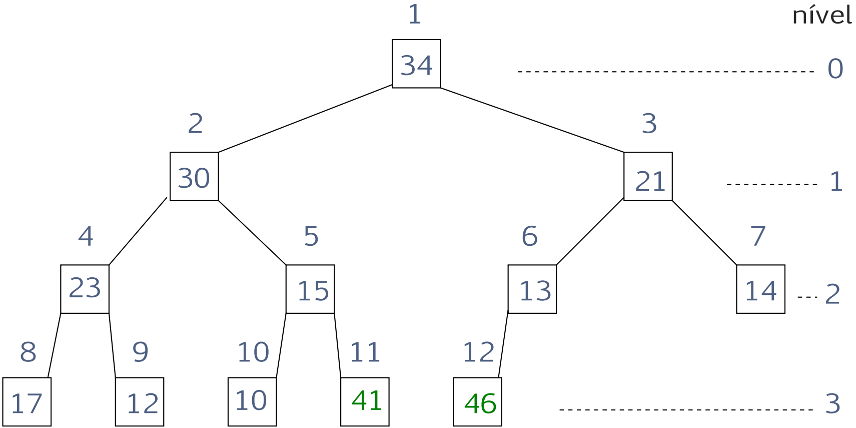
1	2	3	4	5	6	7	8	9	10	11	12
34	30	13	23	15	21	14	17	12	10	41	46

Execução de HEAP-SORT



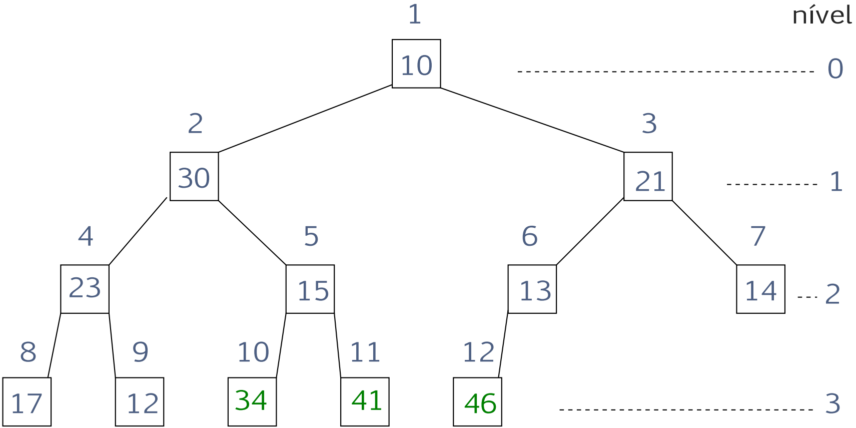
1	2	3	4	5	6	7	8	9	10	11	12
34	30	21	23	15	13	14	17	12	10	41	46

Execução de HEAP-SORT



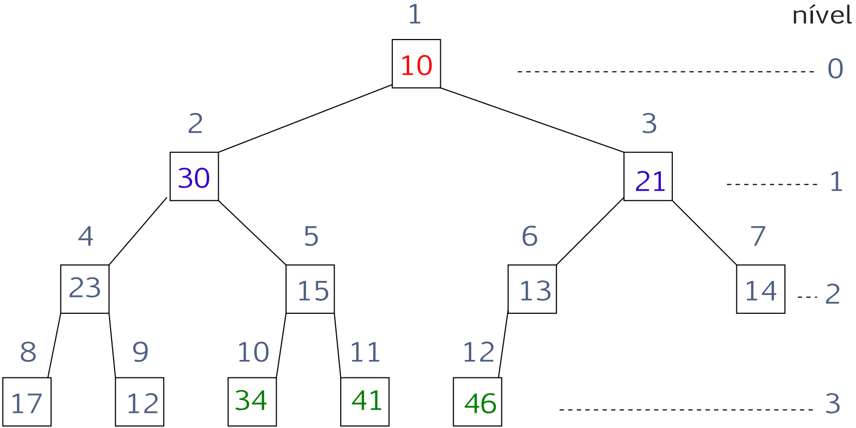
1	2	3	4	5	6	7	8	9	10	11	12
34	30	21	23	15	13	14	17	12	10	41	46

Execução de HEAP-SORT



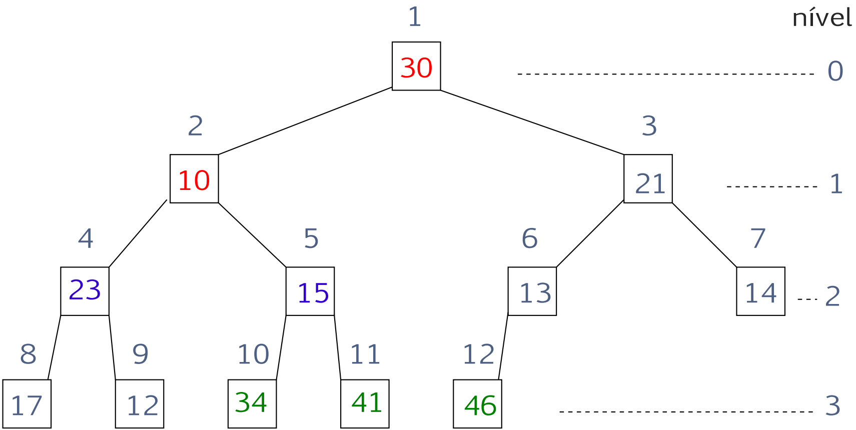
1	2	3	4	5	6	7	8	9	10	11	12
10	30	21	23	15	13	14	17	12	34	41	46

Execução de Heap-Sort



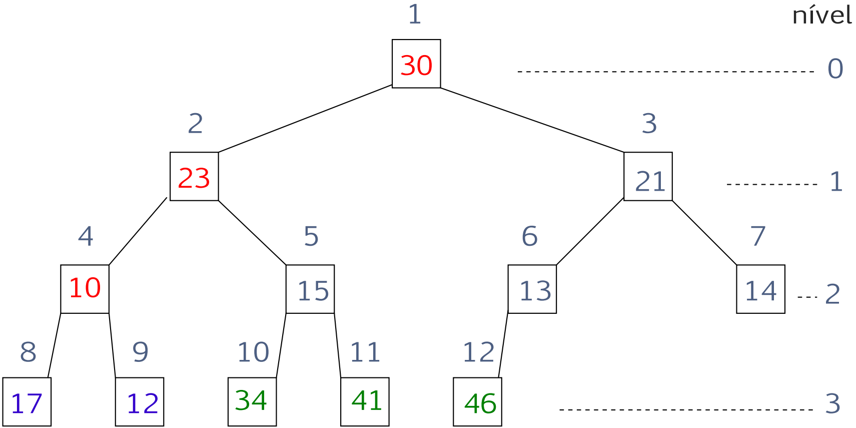
1	2	3	4	5	6	7	8	9	10	11	12
10	30	21	23	15	13	14	17	12	34	41	46

Execução de Heap-Sort



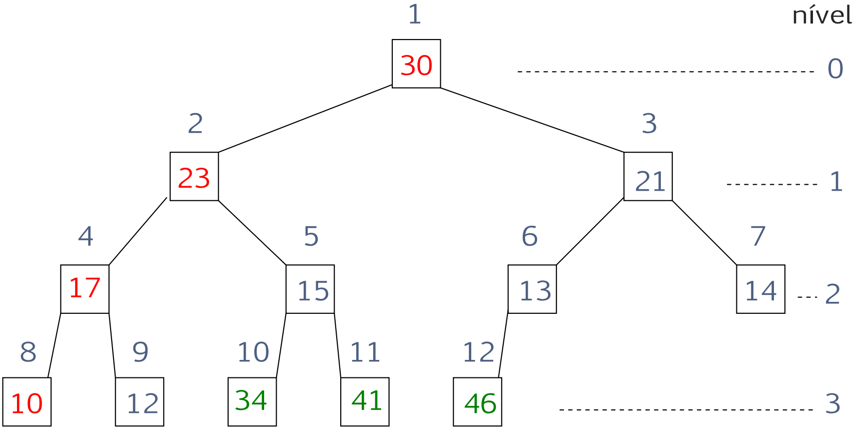
1	2	3	4	5	6	7	8	9	10	11	12
30	10	21	23	15	13	14	17	12	34	41	46

Execução de Heap-Sort



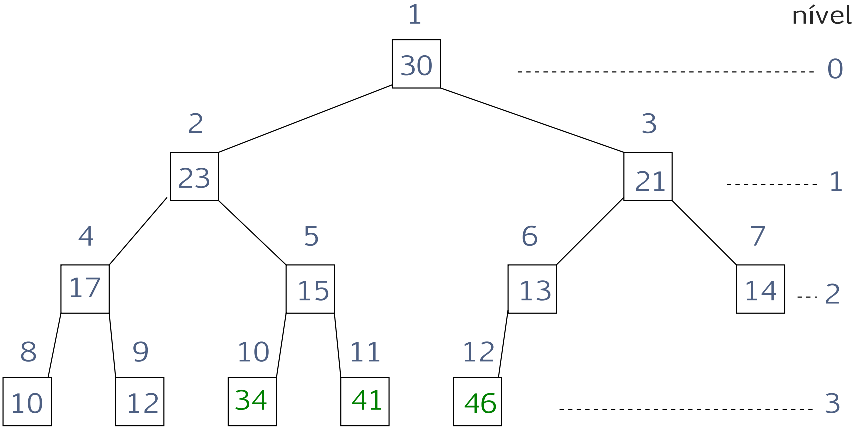
1	2	3	4	5	6	7	8	9	10	11	12
30	23	21	10	15	13	14	17	12	34	41	46

Execução de HEAP-SORT



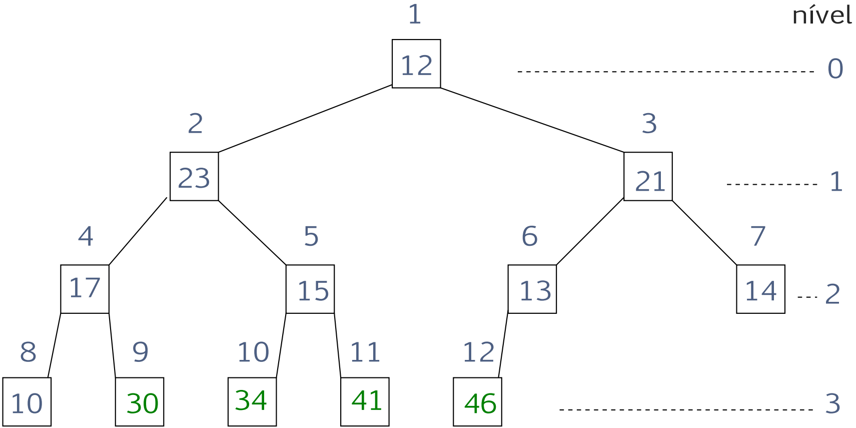
1	2	3	4	5	6	7	8	9	10	11	12
30	23	21	17	15	13	14	10	12	34	41	46

Execução de HEAP-SORT



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

Execução de HEAP-SORT



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

Análise de HEAP-SORT

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

Invariantes

No início de cada iteração vale

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

Análise de Heap-Sort

```
HEAP-SORT(A, n)
    BUILD-MAX-HEAP(A, n)
    para m ← n decrescendo até 2 faça
        A[1] ↔ A[m]
        MAX-HEAPIFY(A, m, 1)
```

Invariantes

No início de cada iteração vale

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

Análise de Heap-Sort

```
Heap-Sort(A, n)
    Build-Max-Heap(A, n)
    para m ← n decrescendo até 2 faça
        A[1] ↔ A[m]
        Max-Heapify(A, m, 1)
```

Invariantes

No início de cada iteração vale

1. $A[m+1\dots n]$ é crescente
2. $A[1\dots m] \le A[m+1]$
3. $A[1\dots m]$ é um max-heap

Análise de HEAP-SORT

HEAP-SORT(A, n)

BUILD-MAX-HEAP(A, n)
para $m \leftarrow n$ **decrescendo até** 2 **faça**
    $A[1] \leftrightarrow A[m]$
    MAX-HEAPIFY$(A, m, 1)$

Invariantes

No início de cada iteração vale

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

Complexidade de HEAP-SORT

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

Consumo de tempo no pior caso? $O(n \log n)$

E no melhor caso?

Complexidade de HEAP-SORT

HEAP-SORT(A, n)	Tempo
BUILD-MAX-HEAP(A, n)	$\Theta(n)$
para $m \leftarrow n$ **decrescendo até** 2 **faça**	$\Theta(n)$
$\quad A[1] \leftrightarrow A[m]$	$\Theta(n)$
$\quad$ MAX-HEAPIFY$(A, m, 1)$	$n \cdot O(\lg n)$

Consumo de tempo no pior caso? $O(n \log n)$

E no melhor caso?

Complexidade de Heap-Sort

Heap-Sort(A, n)		Tempo
1	Build-Max-Heap(A, n)	$\Theta(n)$
2	**para** $m \leftarrow n$ **decrescendo até** 2 **faça**	$\Theta(n)$
3	$A[1] \leftrightarrow A[m]$	$\Theta(n)$
4	Max-Heapify$(A, m, 1)$	$n \cdot O(\lg n)$

Consumo de tempo no pior caso? $O(n \log n)$

E no melhor caso?

Complexidade de HEAP-SORT

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

Consumo de tempo no pior caso? $O(n \log n)$

E no melhor caso?

Complexidade de HEAP-SORT

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

Consumo de tempo no pior caso? $O(n \log n)$
E no melhor caso?

Fila de prioridades

Filas de prioridades

Definição

Uma fila de prioridades é um tipo abstrato de dados que consiste de uma coleção S de itens com prioridades associadas e permite as operações

- Maximum(S) devolve um elemento de maior prioridade
- Extract-Max(S) remove um elemento de maior prioridade
- Increase-Key(S, x, p) aumenta a prioridade de x para p
- Insert(S, x, p) insere um elemento x com prioridade p

Filas de prioridades

Definição

Uma fila de prioridades é um tipo abstrato de dados que consiste de uma coleção S de itens com prioridades associadas e permite as operações

- $\textsc{Maximum}(S)$ devolve um elemento de maior prioridade
- $\textsc{Extract-Max}(S)$ remove um elemento de maior prioridade
- $\textsc{Increase-Key}(S, x, p)$ aumenta a prioridade de x para p
- $\textsc{Insert}(S, x, p)$ insere um elemento x com prioridade p

Filas de prioridades

Definição

Uma fila de prioridades é um tipo abstrato de dados que consiste de uma coleção S de itens com prioridades associadas e permite as operações

- MAXIMUM(S) devolve um elemento de maior prioridade
- EXTRACT-MAX(S) remove um elemento de maior prioridade
- INCREASE-KEY(S, x, p) aumenta a prioridade de x para p
- INSERT(S, x, p) insere um elemento x com prioridade p

Filas de prioridades

Definição

Uma fila de prioridades é um tipo abstrato de dados que consiste de uma coleção S de itens com prioridades associadas e permite as operações

- $\textsc{Maximum}(S)$ devolve um elemento de maior prioridade
- $\textsc{Extract-Max}(S)$ remove um elemento de maior prioridade
- $\textsc{Increase-Key}(S, x, p)$ aumenta a prioridade de x para p
- $\textsc{Insert}(S, x, p)$ insere um elemento x com prioridade p

Filas de prioridades

Definição

Uma fila de prioridades é um tipo abstrato de dados que consiste de uma coleção S de itens com prioridades associadas e permite as operações

- $\textsc{Maximum}(S)$ devolve um elemento de maior prioridade
- $\textsc{Extract-Max}(S)$ remove um elemento de maior prioridade
- $\textsc{Increase-Key}(S, x, p)$ aumenta a prioridade de x para p
- $\textsc{Insert}(S, x, p)$ insere um elemento x 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 $A[1] \leftarrow A[n]$
2 $n \leftarrow n - 1$
3 Max-Heapify$(A, n, 1)$

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

Implementação com max-heap

Heap-Max(A, n)
1 **devolva** $A[1]$

Complexidade de tempo: $\Theta(1)$

Heap-Extract-Max(A, n)
1 $A[1] \leftarrow A[n]$
2 $n \leftarrow n - 1$
3 Max-Heapify $(A, n, 1)$

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

Implementação com max-heap

Heap-Max(A, n)
1 **devolva** $A[1]$

Complexidade de tempo: $\Theta(1)$

Heap-Extract-Max(A, n)
1 $A[1] \leftarrow A[n]$
2 $n \leftarrow n - 1$
3 Max-Heapify$(A, n, 1)$

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

Implementação com max-heap

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

Complexidade de tempo: $\Theta(1)$

```
HEAP-EXTRACT-MAX(A, n)
1    A[1] ← A[n]
2    n ← n − 1
3    MAX-HEAPIFY(A, n, 1)
```

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

Implementação com max-heap

Heap-Increase-Key$(A, i, chave)$

$A[i] \leftarrow chave$

enquanto $i > 1$ **e** $A[\lfloor i/2 \rfloor] < A[i]$ **faça**

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

  $i \leftarrow \lfloor i/2 \rfloor$

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

Max-Heap-Insert$(A, n, chave)$

$n \leftarrow n + 1$

$A[n] \leftarrow -\infty$

Heap-Increase-Key$(A, n, chave)$

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

Implementação com max-heap

```
HEAP-INCREASE-KEY(A, i, chave)
    A[i] ← chave
    enquanto i > 1 e A[⌊i/2⌋] < A[i] faça
        A[i] ↔ A[⌊i/2⌋]
        i ← ⌊i/2⌋
```

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

```
MAX-HEAP-INSERT(A, n, chave)
    n ← n + 1
    A[n] ← −∞
    HEAP-INCREASE-KEY(A, n, chave)
```

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

Implementação com max-heap

Heap-Increase-Key$(A, i, chave)$

$A[i] \leftarrow chave$
enquanto $i > 1$ **e** $A[\lfloor i/2 \rfloor] < A[i]$ **faça**
  $A[i] \leftrightarrow A[\lfloor i/2 \rfloor]$
  $i \leftarrow \lfloor i/2 \rfloor$

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

Max-Heap-Insert$(A, n, chave)$

$n \leftarrow n + 1$
$A[n] \leftarrow -\infty$
Heap-Increase-Key$(A, n, chave)$

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

Implementação com max-heap

```
HEAP-INCREASE-KEY(A, i, chave)
    A[i] ← chave
    enquanto i > 1 e A[⌊i/2⌋] < A[i] faça
        A[i] ↔ A[⌊i/2⌋]
        i ← ⌊i/2⌋
```

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

```
MAX-HEAP-INSERT(A, n, chave)
    n ← n + 1
    A[n] ← −∞
    HEAP-INCREASE-KEY(A, n, chave)
```

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