Selection Sort, Heap Sort & Fila de Prioridade

Disclaimer

Slides baseados nos slides dos Professores

- Cândida Nunes da Silva
- Carla N. Lintzmayer
- Cid Carvalho
- Lehilton Pedrosa
- Orlando Lee

Selection Sort

Ideia:

- o subVetor $A[1..i-1]$ está ordenado
- $max(A[1..i-1]) \leq Min(A[i..n])$
- Substituímos a posição $A[i]$ pelo mínimo em $A[i..n]$

```
SelectionSort(A,n)
    Para  i ← 1  até  n-1
          min ← i
          para  j ← i+1  até  n
                Se  A[j] < A[min]
                    min ← j
          A[i] ↔ A[min]
```

Análise de tempo de Execução

```
SelectionSort(A, n)
    Para i ← 1 até n-1
        min ← i
        para j ← i+1 até n
            Se A[j] < A[min]
                min ← j
        A[i] ↔ A[min]
```

Análise de tempo de Execução

```
SelectionSort(A, n)
    Para i ← 1 até n-1                          Θ(n)
        min ← i                                 Θ(n)
        para j ← i+1 até n                      ⎫
            Se A[j] < A[min]                    ⎬ O(n) · Θ(n) = O(n²)
                min ← j                         ⎭
        A[i] ⇔ A[min]                           Θ(n)
```

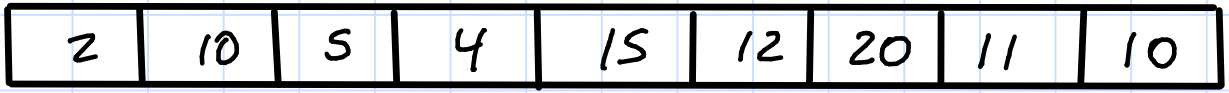
T(n) é $O(n^2)$

Versão Alternativa

Vamos reescrever esse algonitmo para ordenar a partir do final

Ex

A =



Ideia

- A[i+1..n] está ordenado
- A[1..i] ≤ A[i+1]

```
Selection-Sort (A, n)
    Para  i ← n  decrescendo até 2
          max ← Maximum (A, i)
          A[i] ↔ A[max]

Maximum (A, i)
    max ← i
    para j ← i-1  decrescendo até 1
          se  A[j] > A[max]
                max ← j
    Devolva  A[max]
```

Análise de Selection-Sort

Assuma que Maximum(A, i) leva tempo $O(t(i))$

```
Selection-Sort (A, n)
    Para i ← n decrescendo até 2
        max ← Maximum (A, i)
        A[i] ↔ A[max]
```

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

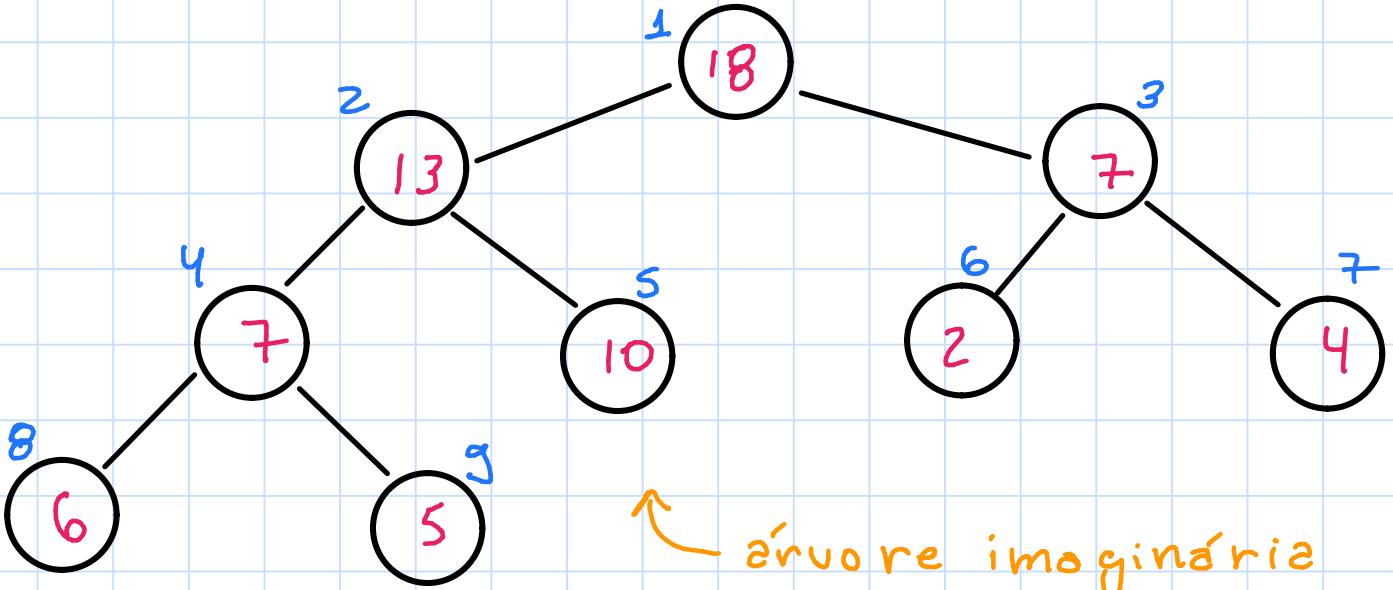
Vamos tentar melhorar Maximum

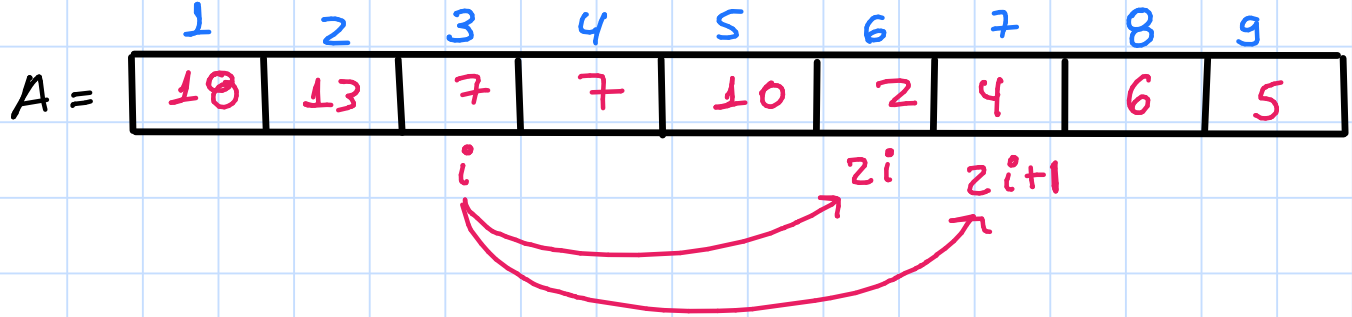
(Max)-Heap

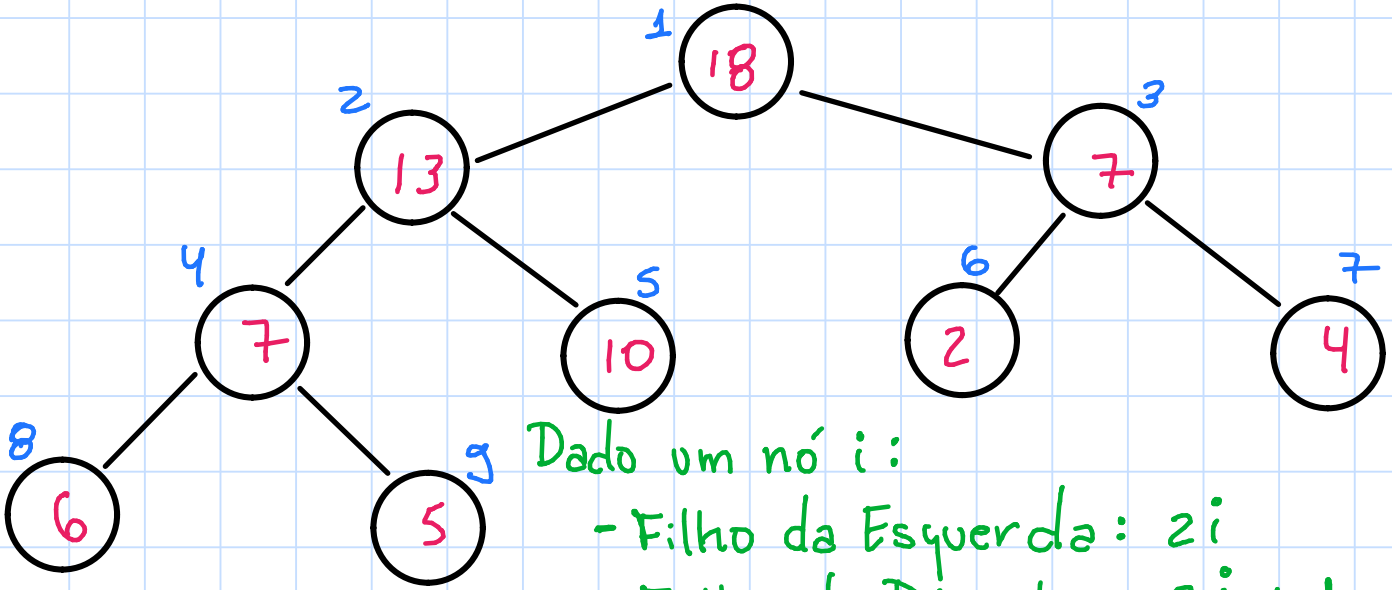
Heap

- Um heap é uma árvore binária armazenada em um vetor $A[1..n]$

	1	2	3	4	5	6	7	8	9
A =	18	13	7	7	10	2	4	6	5

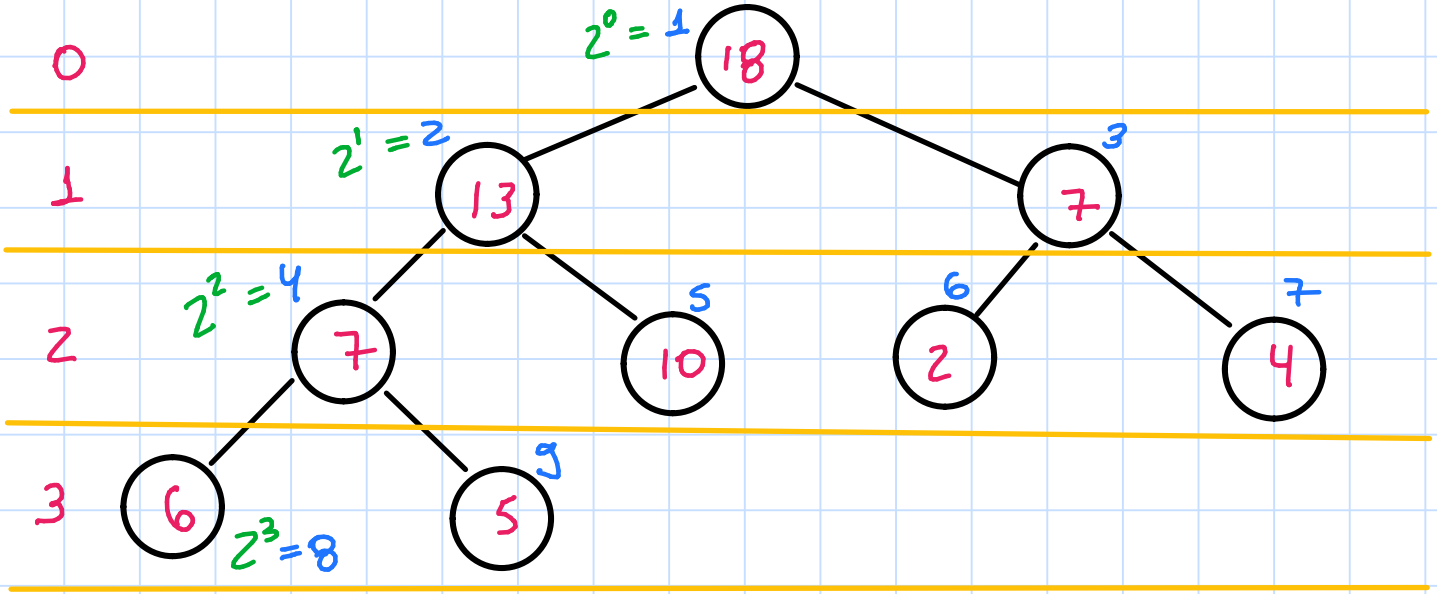






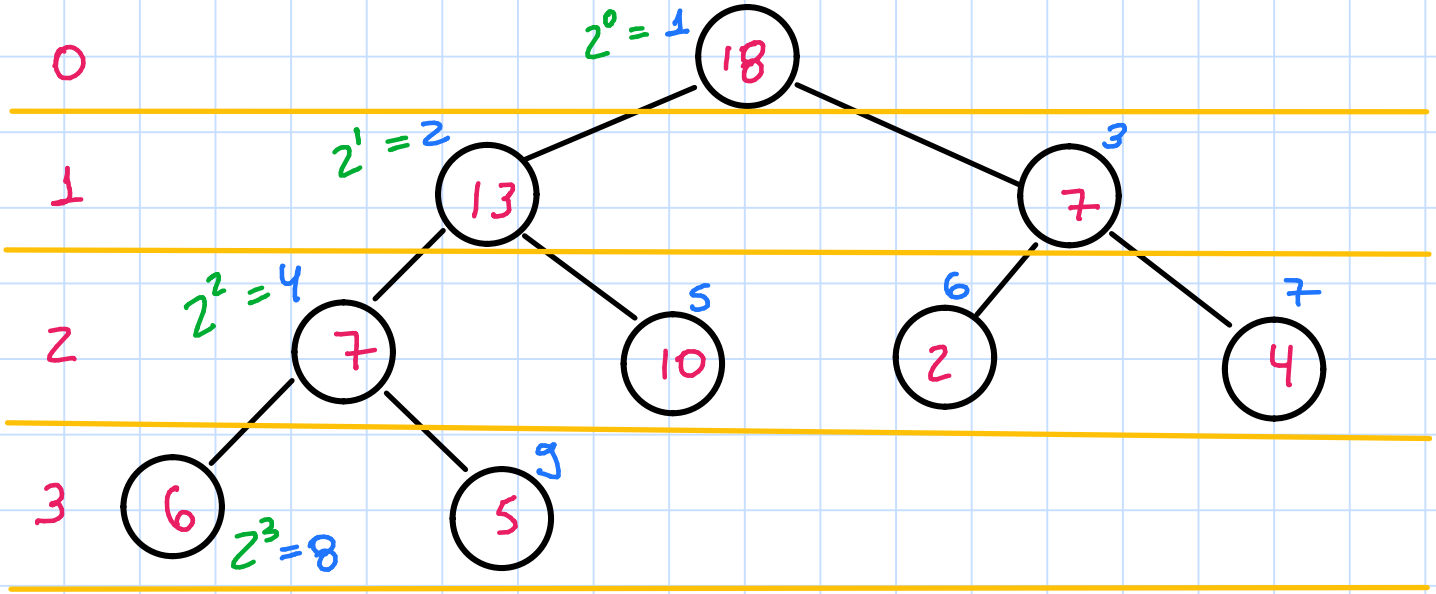
Dado um nó i:

- Filho da Esquerda: $2i$
- Filho da Direita: $2i+1$
- Pai: $\lfloor i/2 \rfloor$



Um heap é uma árvore completa

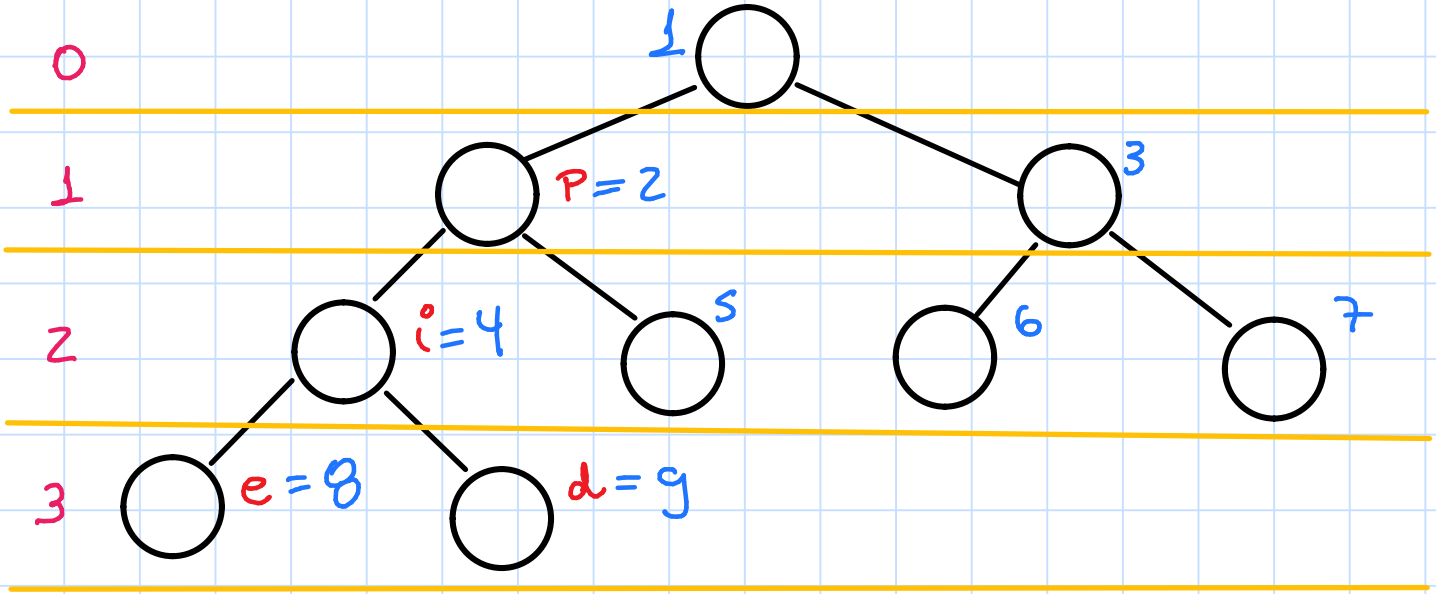
- Cada nível $\ell = 0, 1, 2, \ldots$ tem 2^ℓ nós (com exceção do último)



Seja i um vértice no nível ℓ

$$i = \sum_{i=0}^{\ell-1} 2^i + a = 2^\ell - 1 + a, \quad \text{onde} \quad 1 \leq a \leq 2^\ell$$

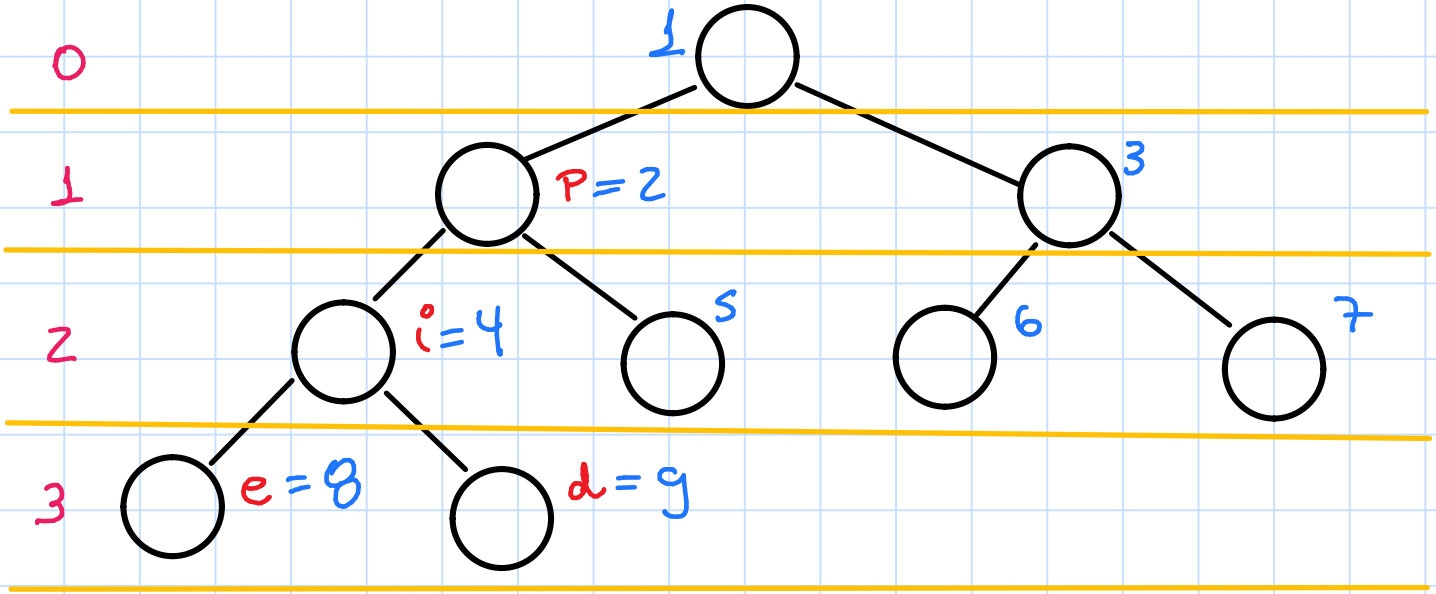
Seja e, d e p os rótulos, respectivamente, do filho da esquerda, fil. da direita e do pai $\Rightarrow$



$$i = 2^{l} + a - 1$$

filho da esquerda $= 2^{l+1} + 2(a-1) = 2\left(2^{l} + a - 1\right)$

$= 2i$

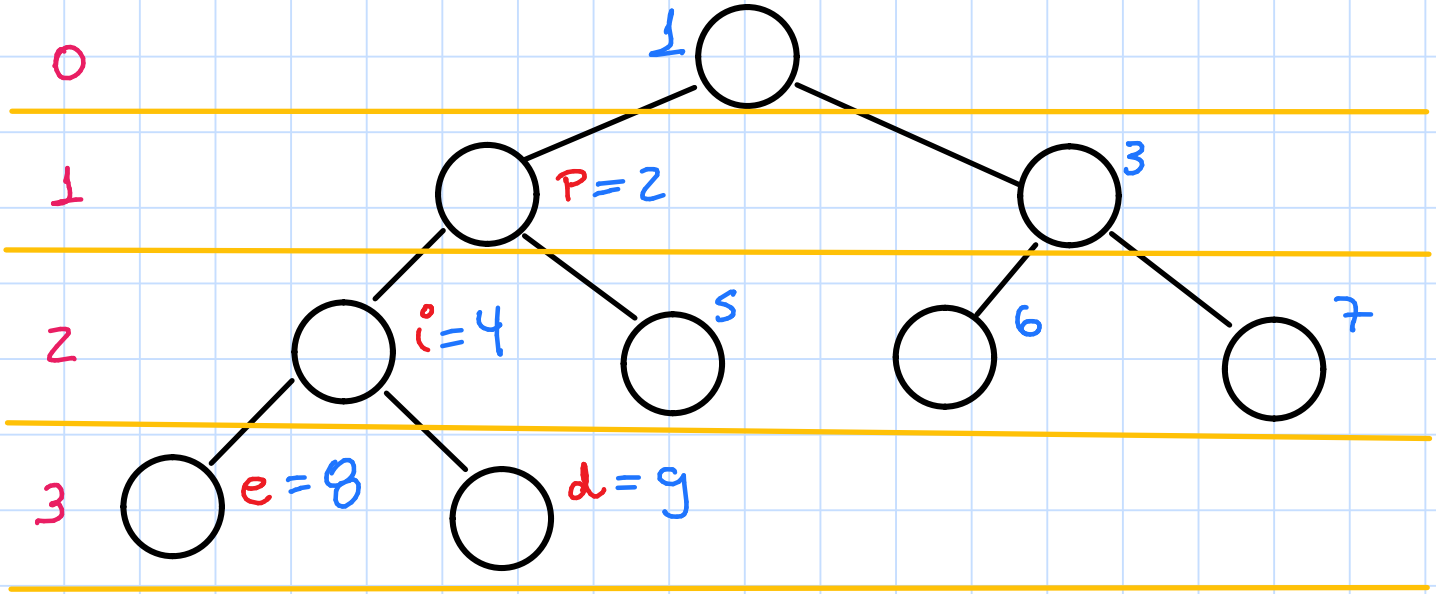


$$i = 2^l + a - 1$$

filho da direita $= 2^{l+1} + 2(a-1) + 1$

$= 2(2^l + a - 1) + 1$

$= 2i + 1$



Se i é o filho da esquerda

$$i = 2p \iff \frac{i}{2} = p$$

Se i é o filho da direita

$$i = 2p + 1 \iff \frac{i-1}{2} = p$$

$$= \left\lfloor \frac{i}{2} \right\rfloor$$

ímpar

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

- Note que o rótulo do primeiro nó em um nível ℓ é

$$i = 2^{\ell} + a - 1 = 2^{\ell} + 1 - 1 = 2^{\ell}$$

- Se i está em um nível ℓ, então

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

$$\lg 2^{\ell} \leq \lg i$$

$$\ell \leq \lg i$$

$$\ell \leq \lfloor \lg i \rfloor$$

inteiro

$$\lg i < \lg 2^{\ell+1}$$

$$\lg i - 1 < \ell$$

AFIRMAÇÃO $\lfloor \lg i \rfloor \leq \ell \Rightarrow$

$\lg i - 1 < \ell$

se $\lg i$ é inteiro:

$\lg i - 1 < \ell \Rightarrow \lg i \leq \ell$, que podemos escrever como

$$\lfloor \lg i \rfloor \leq \ell$$

se $\lg i$ não é inteiro

$\lg i - 1 < \ell \Rightarrow \lceil \lg i - 1 \rceil \leq \ell \Leftrightarrow \lceil \lg i \rceil - 1 \leq \ell$

$\lfloor \lg i \rfloor + 1 - 1 \leq \ell \Leftrightarrow \lfloor \lg i \rfloor \leq \ell$

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

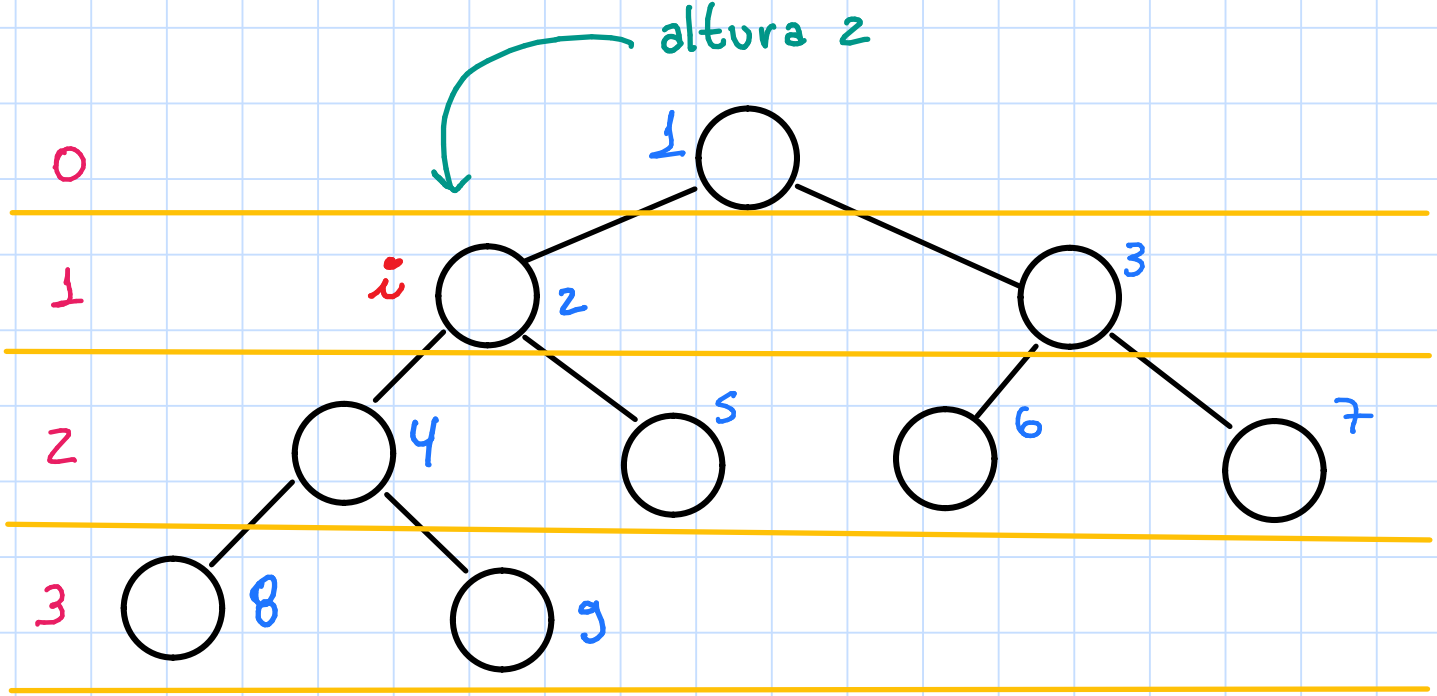
$\Rightarrow$ como consequência, a árvore de uma heap de n elementos tem

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

níveis

↳ Lembre-se que o 1º nível tem rótulo zero

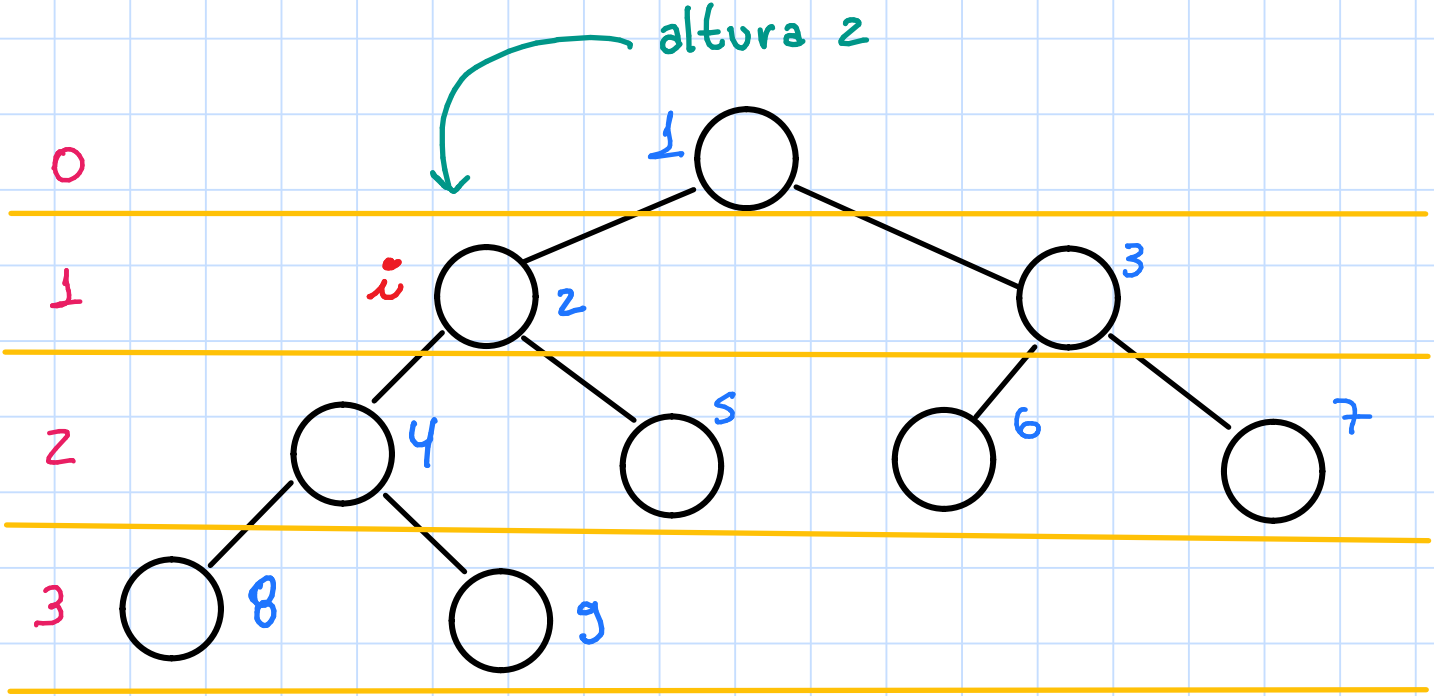
Altura de um nó i é o maior nº de arestas do no i até uma folha na subárvore enraizada em i



Altura da árvore é a altura do nó raiz

Árvore de uma heap tem $\lfloor \lg n \rfloor + 1$ níveis

$\Rightarrow$ altura de um heap é $\lfloor \lg n \rfloor = O(\lg n)$



Funções

Pai (i)
    Devolva $\lfloor i/2 \rfloor$

Filho-Esquerda (i)
    Devolva $2i$

Filho-Direita (i)
    Devolva $2i+1$

Heap Máximo e Mínimo

Há dois tipos de Heap
- máximo ← Se falarmos simplesmente Heap, Vamos assumir que é Heap Máximo.
- mínimo

Em um heap mínimo, cada nó é menor que seus filhos, i.e.,

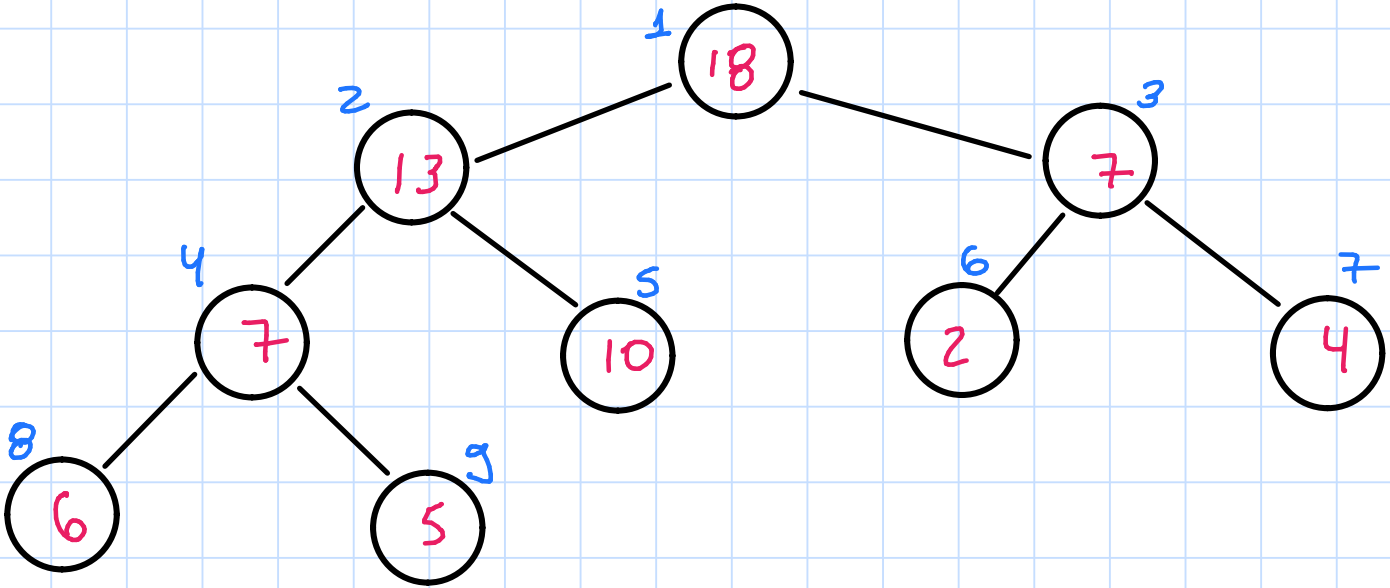
$$A[pai(i)] \leq A[i] \quad \forall \quad 2 \leq i \leq n$$

Em um heap máximo, cada nó é maior que seus filhos, i.e.,

$$A[pai(i)] \geq A[i] \quad \forall \quad 2 \leq i \leq n$$

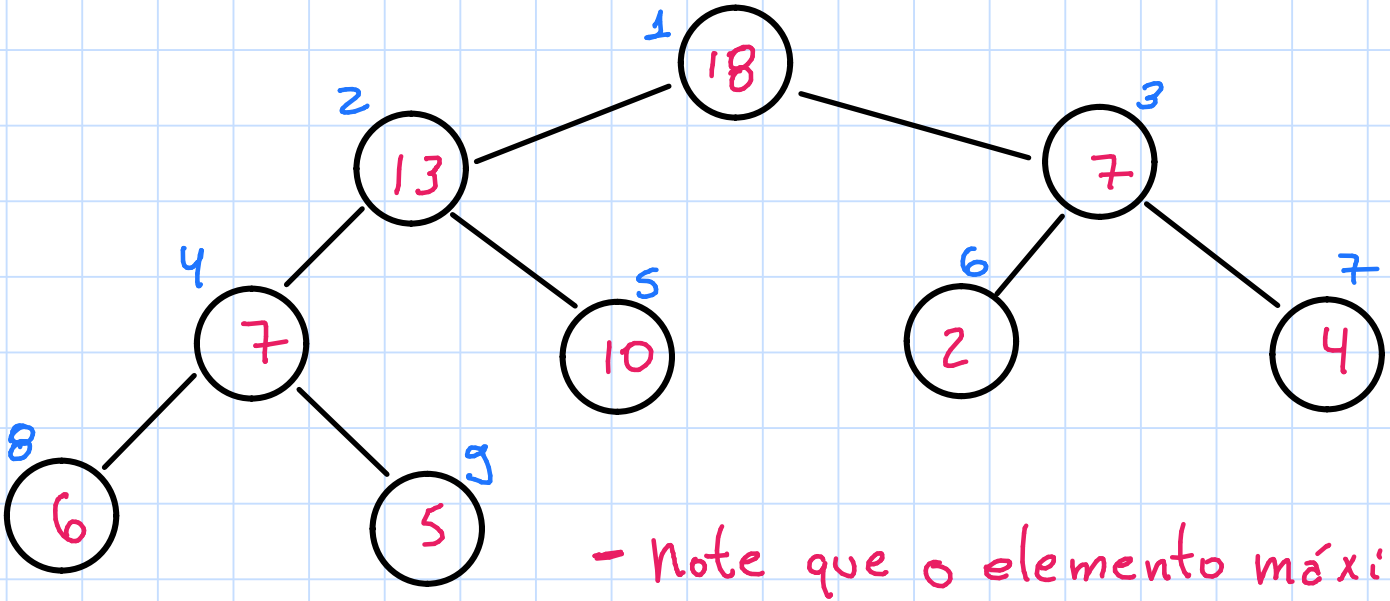
Exemplo de Heap (Máximo)

	1	2	3	4	5	6	7	8	9
A =	18	13	7	7	10	2	4	6	5



Exemplo de Heap (Máximo)

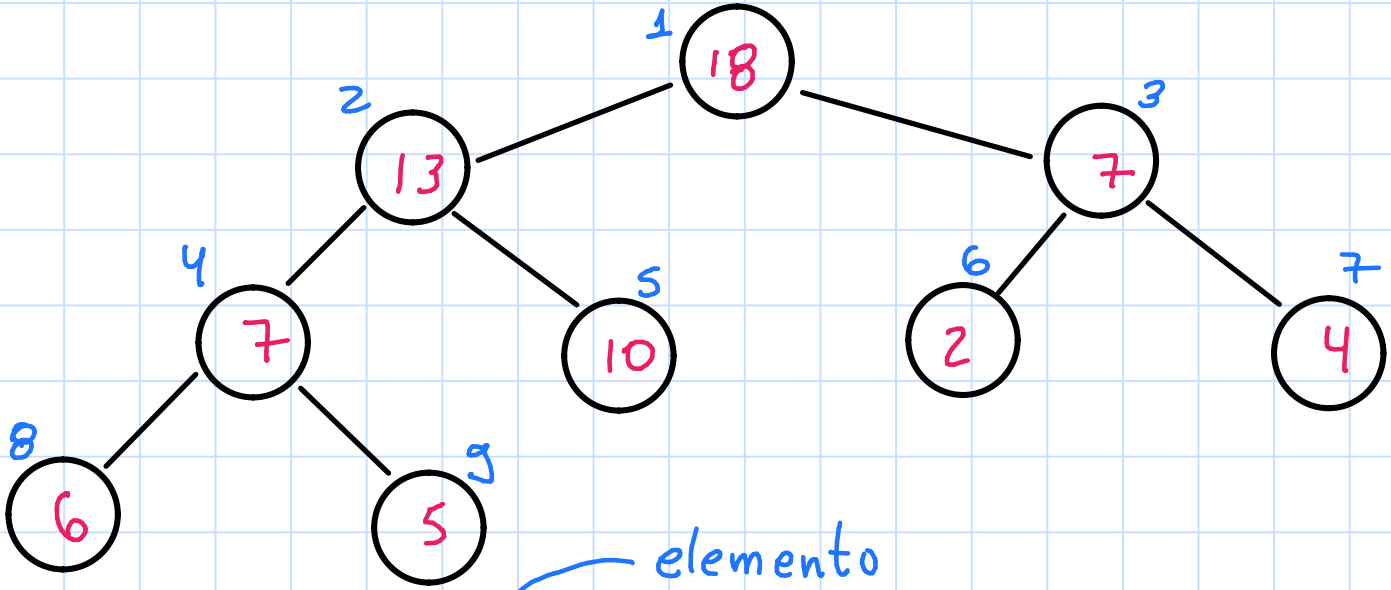
	1	2	3	4	5	6	7	8	9
A =	18	13	7	7	10	2	4	6	5



- Note que o elemento máximo está na raiz
- Cada subárvore tbm é um heap

Operações Heap: Mudança de Valor

	1	2	3	4	5	6	7	8	9
A =	18	13	7	7	10	2	4	6	5



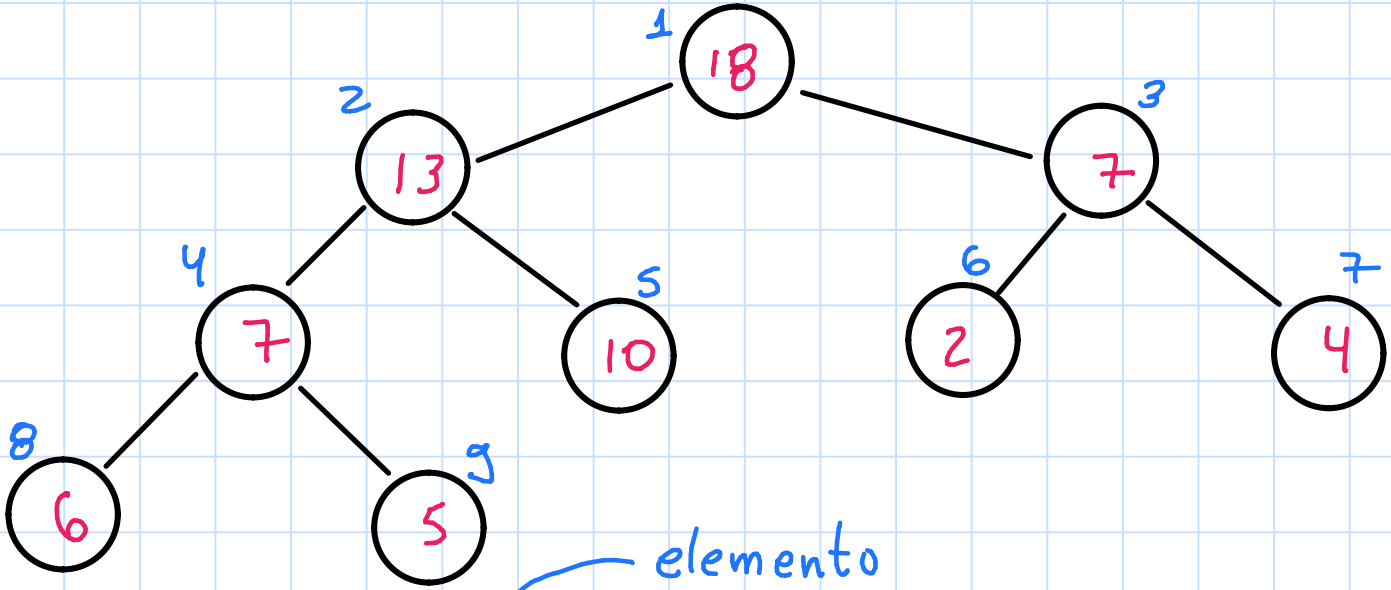
elemento

AlteraHeap(A, 1, 3)

novo valor

Operações Heap: Mudança de Valor

	1	2	3	4	5	6	7	8	9
A =	18	13	7	7	10	2	4	6	5



elemento

AlteraHeap(A, 9, 15)

novo valor

CorrigeHeapDescendo(A, i)

Assumindo que as subárvores do filho da esquerda e do filho da direita são heaps, CorrigeHeapDescendo transforma a subárvore enraizada em i em uma heap

```
Corrige Heap Descendo (A, n, i)

1    e ← Filho-Esquerda(i)
2    d ← Filho-Direita(i)
3    maior ← i
4    se  e ≤ n  e  A[e] > maior
5        maior ← e
6    se  d ≤ n  e  A[d] > maior
7        maior ← d
8    se  maior ≠ i
9        A[i] ↔ A[maior]
10       CorrigeHeapDescendo(A, n, maior)
```

Lema Dado um vetor A e um índice i tal que as subárvores enraizadas no filho da esquerda e direita são heaps, o algoritmo

CorrigeHeapDescendo(A, i)

transforma a subárvore enraizada em i em um heap.

Ideia da demonstração: indução no altura h do nó i

- Se $h=0$, então é folha e o algoritmo está correto
- se $h>0$, suponha que o algoritmo funciona para árvores com alturas menores que h. Antes da linha 8 $A[maior] \geq A[i], A[2i], A[2i+1]$. Após a linha 9, $A[i] \geq A[2i], A[2i+1]$. Pela H.I. CorrigeHeapDescendo transforma a subárvore com raiz <u>maior</u> em Heap

□

Complexidade de CorrigeHeapDescendo

CorrigeHeapDescendo (A, n, i)

```
1   e ← Filho-Esquerda(i)
2   d ← Filho-Direita(i)
3   maior ← i
4   se e ≤ n e A[e] > maior
5       maior ← e
6   se d ≤ n e A[d] > maior
7       maior ← d
8   se maior ≠ i
9       A[i] ↔ A[maior]
10      CorrigeHeapDescendo(A, n, maior)
```

Complexidade de CorrigeHeapDescendo

```
CorrigeHeapDescendo (A, n, i)
1    e ← Filho-Esquerda(i)
2    d ← Filho-Direita(i)
3    maior ← i
4    se  e ≤ n  e  A[e] > maior
5        maior ← e
6    se  d ≤ n  e  A[d] > maior
7        maior ← d
8    se  maior ≠ i
9        A[i] ↔ A[maior]
10       CorrigeHeapDescendo(A, n, maior)
```

$\Theta(1)$

$T(h-1)$

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

$= O(h)$

$= O(\lg n)$

CorrigeHeapSubindo(A, i)

```
1    Se  i ≥ 2  e  A[i] > A[pai(i)]
2        A[i] ↔ A[p]
3        CorrigeHeapSubindo(A, pai(i))
```

Complexidade de CorrigeHeapSubindo(A, i)

CorrigeHeapSubindo(A, i)

```
1   Se i ≥ 2 e A[i] > A[pai(i)]
2       A[i] ↔ A[p]
3       CorrigeHeapSubindo(A, pai(i))
```

Complexidade de CorrigeHeapSubindo(A, i)

CorrigeHeapSubindo(A, i)

```
1  Se i ⩾ 2  e  A[i] > A[pai(i)]        ⎫
2      A[i] ↔ A[p]                      ⎬ Θ(1)
3      CorrigeHeapSubindo(A, pai(i))    ⎭ T(h-1)
```

Seja h a altura do nó i

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

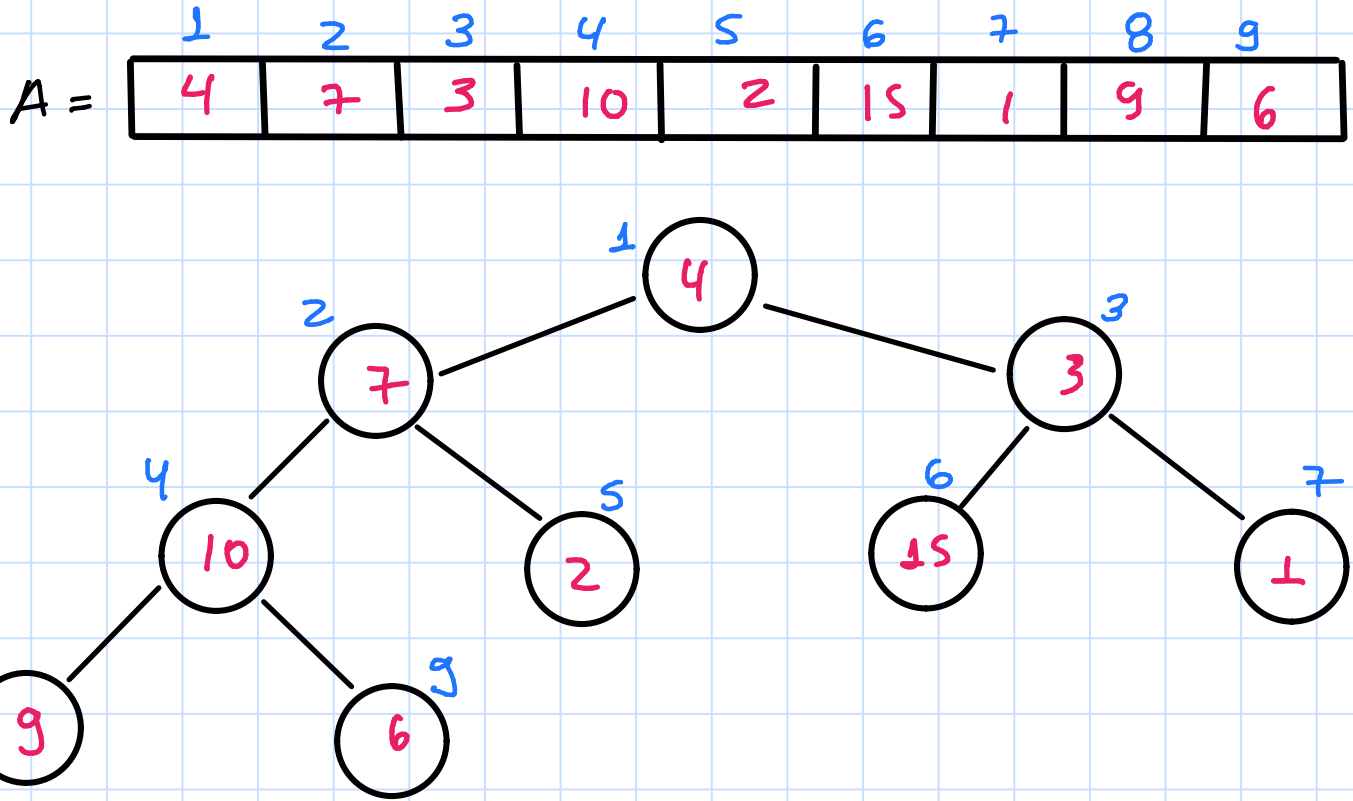
$$= O(\lg n)$$

```
AlteraHeap(A, n, i, k)
    t ← A[i]
    A[i] ← k
    Se  A[i] > t
        CorrigeHeapSubindo(A, i)
    Se  A[i] < t
        CorrigeHeapDescendo(A, n, i)
```

- Correção é uma consequência direta da Correção de CorrigeHeapSubindo e CorrigeHeapDescendo

- Tempo de Execução: $O(\lg n)$

Construção da Heap



elementos no vetor

ConstroiHeap (A, n)

para $i = \lfloor n/2 \rfloor$ decrescendo até 1

CorrigeHeapDescendo (A, n, i)

Porque começamos em $\lfloor n/2 \rfloor$?

$2i \leq n \iff i \leq \frac{n}{2}$

Filho esquerda existe

Como i é inteiro $i \leq \lfloor \frac{n}{2} \rfloor$

Complexidade de ConstroiHeap

```
ConstroiHeap(A, n)
1 para i = ⌊n/2⌋ decrescendo até 1        ] Θ(n)
2     CorrigeHeapDescendo(A, n, i)        ] Θ(n) · O(lg n)
```

Uma análise rápida nos mostra que

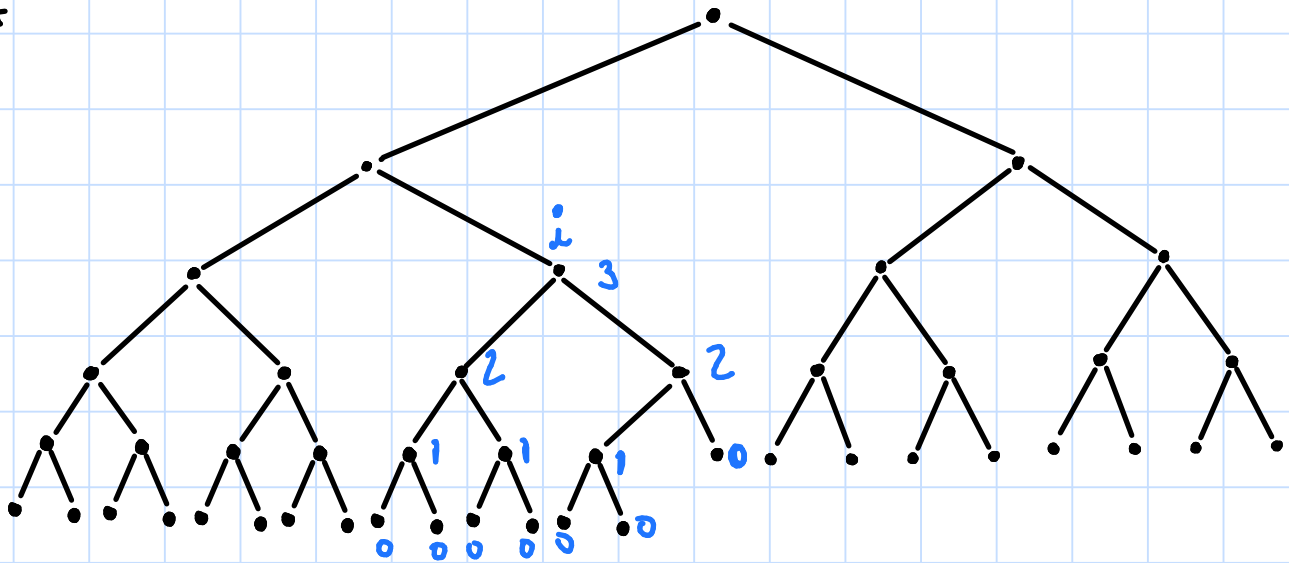
$$T(n) = O(n \lg n)$$

Uma análise mais minuciosa nos mostra que é

$$T(n) = \Theta(n)$$

⇒ claramente é $\Omega(n)$ (linha 1)

A



Pera um nó i de altura h, CorrigeHeapDescendo(A, i) leva tempo O(h)

- temos 1 nó de altura h
- no máximo 2 de altura h-1
- no máximo 4 de altura h-2

Lema Em uma heap com n elementos, existem no máximo

$$\lceil n/2^{h+1} \rceil$$

nós de altura h.

Demonstração

Se $h=0$, então existem $\lceil n/2 \rceil = \lceil n/2^{0+1} \rceil = \lceil n/2^{h+1} \rceil$ elementos que não são folhas

$2i = n \qquad i = \frac{n}{2} = \lfloor n/2 \rfloor \qquad n = \lfloor \frac{n}{2} \rfloor + \lceil \frac{n}{2} \rceil$

$2i+1 = n \qquad i = \frac{n-1}{2} = \lfloor \frac{n}{2} \rfloor$

Se $h > 0$, então por hipótese de indução existem no máximo

$$\lceil n/2^{h} \rceil$$

Se $h > 0$, então por hipótese de indução existem no máximo

$$\lceil n/2^h \rceil$$

nós de altura $h-1$. Note que o número de nós com altura h é a metade do número de nós com altura $h-1$. Portanto, o número máximo de nós com altura h é

$$\left\lceil \frac{\lceil n/2^h \rceil}{2} \right\rceil = \left\lceil \frac{n}{2^{h+1}} \right\rceil$$

□

Assim, temos que o tempo de execução de constrói Heap é

$$T(n) \leq \sum_{h=1}^{\lfloor \lg n \rfloor} \left\lceil \frac{n}{2^{h+1}} \right\rceil O(h)$$

o laço da linha 1 este iterando sobre os vértices que não são folha (altura entre 1 e $\lfloor \lg n \rfloor$)

$$\leq \sum_{h=1}^{\lfloor \lg n \rfloor} \left\lceil \frac{n}{2^{h+1}} \right\rceil C \cdot h = \sum_{h=1}^{\lfloor \lg n \rfloor} \left\lceil \frac{n/2^h}{2} \right\rceil C \cdot h$$

$$\leq \sum_{h=1}^{\lfloor \lg n \rfloor} n/2^h \cdot C \cdot h$$

$$= Cn \sum_{h=1}^{\lfloor \lg n \rfloor} h/2^h$$

$$\leq 4Cn = O(n) \leq 4$$

$\left\lceil \frac{n/a}{2} \right\rceil \leq \frac{n}{a}$

Caso n/a par

$\frac{\frac{n}{a}}{2} \leq \frac{n}{a}$

Caso n/a ímpar

$\frac{n/a + 1}{2} \leq n/a$

$n/a + 1 \leq 2\, n/a$

$1 \leq n/a$

$a \leq n$

Próx. Slide

Vamos provar que $\sum_{i=0}^{m} \frac{i}{2^i} \leq 4$, para qualquer $m \geq 0$

$$\Longrightarrow \sum_{h=1}^{\lfloor \lg n \rfloor} h/2^h \leq 4$$

Vamos provar que $\sum_{i=0}^{m} \frac{i}{2^i} \leq 4$, para qualquer $m \geq 0$

Seja $S = \sum_{i=0}^{m} \frac{i}{2^i} = \frac{1}{2} + \frac{2}{2^2} + \frac{3}{2^3} + \frac{4}{2^4} + \cdots + \frac{m}{2^m}$

Portanto $\frac{S}{2} = \frac{1}{2^2} + \frac{2}{2^3} + \frac{3}{2^4} + \cdots + \frac{m}{2^{m+1}}$

Note que

$$\frac{S}{2} = S - \frac{S}{2} = \left(\frac{1}{2} + \frac{2}{2^2} + \frac{3}{2^3} + \cdots + \frac{m}{2^m}\right) - \left(\frac{1}{2^2} + \frac{2}{2^3} + \cdots + \frac{m}{2^{m+1}}\right)$$

$$= \frac{1}{2} + \frac{1}{2^2} + \frac{1}{2^3} + \cdots + \frac{1}{2^m} - \frac{m}{2^{m+1}}$$

$$\frac{S}{2} = \frac{1}{2} + \frac{1}{2^2} + \frac{1}{2^3} + \cdots + \frac{1}{2^m} - \frac{m}{2^{m+1}}$$

$$S = 2\frac{S}{2} = 2\left(\frac{1}{2} + \frac{1}{2^2} + \frac{1}{2^3} + \cdots + \frac{1}{2^m}\right) - \frac{2m}{2^{m+1}}$$

$$\leq 2\left(\frac{1}{2} + \frac{1}{2^2} + \frac{1}{2^3} + \cdots + \frac{1}{2^m}\right)$$

$$= 2\sum_{i=1}^{m}\left(\frac{1}{2}\right)^i = 2\left[\frac{\left(\frac{1}{2}\right)^{m+1} - 1}{\frac{1}{2} - 1}\right]$$

$$= 2\left[\frac{\left(\frac{1}{2}\right)^{m+1} - 1}{-\frac{1}{2}}\right] = -4\left[\left(\frac{1}{2}\right)^{m+1} - 1\right]$$

$$= 4 - 4\left(\frac{1}{2}\right)^{m+1} \leq 4$$

Correção

```
ConstroiHeap(A, n)
1 para  i = ⌊n/2⌋  decrescendo até 1
2       CorrigeHeapDescendo(A, n, i)
```

Tco ConstroiHeap(A,n) transforma um vetor de números A em um Heap.

P(t) = "Antes da t-ésima iteração começar, vale que

- $i = \lfloor n/2 \rfloor + 1 - t$, e
- a árvore enraizada em A[j], para $i+1 \leq j \leq n$ é uma heap "

Heap - Sort

```
Selection-Sort (A, n)
    Para  i ← n  decrescendo até 2
          max ← Maximum (A, i)
          A[i] ↔ A[max]

Maximum (A, i)
    max ←   i
    para  j ← i-1  decrescendo até 1
          se   A[j] > A[max]
               max ←   j
    Devolva  max
```

Heap

Selection-Sort (A, n)

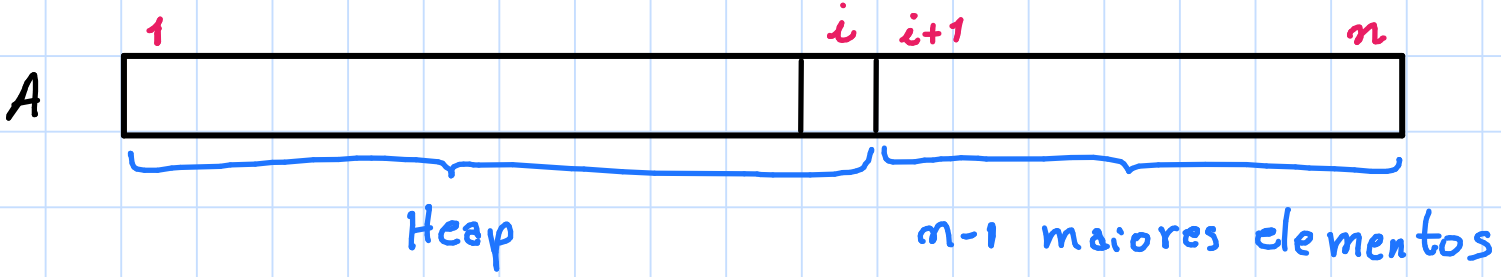
ConstroiHeap(A, n)

Para $i \leftarrow n$ decrescendo até 2

~~max~~ $\leftarrow$ Maximum(A, i)

$A[1] \leftrightarrow A[\text{max}]$ (i)

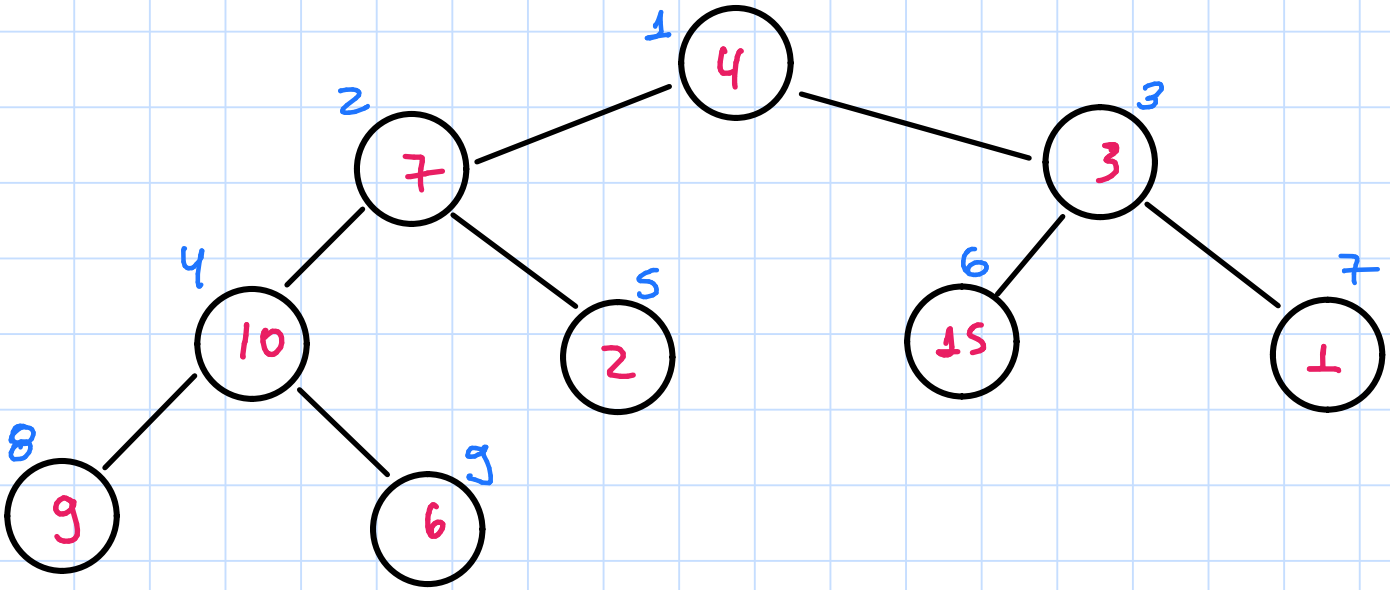
CorrigeHeapDescendo$(A, i-1, 1)$



Exemplo

$A =$

1	2	3	4	5	6	7	8	9
4	7	3	10	2	15	1	9	6



```
HeapSort (A, n)

1  ConstroiHeap (A, n)
2  Para  i ← n  decrescendo até 2
3        A[1] ↔ A[i]
4        CorrigeHeapDescendo (A, i-1, 1)
```

Tempo de Execução

HeapSort (A, n)

```
1  ConstroiHeap (A, n)
2  Para  i <- n   decrescendo até 2
3        A[1] <-> A[i]
4        CorrigeHeapDescendo (A, i-1, 1)
```

Tempo de Execução

HeapSort (A, n)

1 ConstroiHeap (A, n) $O(n)$
2 Para i ← n decrescendo até 2 $O(n)$
3 A[1] ↔ A[i] $O(n)$
4 CorrigeHeapDescendo (A, i-1, 1) $O(n) \cdot O(\lg n)$

$T(n) = O(n \lg n)$

Correção

```
HeapSort (A, n)
1  ConstroiHeap (A, n)
2  Para  i ← n  decrescendo até 2
3        A[1] ↔ A[i]
4        CorrigeHeapDescendo (A, i-1, 1)
```

Teo O algoritmo HeapSort(A, n) ordena um vetor A de n elementos

$P(t)$ = " Antes da t-ésima iteração começar,

- $i = n + 1 - t$
- $A[1..\underbrace{n+1-t}_{i}]$ é um heap
- $A[\underbrace{n+2-t}_{i+1}..n]$ está ordenado
- $A[1..\underbrace{n+1-t}_{i}] \leq \min(A[n+2-t..n])$
- Contém uma permutação dos elementos iniciais "

Base ($t = 1$)

- $i = n + 1 - t = n$
- $A[1..\underbrace{n+1-t}_{i}] = A[1..n]$ é um heap
- $A[\underbrace{n+2-t}_{i+1}..n] = A[n+1..n] = \emptyset$ está ordenado
- $A[1..\underbrace{n+1-t}_{i}] = A[1..n] \leq \min(A[i+1..n] = A[n+1..n])$

Passo: P(t) Vale, iteração t ocorre, Vemos provar P(t+1)

P(t) = Antes da t-ésima iteração começar,

① $i = n + 1 - t$

② $A[1..\underbrace{n+1-t}_{i}]$ é um heap

③ $A[\underbrace{n+2-t}_{i+1}..n]$ está ordenado

④ $A[1..\underbrace{n+1-t}_{i}] \leq \min(A[\underbrace{n+2-t}_{i+1}..n])$

Seja $x = A[1]$. Por ②, $x \geq A[1..n+1-t]$ e, por ④, $x \leq \min(A[n+2-t..n])$. A linha 3 executa e coloca x em $A[i] = A[n+1-t]$. Portanto $A[n+1-t..n]$ está ordenado e $A[1..n-t] \leq \min(A[n+1-t..n])$ (por 4 e 2). Após a linha 4 temos que $A[1..i-1] = A[1..n-t]$ é uma heap. O laço finaliza e i é decrementado, passando a valer $i = n - t$.

□

Teo O algoritmo HeapSort(A, n) ordena um vetor A de n elementos

Demonstração

- Pela correção de ConstroiHeap, após a exe. da linha 1, A é uma Heap.
- Pela invariante P(t), o laço termina após n iterações
- Por P(n), temos que

$$A[2..n] \text{ está ordenado e}$$
$$A[1..1] = A[1] \leq \min(A[2..n])$$

- Portanto A está ordenado □

Ordenação

	Tempo de Execução
InsertionSort	$O(n^2)$
MergeSort	$O(n \lg n)$
SelectionSort	$O(n^2)$
HeapSort	$O(n \lg n)$
QuickSort	$O(n^2)$

Qualquer algoritmo baseado em comparação requer $\Omega(n \lg n)$ no pior caso.

Exemplo de algoritmo que não é baseado em comp.

```
Counting-Sort (A, n)        ▷ Sabemos que 1 ≤ A[i] ≤ C ∀ i
    Seja B[1..C] um vetor
    Para i = 1..C
        B[i] = 0
    Para i = 1 até n
        B[A[i]] ← B[A[i]] + 1
    k ← 0
    Para j ← 1 até C
        para i ← 1 até B[j]
            A[k+i] ← j
        k ← k + B[j]
```

Exemplo de algoritmo que não é baseado em comp.

```
Counting-Sort (A, n)        ▷ Sabemos que 1 ≤ A[i] ≤ C  ∀ i
    Seja B[1..C] um vetor
    Para i = 1..C                          ⎫
        B[i] = 0                           ⎭ Θ(c)
    Para i = 1 até n                       ⎫
        B[A[i]] ← B[A[i]] + 1               ⎭ Θ(n)
    k ← 0                                  ⎫
    Para j ← 1 até C                       ⎭ Θ(c)
        para i ← 1 até B[j]                ⎫
            A[k+i] ← j                     ⎬ Θ(n)
        k ← k + B[j]                       ⎭
```

$T(n) = \Theta(n)$

Fila de Prioridades

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

- `Maximum(S)` devolve o elemento de maior prioridade
- `Extract-Max(S)` remove o elemento de maior prioridade
- `Increase-key(S, x, p)` altera a prioridade de x para p
- `Insert(S, x, p)` insere um elemento x com prioridade p

Fila de Prioridade Pode ser implementada como

- Vetor
- Heap
- outros

```
InsereHeap(A, n, p)
    A[n+1] ← p
    CorrigeHeapSubindo(A, n+1)
    Devolve A, n+1

RemoveHeap(A, n, i)
    A[i] ← A[n]
    CorrigeHeapDescendo(A, n-1, i)
    Devolve A, n-1
```