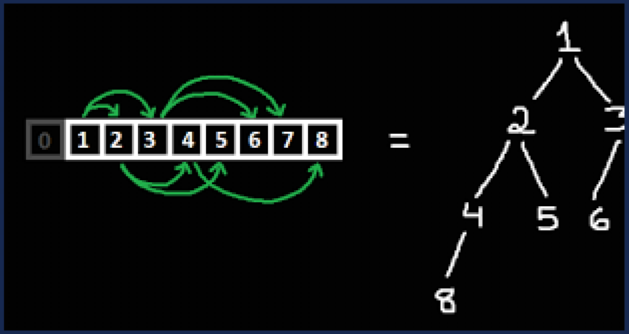
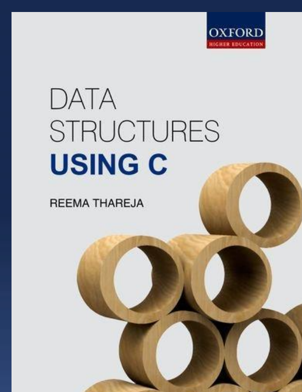


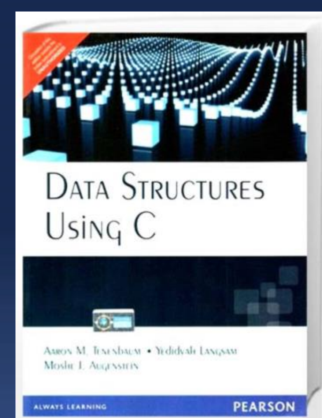
Unidade 14 – Heaps e Filas de Prioridade

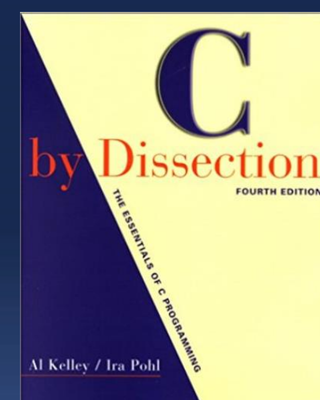
Prof. Aparecido V. de Freitas
Doutor em Engenharia
da Computação pela EPUSP
aparecidovfreitas@gmail.com

Bibliografia

- Algoritmos – Teoria e Prática – **Cormen** – Segunda Edição – Editora Campus, 2002
- Data Structures using C – Oxford University Press – 2014
- Data Structures Using C – A. Tenenbaum, M. Augensem, Y. Langsam, Pearson 1995
- C By Dissection – Kelley, Pohh – Third Edition – Addison Wesley

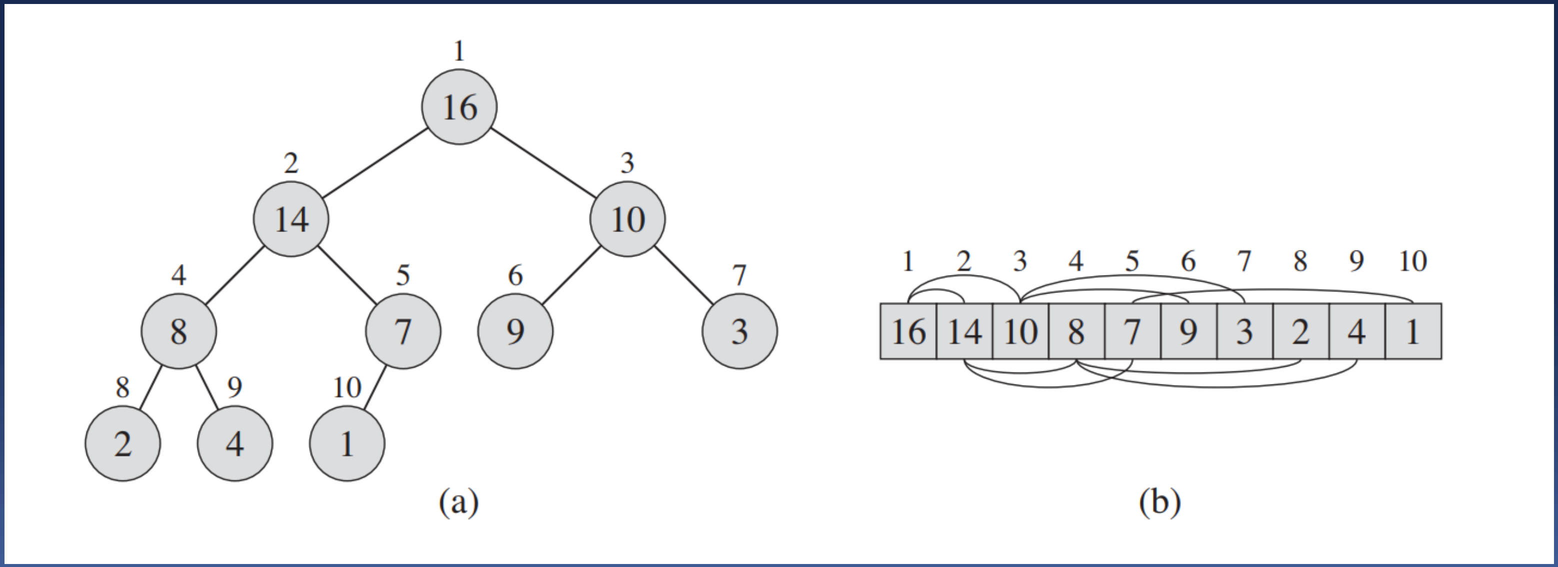






Heaps

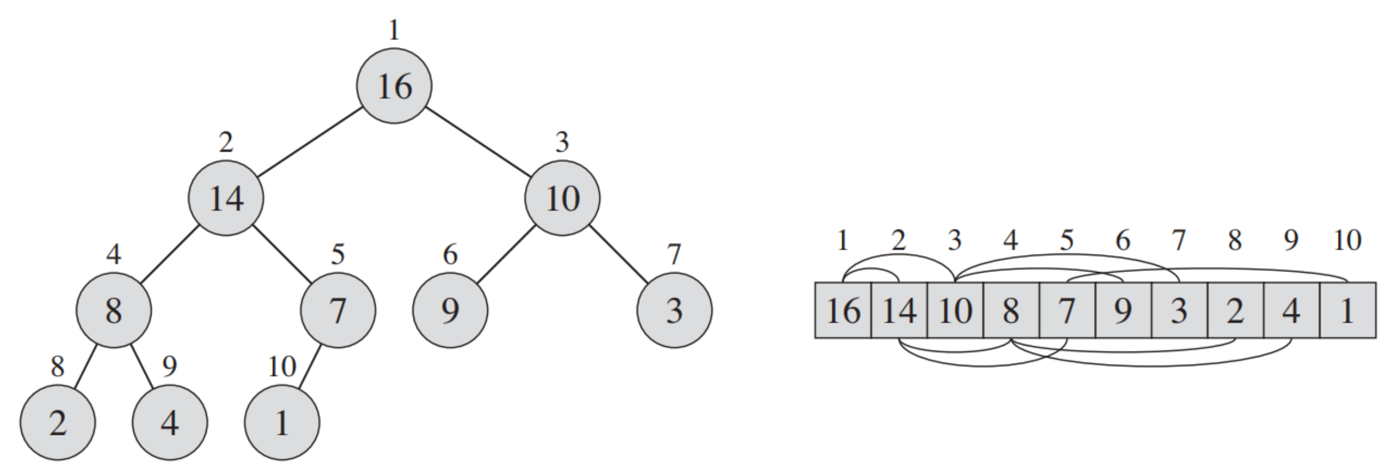
- A estrutura de dados **heap** (binário) pode ser vista como uma **árvore binária praticamente completa**.
- Cada **nó** da árvore corresponde a um elemento do array que armazena o valor do nó;
- A árvore está completamente preenchida em todos os níveis, exceto talvez no nível mais baixo, que é preenchido à esquerda até certo ponto;
- **Heaps** são usualmente implementados por meio de arrays.



Heaps - Propriedades

É uma estrutura de dados que pode ser visualizada como uma árvore binária quase completa.
Cada nó da árvore é ocupado por um elemento e temos as seguintes propriedades:

- A árvore é completa até o penúltimo nível
- No último nível as folhas estão o mais à esquerda possível
- O conteúdo de um nó é maior ou igual ao conteúdo dos nós na subárvore enraizada nele (max-heap)



(a) (b)

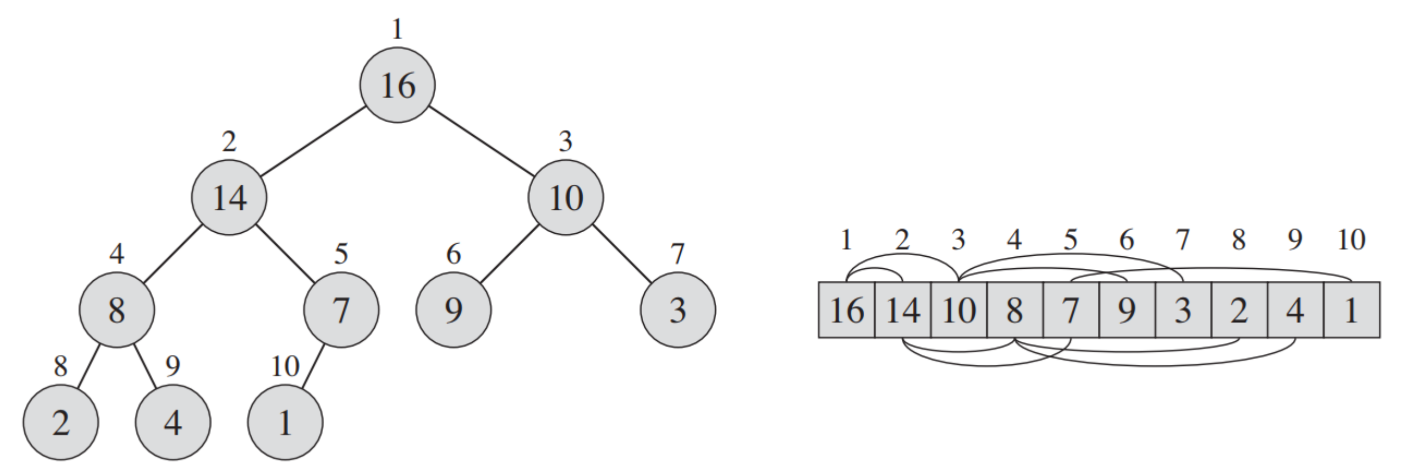
- **Observação**: Se o conteúdo de um nó é menor ou igual ao conteúdo dos nós da subárvore enraizada por ele, tem-se um (min-heap).

Heaps – Observação

É uma estrutura de dados que pode ser visualizada como uma árvore binária quase completa.
Cada nó da árvore é ocupado por um elemento e temos as seguintes propriedades:

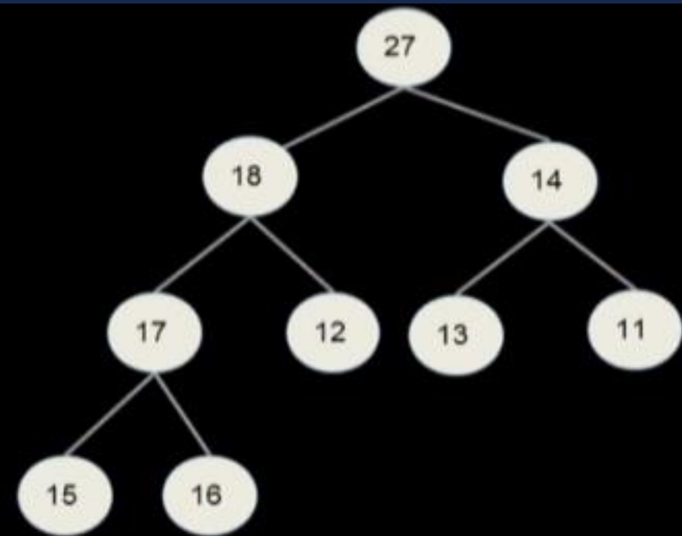
- A árvore é completa até o penúltimo nível
- No último nível as folhas estão o mais à esquerda possível
- O conteúdo de um nó é maior ou igual ao conteúdo dos nós na subárvore enraizada nele (max-heap)

- **Observação**: Essas propriedades garantem que um **heap** pode ser implementado em um array A[1..m].



(a) (b)

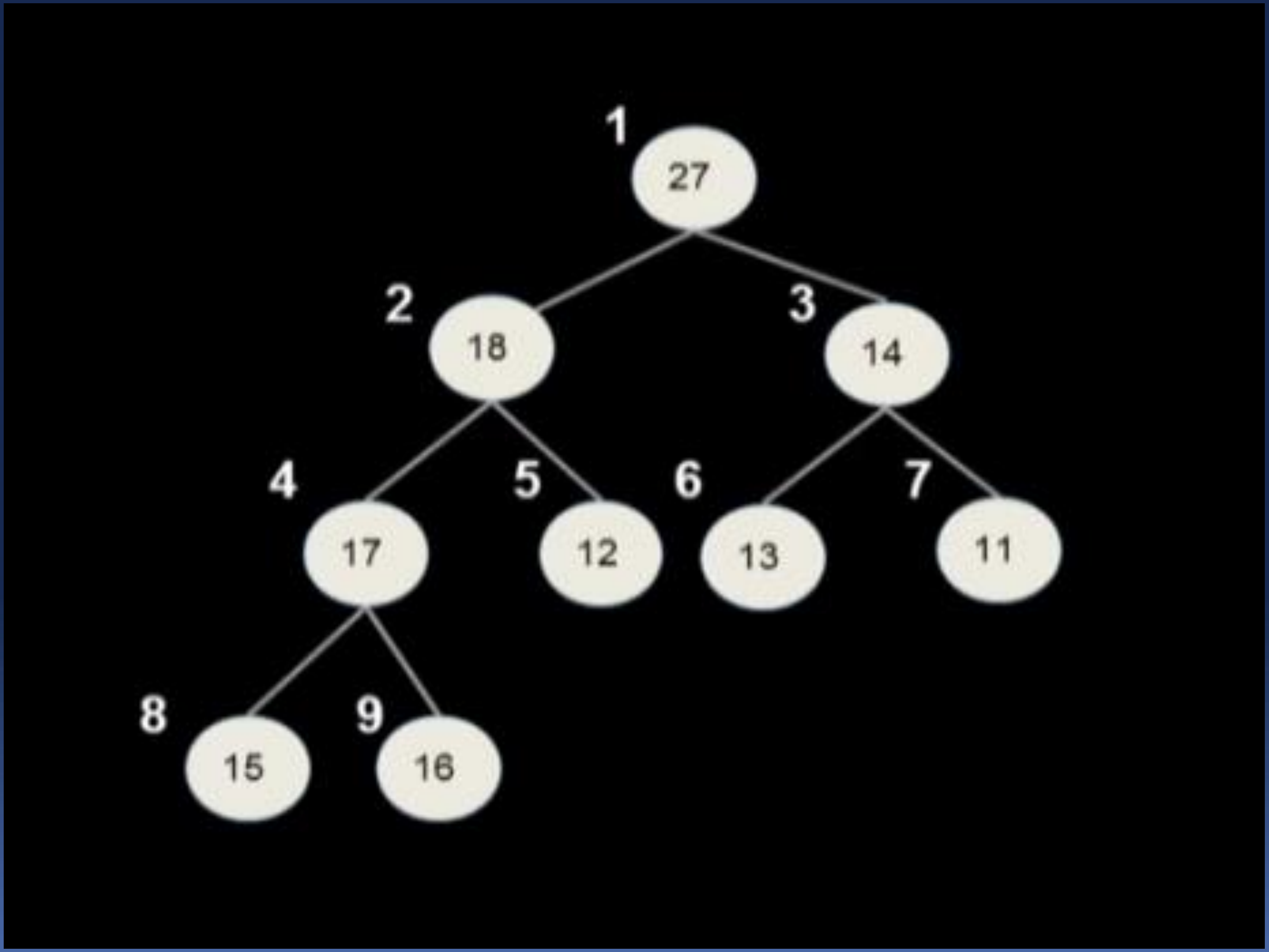
Heap - Exemplo



- A árvore é completa até o penúltimo nível
- No último nível as folhas estão o mais à esquerda possível
- O conteúdo de um nó é **maior ou igual** ao conteúdo dos nós na subárvore enraizada nele (**max-heap**)

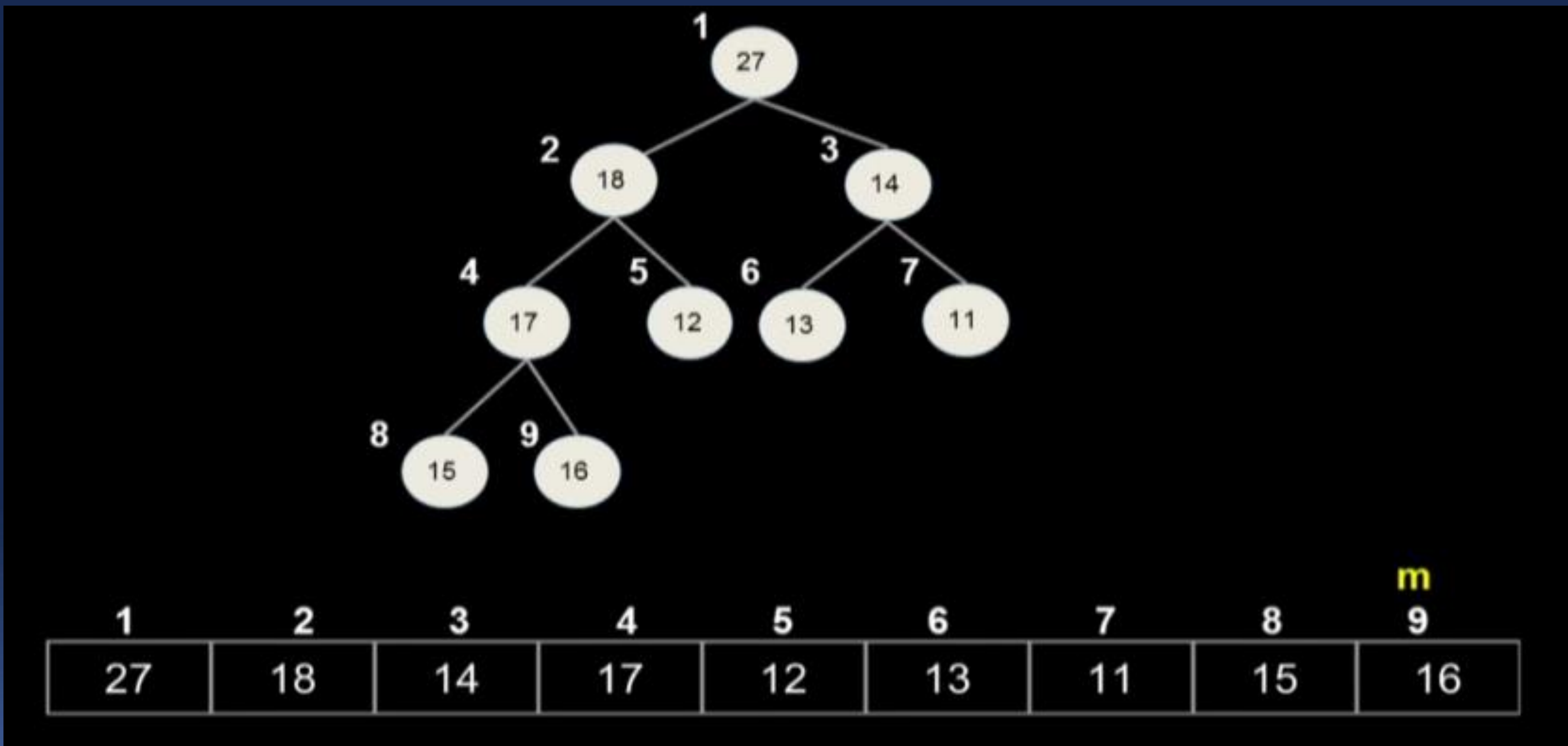
Heap – Implementação

- A implementação do heap, pelas suas propriedades, **não** exige que se crie uma árvore;
- Pode-se assim, implementar um **heap** por meio de um **array**;



Heap - Implementação

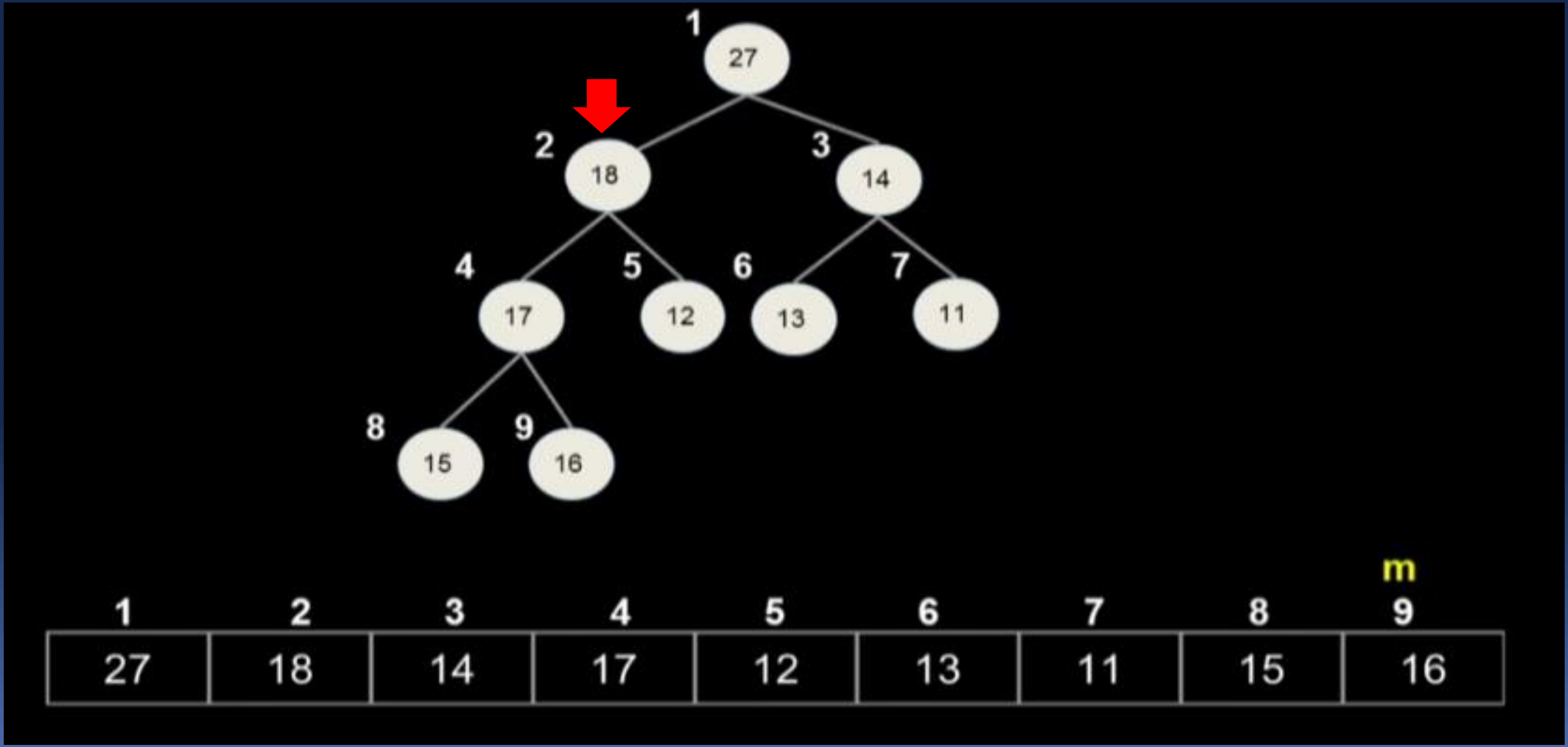
- Assim, a **implementação** de um heap é feita por meio de um array.



Heap - Implementação

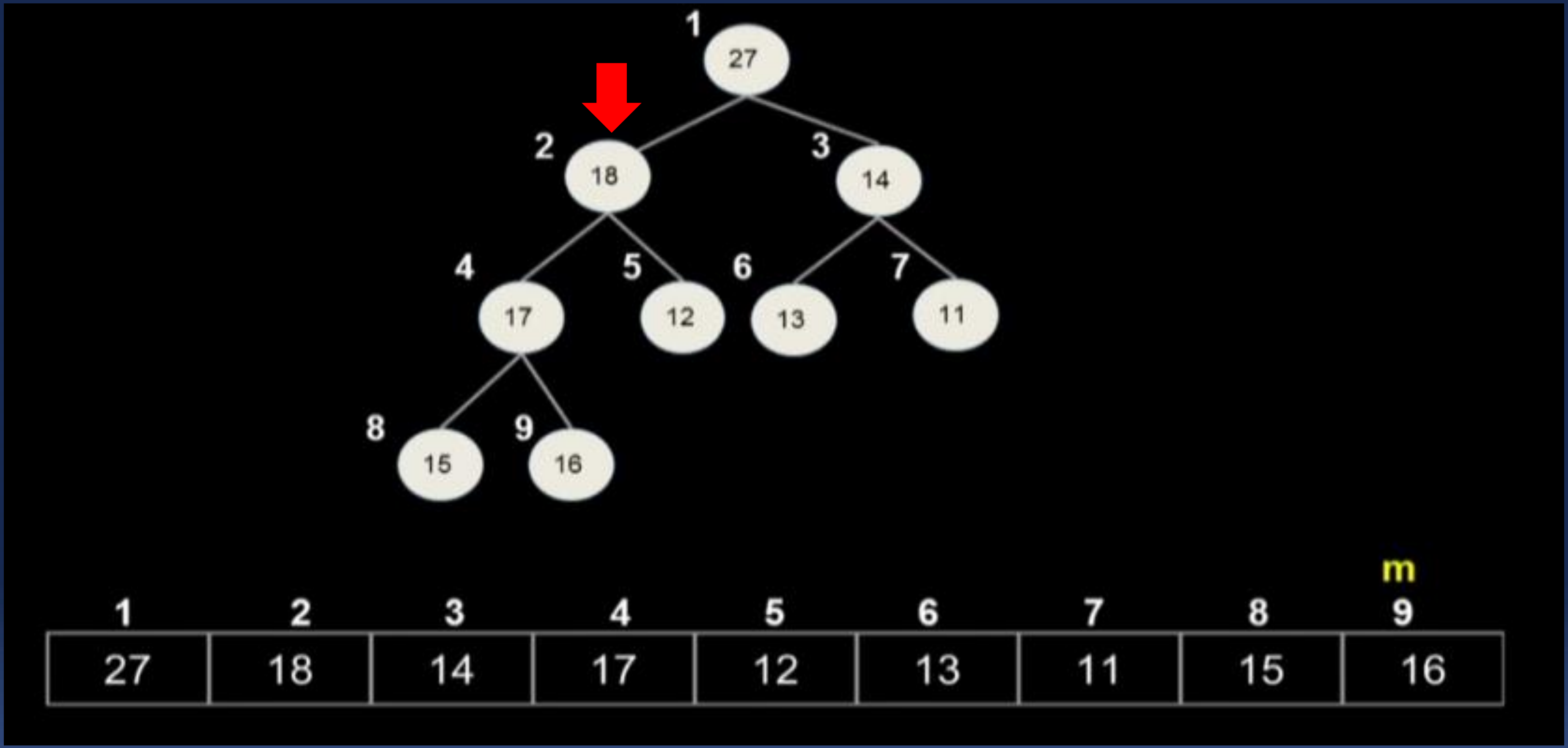
- Dado um elemento do **heap**, como determinar os valores correspondentes a seus filhos?

- Por exemplo, quem são os filhos do nó de **índice 2**?



Heap - Implementação

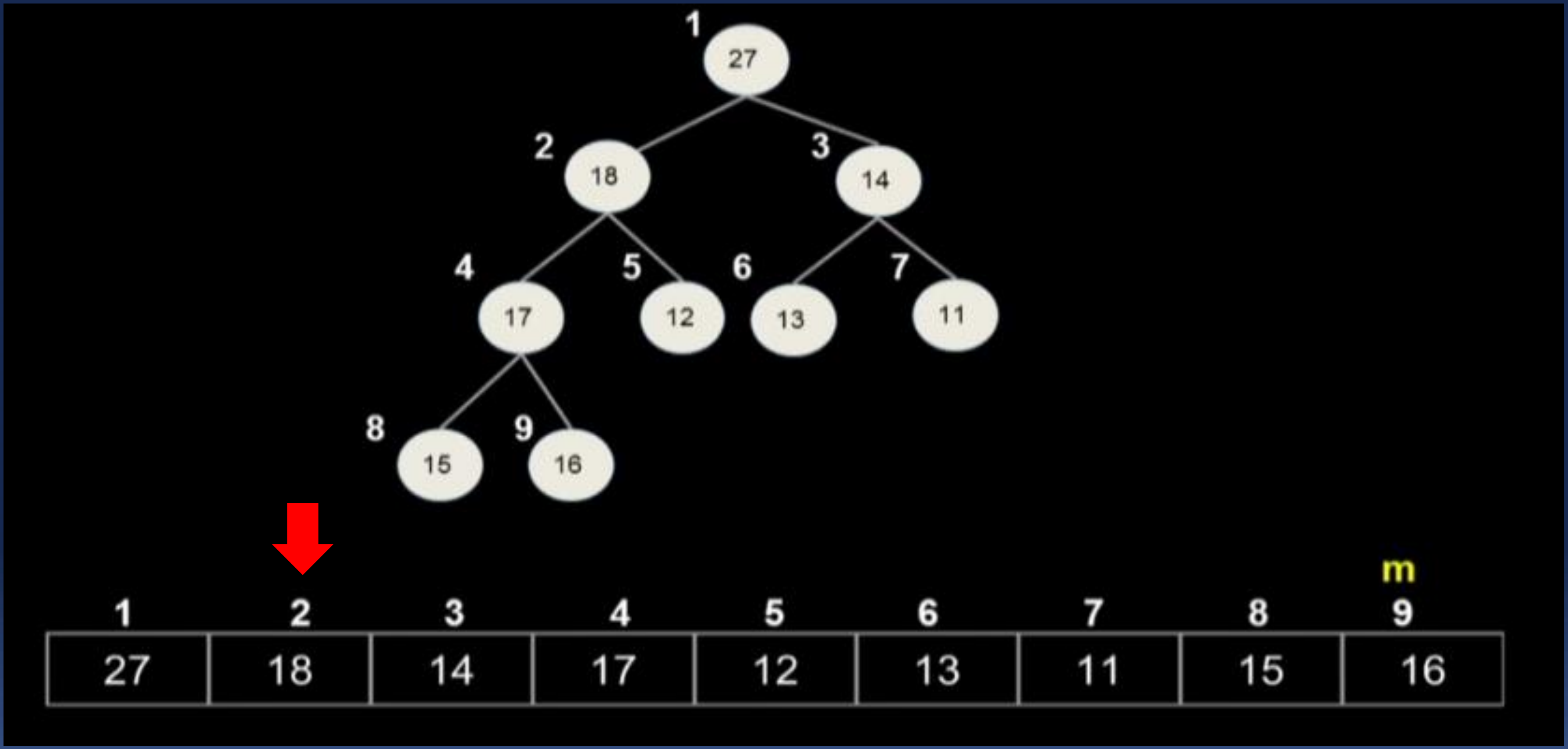
- Por exemplo, quem são os filhos do nó de **índice 2**?



- Observando a árvore correspondente ao heap, é fácil verificar-se que os filhos de 18 são 17 e 12.
- Mas, como determiná-los no array de implementação?

Heap - Implementação

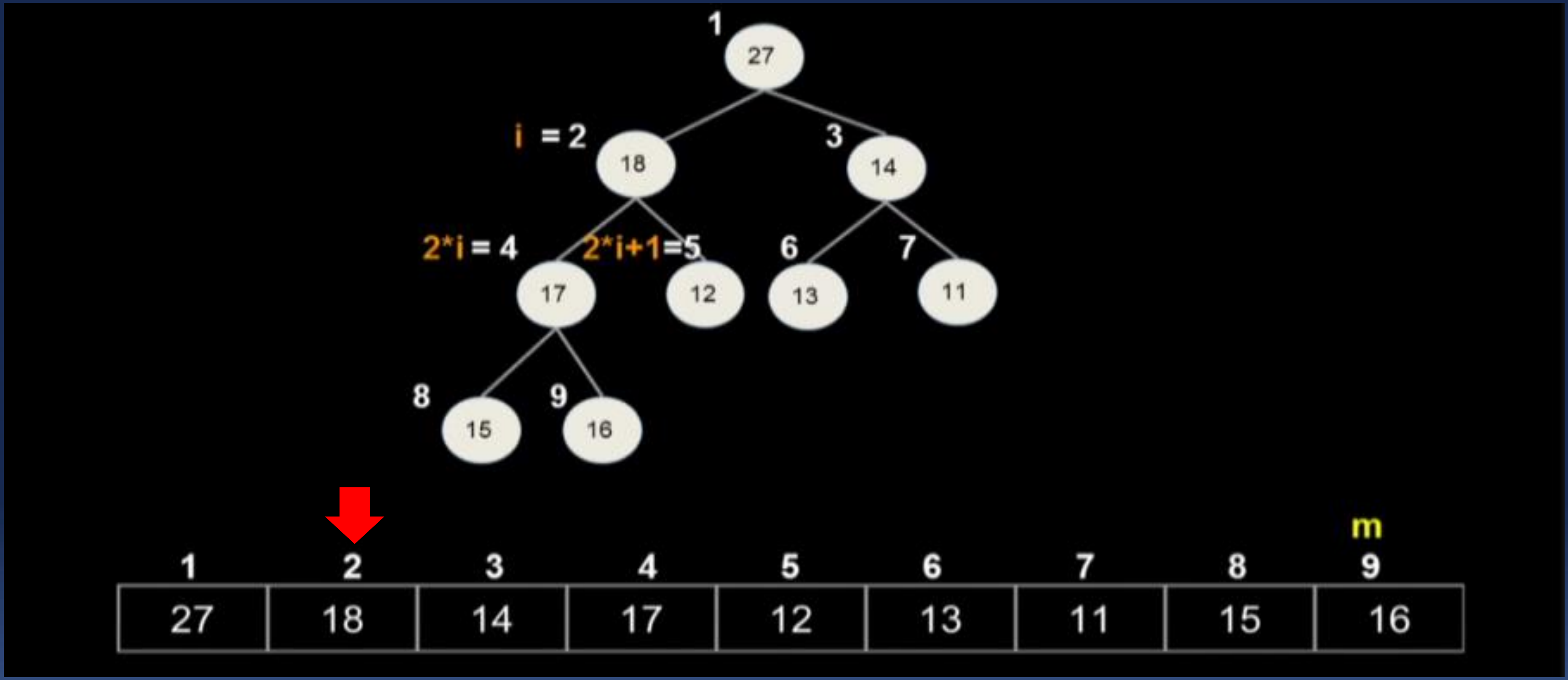
- Por exemplo, quem são os filhos do nó de **índice 2**?



- Mas, como determiná-los no array de implementação?

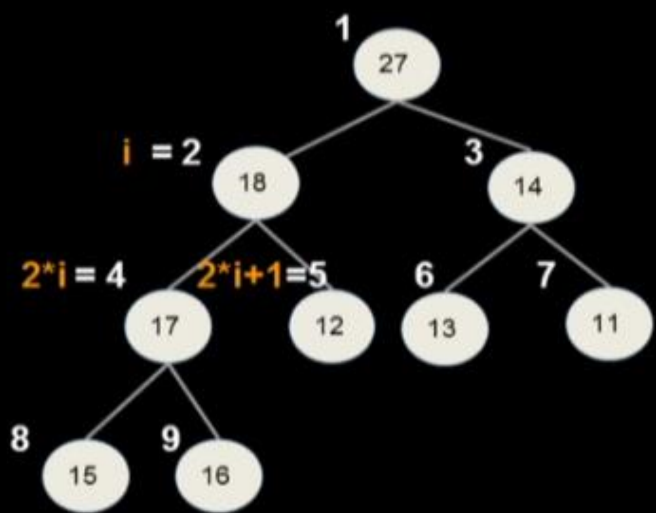
Heap - Implementação

- Por exemplo, quem são os filhos do nó de **índice 2**?



- No array de implementação, o filho a esquerda no nó com endereço **i**, está na posição **2*i**;
- No array de implementação, o filho a direita do nó com endereço **i**, está na posição **2*i + 1**.

Heap - Implementação



Para qualquer nó i:
2i é o filho esquerdo de i
2i+1 é o filho direito de i
Quem é o pai de i?

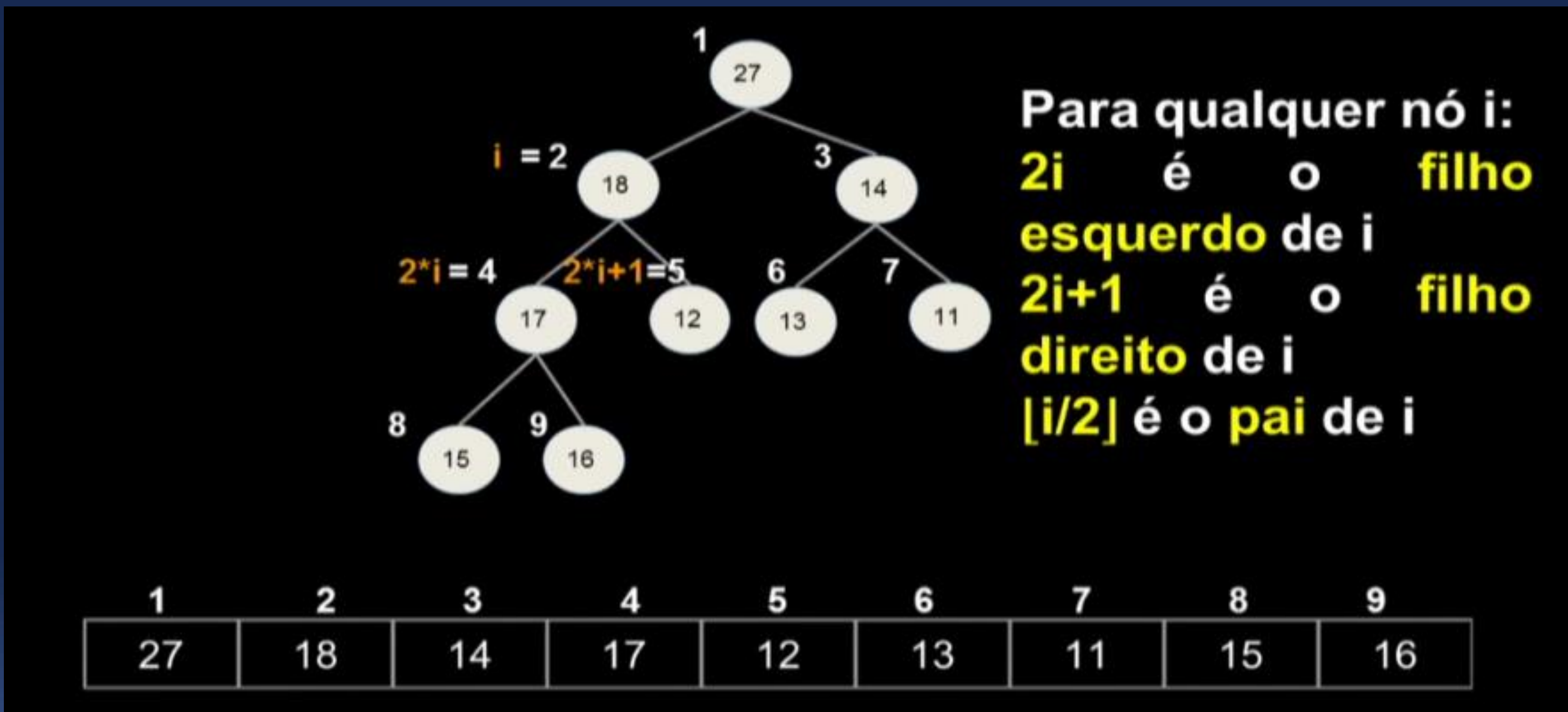
1	2	3	4	5	6	7	8	9
27	18	14	17	12	13	11	15	16

Heap - Implementação

- Dado um nó em um endereço i, qual o endereço de seu pai?

Heap - Implementação

- Dado um nó em um endereço **i**, qual o endereço de seu pai?



- No array de implementação, o pai de um nó em um endereço **i** está na posição **|_ i/2 _|** .

Heap - Propriedades

PARENT(i)

return $\lfloor i/2 \rfloor$

LEFT(i)

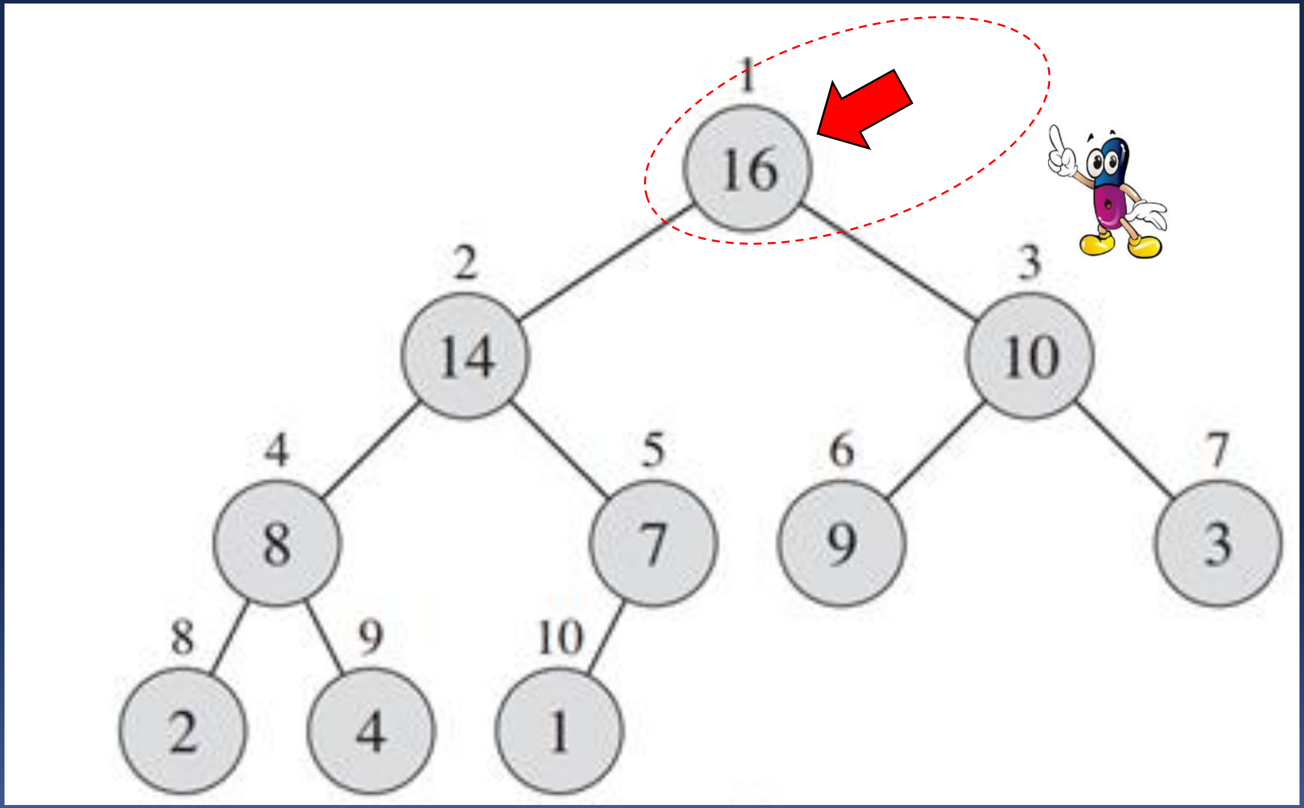
return $2i$

RIGHT(i)

return $2i + 1$

Heap - Observação

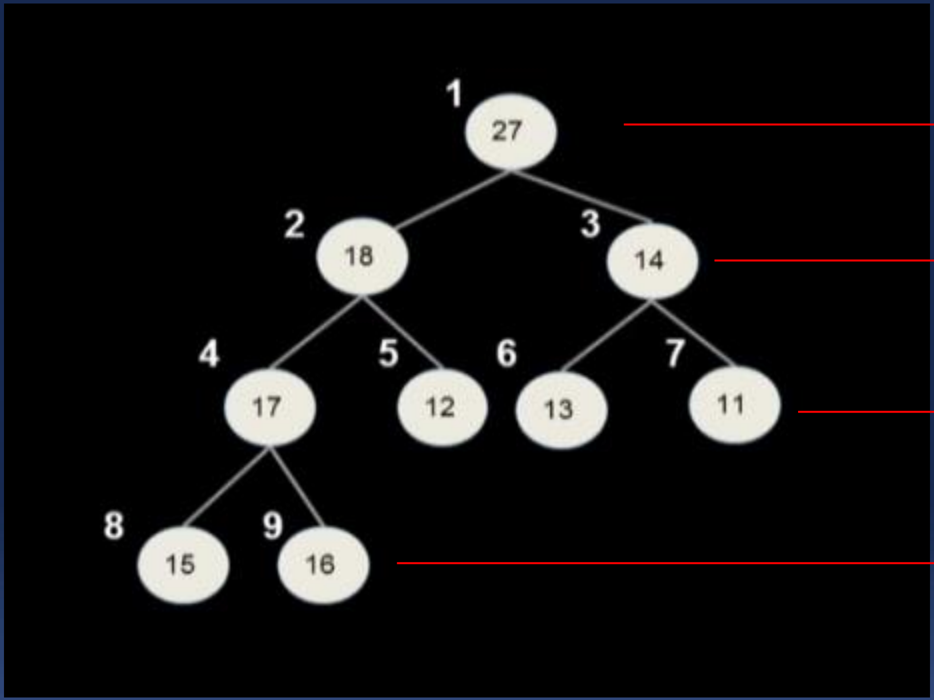
- Em um **max-heap**, o maior elemento da árvore, por definição, está na **raiz**;



Níveis de um Heap

- Cada nível **p**, tem exatamente 2^p nós, exceto talvez no último nível;

- Por exemplo, no heap abaixo, o nível 2 tem exatamente 4 nós.



Nível 0

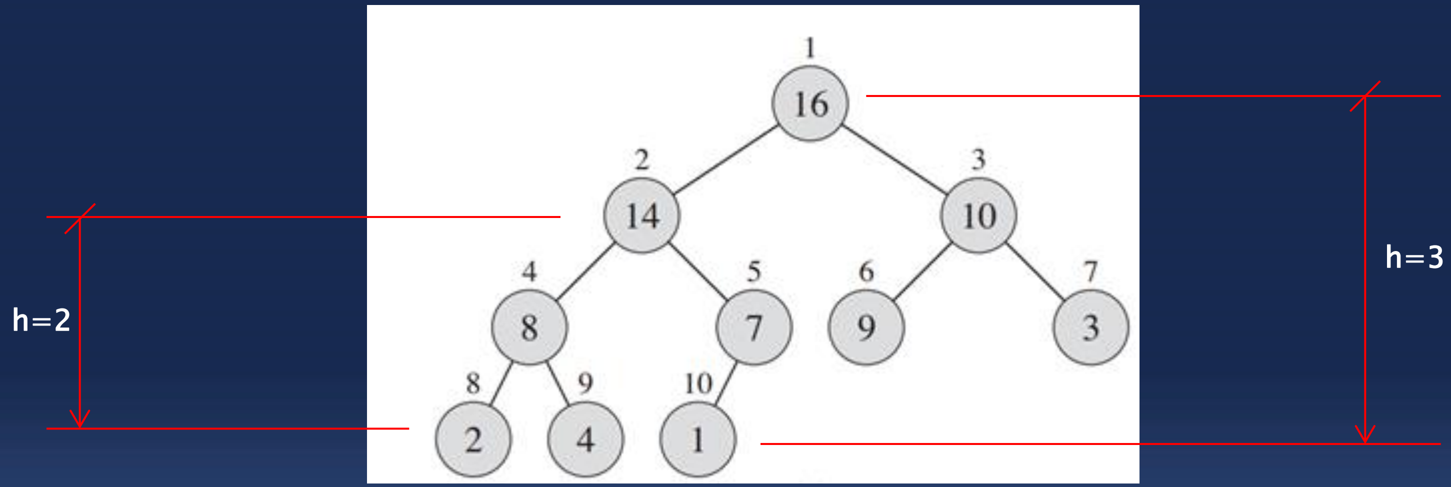
Nível 1

Nível 2

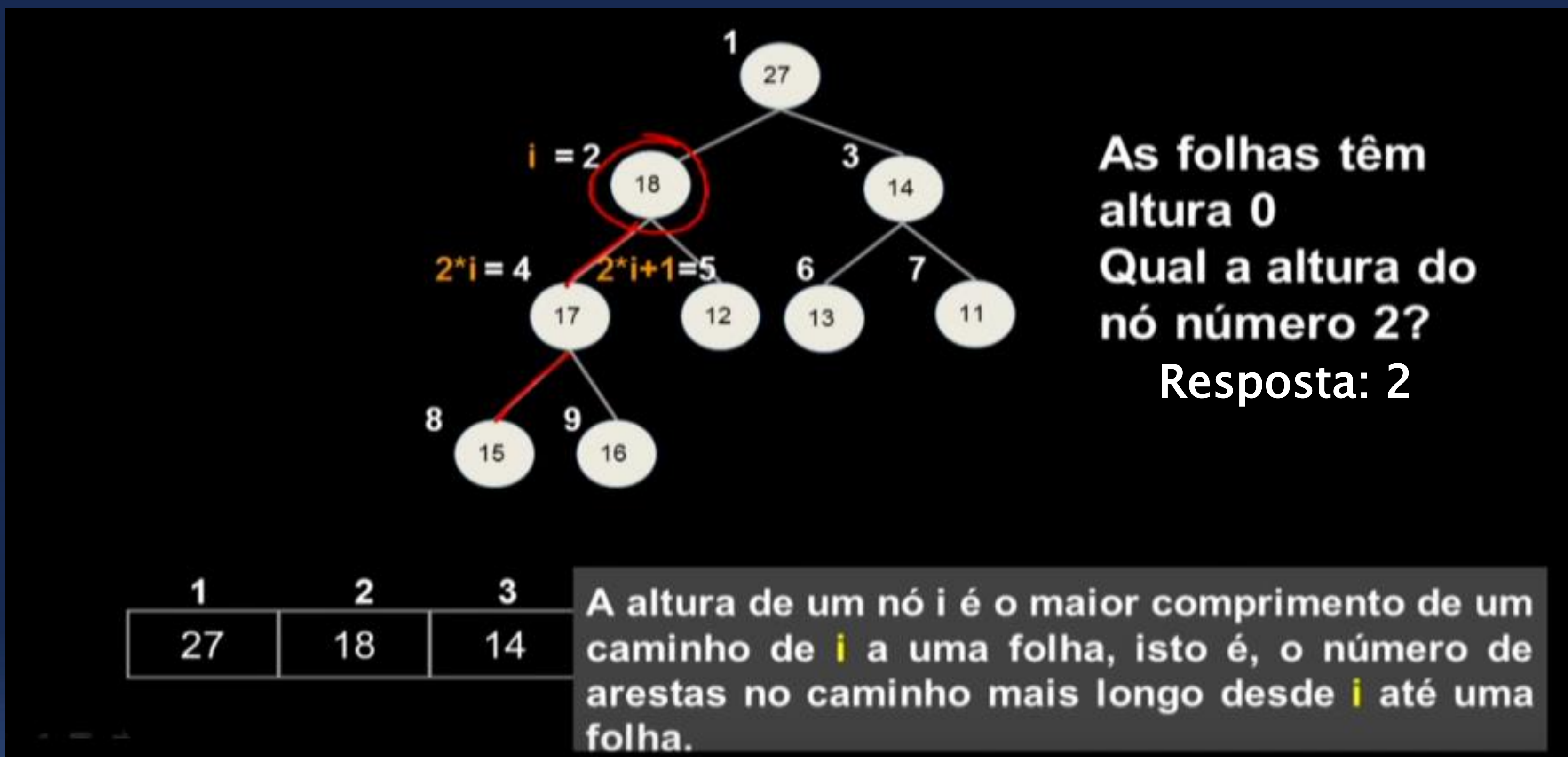
Nível 3

Altura de nós do Heap

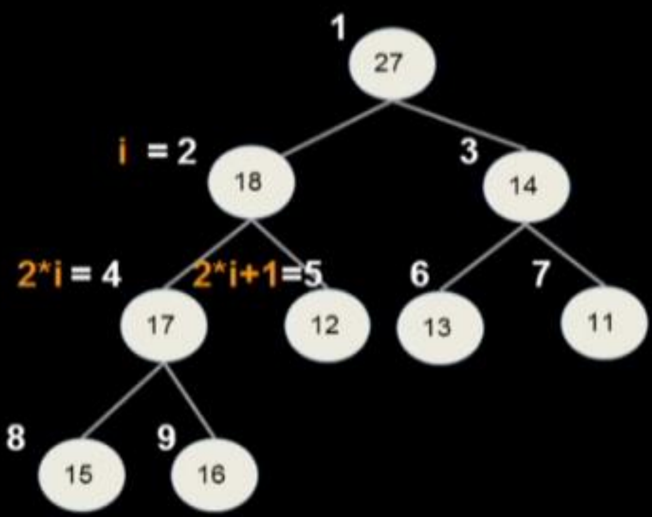
A altura de um nó i é o maior comprimento de um caminho de i até uma folha, isto é, o número de arestas no caminho mais longo desde i até uma folha.



Altura de nós do Heap



Altura de nós do Heap



Pode ser demonstrado que a altura de um nó i é:

$$h = \lfloor lg(m/i) \rfloor$$

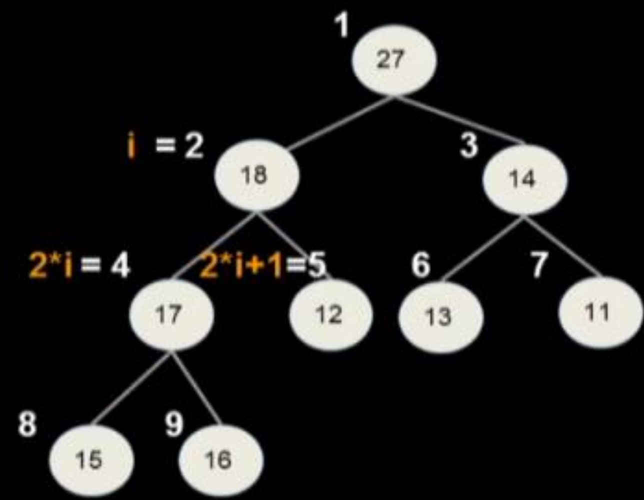
A altura da raiz é:

$$h = \lfloor lg(m) \rfloor$$

1	2	3
27	18	14

A altura de um nó i é o maior comprimento de um caminho de i a uma folha, isto é, o número de arestas no caminho mais longo desde i até uma folha.

Altura de uma árvore



Pode ser demonstrado que a altura de um nó i é:

$$h = \lfloor lg(m/i) \rfloor$$

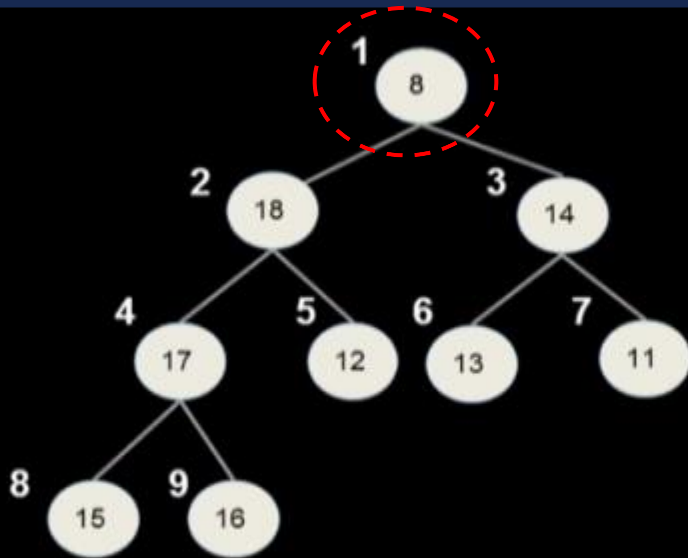
A altura da árvore é:

$$h = \lfloor lg(m) \rfloor$$

1	2	3
27	18	14

A altura de um nó i é o maior comprimento de um caminho de i a uma folha, isto é, o número de arestas no caminho mais longo desde i até uma folha.

Manutenção do Heap

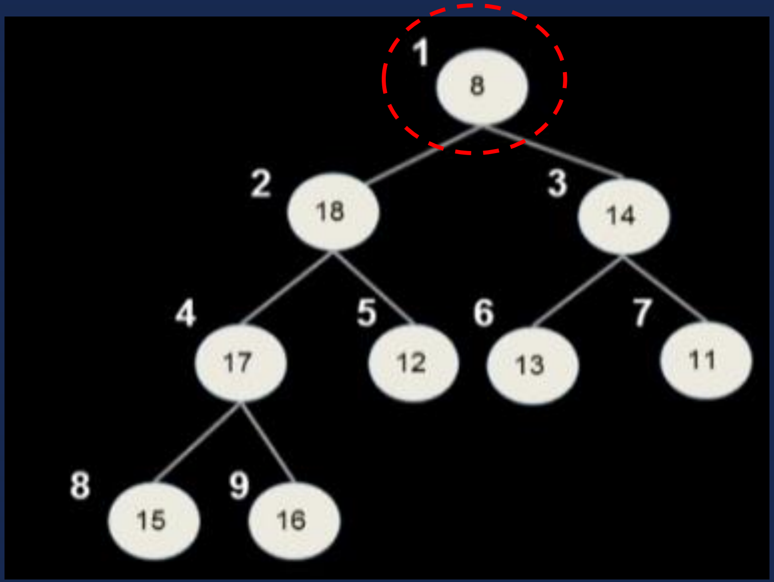


É quase um heap.

Apenas o elemento da raiz não satisfaz a propriedade de heap.

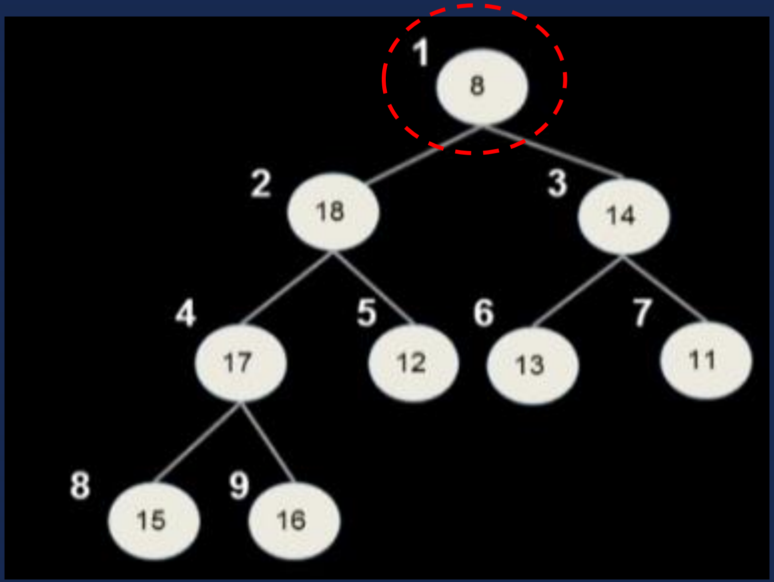
1	2	3	4	5	6	7	8	9
8	18	14	17	12	13	11	15	16

Manutenção do Heap



O que se deve fazer para que a árvore se torne um heap?

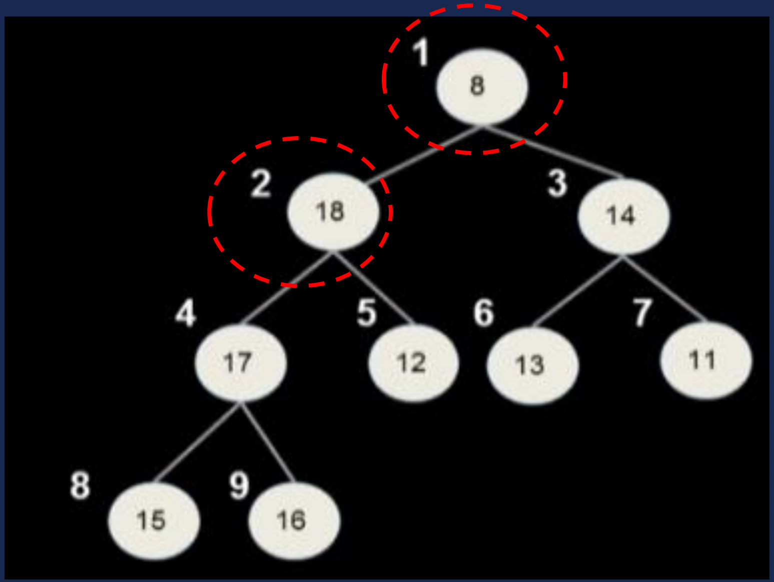
Manutenção do Heap



- Uma ideia seria mover o nó com o valor **8** para baixo, até deixá-lo numa posição conveniente que atenda a propriedade do heap;

1	2	3	4	5	6	7	8	9
8	18	14	17	12	13	11	15	16

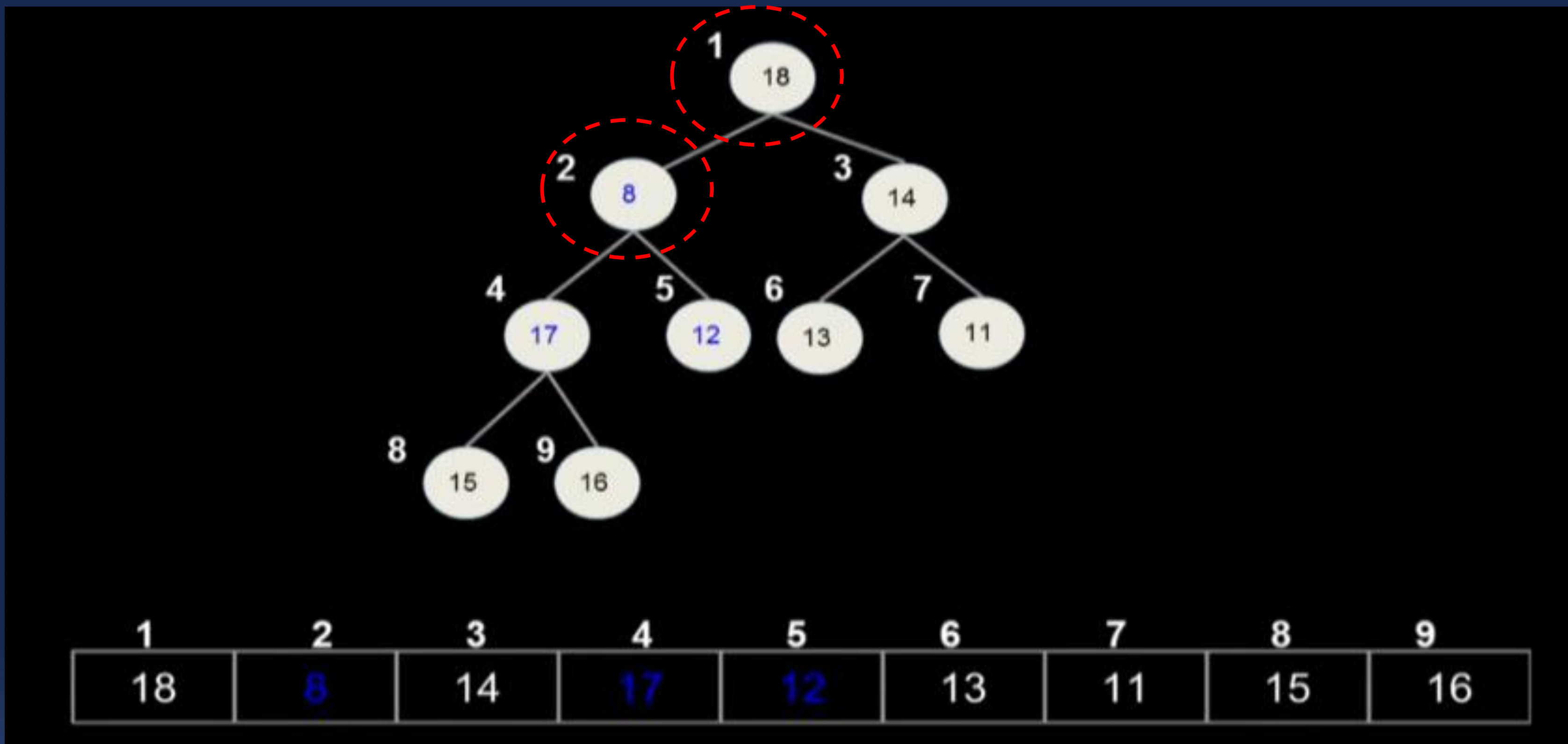
Manutenção do Heap



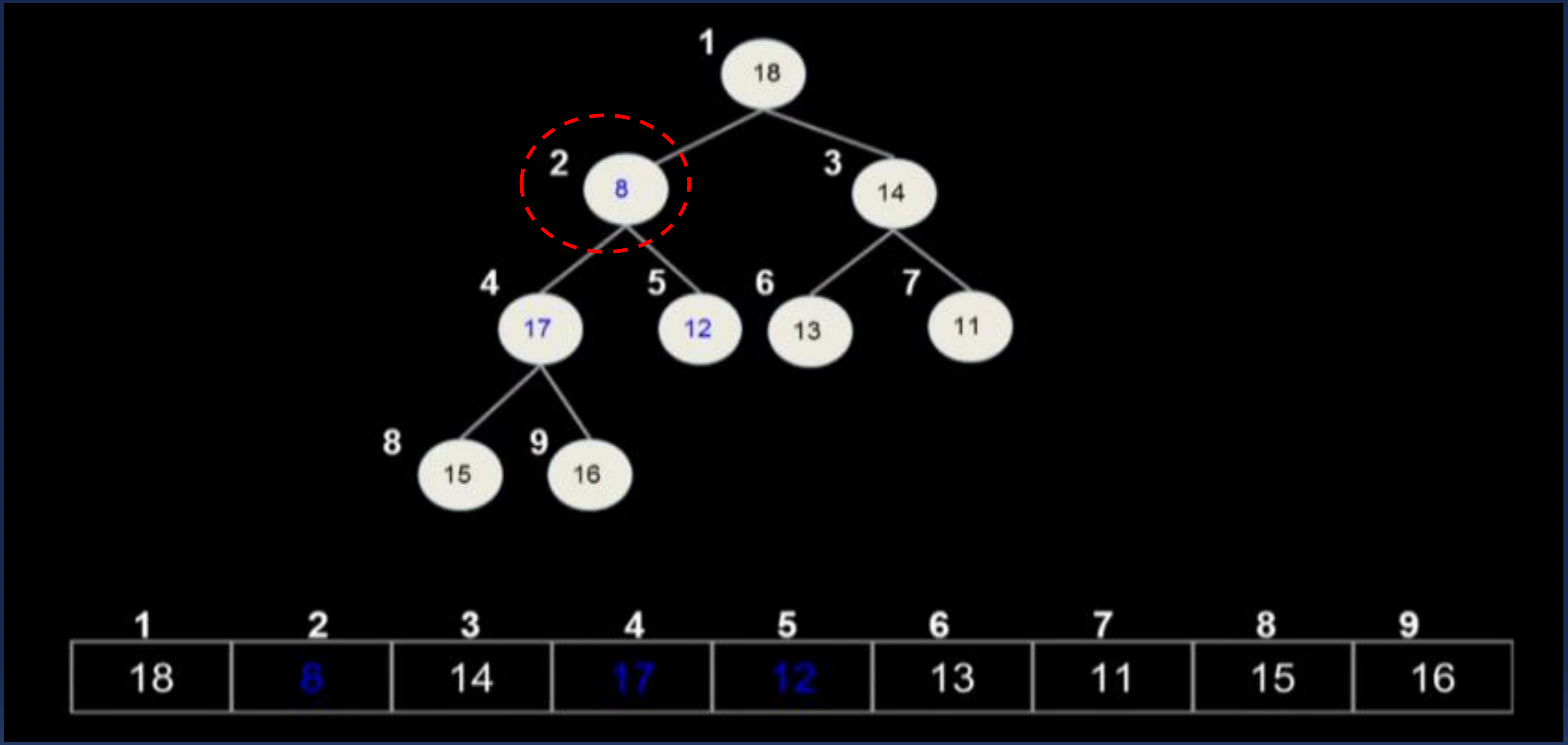
- Primeiramente, vamos comparar o 8 com os seus filhos;
- Vamos trabalhar com o maior dos filhos, no caso o nó com o valor 18;
- Proceda-se à troca do 8 com o 18;

1	2	3	4	5	6	7	8	9
8	18	14	17	12	13	11	15	16

Manutenção do Heap

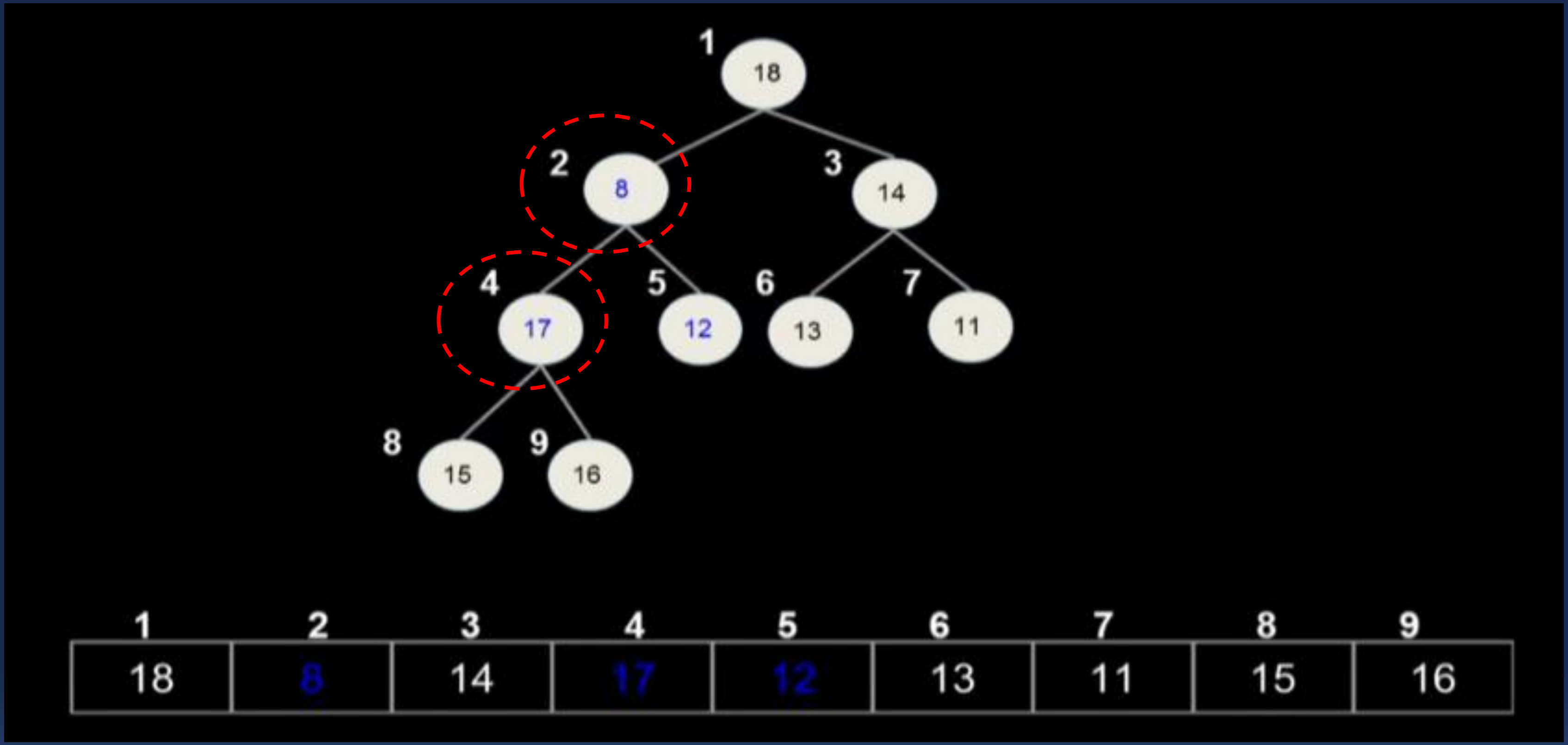


Manutenção do Heap



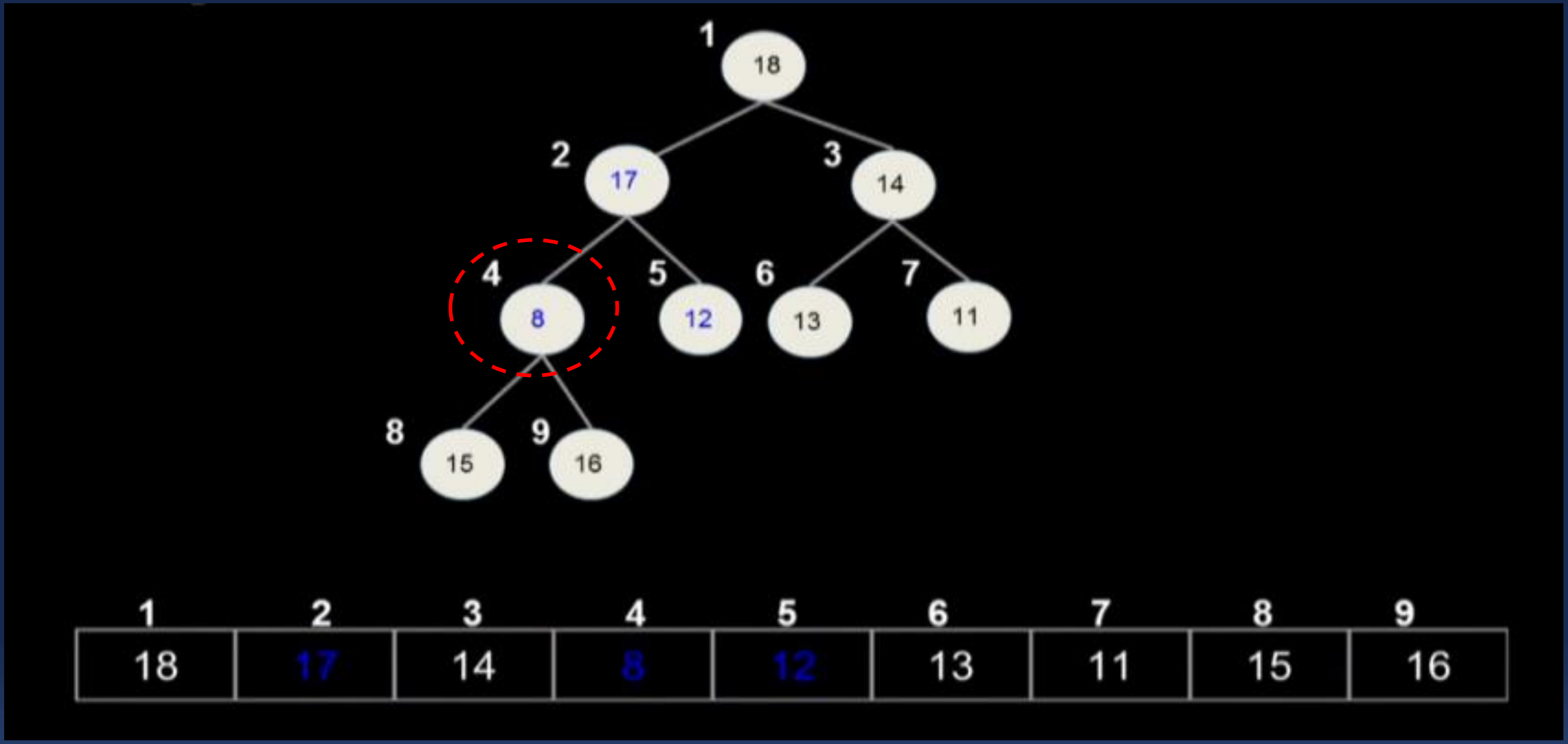
- Fazemos agora o mesmo com os filhos do nó 8;
- Compara-se o nó 8 com os seus filhos 17 e 12

Manutenção do Heap



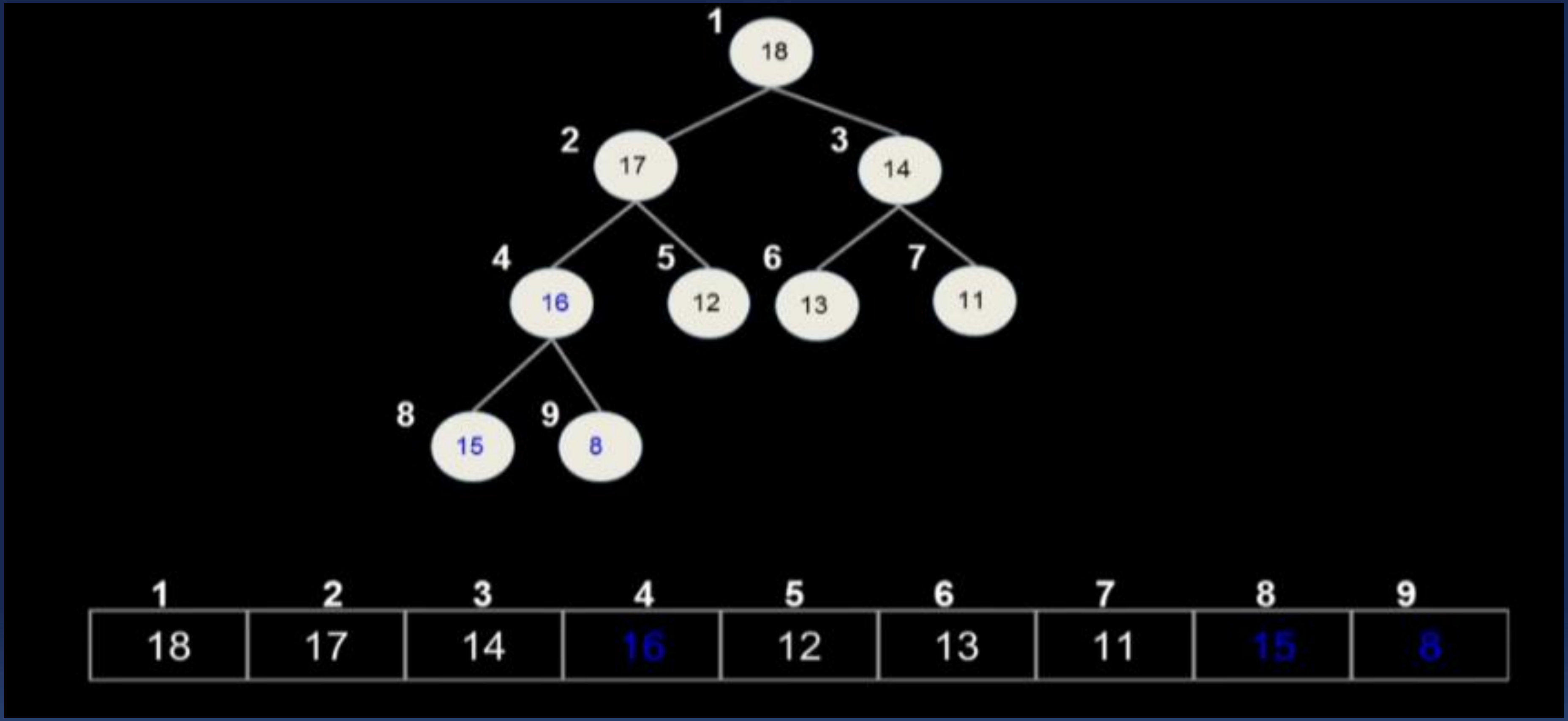
- Proceda-se à troca do nó 8 com o maior dos filhos que é o 17

Manutenção do Heap



- O processo continua até a árvore se tornar um heap (heapfy);

Manutenção do Heap



- A árvore agora é um **heap**;

Algoritmo Max-Heapfy

Manutenção da propriedade de heap

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

Recebe o vetor A [1 ...m] e o índice i, tal que as árvores com raízes nos filhos esquerdo e direito do nó i são max-heaps.

Algoritmo Max-Heapfy

Manutenção da propriedade de heap

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

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

$T(h) = O(h)$

Como: a altura de um nó i é:

$h = \lfloor lg(m/i) \rfloor$

$T(h) = O(h) = O(\lfloor lg(m/i) \rfloor)$

$= O(lg(m/i)) = O(lg(m))$

Construção de um Max Heap

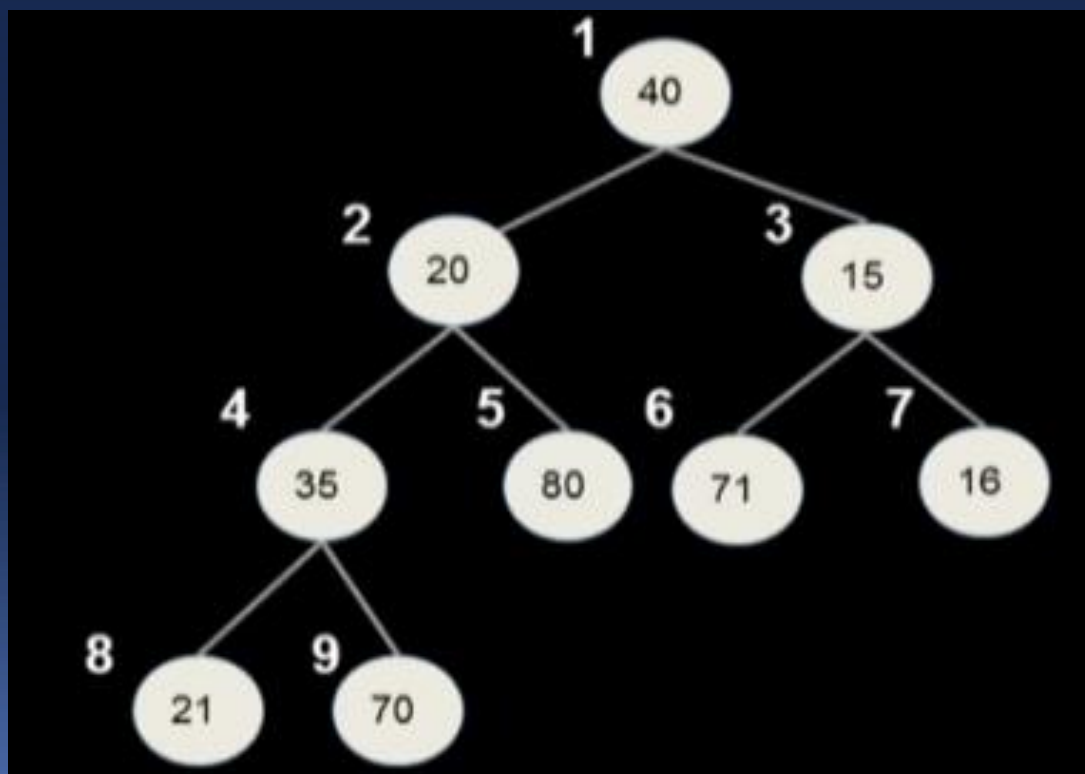
- Dado um array de valores, como construir um **max heap** a partir deste array?

1	2	3	4	5	6	7	8	9
40	20	15	35	80	71	16	21	70

Construção de um Max Heap

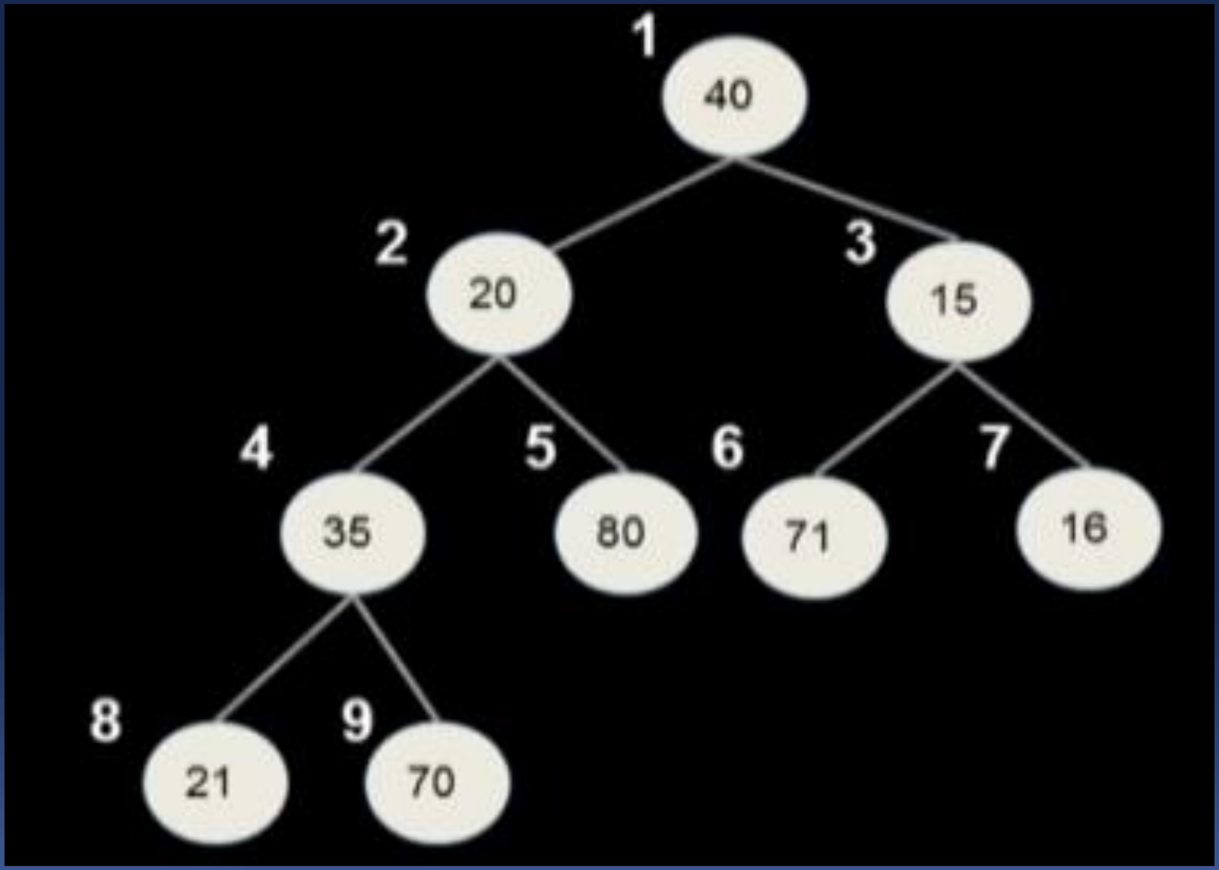
1	2	3	4	5	6	7	8	9
40	20	15	35	80	71	16	21	70

- Árvore Inicial



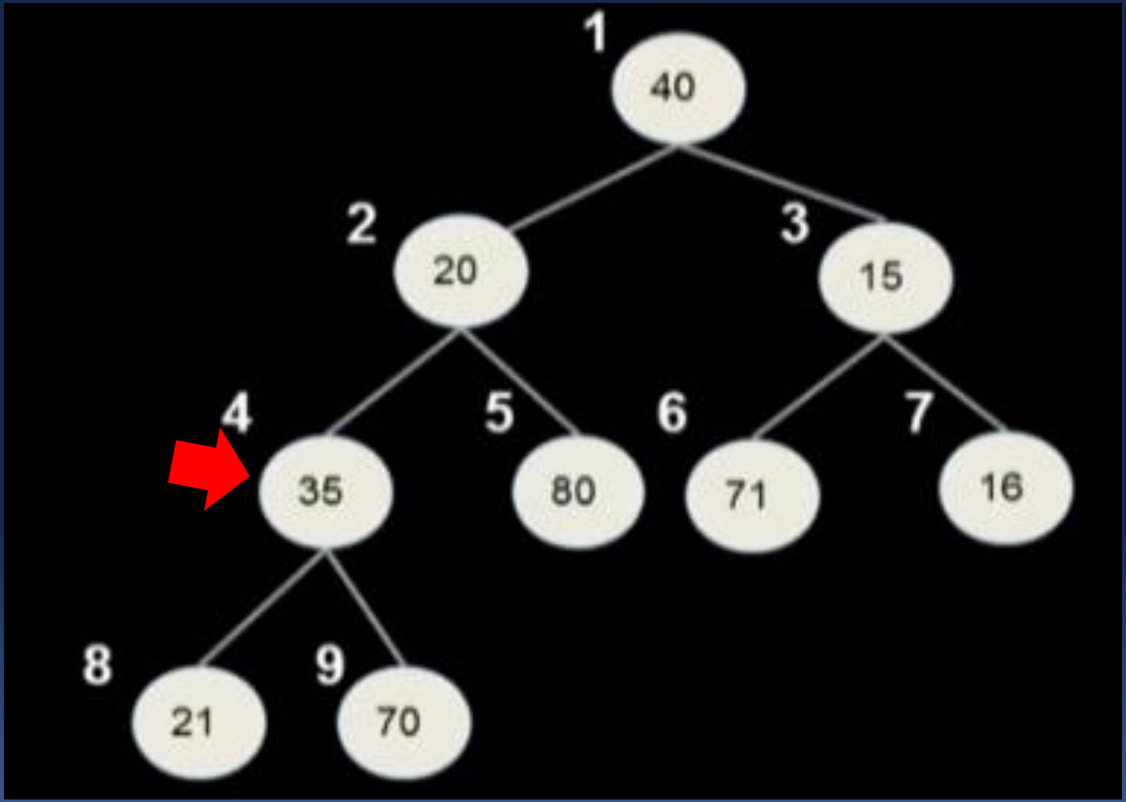
Construção de um Max Heap

- Os nós folhas cumprem a propriedade de heap pois eles não têm filhos!



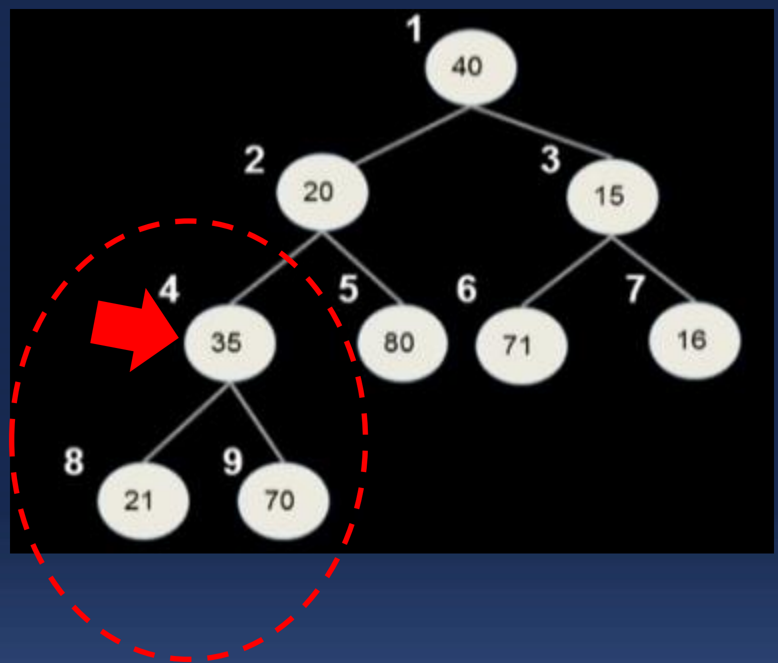
Construção de um Max Heap

- O nó **35** não cumpre a propriedade de heap;



Construção de um Max Heap

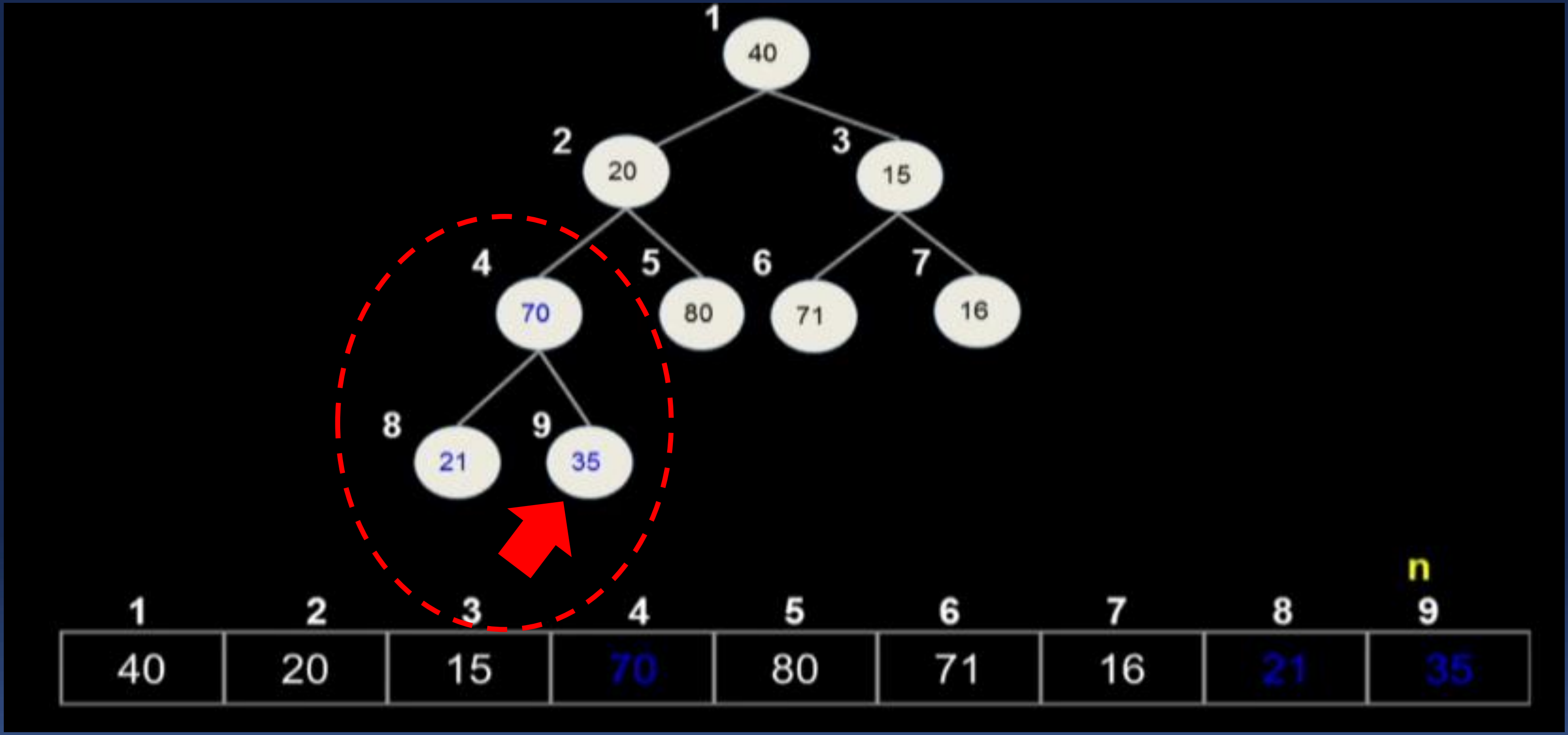
- O nó **35** não cumpre a propriedade de heap;
- Vamos trocá-lo com o nó **70** que é o maior dos filhos;



QualitSys

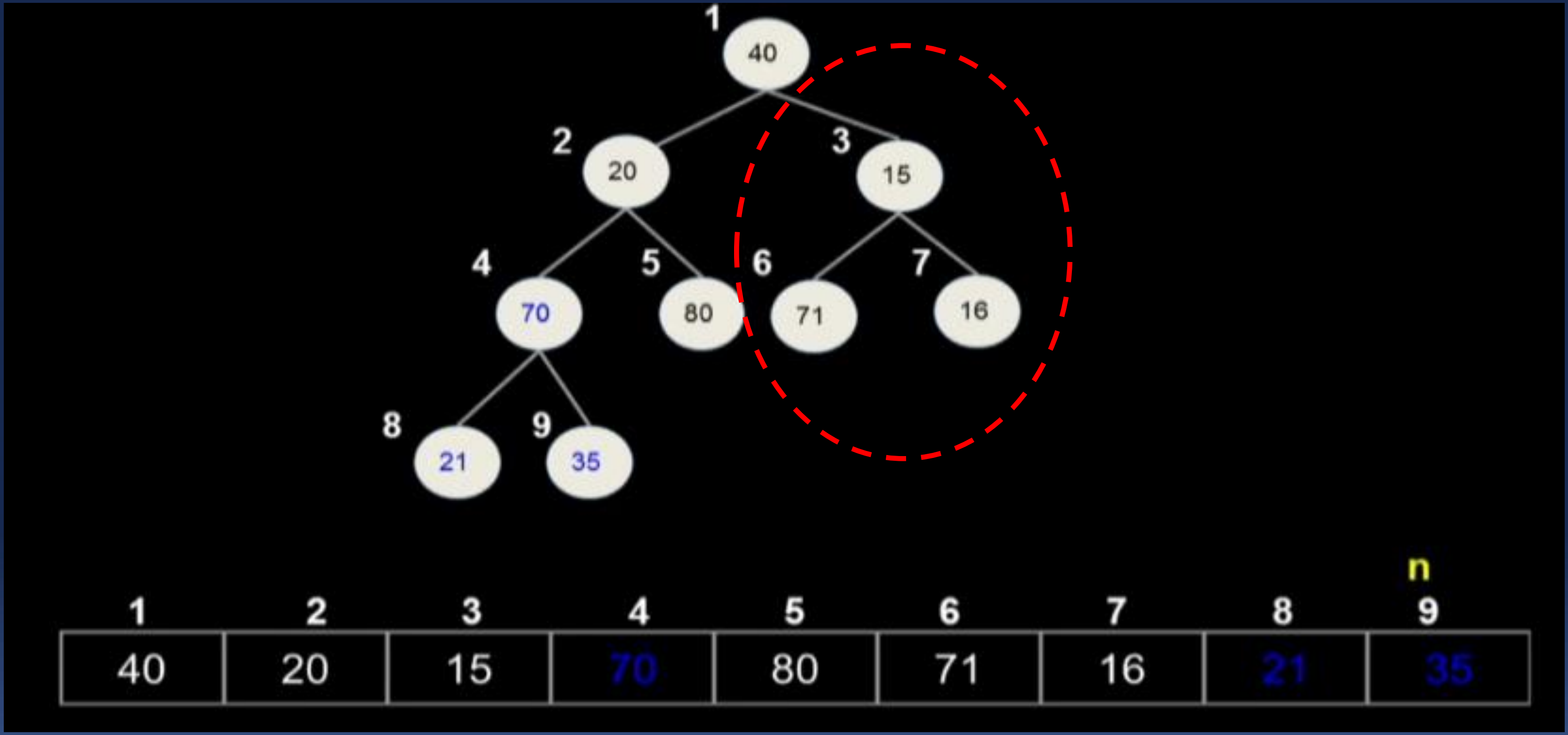
Construção de um Max Heap

- O nó 35 não cumpre a propriedade de heap;
- Vamos trocá-lo com o nó 70 que é o maior dos filhos;



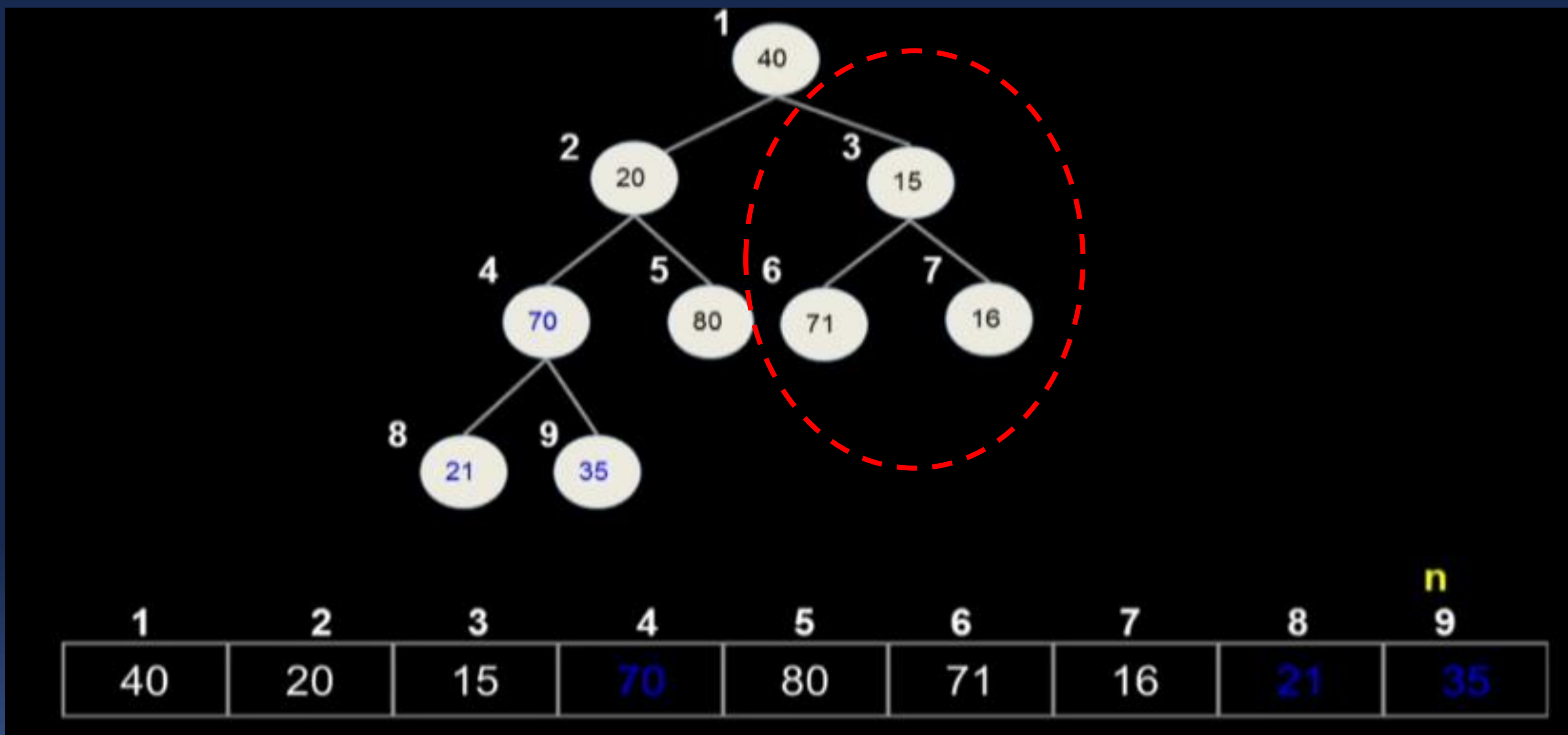
Construção de um Max Heap

- O processo continua com o nó **15**;



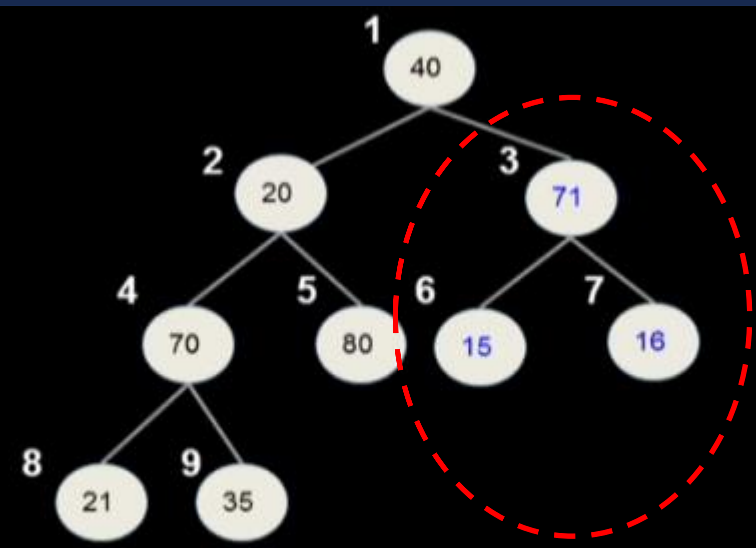
Construção de um Max Heap

- O processo continua com o nó **15**;
- O nó com valor **15** é trocado com o nó **71.**



Construção de um Max Heap

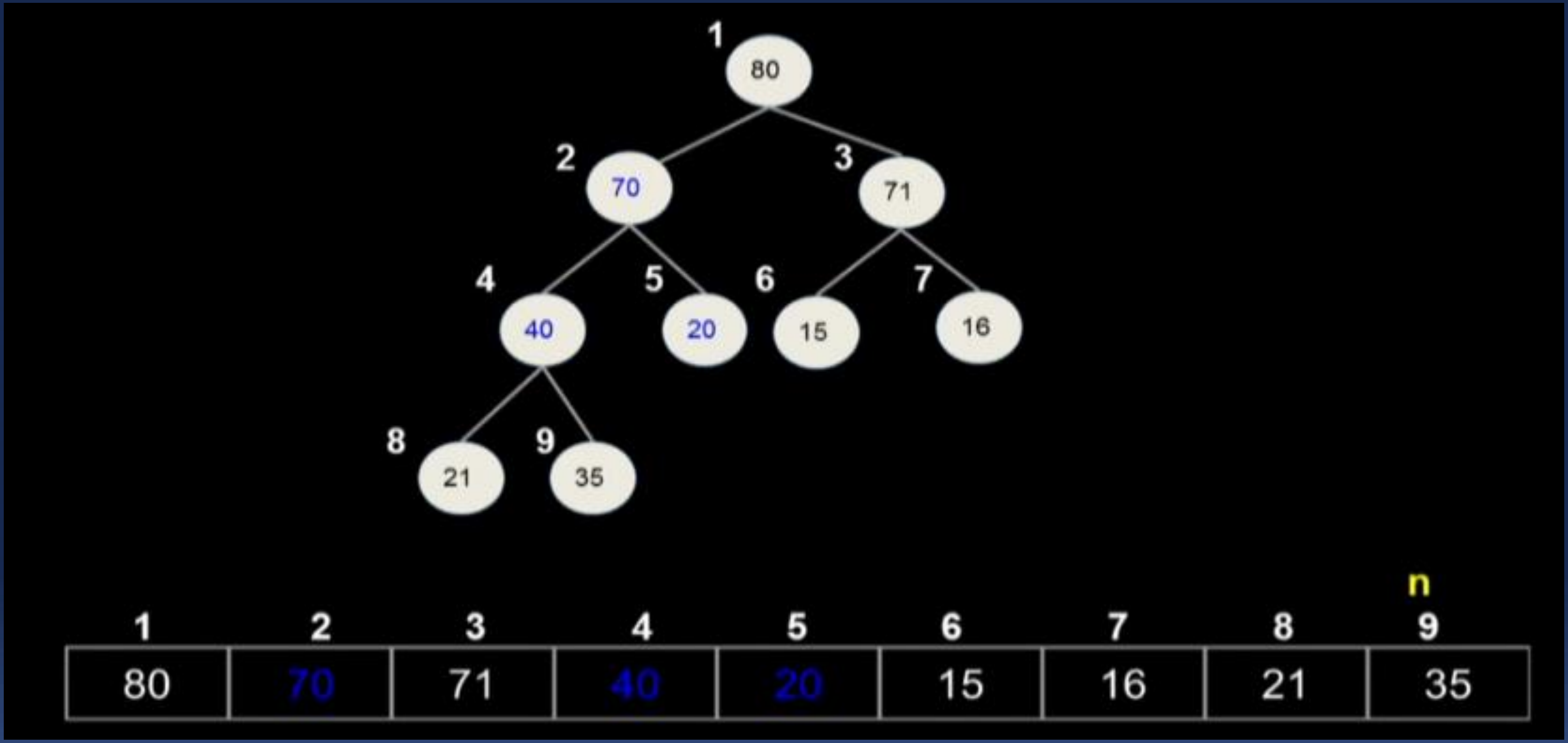
- O processo continua com o nó **15**;
- O nó com valor **15** é trocado com o nó **71**



1	2	3	4	5	6	7	8	9
40	20	71	70	80	15	16	21	35

Construção de um Max Heap

- O processo continua;
- Para cada nó não folha, aplica-se o algoritmo max-heapfy, até finalizar-se a árvore com a estrutura de **heap**.



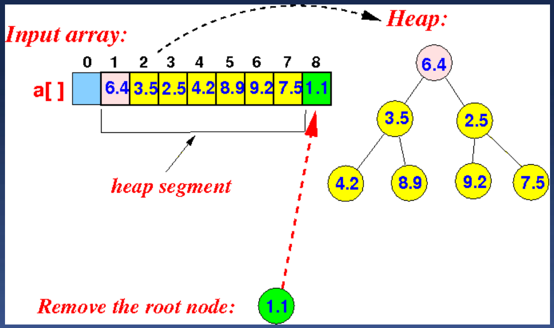
Construção de um Max Heap

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

- Mostra-se que a complexidade do algoritmo BUILD-MAX-HEAP é **O(n)**

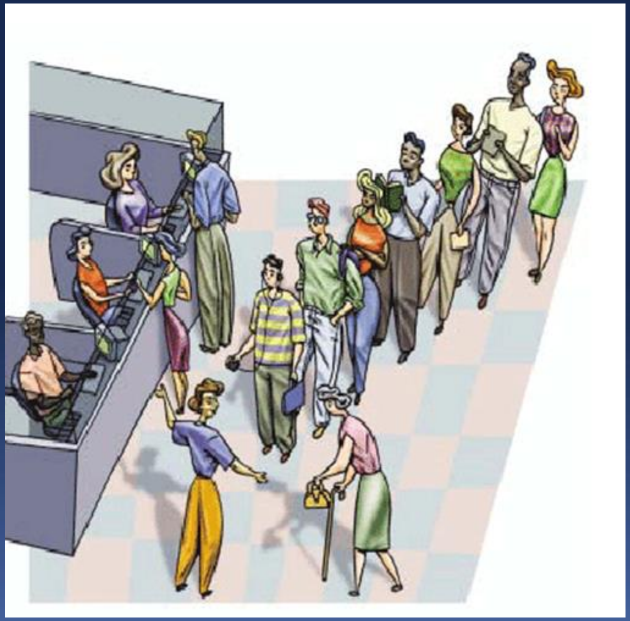
Aplicações de Heap

- Fila de Prioridade
- Heapsort



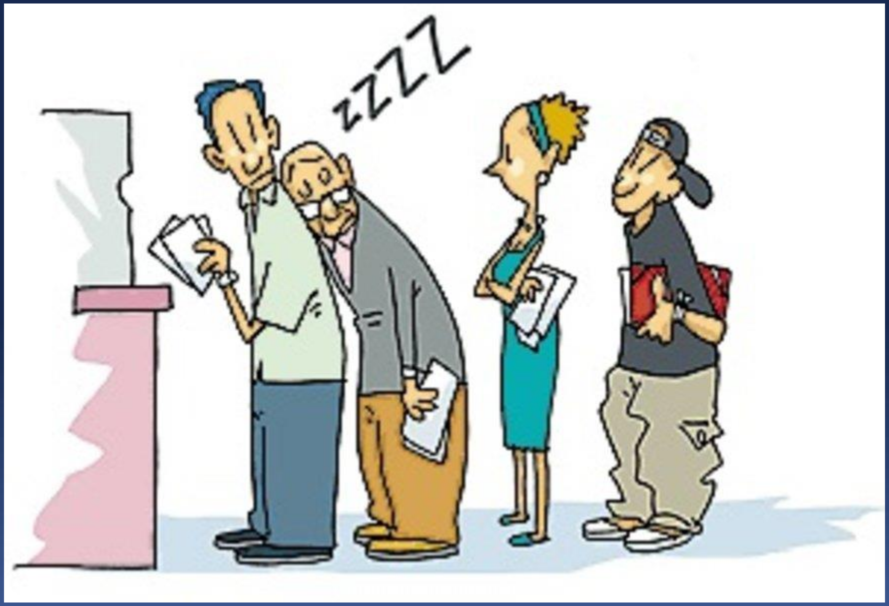
Fila de Prioridade

- É uma fila que permite que elementos sejam adicionados asssociados a uma **prioridade**;
- Cada elemento na fila deve possuir um dado adicional que representa sua **prioridade** de atendimento;
- Uma regra explícita define que o elemento de maior **prioridade** (o que tem o maior número associado) deve ser o primeiro a ser removido da fila, quando uma remoção for requerida;



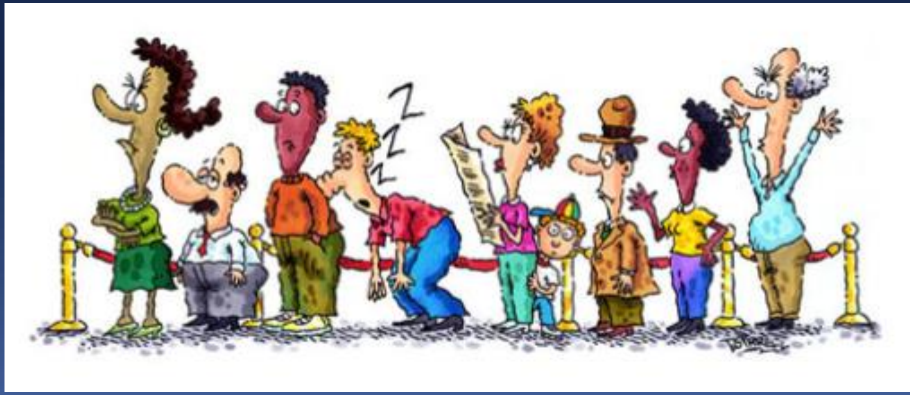
Fila de Prioridade

- São empregadas em diversas aplicações;
- Por exemplo, em **filas** de bancos geralmente há um esquema de prioridade em que clientes preferenciais, idosos ou mulheres grávidas possuem maior prioridade de atendimento quando comparados aos demais clientes;

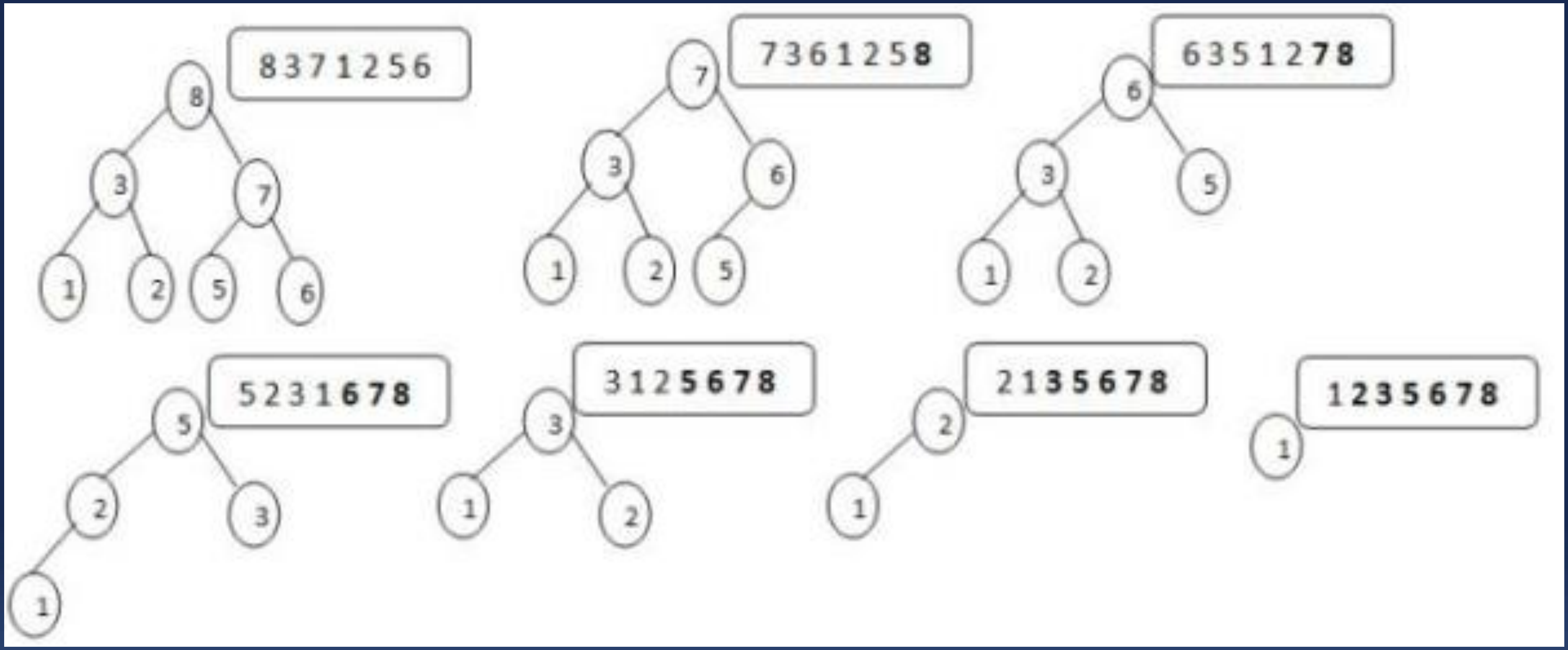


Fila de Prioridade - Operações

- **Inserir** com prioridade;
- **Remover** do elemento com mais alta prioridade;
- **Alterar** a prioridade de um determinado elemento;
- **Retornar** o número de elementos existentes na fila de prioridade;
- **Testar** a existência de elementos de mesma prioridade.



Implementação HeapSort



QualitSys

Implementação HeapSort

```
package br.uscs;

import java.util.Arrays;

public class HeapSort {

        public static int tamanho;

        public static void main(String[] args) {

                int[] lista = {5,6,2,1,9,10,12,0,3,7,14,99,34,77};

                System.out.println("lista antes do HeapSort: \n");
                System.out.println(Arrays.toString(lista)) ;

                System.out.println("\n\nlista após o HeapSort: \n");
                heapSort(lista);

                System.out.println(Arrays.toString(lista)) ;
}
```

Implementação HeapSort

```
public static void maxHeapify (int[] A, int pai) {

       int esq = 2 * pai + 1;
       int dir = (2 * pai) + 2;
       int maior = pai;

       if (esq <= tamanho && A[esq] > A[maior])
              maior = esq;

       if (dir <= tamanho && A[dir] >  A[maior])
              maior = dir;

       if (maior != pai) {
              int aux = A[pai];
              A[pai] = A[maior];
              A[maior] = aux;
              maxHeapify(A, maior);
       }
}
```

Implementação HeapSort

```
public static void buildMaxHeap(int[]A) {

      tamanho = A.length - 1;

      for (int pai = tamanho/2; pai >=0; pai--) {
            maxHeapify(A,pai);
      }
}
```

Implementação HeapSort

```
public static void heapSort(int[] A) {
        buildMaxHeap(A);

        for (int i = tamanho; i>0; i--) {
                int aux = A[i];
                A[i] = A[0];
                A[0] = aux;
                tamanho--;
                maxHeapify(A,0);
        }
  }
}
```