ACH2002

Aula 15

Heapsort

(adaptados dos slides de aula da Profa. Fátima L. S. Nunes)

EACH USP

Aulas passadas

- Algoritmos de ordenação elementares
  - InsertionSort
  - SelectionSort
  - BubbleSort
  - ShellSort
- Algoritmos de ordenação eficientes
  - MergeSort

Aula de hoje

- Algoritmos de ordenação elementares
  - InsertionSort
  - SelectionSort
  - BubbleSort
  - ShellSort
- Algoritmos de ordenação eficientes
  - MergeSort
  - **HeapSort**

HeapSort

- Algoritmo tem este nome porque utiliza uma estrutura de dados chamada ***heap*** para auxiliar na ordenação.
- Utiliza o mesmo princípio da ordenação por seleção:
  - encontrar o menor item do arranjo e trocar com o elemento que está na primeira posição;
  - em seguida, encontrar o segundo menor e trocar com o elemento da segunda posição;
  - e assim por diante...

HeapSort

- Quantas comparações são necessárias para encontrar o menor item em um arranjo de ***n*** elementos?

HeapSort

- Quantas comparações são necessárias para encontrar o menor item em um arranjo de ***n*** elementos?
  - n-1 comparações!
- Será que este custo pode ser reduzido?

HeapSort

- Quantas comparações são necessárias para encontrar o menor item em um arranjo de ***n*** elementos?
  - n-1 comparações!
- Será que este custo pode ser reduzido?
  - Sim ⇒ estabelecendo-se uma fila de prioridades ⇒ estrutura denominada ***heap***

Fila de prioridades

- Na prática, um array que estabelece uma determinada ordem de execução/busca...
- usadas em diversas aplicações de Computação;
- operações mais comuns:
  - adicionar um novo item;
  - encontrar o item com menor (ou maior) valor;
  - retirar o item com menor (ou maior) valor;
  - alterar a prioridade de um item;
  - remover um item qualquer;
  - etc

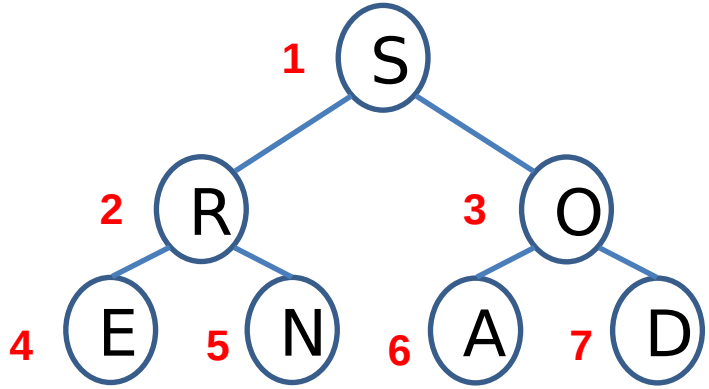
EACH USP

Heap

- Estrutura de dados usada para implementar fila de prioridades.
- Proposto por J. W. J Williams, em 1964.
- Definição:
  - um ***heap*** é uma estrutura de dados contendo uma sequência de itens com chaves: *c[1],c[2],..., c[n]* tal que *c[i]* ≥ *c[2i]* e *c[i]* ≥ *c[2i+1]* , para todo *i=1, 2, ..., n/2*.

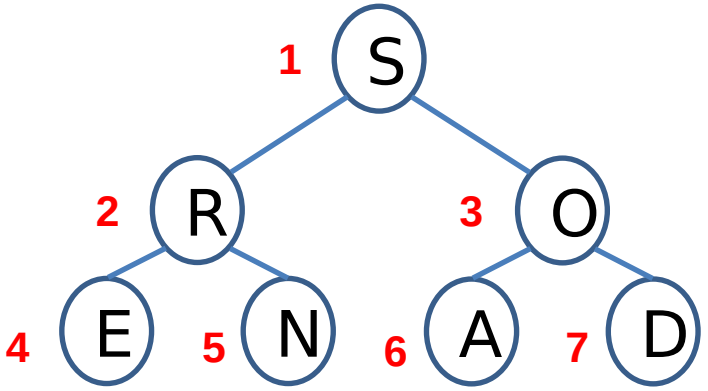
Heap

- Definição:
  - um ***heap*** é uma estrutura de dados contendo uma sequência de itens com chaves: *c[1],c[2],..., c[n]* tal que *c[i] ≥ c[2i]* e *c[i] ≥ c[2i+1]* , para todo *i=1, 2, ..., n/2*.
  - sequência é facilmente visualizada se for desenhada como uma **árvore binária completa**: as linhas que saem de uma chave levam a duas chaves menores de nível inferior.



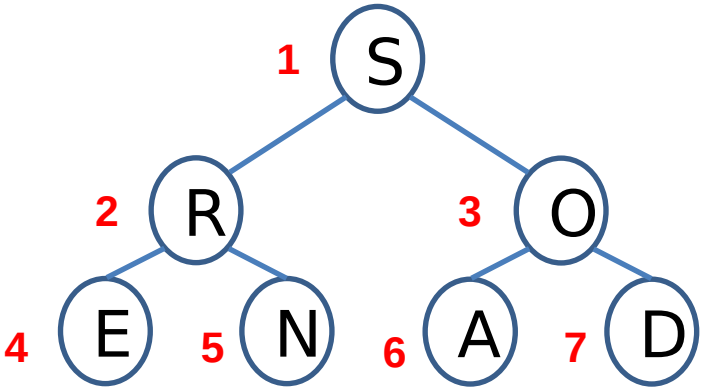
Heap

- Definição:
  - **árvore binária completa**: árvore binária com os nós numerados de ***1*** a ***n***.
  - primeiro nó é chamado raiz;
  - nó $\lfloor k/2 \rfloor$ é o pai do nó k, para $1 < k \leq n$;
  - nós $2k$ e $2k+1$ são filhos à esquerda e à direita do nó k, para $1 \leq k \leq \lfloor n/2 \rfloor$.



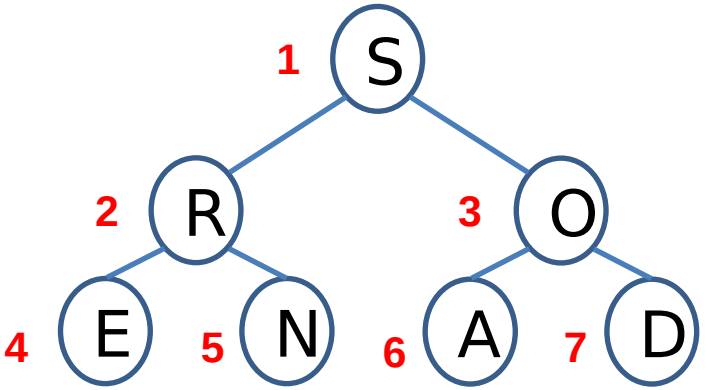
Heap

- Uma **árvore binária completa** e, consequentemente, um ***heap*,** pode ser representado por um *array*.
- Como seria um *array* para representar esta árvore?



Heap

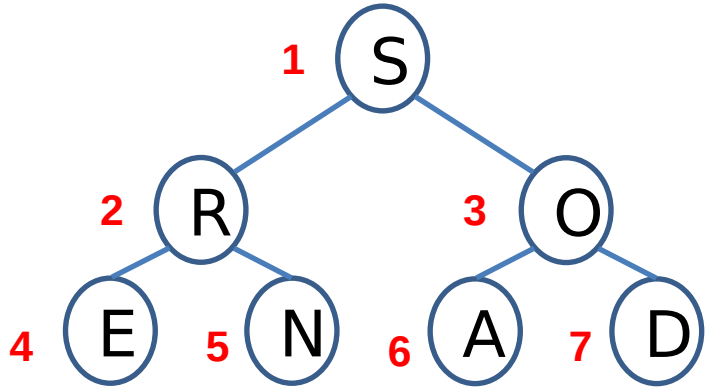
- Uma **árvore binária completa** e, consequentemente, um ***heap*,** pode ser representado por um *array*.
- Como seria um *array* para representar esta árvore?



1	2	3	4	5	6	7
S	R	O	E	N	A	D

Heap

- Como seria um *array* para representar esta árvore?

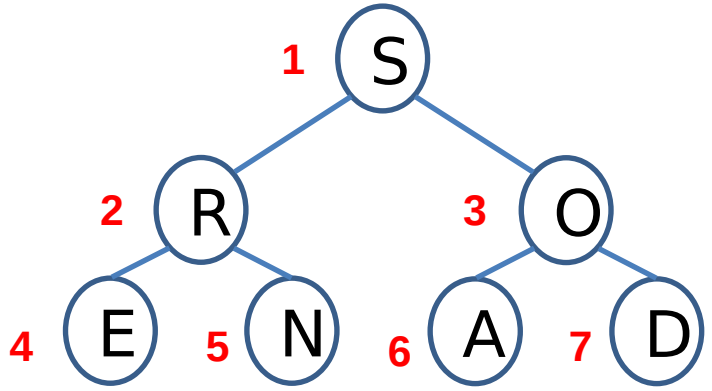


1	2	3	4	5	6	7
S	R	O	E	N	A	D

- Quais são os filhos do nó ***i,*** *se existirem*?
- Qual é o pai de um nó ***i,*** *se existir?*
- Em que posição está o maior elemento?

Heap

- Como seria um *array* para representar esta árvore?

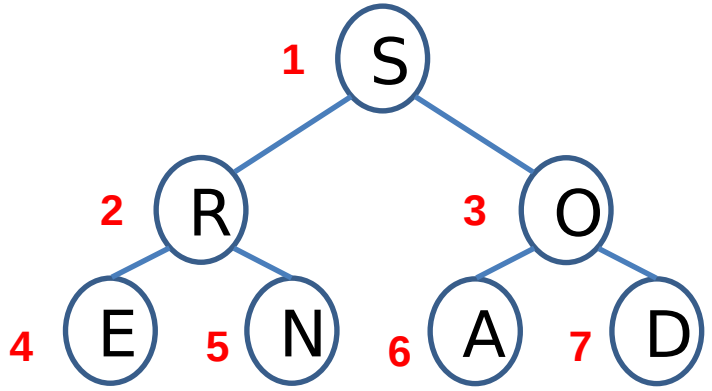


1	2	3	4	5	6	7
S	R	O	E	N	A	D

- Quais são os filhos do nó ***i,*** *se existirem*? ***2i*** e ***2i+1***
- Qual é o pai de um nó ***i,*** *se existir?*
- Em que posição está o maior elemento, neste caso?

Heap

- Como seria um *array* para representar esta árvore?

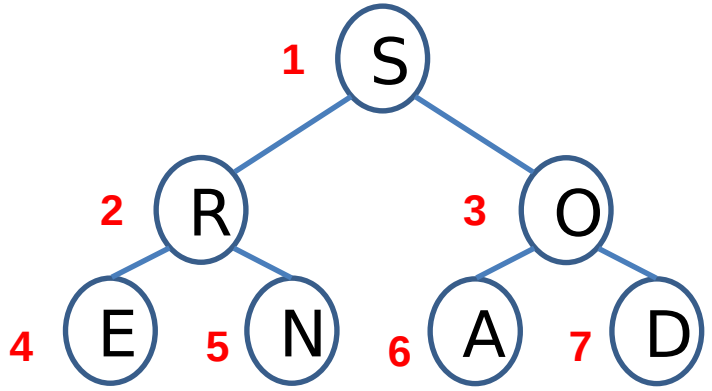


1	2	3	4	5	6	7
S	R	O	E	N	A	D

- Quais são os filhos do nó ***i,*** *se existirem*? ***2i*** e ***2i+1***
- Qual é o pai de um nó ***i,*** *se existir*? ***i*** ***div*** ***2***
- Em que posição está o maior elemento, neste caso?

Heap

- Como seria um *array* para representar esta árvore?



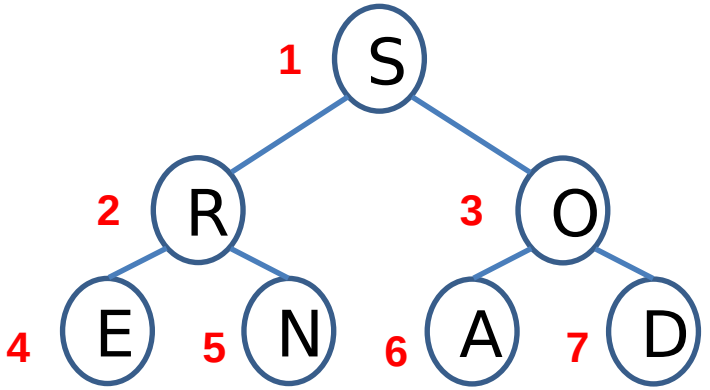
1	2	3	4	5	6	7
S	R	O	E	N	A	D

- Quais são os filhos do nó ***i,*** *se existirem*? ***2i*** e ***2i+1***
- Qual é o pai de um nó ***i,*** *se existir*? ***i*** ***div*** ***2***
- Em que posição está o maior elemento, neste caso? ***1***

Heap

- Algoritmos:

1	2	3	4	5	6	7
S	R	O	E	N	A	D

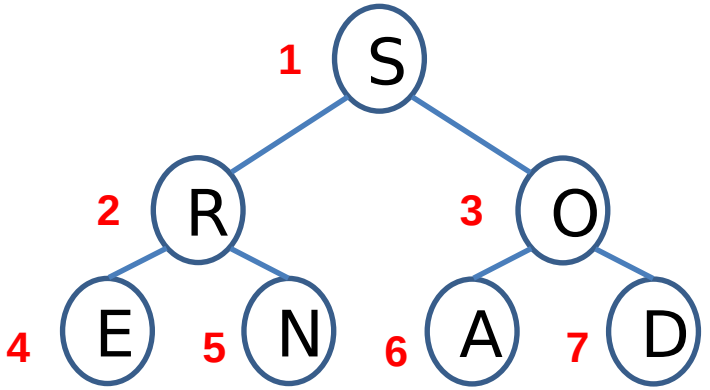



```
pai(i)
   retorna ???

filho_esquerdo(i)
   retorna ???

filho_direito(i)
   retorna ???
```

Heap

- Algoritmos:

1	2	3	4	5	6	7
S	R	O	E	N	A	D

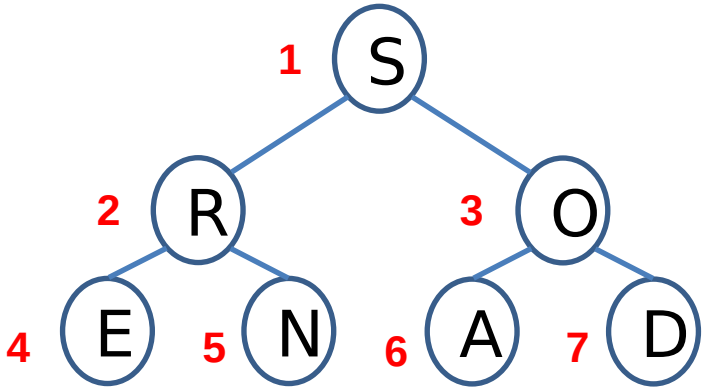



```
pai(i)
   retorna ⌊i/2⌋

filho_esquerdo(i)
   retorna ???

filho_direito(i)
   retorna ???
```

Heap

- Algoritmos:

1	2	3	4	5	6	7
S	R	O	E	N	A	D

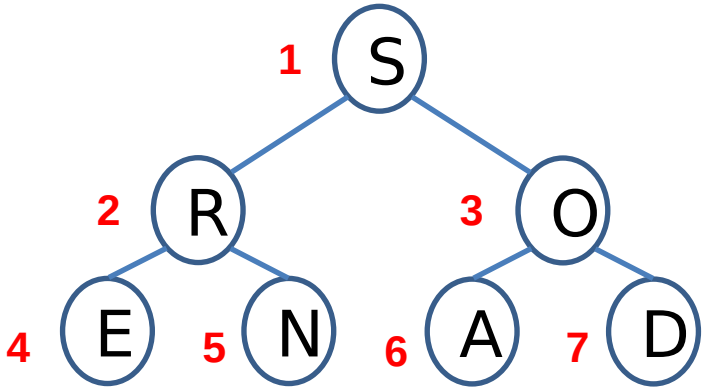



```
pai(i)
   retorna ⌊i/2⌋

filho_esquerdo(i)
   retorna 2i

filho_direito(i)
   retorna ???
```

Heap

- Algoritmos:

1	2	3	4	5	6	7
S	R	O	E	N	A	D

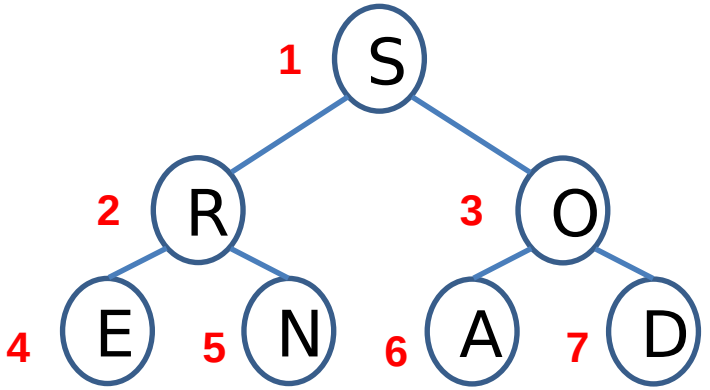



```
pai(i)
   retorna ⌊i/2⌋

filho_esquerdo(i)
   retorna 2i

filho_direito(i)
   retorna 2i+1
```

Heap

- Algoritmos:

1	2	3	4	5	6	7
S	R	O	E	N	A	D



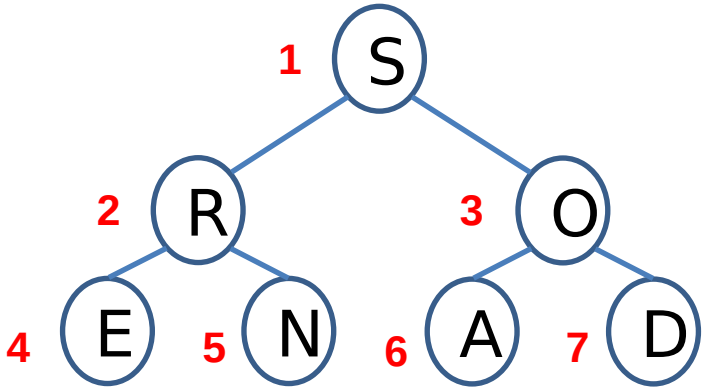

```
pai(i)
   retorna ⌊i/2⌋

filho_esquerdo(i)
   retorna 2i

filho_direito(i)
   retorna 2i+1
```

Complexidades dessas operações:

Heap

- Algoritmos:

1	2	3	4	5	6	7
S	R	O	E	N	A	D

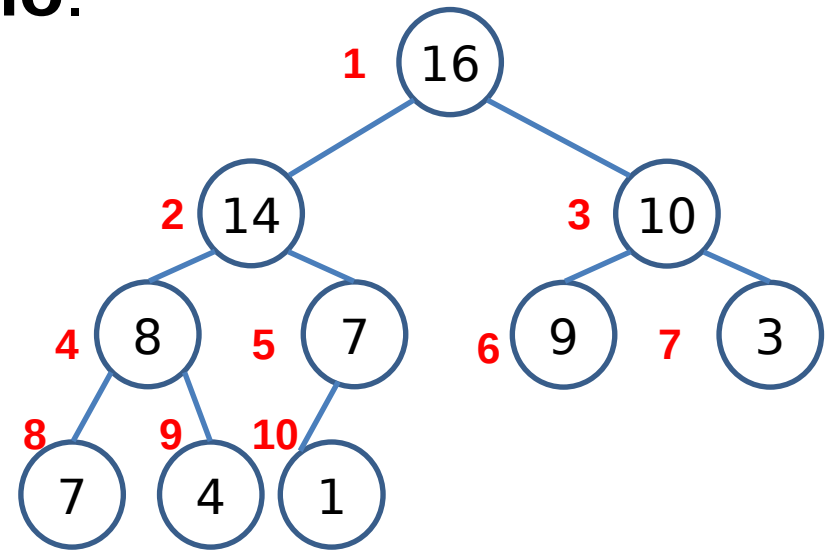



```
pai(i)
   retorna ⌊i/2⌋

filho_esquerdo(i)
   retorna 2i

filho_direito(i)
   retorna 2i+1
```

Complexidades dessas operações:
O(1) !!!

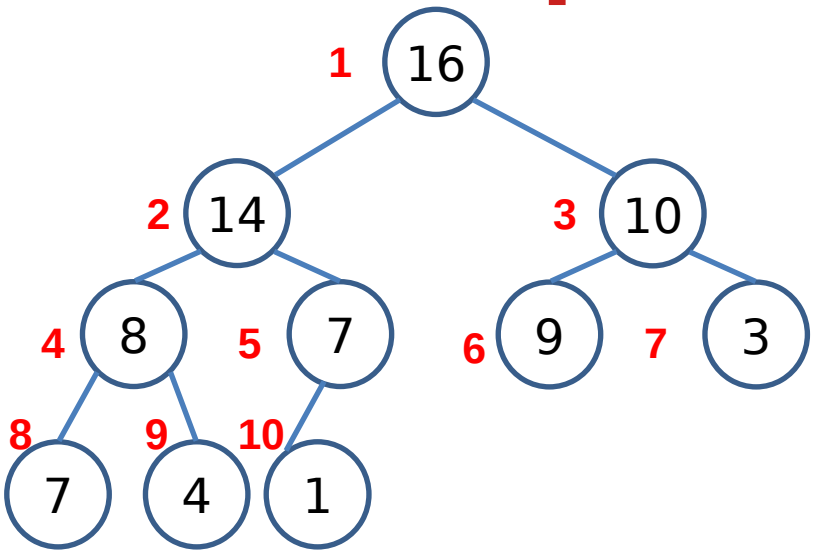
Heap

- Dado um arranjo A que representa um ***heap***, definimos dois tipos de ***heap***:
  - ***heap mínimo***: *A[pai(i)] ≤ A[i]*
  - ***heap máximo***: *A[pai(i)] ≥ A[i]*
- ***HeapSort*** usa ***heap* máximo**.
- Um exemplo com números:



1	2	3	4	5	6	7	8	9	10
16	**14**	**10**	**8**	**7**	**9**	**3**	**7**	**4**	**1**

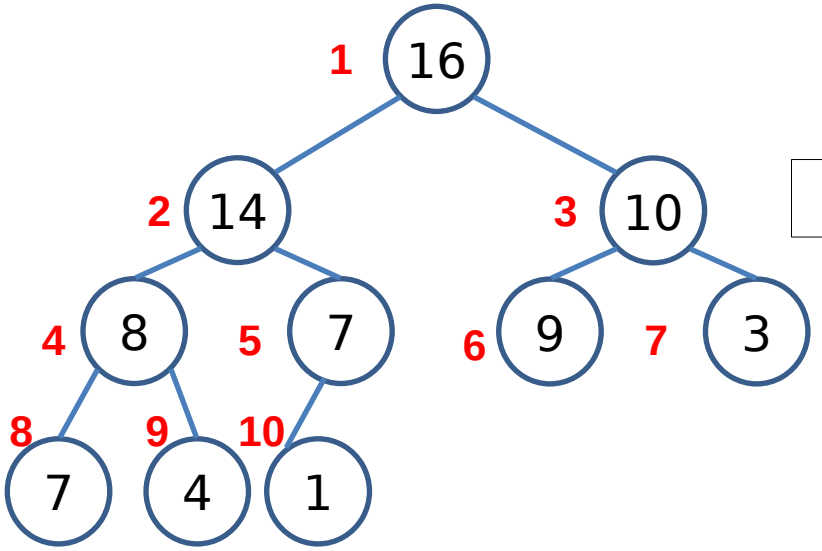
Heap



1	2	3	4	5	6	7	8	9	10
16	14	10	8	7	9	3	7	4	1

- **Altura de um nó** em um *heap*: número de arestas no caminho descendente simples mais longo desde o nó até um nó folha (último nível da árvore)
- **Altura de *heap***: altura de sua raiz.

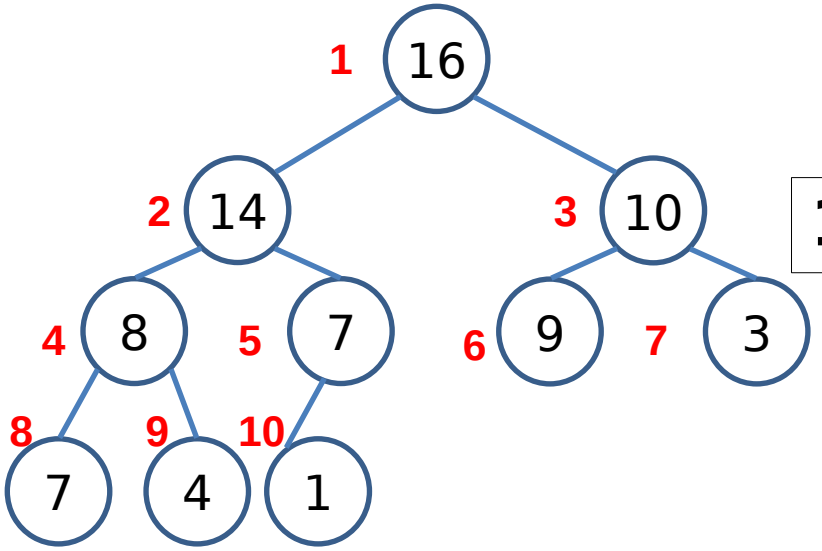
Heap



1	2	3	4	5	6	7	8	9	10
16	14	10	8	7	9	3	7	4	1

- **Exemplos**
  - **Altura do nó 2:**
  - **Altura do nó 9:**
  - **Altura de *heap*:**

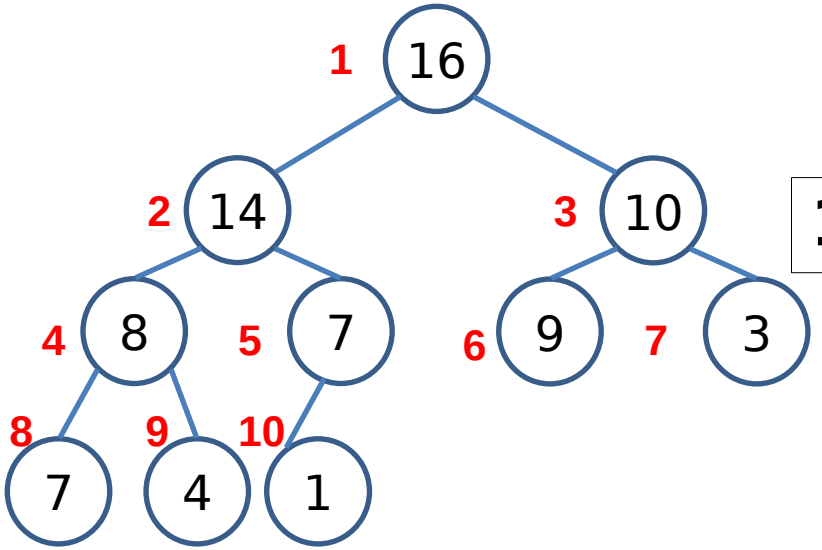
Heap



1	2	3	4	5	6	7	8	9	10
16	14	10	8	7	9	3	7	4	1

- **Exemplos**
  - **Altura do nó 2: 2**
  - **Altura do nó 9:**
  - **Altura de *heap*:**

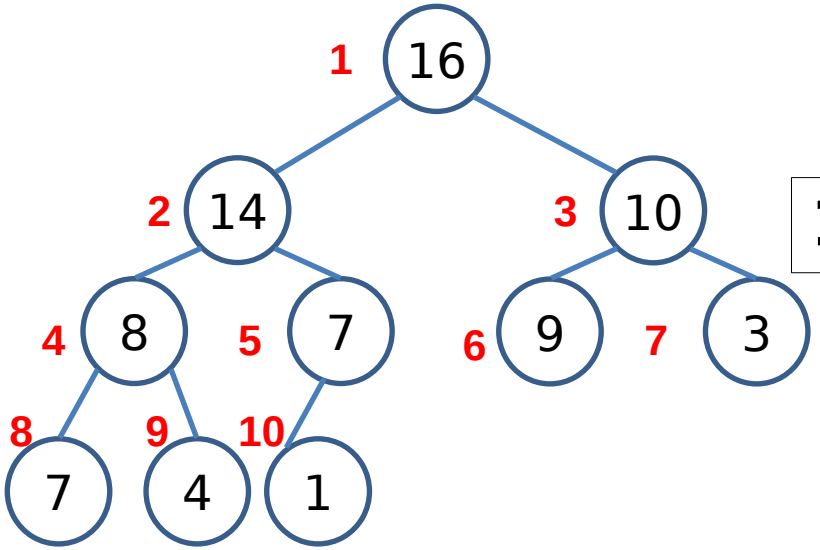
Heap



1	2	3	4	5	6	7	8	9	10
16	14	10	8	7	9	3	7	4	1

- **Exemplos**
  - **Altura do nó 2: 2**
  - **Altura do nó 9: 0**
  - **Altura de *heap*:**

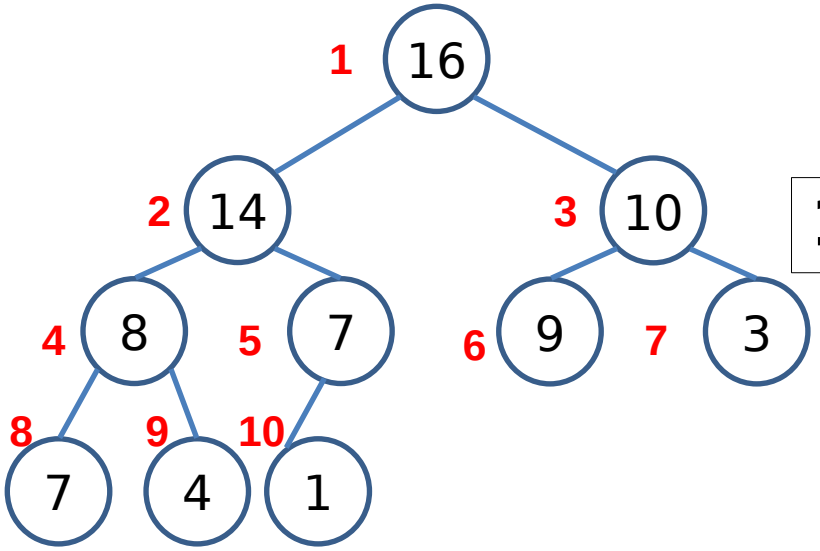
Heap



1	2	3	4	5	6	7	8	9	10
16	14	10	8	7	9	3	7	4	1

- **Exemplos**
  - **Altura do nó 2: 2**
  - **Altura do nó 9: 0**
  - **Altura de *heap*: 3**

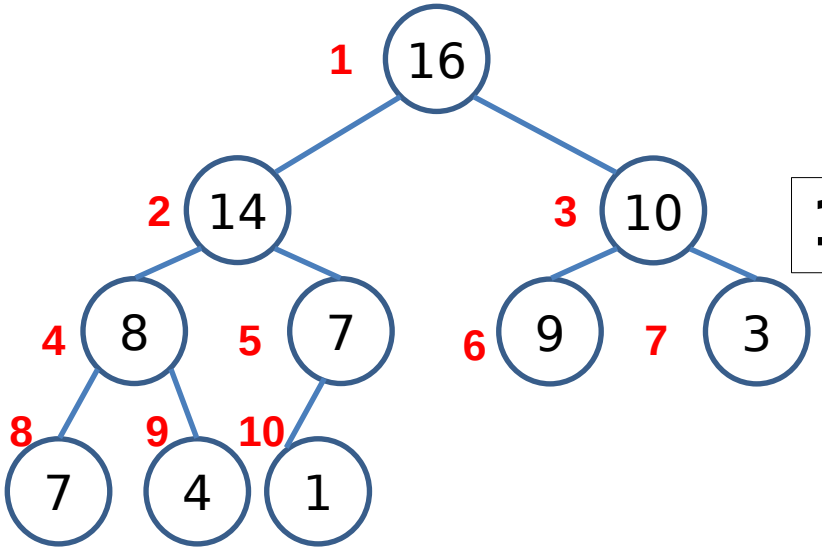
Heap



1	2	3	4	5	6	7	8	9	10
16	**14**	**10**	**8**	**7**	**9**	**3**	**7**	**4**	**1**

- Qual é a complexidade da altura de um *heap* de *n* elementos?
  - árvore binária completa;
  - cada nível é dividido em 2;
  - complexidade da altura:

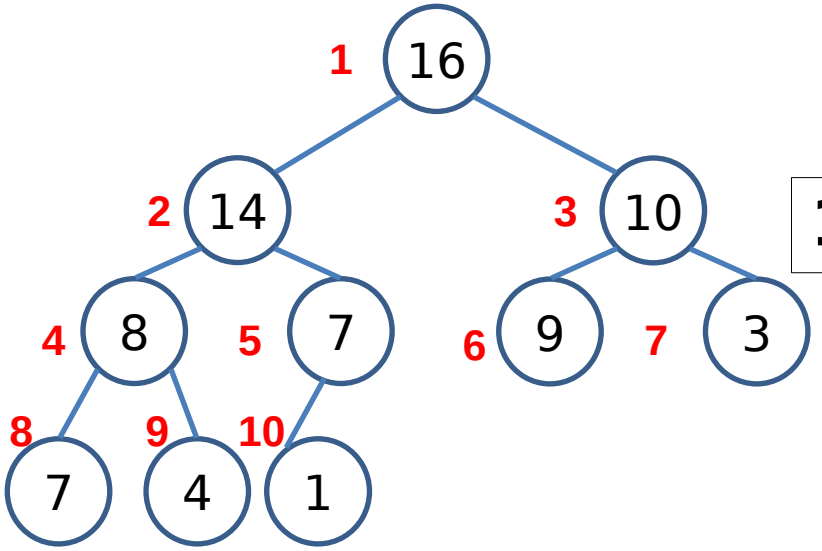
Heap



1	2	3	4	5	6	7	8	9	10
16	**14**	**10**	**8**	**7**	**9**	**3**	**7**	**4**	**1**

- Qual é a complexidade da altura de um *heap* de *n* elementos?
  - árvore binária completa;
  - cada nível é dividido em 2;
  - complexidade da altura: ***Θ(lg n)***

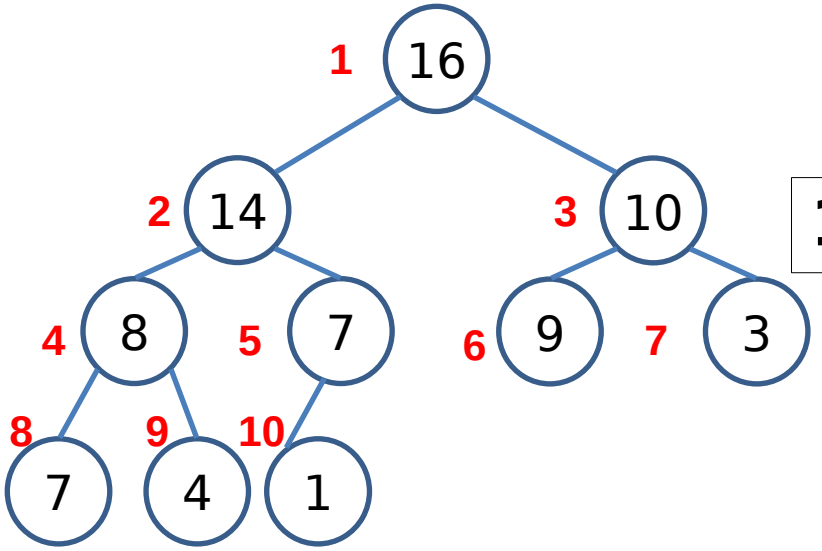
Heap



1	2	3	4	5	6	7	8	9	10
16	14	10	8	7	9	3	7	4	1

- Operações básicas sobre estrutura de *heaps:*
  - ***refaz heap máximo***
  - ***construir heap máximo***

Heap

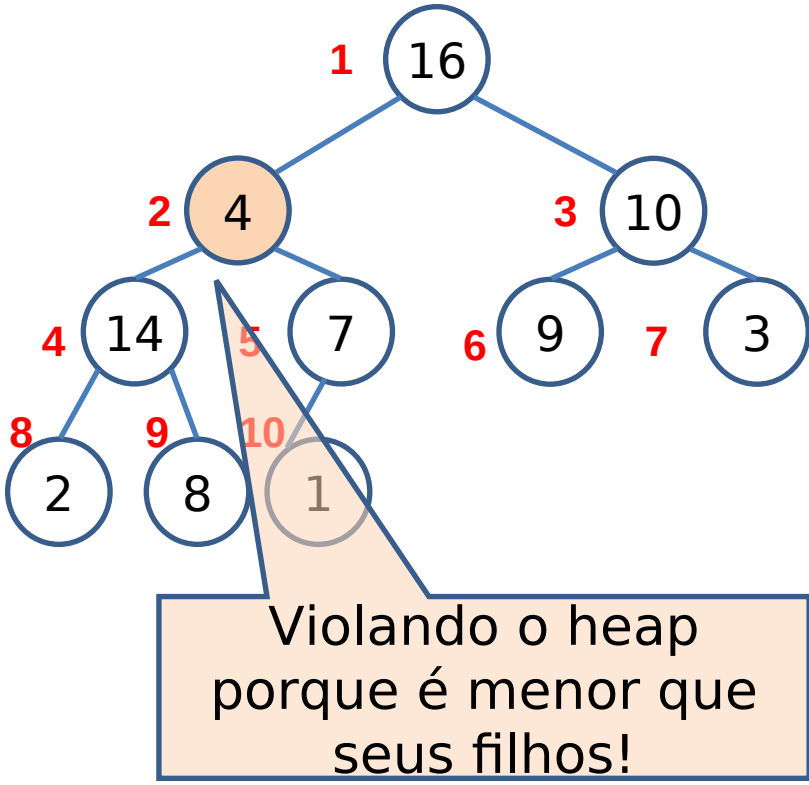


1	2	3	4	5	6	7	8	9	10
16	14	10	8	7	9	3	7	4	1

- Operações básicas sobre estrutura de *heaps:*
  - ***refaz heap máximo:*** *(útil para 1) construir o heap, e 2) depois de remover a raiz, colocando o último elemento do vetor em seu lugar)*
  - ***construir heap máximo***

Refaz Heap Máximo

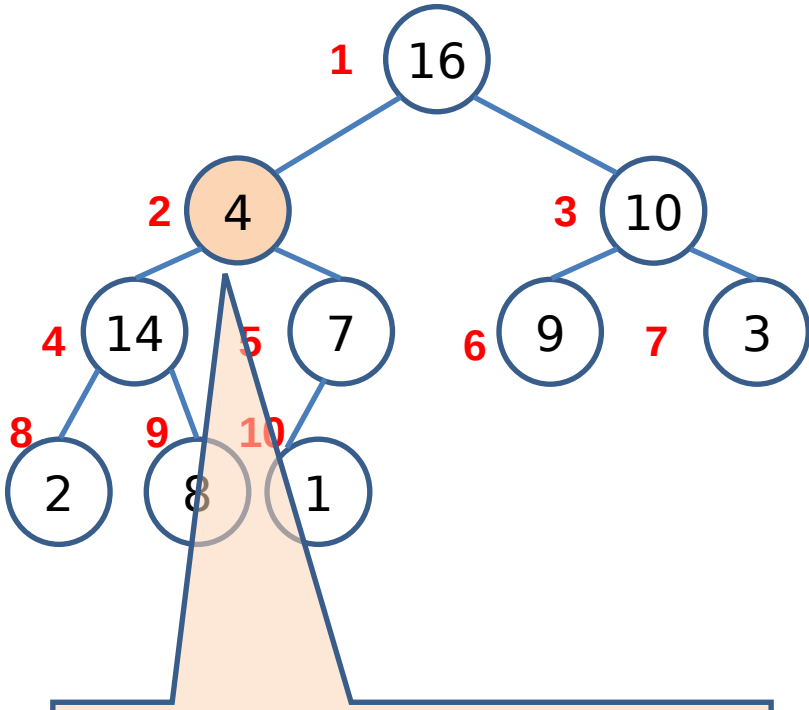
- Algoritmo:



COMO RESOLVER?

Refaz Heap Máximo

- Algoritmo:



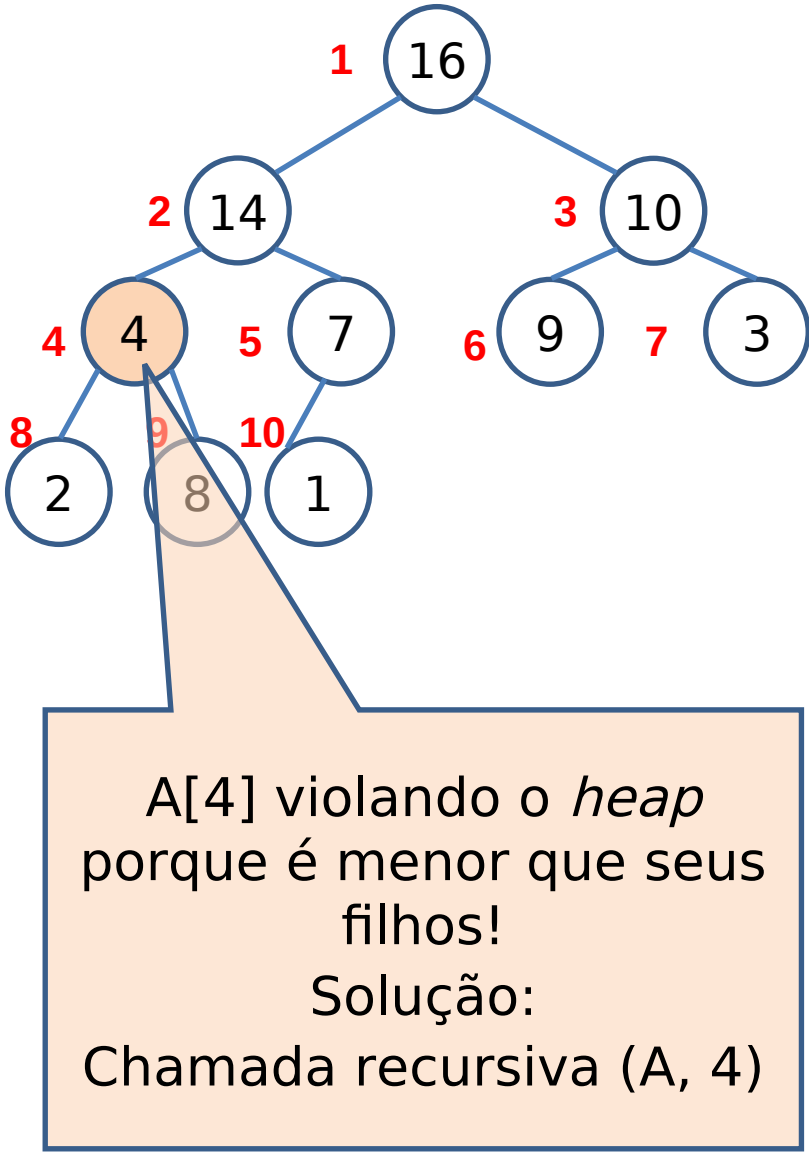
A[2] violando o *heap* porque é menor que seus filhos!

Solução:

Troca A[2] com o maior dos seus dois filhos!

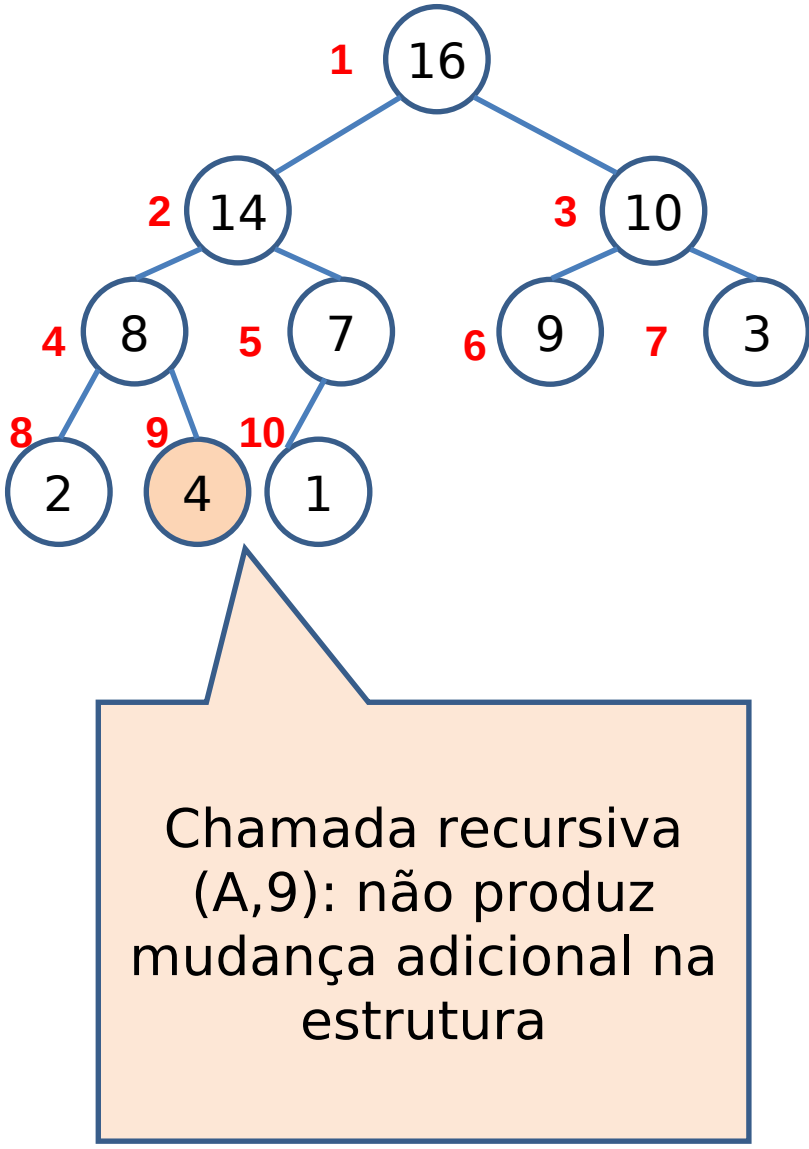
Refaz Heap Máximo

- Algoritmo:



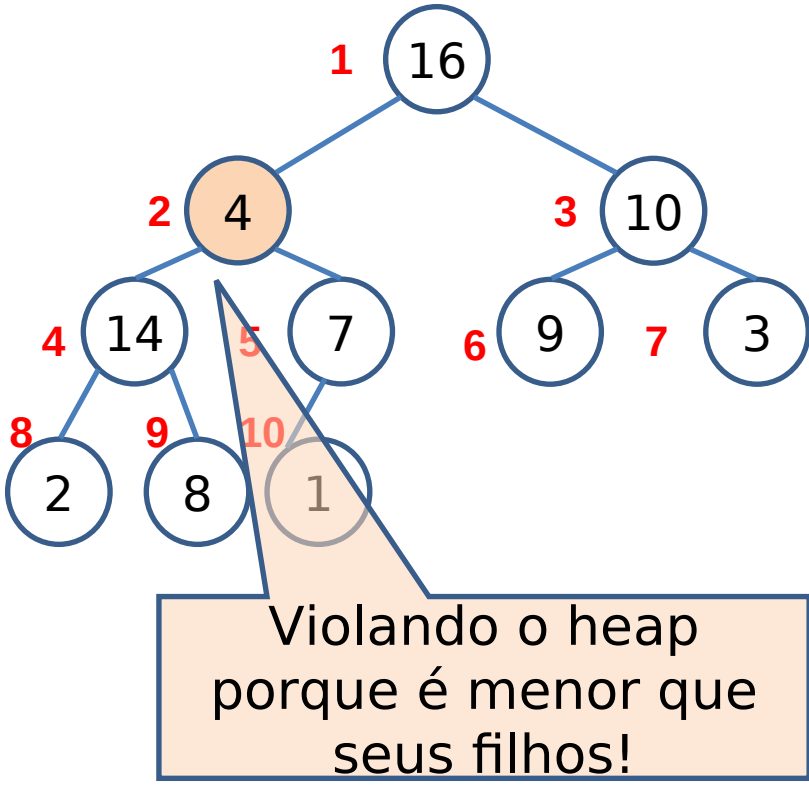
Refaz Heap Máximo

- Algoritmo:



Refaz Heap Máximo

- Algoritmo:

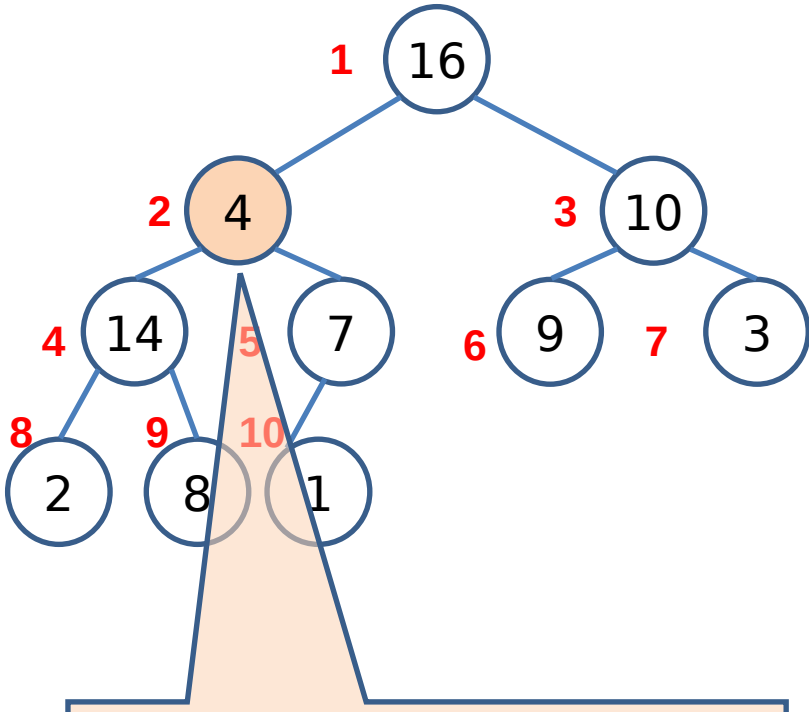


COMO RESOLVER?

Refaz Heap Máximo

- Algoritmo:

```
refazHeapMaximo(A[],i)
l ← filho_esquerdo(i)
r ← filho_direito(i)
se l ≤ tamanho-do-heap[A] e A[l] > A [i]
       maior ← l
senão
       maior ← i
fim se
se r ≤ tamanho-do-heap[A] e A[r] > A[maior]
      maior ← r
fim se
se maior ≠ i
       trocar A[i] ↔ A[maior]
       refazHeapMaximo(A,maior)
fim se
```



A[2] violando o *heap* porque é menor que seus filhos!

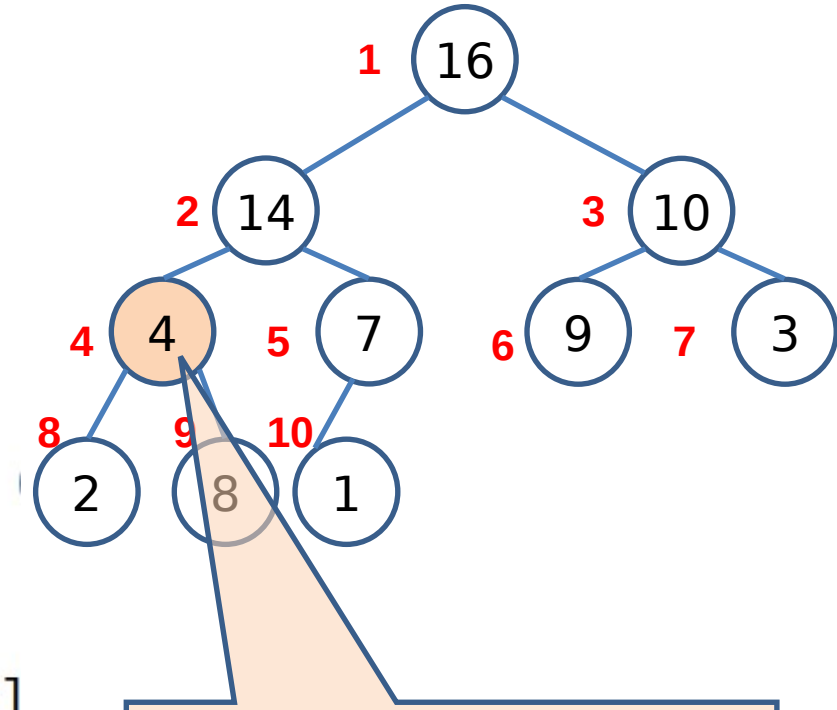
Solução:

Troca A[2] com o maior dos seus dois filhos!

Refaz Heap Máximo

- Algoritmo:

```
refazHeapMaximo(A[],i)
l ← filho_esquerdo(i)
r ← filho_direito(i)
se l ≤ tamanho-do-heap[A] e A[l] > A [i]
        maior ← l
senão
        maior ← i
fim se
se r ≤ tamanho-do-heap[A] e A[r] > A [maior]
        maior ← r
fim se
se maior ≠ i
        trocar A[i] ↔ A[maior]
        refazHeapMaximo(A,maior)
fim se
```



A[4] violando o *heap* porque é menor que seus filhos!

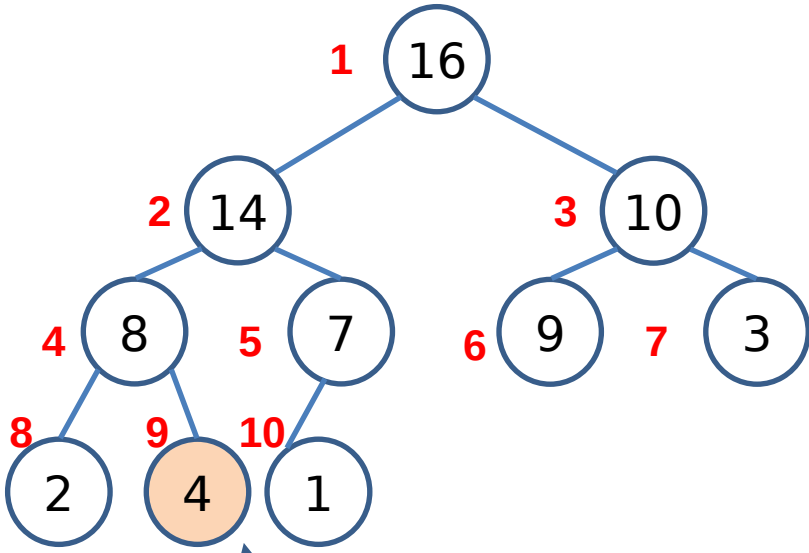
Solução:

Chamada recursiva (a, 4)

Refaz Heap Máximo

- Algoritmo:

```
refazHeapMaximo(A[],i)
l ← filho_esquerdo(i)
r ← filho_direito(i)
se l ≤ tamanho-do-heap[A] e A[l] > A [i]
       maior ← l
senão
       maior ← i
fim se
se r ≤ tamanho-do-heap[A] e A[r] > A [maior]
       maior ← r
fim se
se maior ≠ i
       trocar A[i] ↔ A[maior]
       refazHeapMaximo(A,maior)
fim se
```

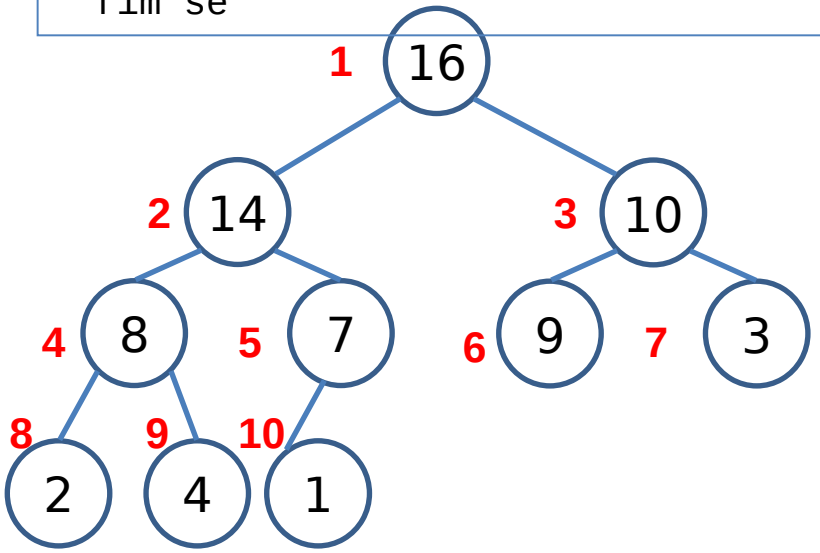


Chamada recursiva (A,9): não produz mudança adicional na estrutura

Refaz Heap Máximo

- Algoritmo:

```
refazHeapMaximo(A[],i)
l ← filho_esquerdo(i)
r ← filho_direito(i)
se l ≤ tamanho-do-heap[A] e A[l] > A [i]
        maior ← l
senão
        maior ← i
fim se
se r ≤ tamanho-do-heap[A] e A[r] > A
[maior]
        maior ← r
fim se
se maior ≠ i
        trocar A[i] ↔ A[maior]
        refazHeapMaximo(A,maior)
fim se
```

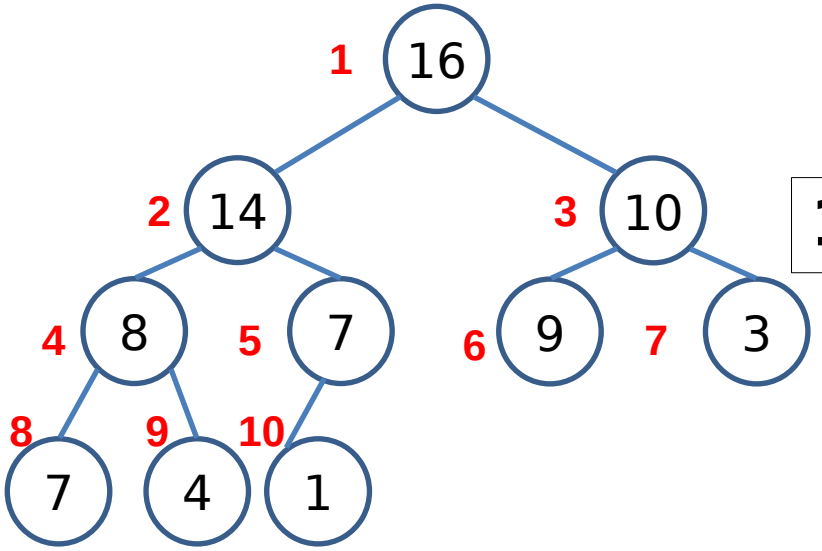


Qual tempo de execução em uma subárvore de tamanho ***n***, com raiz em um dado nó ***i*** ?

- $\Theta(1)$ para corrigir relacionamentos entre os elementos A[i], A[esquerda(i)] e A[direita(i)] *(somente faz a troca no vetor)* +
- tempo de executar refazHeapMaximo em uma subárvore com raiz em um dos filhos do nó *i*. *Ou seja, para um dado valor i = I (da primeira chamada), as chamadas seguintes i pode assumir os valores 2I, 4I, … 2^kI, enquanto $2^k I \leq n$. Logo, o número máximo de chamadas é lg (n/I), o que correponde à altura h do nó I na árvore - O(h).*

Para o nó raiz do heap = *O(lg n)*

Heap

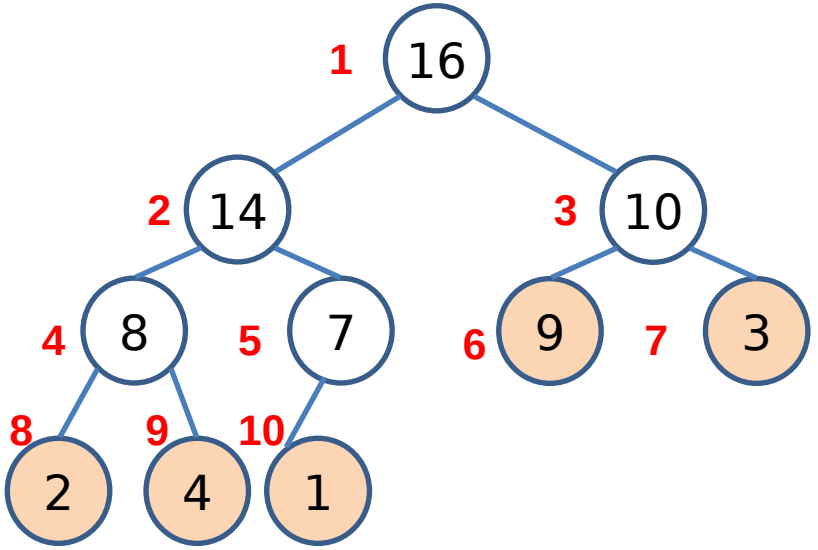


1	2	3	4	5	6	7	8	9	10
16	14	10	8	7	9	3	7	4	1

- Operações básicas sobre estrutura de *heaps:*
  - ***refaz heap máximo***
  - ***construir heap máximo***

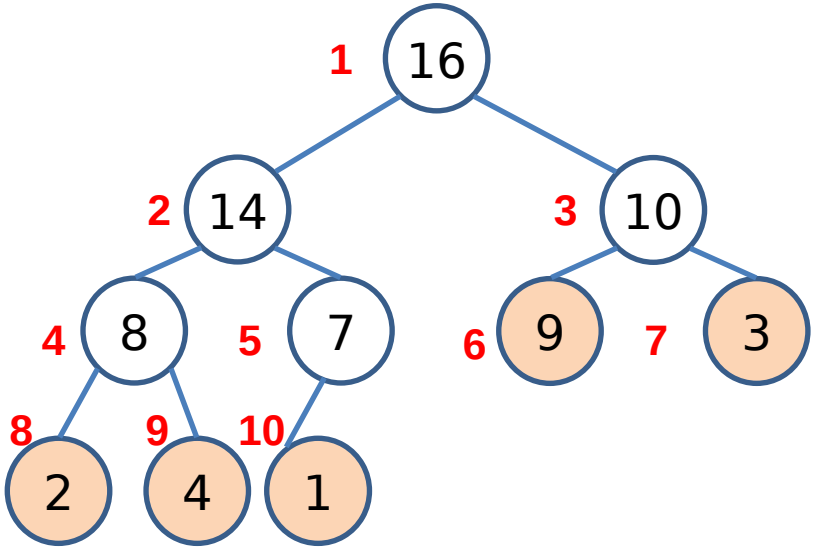
Construção do Heap

- Agora precisamos saber como construir uma estrutura de **heap**
- Dado um arranjo [A..n], quais os índices dos nós folhas(último nível da árvore)?



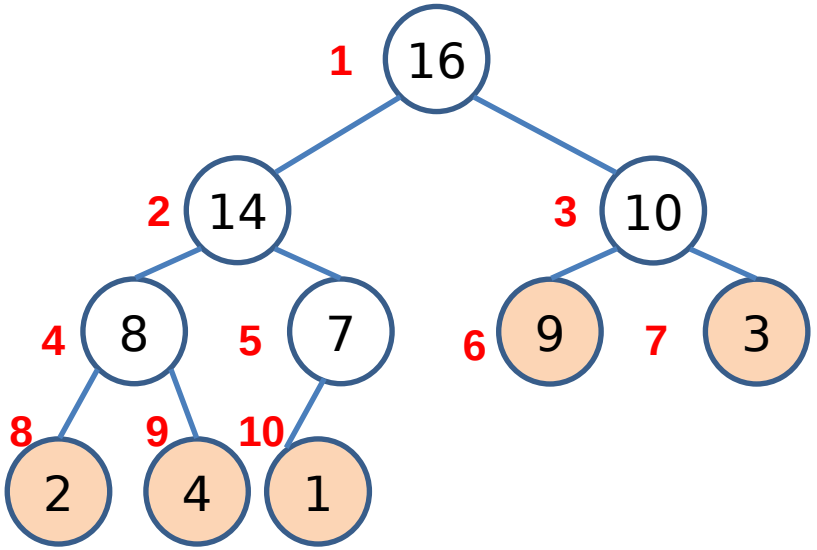
Construção do Heap

- Agora precisamos saber como construir uma estrutura de **heap**:
- Dado um arranjo [A..n], quais os índices dos nós folhas(último nível da árvore)? $(\lfloor n/2 \rfloor + 1) .. n$



Construção do Heap

- Agora precisamos saber como construir uma estrutura de **heap**:
- Dado um arranjo [A..n], quais os índices dos nós folhas(último nível da árvore)? $(\lfloor n/2 \rfloor + 1) .. n$
- Cada nó folha é um ***heap*** de um elemento com o qual podemos começar a construir a árvore.
- O procedimento constroiHeapMaximo percorre os nós restantes e executa o procedimento refazHeapMaximo sobre cada um dos nós folha.

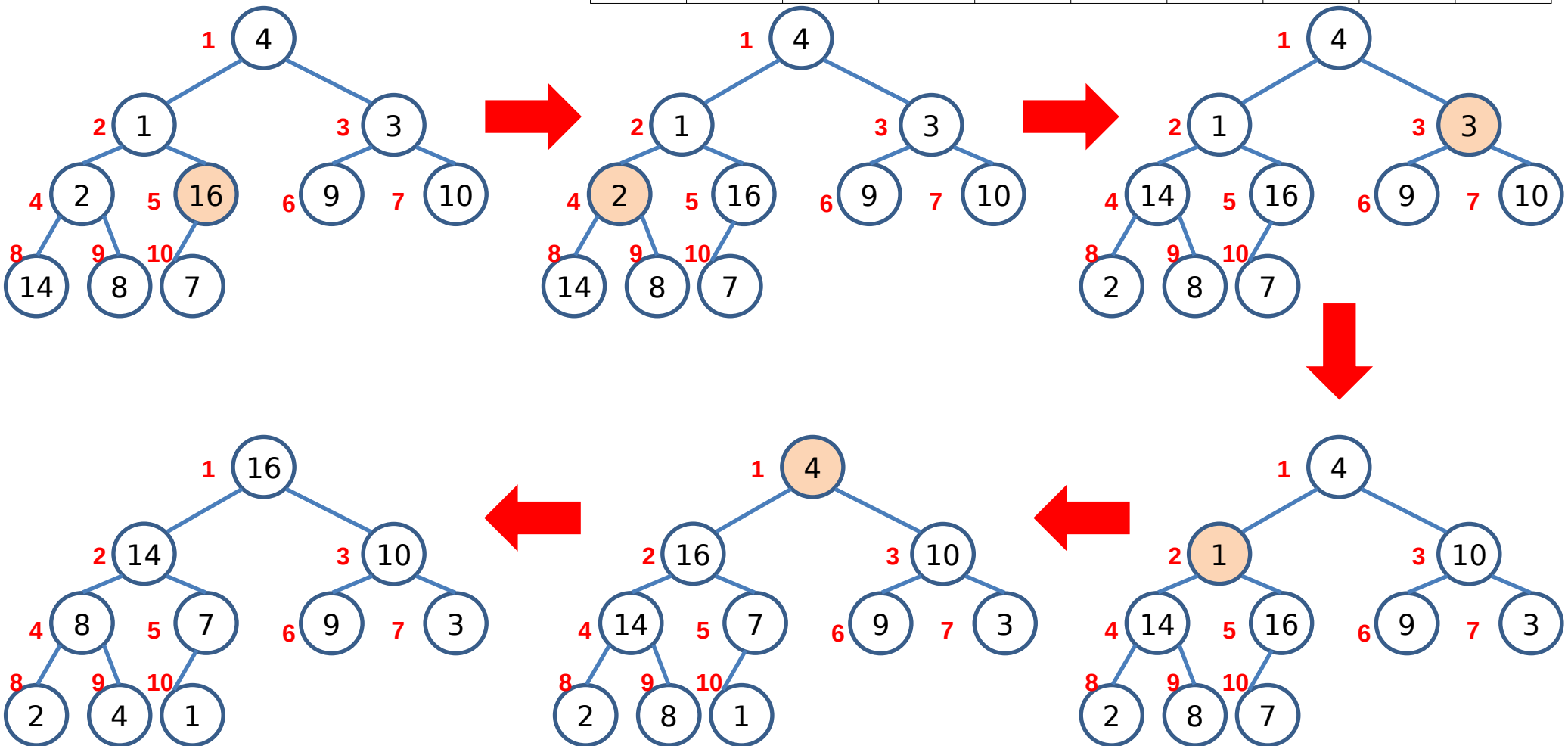



```
constroiHeapMaximo(A[])
tamanhoHeap ← tamanho[A]
para i ← ⌊tamanho[A]/2⌋ até 1, com decremento -1
  faça refazHeapMaximo(A,i)
```

Construção do Heap

```
constroiHeapMaximo(A[])
tamanhoHeap ← tamanho[A]
para i ← ⌊tamanho[A]/2⌋ até 1, com decremento -1
  faça refazHeapMaximo(A,i)
```

1	2	3	4	5	6	7	8	9	10
4	1	3	2	16	9	10	14	8	7



Construção do Heap

```
constroiHeapMaximo(A[])
tamanhoHeap ← tamanho[A]
para i ← ⌊tamanho[A]/2⌋ até 1, com decremento -1
  faça refazHeapMaximo(A,i)
```

- Analisando a complexidade:
  - cada chamada a refazHeapMaximo tem T(n) = O(lg n) e existe O(n) chamadas. Então: *T(n) = O(n lg n).*
  - No entanto, é possível definir essa complexidade mais restritamente.
  - Se analisarmos a complexidade em função da altura da árvore, chegaremos a **O(n)**, pois afinal a complexidade do refazHeapMaximo é O(h), sendo h a altura do nó I no qual é feita a primeira chamada do refazHeapMaximo.

Construção do Heap

```
constroiHeapMaximo(A[])
tamanhoHeap ← tamanho[A]
para i ← ⌊tamanho[A]/2⌋ até 1, com decremento -1
  faça refazHeapMaximo(A,i)
```

- Analisando a complexidade:
  - a complexidade do refazHeapMaximo é O(h), sendo h a altura do nó I no qual é feita a primeira chamada do refazHeapMaximo.
  - Há nós com altura h variando de 0 até lg n
  - Há no máximo $\lceil n/2^{h+1} \rceil$ nós com uma certa altura h
  - Então a complexidade total é:

$$\sum_{h=0}^{\lfloor \lg n \rfloor} \left\lceil \frac{n}{2^{h+1}} \right\rceil O(h) = O\left(n \sum_{h=0}^{\lfloor \lg n \rfloor} \frac{h}{2^h} \right)$$

< 2, pois

$$\sum_{h=0}^{\infty} \frac{h}{2^h} = \frac{1/2}{(1-1/2)^2} = 2 .$$

(x=1/2 na equação A.8 no livro do Cormen)

- Logo

$$O\left(n \sum_{h=0}^{\lfloor \lg n \rfloor} \frac{h}{2^h} \right) = O\left(n \sum_{h=0}^{\infty} \frac{h}{2^h} \right) = O(n) .$$

HeapSort

- Agora que já sabemos como funciona a estrutura de ***heap,*** podemos definir o algoritmo do HeapSort:

HeapSort

- Agora que já sabemos como funciona a estrutura de ***heap,*** podemos definir o algoritmo do HeapSort:
  - construir o *heap* no arranjo *A[1..n];*
  - máximo de A ficará em *A[1]*: colocá-lo na posição correta, trocando com *A[n]*
  - se desprezarmos o nó ***n*** do *heap,* transformaremos *A[1..n-1]* em um *heap* máximo:
    - filhos da raiz continuam sendo *heap* máximos, mas o novo elemento pode violar a propriedade de *heap* máximo
      - deve-se chamar *refazHeapMaximo(A,1*), que deixa um *heap* máximo em *A[1..(n-1)];*
      - *HeapSort* repete este processo para o *heap* de tamanho n-1, descendo até *heap* de tamanho 2.

HeapSort

- Agora que já sabemos como funciona a estrutura de ***heap,*** podemos definir o algoritmo do HeapSort:
  - construir o *heap* no arranjo *A[1..n];*
  - máximo de A ficará em *A[1]*: colocá-lo na posição correta, trocando com *A[n]*
  - se desprezarmos o nó ***n*** do *heap,* transformaremos *A[1..n-1]* em um *heap* máximo:
    - filhos da raiz continuam sendo *heap* máximos, mas o novo elemento pode violar a propriedade de *heap* máximo
      - deve-se chamar *refazHeapMaximo(A,1*), que deixa um *heap* máximo em *A[1..(n-1)];*
      - *HeapSort* repete este processo para o *heap* de tamanho n-1, descendo até *heap* de tamanho 2.

```
HeapSort(A[], n)
constroiHeapMaximo(A)
para i ← n até 2
     trocar A[1] ↔ A[i]
     tamanhoDoHeap[A] ← tamanhoDoHeap[A] – 1
     refazHeapMaximo(A,1)
```

HeapSort

- Agora que já sabemos como funciona a estrutura de ***heap,*** podemos definir o algoritmo do HeapSort:
  - construir o *heap* no arranjo *A[1..n];*
  - máximo de A ficará em *A[1]*: colocá-lo na posição correta, trocando com *A[n]*
  - se desprezarmos o nó ***n*** do *heap,* transformaremos *A[1..n-1]* em um *heap* máximo:
    - filhos da raiz continuam sendo *heap* máximos, mas o novo elemento pode violar a propriedade de *heap* máximo
      - deve-se chamar *refazHeapMaximo(A,1*), que deixa um *heap* máximo em *A[1..(n-1)];*
      - *HeapSort* repete este processo para o *heap* de tamanho n-1, descendo até *heap* de tamanho 2.

```
HeapSort(A[], n)
constroiHeapMaximo(A)
para i ← n até 2
     trocar A[1] ↔ A[i]
     tamanhoDoHeap[A] ← tamanhoDoHeap[A] – 1
     refazHeapMaximo(A,1)
```

Complexidade:

HeapSort

- Agora que já sabemos como funciona a estrutura de ***heap,*** podemos definir o algoritmo do HeapSort:
  - construir o *heap* no arranjo *A[1..n];*
  - máximo de A ficará em *A[1]*: colocá-lo na posição correta, trocando com *A[n]*
  - se desprezarmos o nó ***n*** do *heap,* transformaremos *A[1..n-1]* em um *heap* máximo:
    - filhos da raiz continuam sendo *heap* máximos, mas o novo elemento pode violar a propriedade de *heap* máximo
      - deve-se chamar *refazHeapMaximo(A,1*), que deixa um *heap* máximo em *A[1..(n-1)];*
      - *HeapSort* repete este processo para o *heap* de tamanho n-1, descendo até *heap* de tamanho 2.

```
HeapSort(A[], n)
constroiHeapMaximo(A) - O(n)
para i ← n até 2
     trocar A[1] ↔ A[i]
     tamanhoDoHeap[A] ← tamanhoDoHeap[A] – 1
     refazHeapMaximo(A,1) – O(lg n)
```

Complexidade: O(n lg n)

HeapSort

- Agora que já sabemos como funciona a estrutura de ***heap,*** podemos definir o algoritmo do HeapSort:
  - construir o *heap* no arranjo *A[1..n];*
  - máximo de A ficará em *A[1]*: colocá-lo na posição correta, trocando com *A[n]*
  - se desprezarmos o nó ***n*** do *heap,* transformaremos *A[1..n-1]* em um *heap* máximo:
    - filhos da raiz continuam sendo *heap* máximos, mas o novo elemento pode violar a propriedade de *heap* máximo
      - deve-se chamar *refazHeapMaximo(A,1*), que deixa um *heap* máximo em *A[1..(n-1)];*
      - *HeapSort* repete este processo para o *heap* de tamanho n-1, descendo até *heap* de tamanho 2.

```
HeapSort(A[], n)
constroiHeapMaximo(A) - O(n)
para i ← n até 2
     trocar A[1] ↔ A[i]
     tamanhoDoHeap[A] ← tamanhoDoHeap[A] – 1
     refazHeapMaximo(A,1) – O(lg n)
```

Complexidade: O(n lg n) in loco: ?

HeapSort

- Agora que já sabemos como funciona a estrutura de ***heap,*** podemos definir o algoritmo do HeapSort:
  - construir o *heap* no arranjo *A[1..n];*
  - máximo de A ficará em *A[1]*: colocá-lo na posição correta, trocando com *A[n]*
  - se desprezarmos o nó ***n*** do *heap,* transformaremos *A[1..n-1]* em um *heap* máximo:
    - filhos da raiz continuam sendo *heap* máximos, mas o novo elemento pode violar a propriedade de *heap* máximo
      - deve-se chamar *refazHeapMaximo(A,1*), que deixa um *heap* máximo em *A[1..(n-1)];*
      - *HeapSort* repete este processo para o *heap* de tamanho n-1, descendo até *heap* de tamanho 2.

```
HeapSort(A[], n)
constroiHeapMaximo(A) - O(n)
para i ← n até 2
     trocar A[1] ↔ A[i]
     tamanhoDoHeap[A] ← tamanhoDoHeap[A] – 1
     refazHeapMaximo(A,1) – O(lg n)
```

Complexidade: O(n lg n) ***in loco***: SIM!

HeapSort

- Agora que já sabemos como funciona a estrutura de ***heap,*** podemos definir o algoritmo do HeapSort:
  - construir o *heap* no arranjo *A[1..n];*
  - máximo de A ficará em *A[1]*: colocá-lo na posição correta, trocando com *A[n]*
  - se desprezarmos o nó ***n*** do *heap,* transformaremos *A[1..n-1]* em um *heap* máximo:
    - filhos da raiz continuam sendo *heap* máximos, mas o novo elemento pode violar a propriedade de *heap* máximo
      - deve-se chamar *refazHeapMaximo(A,1*), que deixa um *heap* máximo em *A[1..(n-1)];*
      - *HeapSort* repete este processo para o *heap* de tamanho n-1, descendo até *heap* de tamanho 2.

```
HeapSort(A[], n)
constroiHeapMaximo(A) - O(n)
para i ← n até 2
     trocar A[1] ↔ A[i]
     tamanhoDoHeap[A] ← tamanhoDoHeap[A] – 1
     refazHeapMaximo(A,1) – O(lg n)
```

Complexidade: O(n lg n) ***in loco***: SIM! **Estável:** ?

HeapSort

- Agora que já sabemos como funciona a estrutura de ***heap,*** podemos definir o algoritmo do HeapSort:
  - construir o *heap* no arranjo *A[1..n];*
  - máximo de A ficará em *A[1]*: colocá-lo na posição correta, trocando com *A[n]*
  - se desprezarmos o nó ***n*** do *heap,* transformaremos *A[1..n-1]* em um *heap* máximo:
    - filhos da raiz continuam sendo *heap* máximos, mas o novo elemento pode violar a propriedade de *heap* máximo
      - deve-se chamar *refazHeapMaximo(A,1*), que deixa um *heap* máximo em *A[1..(n-1)];*
      - *HeapSort* repete este processo para o *heap* de tamanho n-1, descendo até *heap* de tamanho 2.

```
HeapSort(A[], n)
constroiHeapMaximo(A) - O(n)
para i ← n até 2
     trocar A[1] ↔ A[i]
     tamanhoDoHeap[A] ← tamanhoDoHeap[A] – 1
     refazHeapMaximo(A,1) – O(lg n)
```

Complexidade: O(n lg n) ***in loco***: SIM! **Estável:** NÃO

Comparando...

	T(n)			C(n)			M(n)			in loco?	estável?
Algoritmo	Melhor caso	Pior caso	Caso médio	Melhor caso	Pior caso	Caso médio	Melhor caso	Pior caso	Caso médio		
InsertionSort	O(n)	$O(n^2)$	$O(n^2)$	O(n)	$O(n^2)$	$O(n^2)$	O(1)	$O(n^2)$	$O(n^2)$	sim	sim
SelectionSort	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(n^2)$	O(n)	O(n)	O(n)	sim	não
BubbleSort											
MergeSort	O(n lgn)	O(n lgn)	O(n lgn)	O(n lgn)	O(n lgn)	O(n lgn)	O(n lgn)	O(n lgn)	O(n lgn)	não	sim
HeapSort	O(n lgn)	O(n lgn)	O(n lgn)							sim	não

- O que podem falar agora do HeapSort (em comparação com os demais?

Comparando...

	T(n)			C(n)			M(n)			in loco?	estável?
Algoritmo	Melhor caso	Pior caso	Caso médio	Melhor caso	Pior caso	Caso médio	Melhor caso	Pior caso	Caso médio		
InsertionSort	O(n)	$O(n^2)$	$O(n^2)$	O(n)	$O(n^2)$	$O(n^2)$	O(n)	$O(n^2)$	$O(n^2)$	sim	sim
SelectionSort	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(n^2)$	O(n)	O(n)	O(n)	sim	não
BubbleSort											
MergeSort	O(n lgn)	O(n lgn)	O(n lgn)	O(n lgn)	O(n lgn)	O(n lgn)	O(n lgn)	O(n lgn)	O(n lgn)	não	sim
HeapSort	O(n lgn)	O(n lgn)	O(n lgn)							sim	não

- O que podem falar agora do HeapSort (em comparação com os demais?
  - Boa escolha se o arquivo é grande e desordenado, precisa economizar memória, e estabilidade não é requisito

HeapSort

- Detalhes de implementação:
  - Como pode ser implementado esse heap?

```
HeapSort(A[], n)
constroiHeapMaximo(A)
para i ← n até 2
     trocar A[1] ↔ A[i]
     tamanhoDoHeap[A] ← tamanhoDoHeap[A] – 1
     refazHeapMaximo(A,1)
```

HeapSort

- Detalhes de implementação:
  - Como pode ser implementado esse heap?

```
typedef struct {
    int[] A;
    int n;      /* tamanho do vetor A */
    int tamanhoDoHeap;   /* não necessariamente igual a n */
} Heap;
```

```
HeapSort(A[], n)
constroiHeapMaximo(A)
para i ← n até 2
     trocar A[1] ↔ A[i]
     tamanhoDoHeap[A] ← tamanhoDoHeap[A] – 1
     refazHeapMaximo(A,1)
```

HeapSort

- Detalhes de implementação:
  - Como pode ser implementado esse heap?

```
typedef struct {
    int[] A;
    int n;      /* tamanho do vetor A */
    int tamanhoDoHeap;   /* não necessariamente igual a n */
} Heap;
```

```
HeapSort(Heap* h)
constroiHeapMaximo(h)
para i ← h→n até 2
     trocar h→A[1] ↔ h→A[i]
     h→tamanhoDoHeap ← h→tamanhoDoHeap – 1
     refazHeapMaximo(h,1)
```

Exercícios

- Qual a complexidade para o número de comparações (C(n)) e movimentações de registros (M(n)) para o Heapsort?
- Façam os exercícios do cap 6 do livro do Cormen (Cap sobre Heapsort, com exceção da seção 6.5 sobre filas de prioridades).

Referências (com exercícios!)

- Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest & Clifford Stein. Algoritmos - 3a. ed. Edição Americana. Editora Campus, 2002. Cap 6
- Paulo Feofiloff. Algoritmos em C. Cap 10 (tem exercícios!!!) https://www.ime.usp.br/~pf/algoritmos-livro/
- Nívio Ziviani. Projeto de Algoritmos com implementações em C e Pascal. Editora Thomson, 3a. Edição, 2004. Cap 4.1.5 - http://www2.dcc.ufmg.br/livros/algoritmos/implementacoes.php
- Apostila Prof. Marcio – cap 6.5