Algoritmos de Ordenação

HeapSort

Professora:

Fátima L. S. Nunes

SISTEMAS DE INFORMAÇÃO

EACH

Algoritmos de Ordenação

- Algoritmos de ordenação que já conhecemos:?
  - *Insertion Sort* (Ordenação por Inserção)
  - *Selection Sort* (Ordenação por Seleção)
  - *Bubble Sort* (Ordenação pelo método da Bolha)
  - *MergeSort* (Ordenação por intercalação)
  - *QuickSort* (Ordenação rápida)

Algoritmos de Ordenação

- Algoritmos de ordenação que já conhecemos:?
  - *Insertion Sort* (Ordenação por Inserção)
  - *Selection Sort* (Ordenação por Seleção)
  - *Bubble Sort* (Ordenação pelo método da Bolha)
  - *MergeSort* (Ordenação por intercalação)
  - *QuickSort* (Ordenação rápida)
  - Hoje: ***HeapSort*** (???)

HeapSort

- O que ***heap***?

- O que ***heap***?
  - De acordo com tradutor da Google:

substantivo

pilha
monte
montão
acervo
porção
acumulação
multidão
grande quantidade de

verbo

amontoar
acumular
empilhar
juntar
encher
amontar
carregar

Algoritmos de Ordenação

- Algoritmos de ordenação que já conhecemos:?
  - *Insertion Sort* (Ordenação por Inserção)
  - *Selection Sort* (Ordenação por Seleção)
  - *Bubble Sort* (Ordenação pelo método da Bolha)
  - *MergeSort* (Ordenação por intercalação)
  - *QuickSort* (Ordenação rápida)
  - Hoje: ***HeapSort*** (Ordenação por monte)

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***

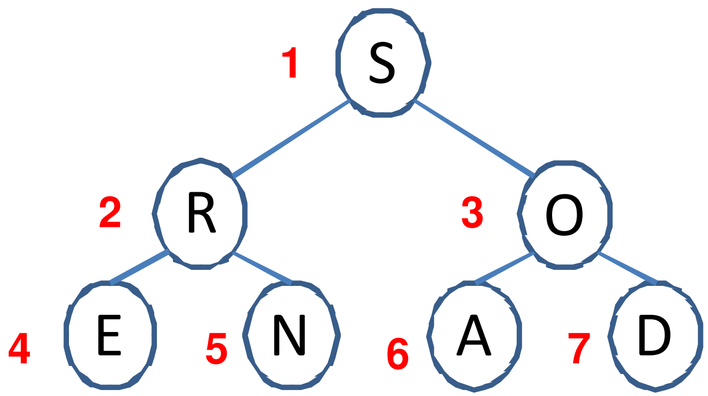
Fila de prioridades

- 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

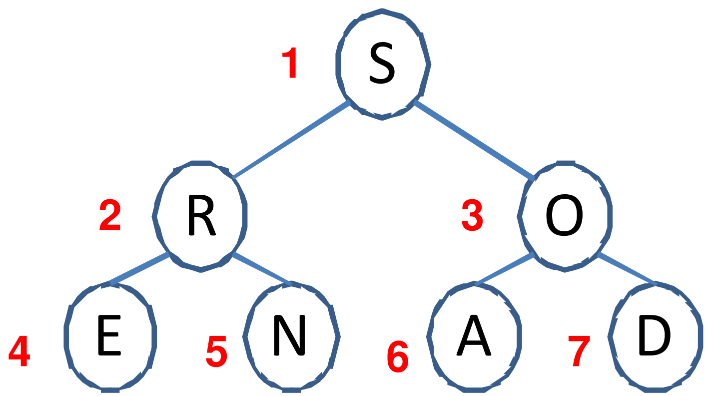
Heap

- 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: $c[1],c[2],\ldots, c[n]$ tal que $c[i] \geq c[2i]$ e $c[i] \geq c[2i+1]$, para todo $i=1, 2, \ldots, n/2$.

- 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.

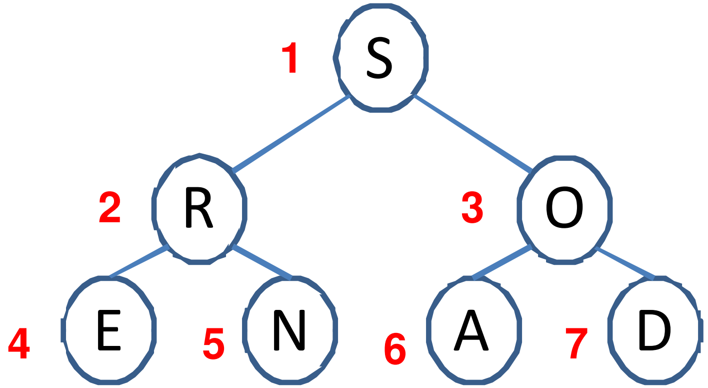


- Definição:
  - **árvore binária completa**: árvore binária com os nós numerados de ***1*** a ***n***.
  - primeiro nó é chamado raiz;
  - nó *[k/2]* é o pai do nó *k*, para $1 < k \leq n$;
  - nós *2k* e *2k+1* são filhos à esquerda e à direita do nó *k*, para $1 \leq k \leq \lfloor k/2 \rfloor$.



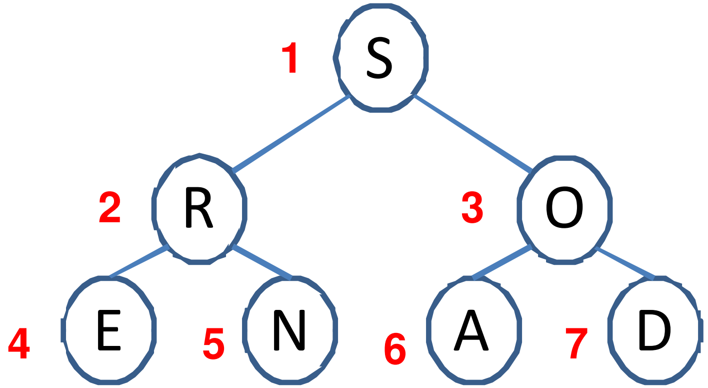
Heap

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



Heap

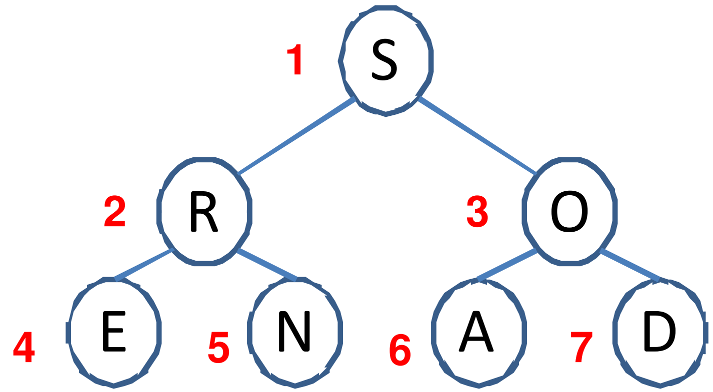
- Uma **árvore binária completa** e, consequentemente, um ***heap,*** pode ser representado por um *array*.
- Como seria um *array* para representar esta árvore?



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

Heap

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

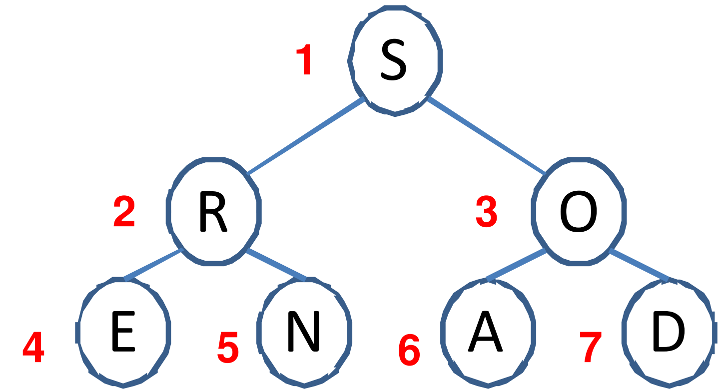


1	2	3	4	5	6	7
S	R	O	E	N	A	D

- 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?

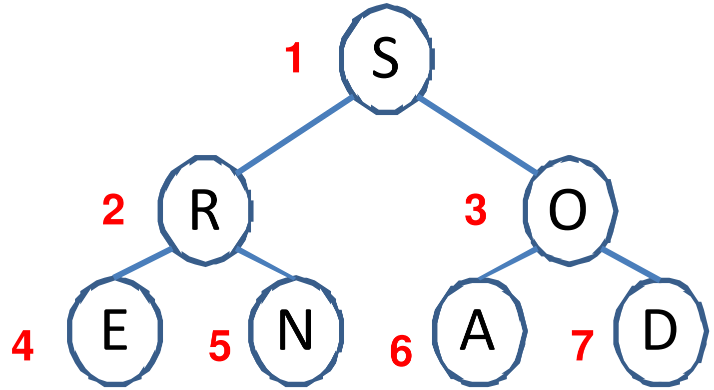


1	2	3	4	5	6	7
S	R	O	E	N	A	D

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

- Algoritmos:

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

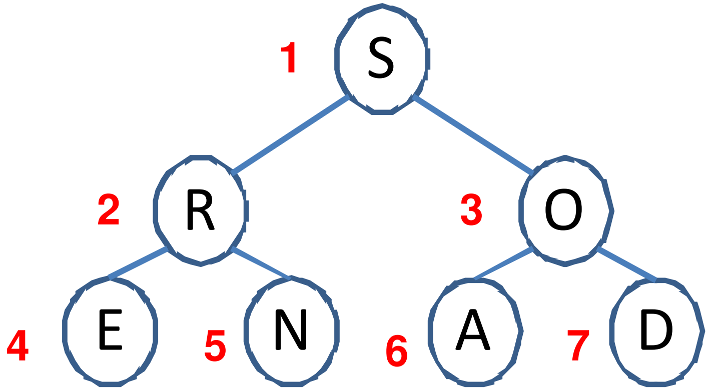



```
pai(i)
   retorna ???

esquerda(i)
   retorna ???

direita(i)
   retorna ???
```

Heap

- Algoritmos:

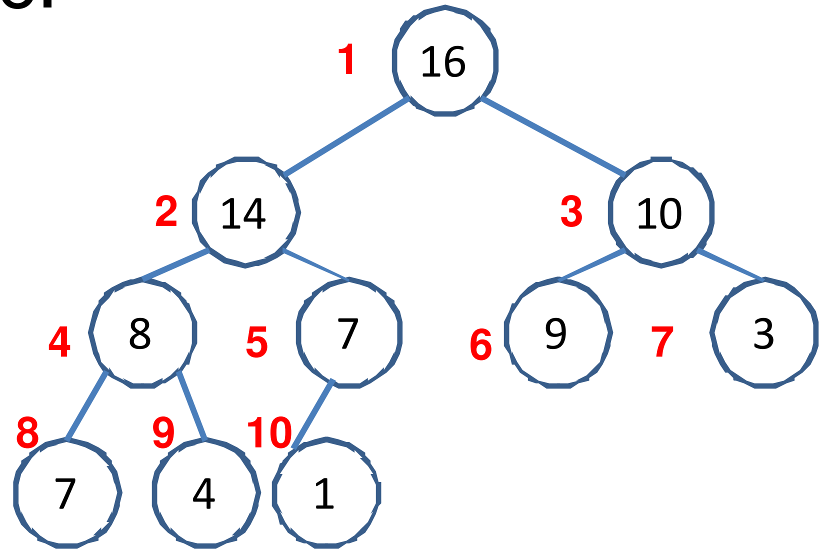
1	2	3	4	5	6	7
S	R	O	E	N	A	D




```
pai(i)
   retorna ⌊i/2⌋

esquerda(i)
   retorna 2i

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

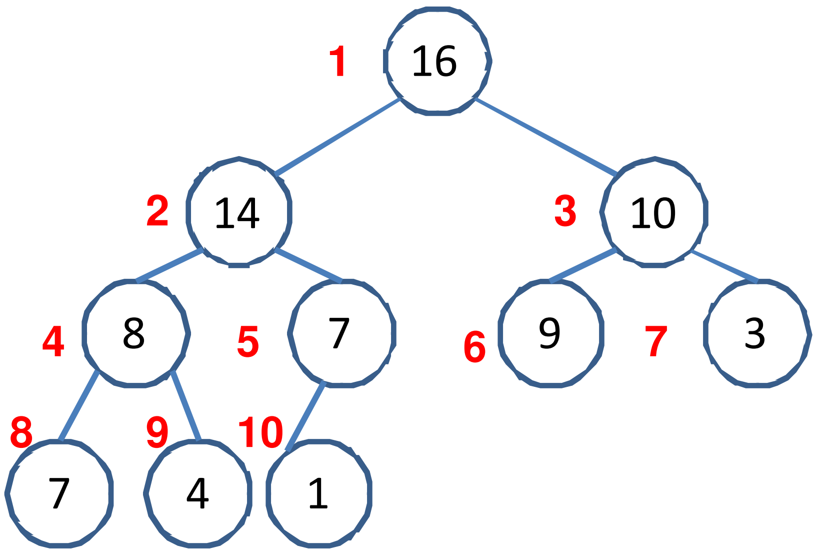
Heap

- 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**

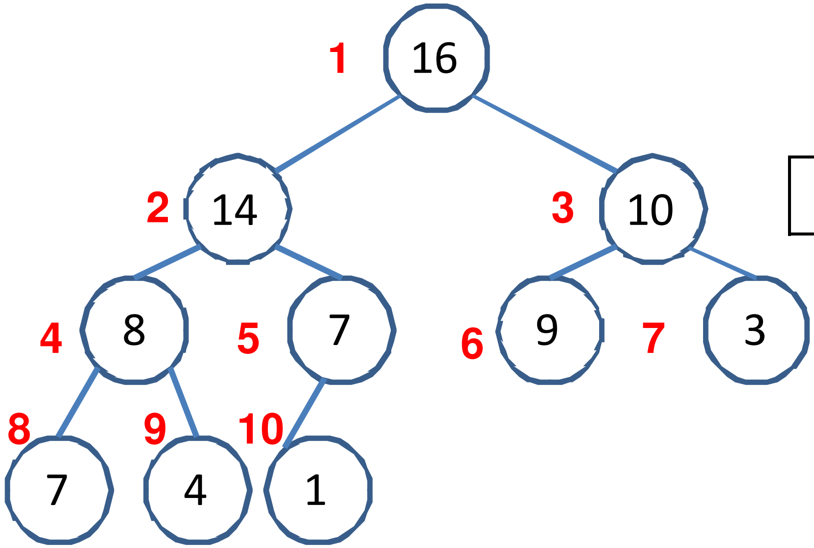
Heap



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.

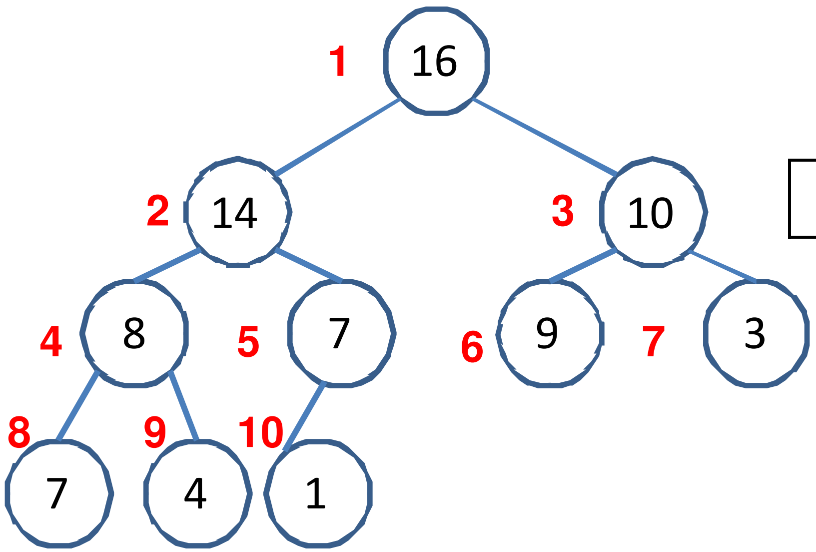
Heap



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

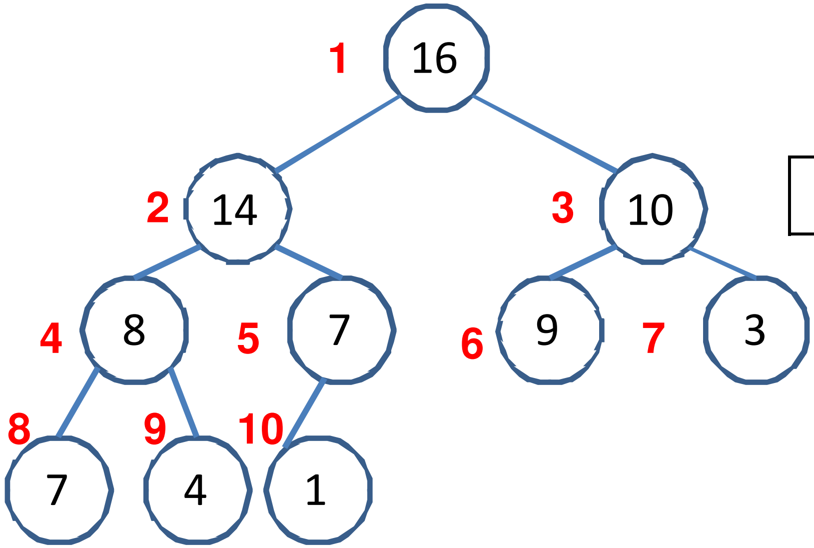
Heap



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 é igual expansão vista anteriormente: $\boldsymbol{\Theta(\lg n)}$

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***

HeapSort

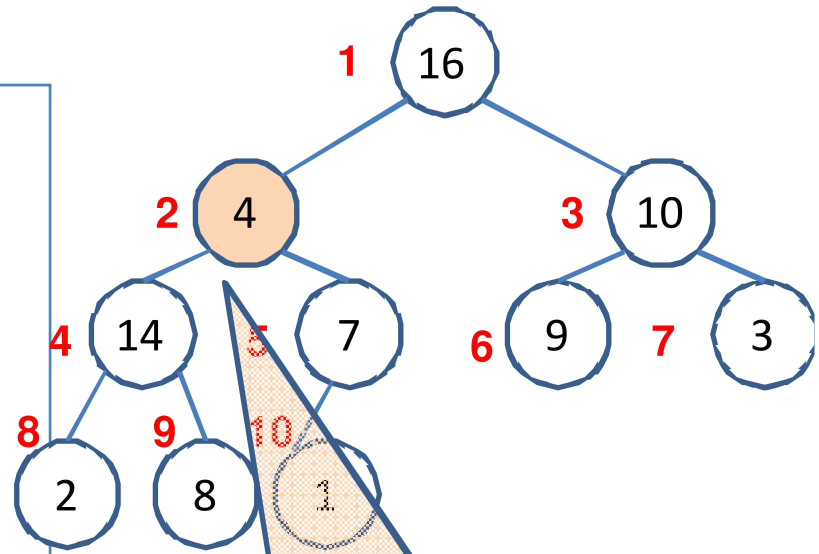
- Algoritmo:

```
refazHeapMaximo(A[],i)
l ← esquerda(i)
r ← direita(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
```

HeapSort

- Algoritmo:

```
refazHeapMaximo(A[],i)
l ← esquerda(i)
r ← direita(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
```



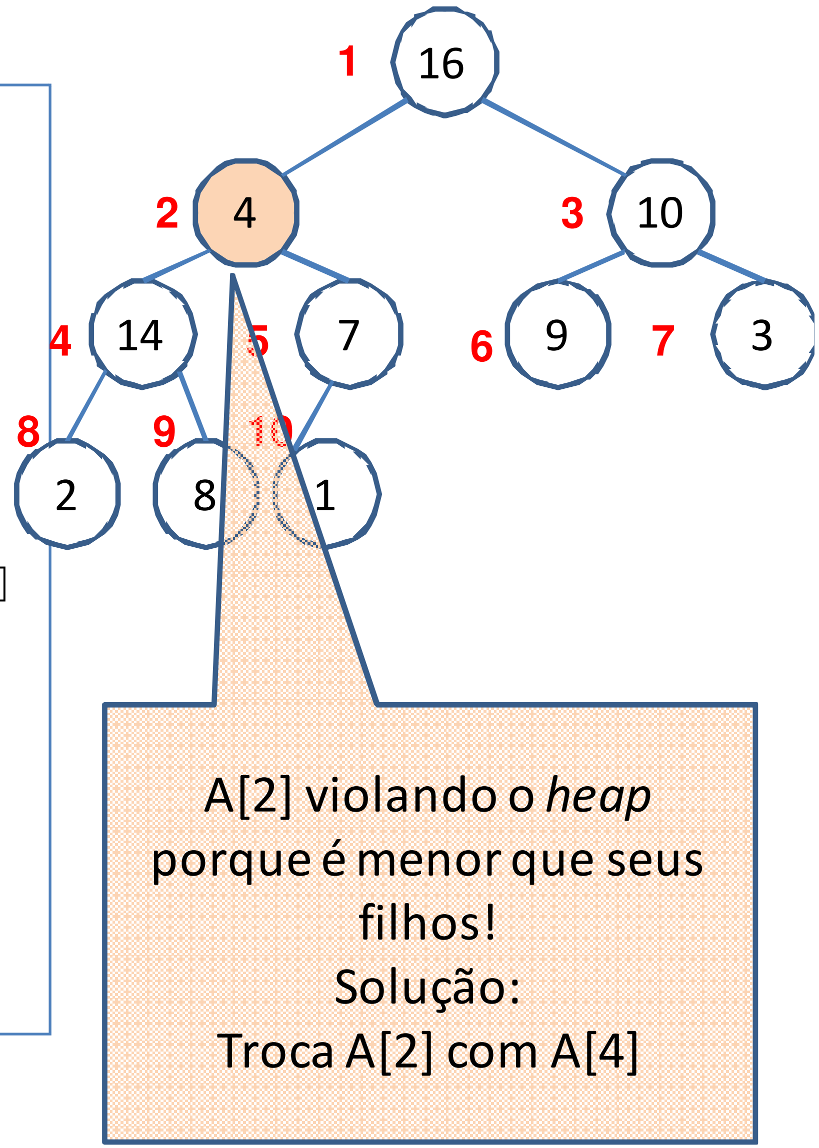
Violando o heap porque é menor que seus filhos!

COMO RESOLVER?
APLIQUE O ALGORITMO E DESCUBRA O QUE ACONTECE!

HeapSort

- Algoritmo:

```
refazHeapMaximo(A[],i)
l ← esquerda(i)
r ← direita(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
```



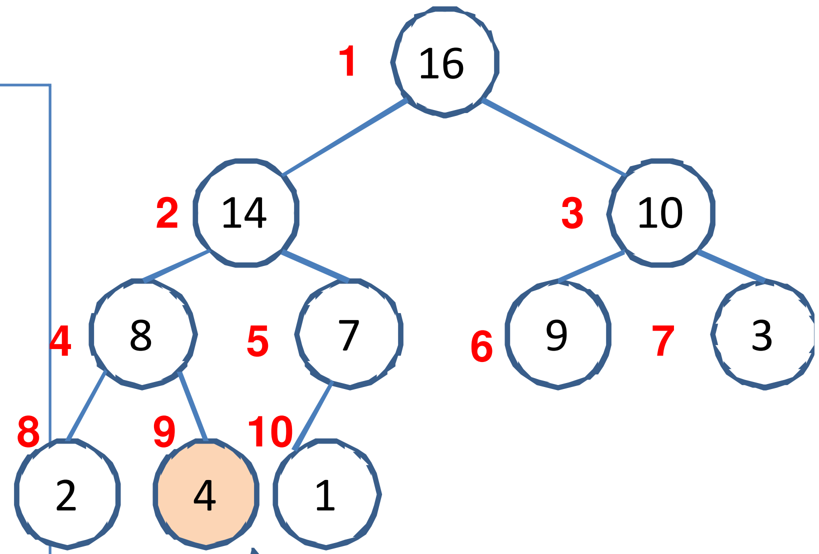
- Algoritmo:

```
refazHeapMaximo(A[],i)
l ← esquerda(i)
r ← direita(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
```

HeapSort

- Algoritmo:

```
refazHeapMaximo(A[],i)
l ← esquerda(i)
r ← direita(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
```

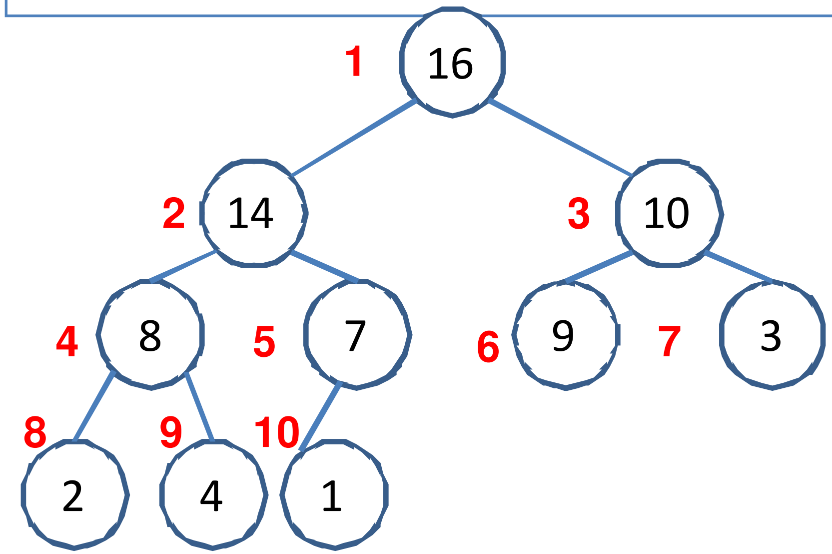


Chamada recursiva (a, 4)
Troca A[4]com A[9]
Chamada recursiva (A,9):
não produz mudança
adicional na estrutura

HeapSort

- Algoritmo:

```
refazHeapMaximo(A[],i)
l ← esquerda(i)
r ← direita(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
```



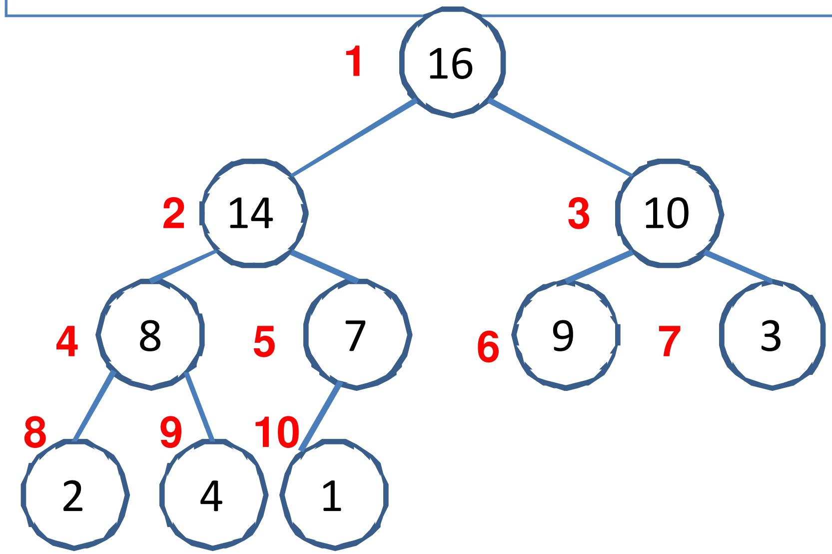
Qual tempo de execução em uma subárvore de tamanho ***n***, com raiz em um dado nó ***i*** ?

- $\Theta(1)$ para corrigir relacionamentos entre os elementos A[i], A[esquerda(i)] e A[direita(i)] *(somente faz a troca) +*
- tempo de executar refazHeapMaximo em uma subárvore com raiz em um dos filhos do nó *i. Subárvores de cada filho têm tamanho máximo = 2n/3 (pior caso última linha está metade cheia)*

HeapSort

- Algoritmo:

```
refazHeapMaximo(A[],i)
l ← esquerda(i)
r ← direita(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*** ?

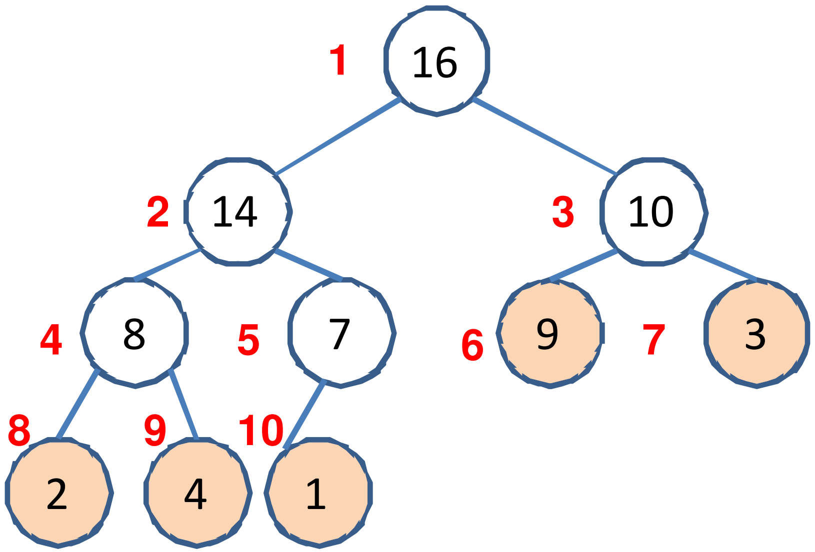
- $\Theta(1)$ para corrigir relacionamentos entre os elementos A[i], A[esquerda(i)] e A[direita(i)] *(somente faz a troca) +*
- tempo de executar refazHeapMaximo em uma subárvore com raiz em um dos filhos do nó *i. Subárvores de cada filho têm tamanho máximo = 2n/3* *(pior caso última linha está metade cheia)*

$T(n) \leq T(2n/3) + \Theta(1) \;\; = O(\log n)$

Para um nó de altura *h*, *T(n)=O(h)*

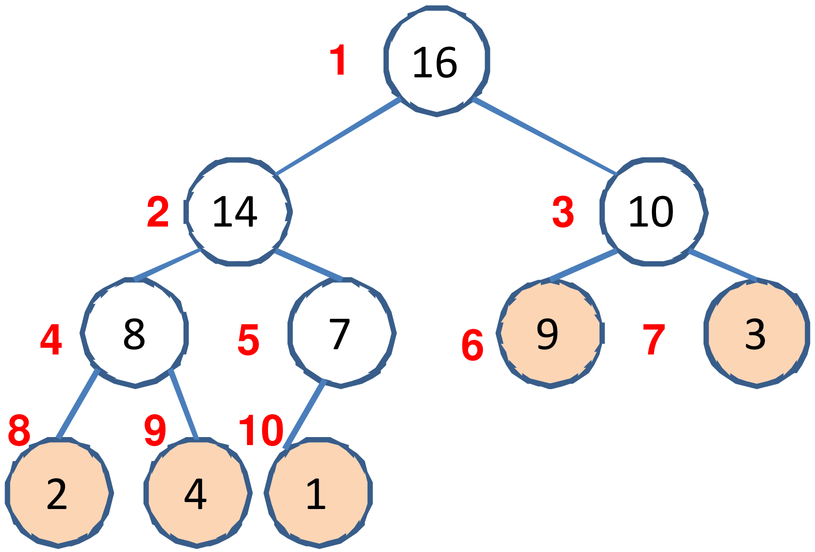
HeapSort

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



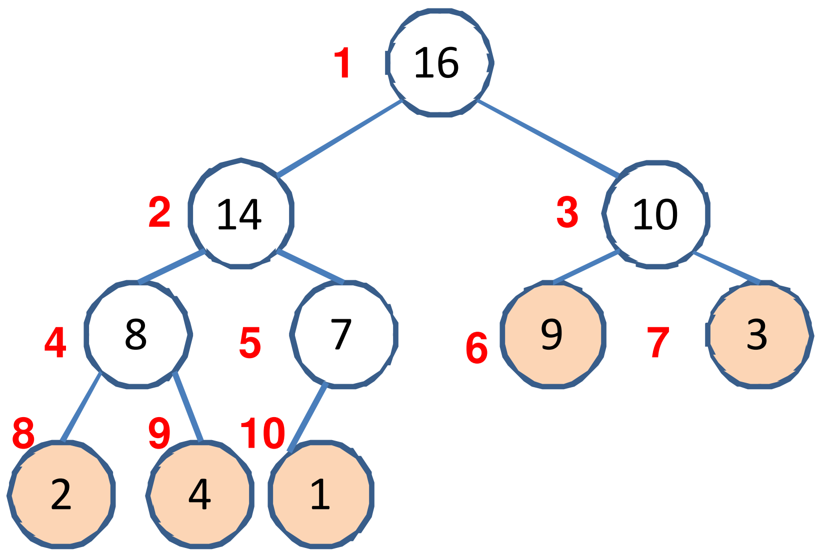
HeapSort

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



HeapSort

- 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.

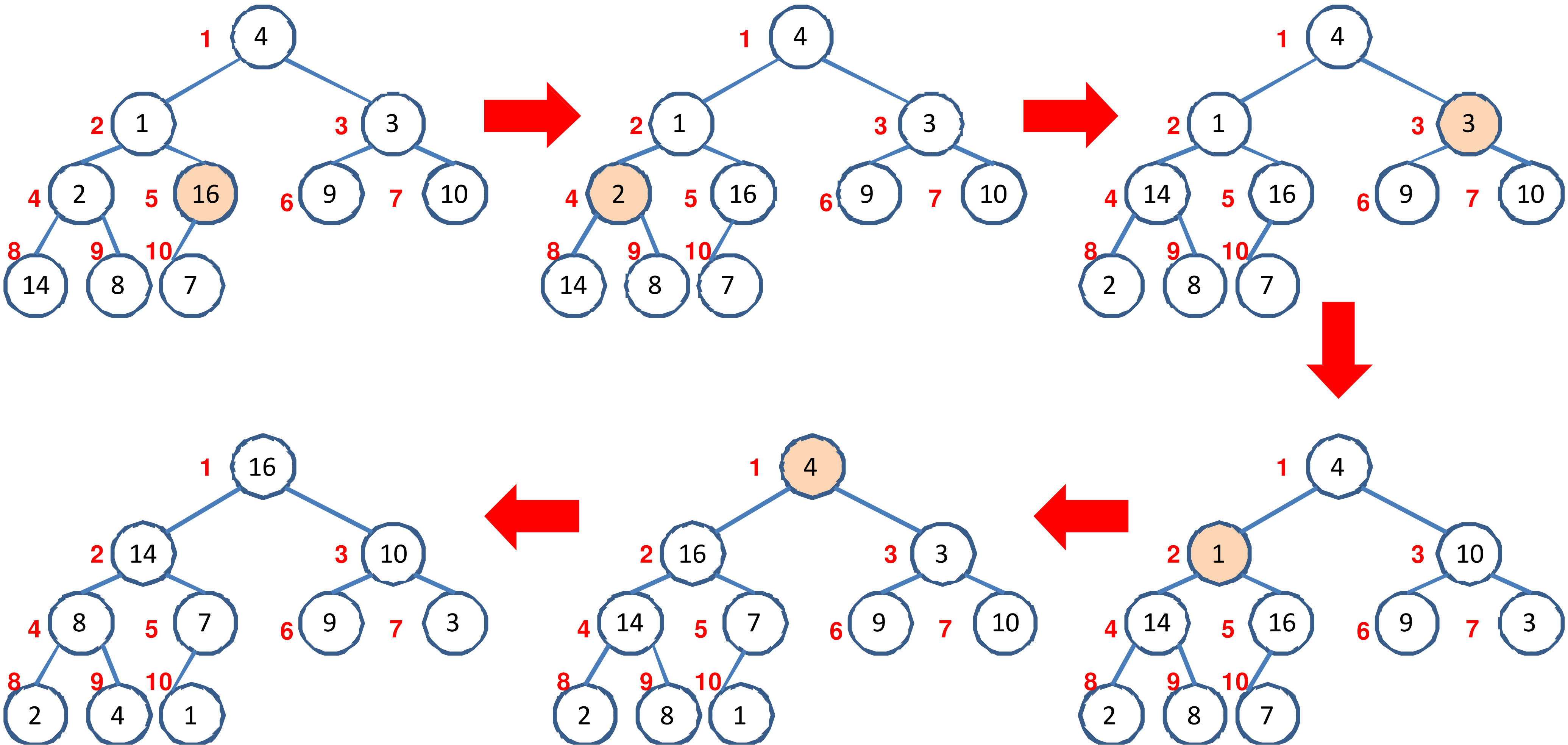



```
constroiHeapMaximo(A[])
tamanhoHeap ← tamanho[A]
para i ← (⌊tamanho[A]/2⌋ até 1, com decremento -1
    faça refazHeapMaximo(A,i)
```

HeapSort

```
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



HeapSort

```
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 constroiHeapMaximo tem T(n) = O(lg n) e existe O(n) chamadas. Então: *T(n) = 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).**

Ver página 109 do Cormen et al.

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[])
constroiHeapMaximo(A)
para i ← tamanho[A] até 2
    trocar A[1] ↔ A[i]
    tamanhoDoHeap[A] ← tamanhoDoHeap[A] - 1
    refazHeapMaximo(A,1)
```

Referências

- Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest & Clifford Stein. Algoritmos - Tradução da 2a. Edição Americana. Editora Campus, 2002.
- Nívio Ziviani. Projeto de Algoritmos com implementações em C e Pascal. Editora Thomson, 2a. Edição, 2004 (texto base)
- Notas de aula – Prof. Delano Beder – EACH-USP

Algoritmos de Ordenação

HeapSort

Professora:

Fátima L. S. Nunes

SISTEMAS DE INFORMAÇÃO

EACH