Heap

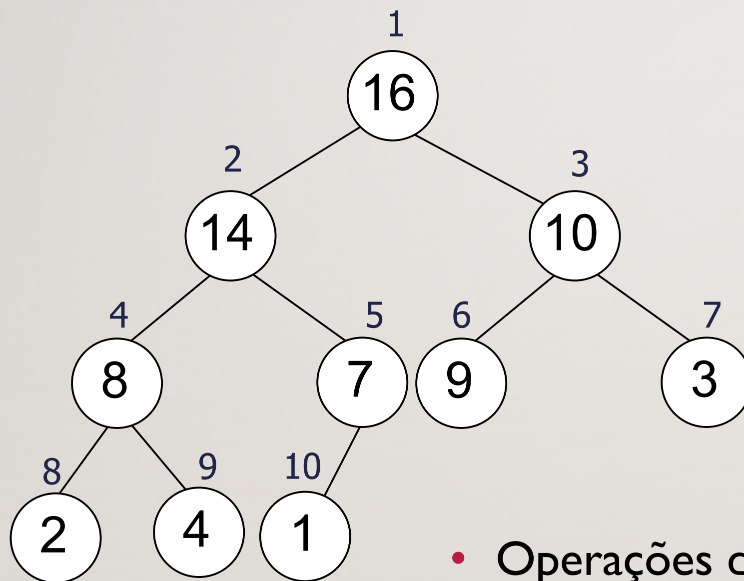
- **Heap**: vetor que pode ser visto como uma árvore binária totalmente preenchida em todos os níveis exceto, possivelmente, o último

- O último nível é preenchido da esquerda para a direita

- O vetor *A* que representa um heap possui dois atributos: tamanho (length[*A*]) e número de elementos no vetor (heap_size[*A*])

Heap

- Um heap visto como uma árvore binária ou como um *array* unidimensional



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

- Operações de <u>consulta</u> sobre um <u>nodo</u> i

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

Esquerda(i)
return(2*i)

Direita(i)
return(2*i + 1)

Propriedade do Heap

- O valor de um nodo é sempre menor ou igual ao valor de seu nodo pai
  - $A[\text{Pai}(i)] \geq A[i], \quad \forall\ i \leq \text{heap_size}[A]$
  - O elemento de maior valor encontra-se armazenado na raiz da árvore

Definições

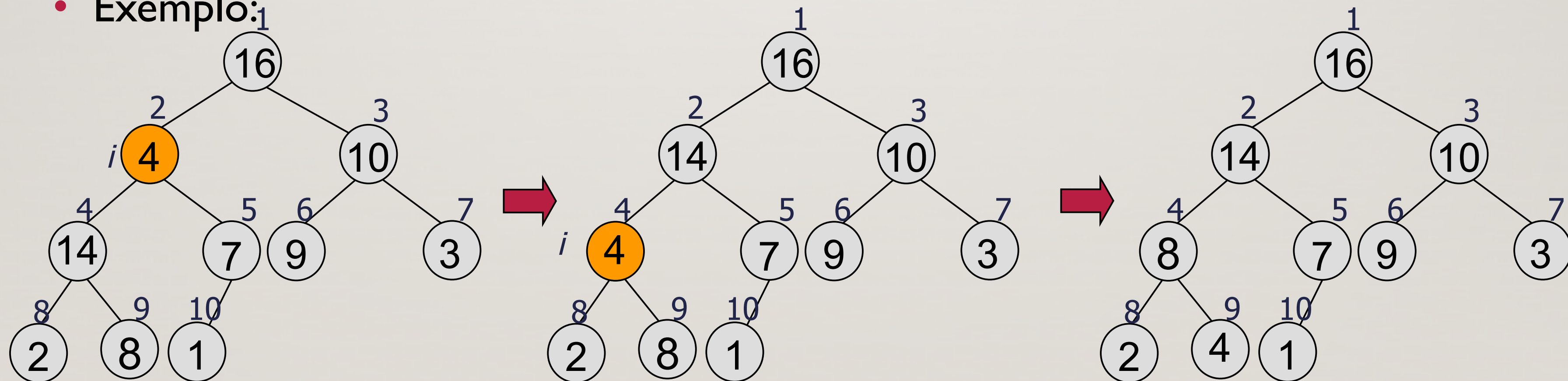
- A altura de um nodo em uma árvore corresponde ao número de arestas no caminho descendente mais longo daquele nodo até um nodo folha
- A altura de um heap de *n* elementos é $\Theta(log_2 n)$ – baseado em uma árvore binária completa
- As operações básicas sobre heaps executam em tempo no máximo proporcional a altura da árvore e, portanto, $O(log_2 n)$

Procedimentos sobre Heaps

- **Heapify**
  - Garante a manutenção da propriedade do Heap. Executa em $O(log_2 n)$
- **Build-Heap**
  - Produz um heap a partir de um vetor não ordenado. Executa em $O(n)$
- **Heapsort**
  - Procedimento de ordenação local. Executa em $O(nlog_2 n)$

Procedimento MaxHeapify

- Reorganiza heaps
- Supõe que as árvores binárias correspondentes a *Esquerda(i)* e *Direita(i)* são heaps, mas *A[i]* pode ser menor que seus filhos
- Exemplo:



Procedimento MaxHeapify

```
Maxheapify ( A, i )
  e ← Esquerda(i);
  d ← Direita(i);
  maior ← i;
  se  (e ≤ heap_size[A] and A[e] > A[maior]) então
     maior ← e;                          /* filho da esquerda é maior */
  se  (d ≤ heap_size[A] and A[d] > A[maior]) entao
     maior ← d;                          /* filho da direita é maior */
  se  (maior ≠ i) entao
       exchange(A[i] ↔ A[maior]);
       heapify(A, maior);
```

Procedimento Build-Heap

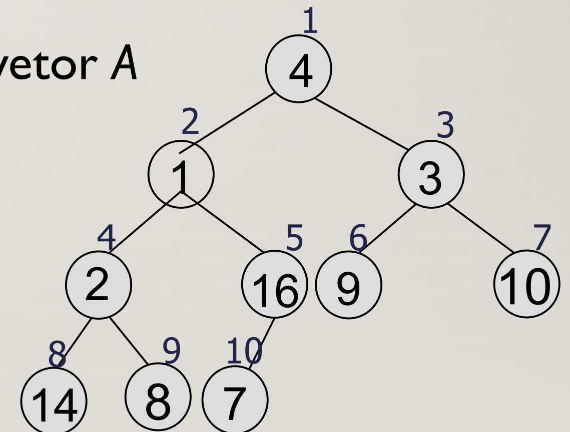
- Utiliza o procedimento MaxHeapify de forma *bottom-up* para transformar um vetor *A[1..n]* em um heap com *n* elementos

- A[(⌊n/2⌋+1)] a A[n] correspondem às folhas da árvore e portanto são heaps de um elemento

- Basta chamar MaxHeapify para os demais elementos do vetor *A*

```
build-heap ( A )
   heap_size[A] ← length[A];
   para  i ← ⌊length[A]/2⌋ até 1 faça
      MaxHeapify(A, i);
```

{4, 1, 3, 2, 16, 9, 10, 14, 8, 7}



Procedimento Heapsort

- Constrói um heap a partir de um vetor de entrada
- Como o maior elemento está localizado na raiz (*A[1]*), este pode ser colocado em sua posição final, trocando-o pelo elemento *A[n]*
- Reduz o tamanho do heap de uma unidade, chama Heapify(A,1) e repete o passo anterior até que o heap tenha tamanho = 2

Procedimento Heapsort

```
heapsort (A)
  build_heap(A);
  para  i ← length[A] até 2 faça
      troca(A[i] ↔ A[1]);
      heap_size[A] ← heap_size[A] –1;
      heapify(A,1);
```

Exercícios

1. Ilustre a operação MaxHeapify sobre o arranjo A= {27,17,3,16,13,10,1,5,7,12,4,8,9,0}, mostre as árvores equivalentes.

2. A sequência {23,17,14,6,13,10,1,5,7,12} é um heap máximo?

3. Construa um procedimento MinHeapify, demonstrando sua utilização.

4. Utilize o procedimento Build-Heap para construir um heap a partir do vetor

   4 1 3 2 16 9 10 14 8 7

5. Ilustre a operação heapsort para o arranjo A = {5,13, 2, 25,7,17,20,8,4}