

DCC405 – ESTRUTURA DE DADOS II

Aula 13 – Fila de Prioridade e Heap

Fila de Prioridade

- A **fila de prioridade** nada mais é que uma **fila** comum que permite que elementos sejam adicionados **associados com 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 é requerida.

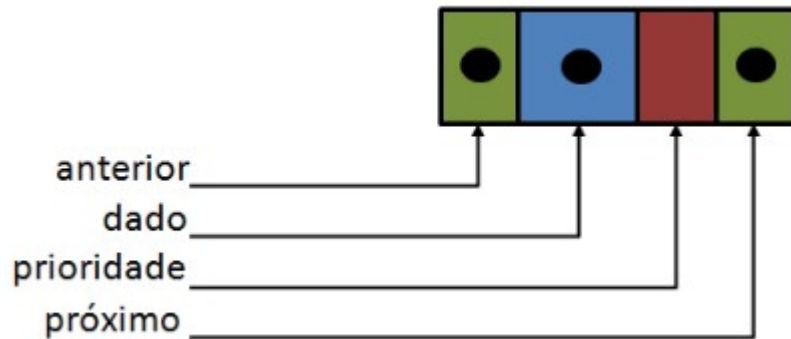


Fila de Prioridade

- As filas de prioridade são utilizadas em diversas situações. Por exemplo nas filas do banco existe um esquema de prioridade em que clientes preferenciais, idosos ou mulheres grávidas possuem uma mais alta prioridade de atendimento se comparados aos demais clientes.
- Fila de processos a serem executados pelo processador (sistemas operacionais)
- Paginação
- Algoritmo Dijkstra
- Heapsort

Fila de Prioridade - Estrutura

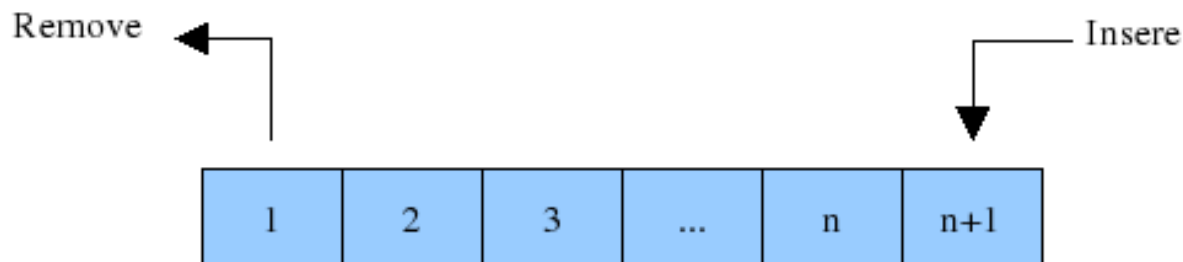
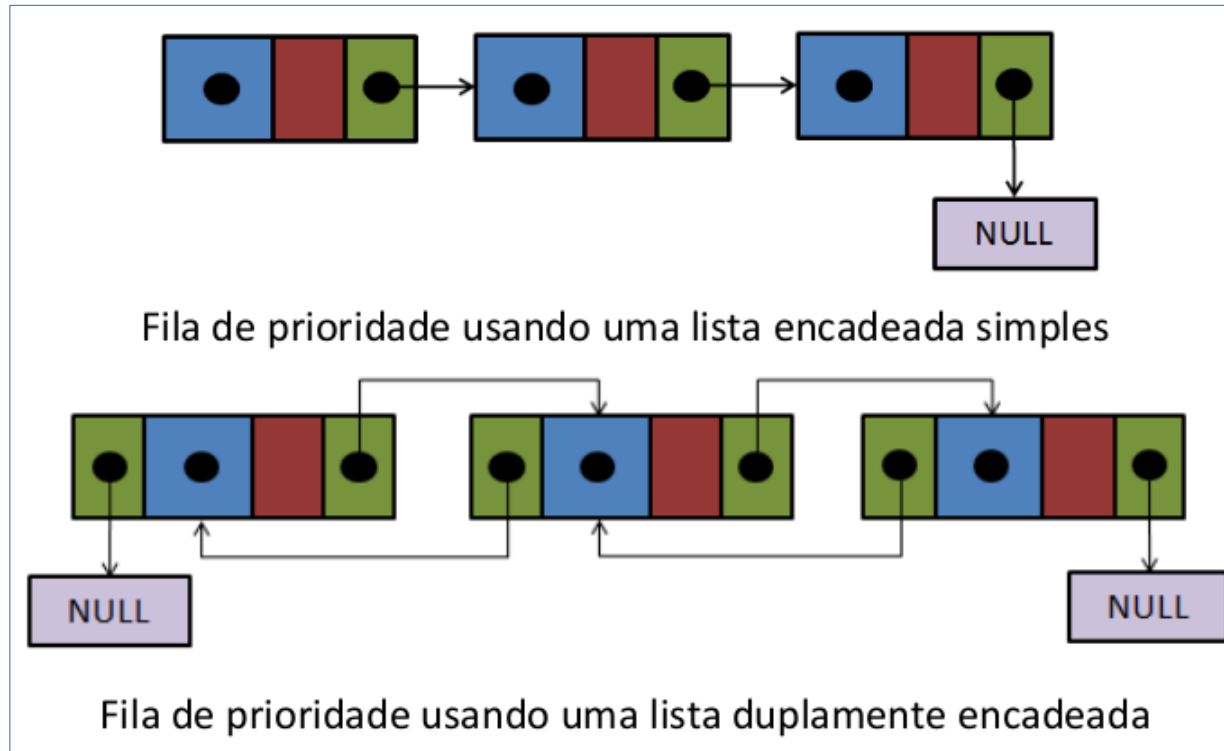
- Como discutido anteriormente, os elementos ou células de um nó de uma **fila de prioridade**



```
struct no {  
    int valor;  
    int prioridade;  
    struct no *prox;  
    struct no *ant;  
};  
typedef struct no No;
```

Os nós em uma fila podem ser encadeadas de maneira simples ou dupla. A escolha do método de encadeamento deve ser dirigida pela necessidade imposta pela aplicação a qual a estrutura de dados se presta assim como por outros fatores **tal como o fato dos dados serem inseridos de maneira ordenada ou não**;

Fila de Prioridade



?

Fila de Prioridade - Operações

- As filas de prioridade preveem duas operações básicas que são na realidade uma extensão das operações básicas de uma fila comum. São elas:
 - **Inserir com prioridade**
 - **Remover elemento de mais alta prioridade**
- Adicionalmente, versões mais avançadas de filas de prioridade podem requerer as seguintes operações:
 - Alterar prioridade de um dado elemento;
 - Número de elementos na fila;
 - Testar a existência de elementos de mesma prioridade;

Encontrar o maior elemento de uma lista

- Antes de começarmos a estudar como implementar filas de prioridade é necessário desenvolver uma maneira de encontrar o maior elemento dentre os existentes em uma dada lista de dados.



Encontrar o maior elemento de uma lista

Existem várias formas de encontrar o maior elemento de uma lista de números. No entanto, para fins de discussão, consideremos a versão “ingênua” abaixo:

```
Entrada:  lista L de números
Saída:    m / m > e  $\forall e \in L$ 

m encontrarMaior(L)
  m ← L(1)
  t ← |L|
  PARA c = 2 ATE t
    SE L(c) > m
      m ← L(c)
  FIM
FIM
```

Encontrar o maior elemento desta lista sempre demorará $O(n)$ para cada pesquisa. Embora $O(n)$ seja uma boa complexidade computacional, as três regras de análise de algoritmos nos impõem a seguinte pergunta:

Podemos fazer melhor?

Fila de Prioridade

- Analisando o algoritmo do slide anterior mais de perto.
Adicionar um novo elemento na lista pode ser feito muito rapidamente, de fato em **$O(1)$** . No entanto, se mudarmos a forma de inserir os novos elementos da lista, podemos manter o maior elemento sempre em uma posição acessível, tornando a resolução do problema de encontrar o maior elemento é checar até o último elemento da lista.
- Se mudarmos a forma de inserir os novos elementos da lista podemos manter o maior elemento sempre em uma posição acessível, tornando a resolução do problema de encontrar o maior elemento factível em **$O(1)$** . Note que, no entanto, a inserção dos novos elementos não terá mais complexidade $O(1)$ mas sim, possivelmente **$O(n)$** ou ainda complexidades de mais alta ordem.

Fila de Prioridade - Implementação

- **Filas de prioridade podem ser implementadas de diversas maneiras.**
 - Array estático (ver viabilidade);
 - Listas encadeadas simples;
 - Listas duplamente encadeadas;
- **As listas encadeadas ainda podem seguir uma estratégia em que os elementos são inseridos de maneira ordenada ou não:**
 - **Não ordenada** – inserção tal qual em uma fila comum (sempre no final da fila), estratégia esta chamada de filas não ordenadas;
 - **Ordenada** – Antes de um elemento ser inserido a fila, a posição correta (de acordo com a sua prioridade) deve ser identificada.

Fila de Prioridade

- Na implementação **não ordenada**, utiliza-se uma lista encadeada como estrutura de dados base. A inserção de elementos acontece tal como na fila normal, no entanto, a remoção de itens se dá por meio da seleção do elemento de maior prioridade da fila. Para tal, toda a lista deve ser percorrida de modo a identificar o elemento com maior prioridade.
- O método **Inserir_com_prioridade(Fila, prioridade)** deve ser composto dos seguintes passos:
 - Inserir elemento no final da fila
 - Atualizar a prioridade do elemento
 - Ajustar ponteiros para manter a estrutura de dados consistente
- O método para remoção do elemento de mais alta prioridade deve ser composto dos seguintes passos:
 - Percorrer todos os elementos da lista para identificar o de maior prioridade
 - Remover tal elemento da lista
 - Ajustar ponteiros para manter a estrutura de dados consistente
 - Retornar o elemento identificado

Fila de Prioridade

- Uma outra possibilidade é manter a fila organizada de alguma forma para que o elemento demais alta prioridade fique sempre acessível e identificável rapidamente, fazendo com que o acesso a tal elemento seja imediato e que a sua remoção da lista tenha uma complexidade computacional baixa (idealmente $O(1)$).

A primeira maneira que veremos para produzir uma fila de prioridade com tais características será via a utilização de uma fila ordenada pela ordem de prioridade de seus elementos.

- Inserção $O(n)$
- Remoção $O(1)$

Fila de Prioridade

- No final das contas, a diferença das implementações ordenada e não ordenada de filas de prioridade usando filas comuns apenas difere de em qual operação possuirá maior complexidade, se na inserção ou remoção.
- Ambas as implementação solucionam o problema, no entanto, a pergunta persiste:

“podemos fazer melhor?”

- A resposta é sim, Existe uma forma de implementar filas de prioridade com complexidade $O(1)$ para acesso do elemento de maior prioridade e **$O(n \log n)$** para inserção e remoção de elementos. No entanto, devemos estudar primeiro, outra estrutura de dados chamada **Heap**.

Fila de Prioridade

- Implementar algoritmo

Heap - Roteiro

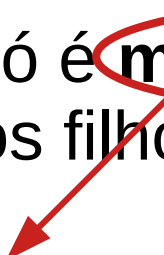
- Definição
- Manutenção da propriedade de heap
- Construção de um max-heap
- Operações de inserção e remoção

Heap

- É uma estrutura de dados que pode ser visualizada como uma **árvore binária** quase completa. (*heap-binário*)
- 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 mais à esquerda possível
 - O conteúdo de um nó é **maior ou igual** ao conteúdo dos seus nós filhos (*max-heap*)

Heap

- É uma estrutura de dados que pode ser visualizada como uma **árvore binária** quase completa. (*heap-binário*)
- 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 mais à esquerda possível
 - O conteúdo de um nó é **maior ou igual** ao conteúdo dos seus nós filhos (**max-heap**)



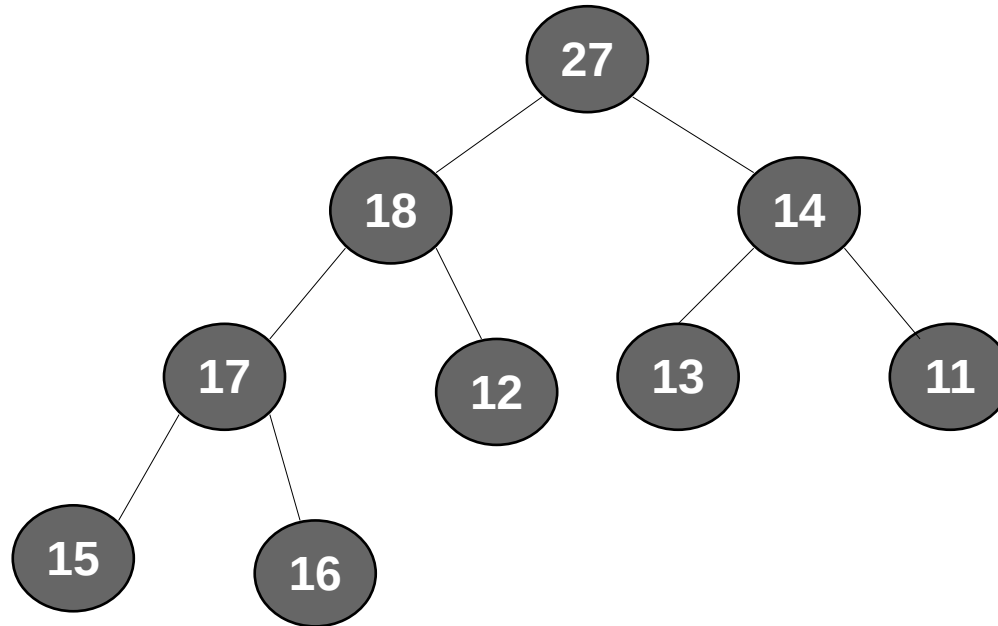
Se inverter essa propriedade para **menor ou igual** temos um (**min-heap**)

Heap

- Essas condições garantem que essa estrutura possa ser armazenada em **um vetor** $A[1..n]$;

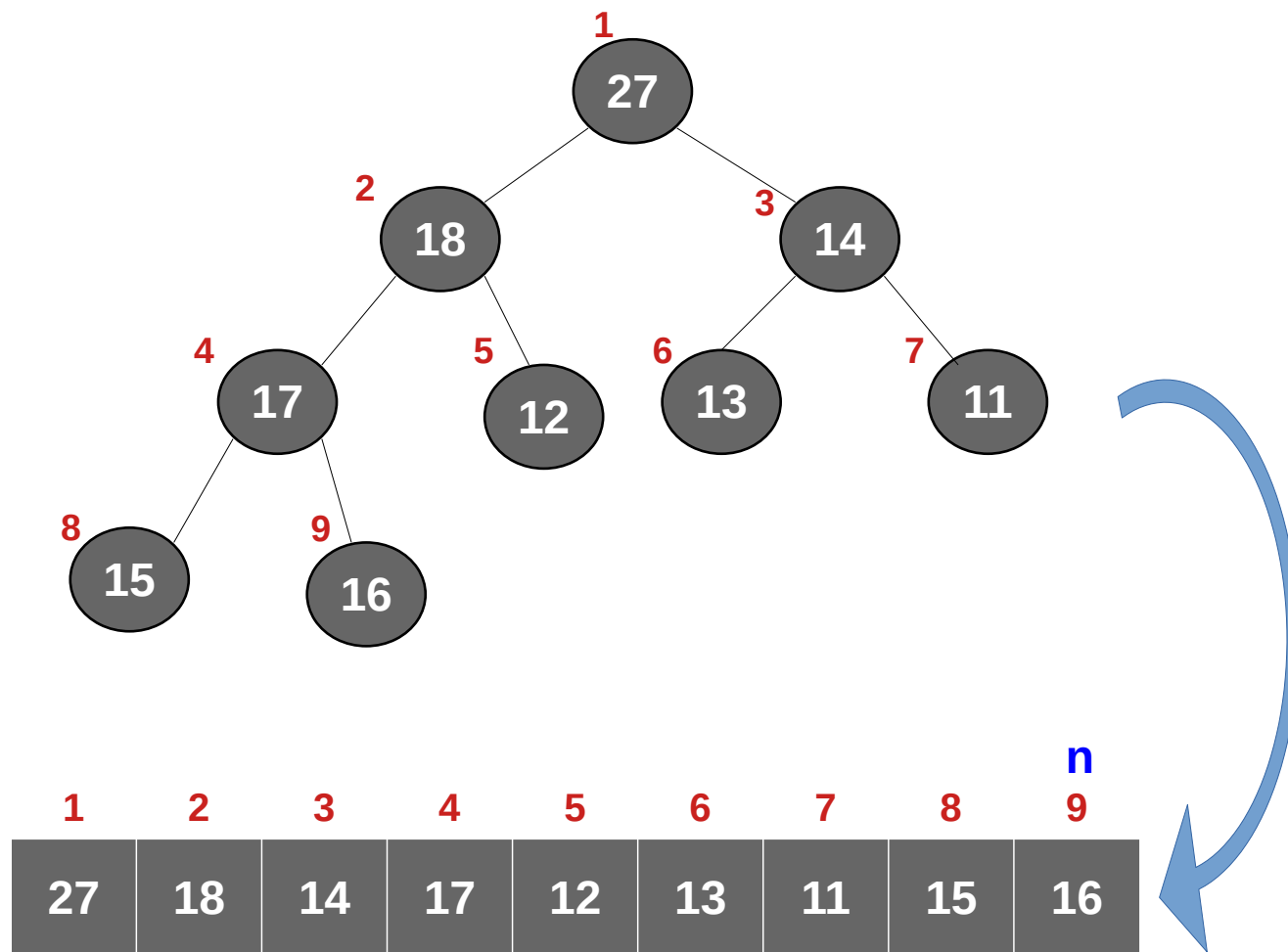


Heap

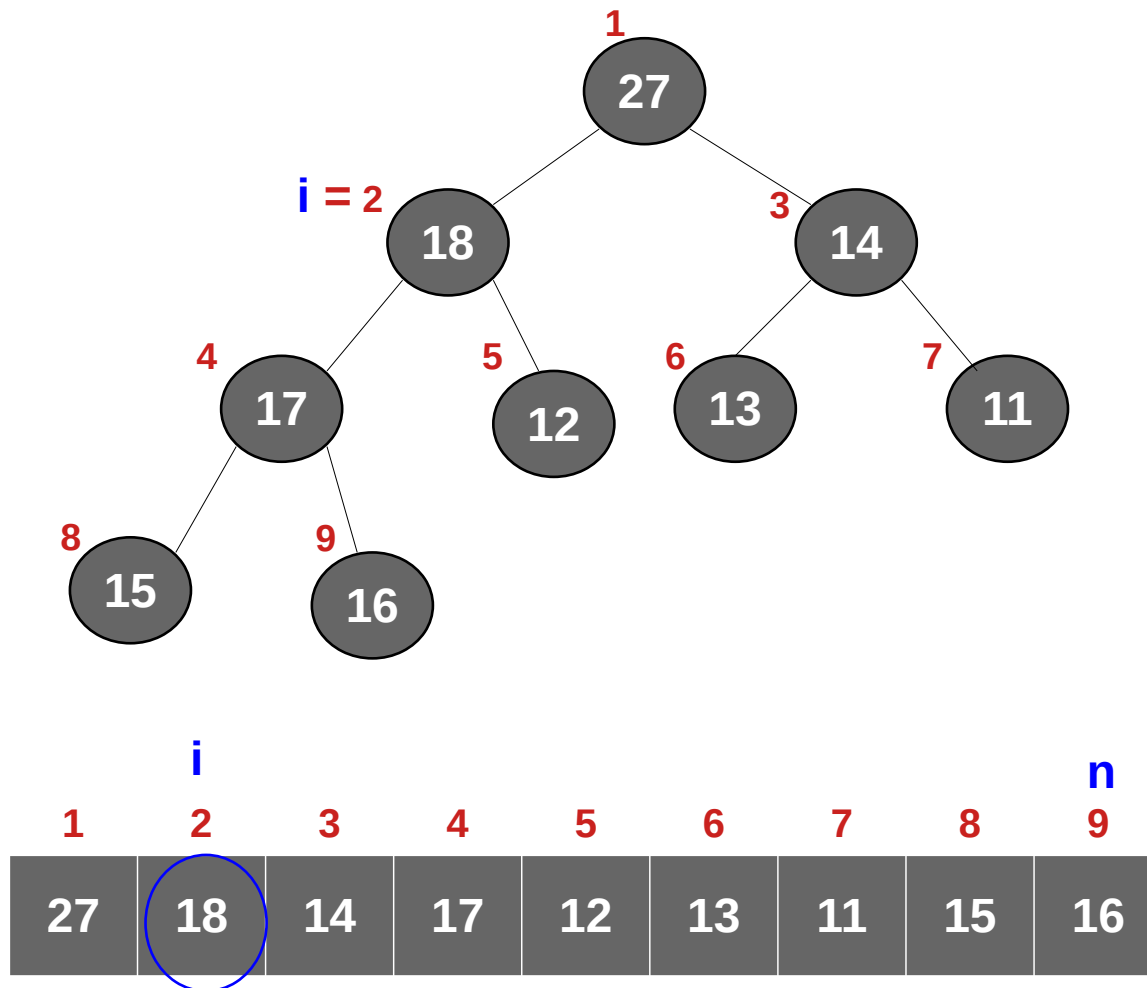


- - A árvore é completa até o penúltimo nível
 - No último nível as folhas estão mais à esquerda possível
 - O conteúdo de um nó é **maior ou igual** ao conteúdo dos seus nós filhos (**max-heap**)

Heap

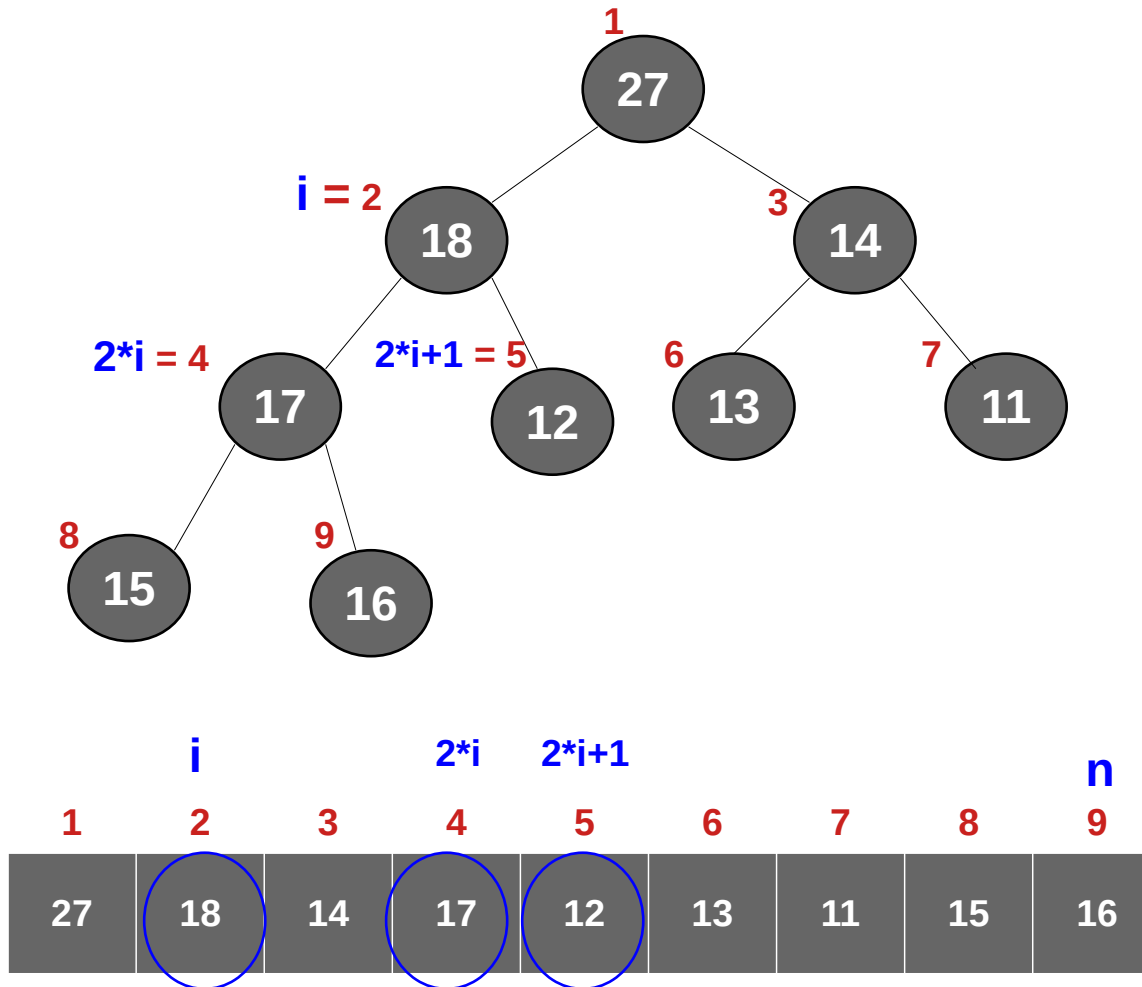


Heap

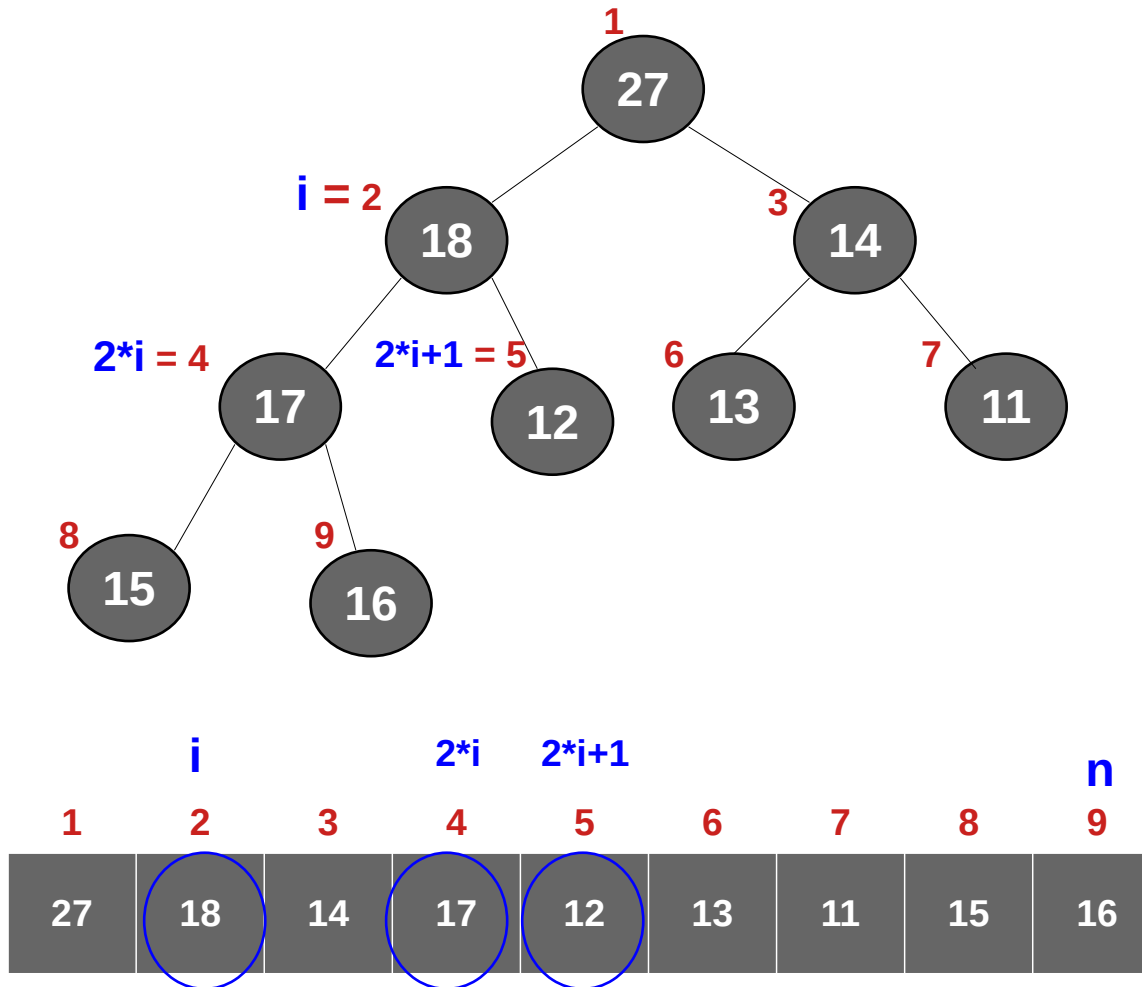


Que operação eu posso fazer, tendo o valor do i para saber quais são os dois filhos dele?

Heap



Heap



Principal propriedade **Heap Binária**

Para qualquer nó i :

→ $2i$ é o filho a esquerda

```
left(i) {  
    return 2i;  
}
```

→ $2i+1$ é o filho a direita

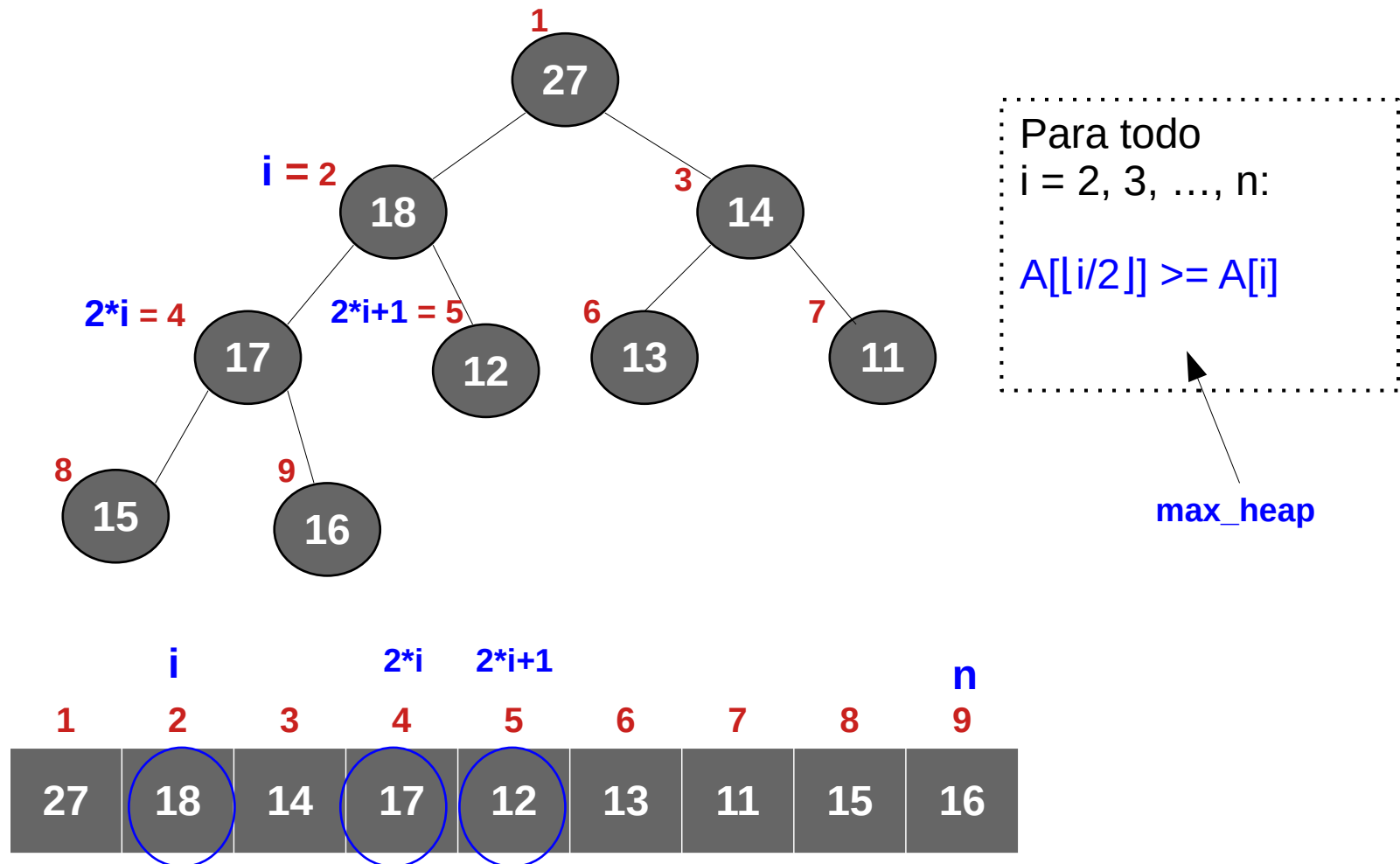
```
right(i) {  
    return 2i+1;  
}
```

Quem é o pai de i ?

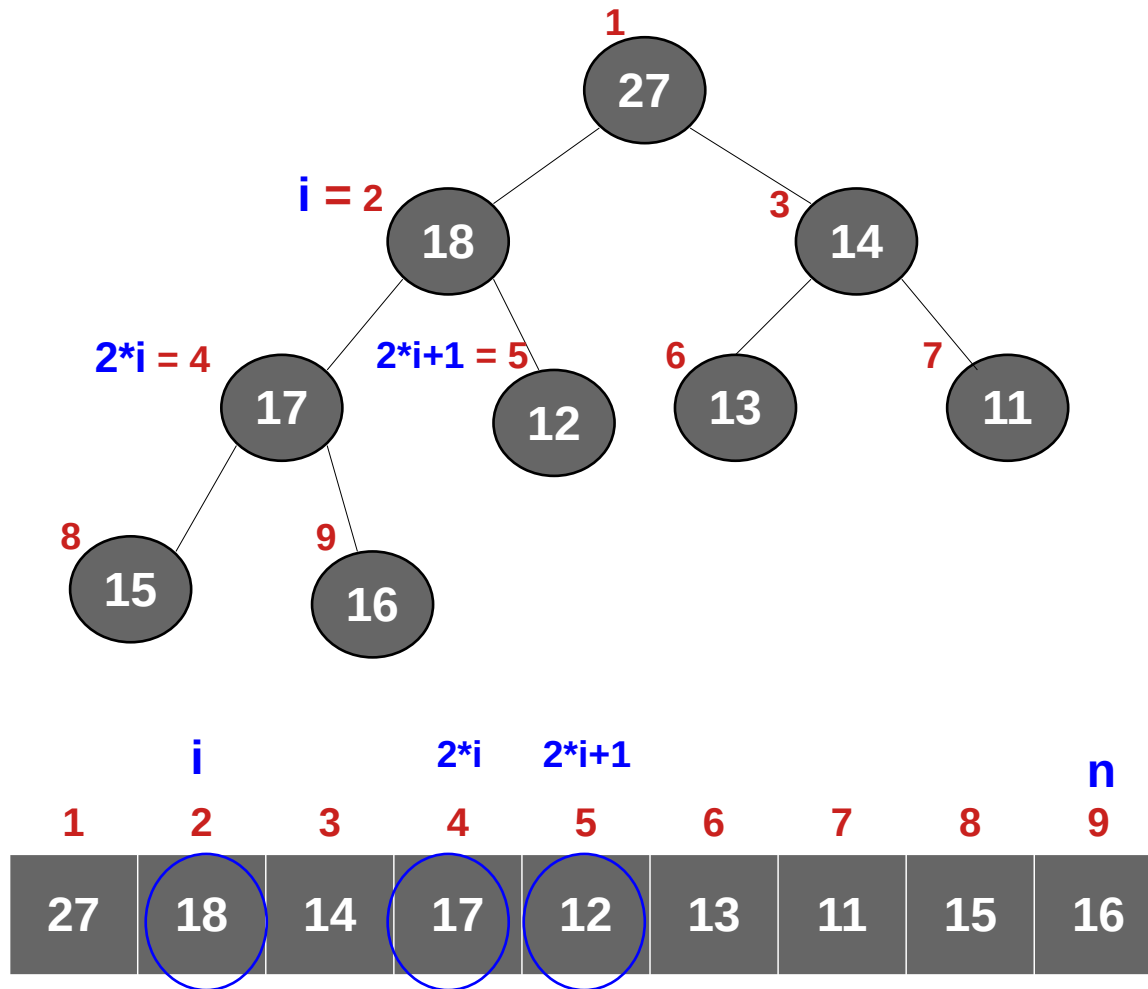
→ $\lfloor i/2 \rfloor$ é o pai de i

```
parent(i) {  
    return  $\lfloor i/2 \rfloor$ ;  
}
```

Heap



Heap



Nível

0

$1 \rightarrow 2^1 = 2$ nós

$2 \rightarrow 2^2 = 4$ nós

3

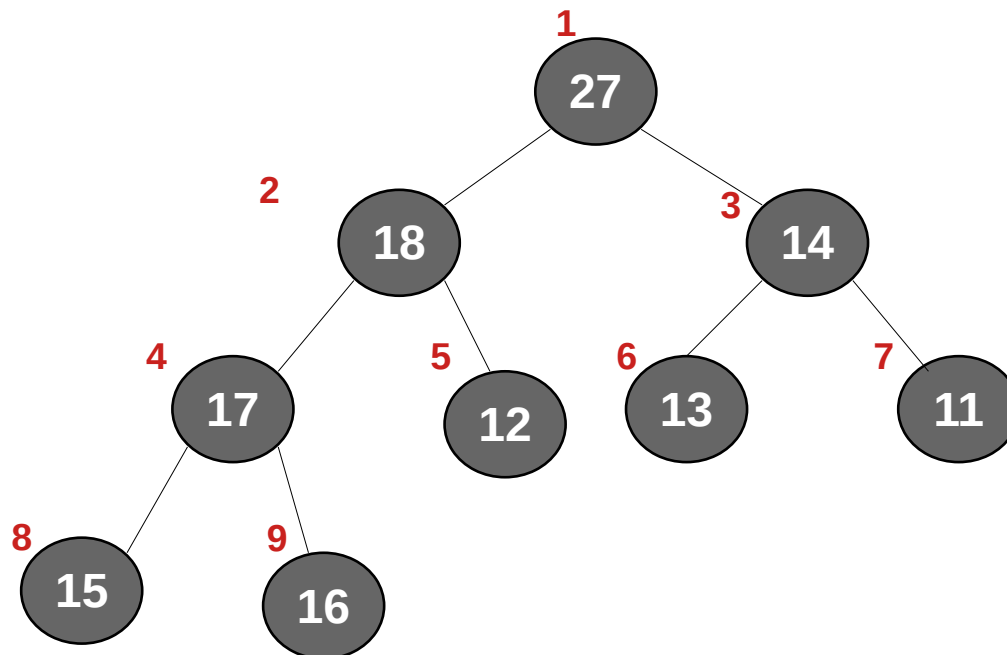
Cada **nível p** tem
exatamente **2^p nós**,
exceto talvez o último
nível.

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**.

Heap

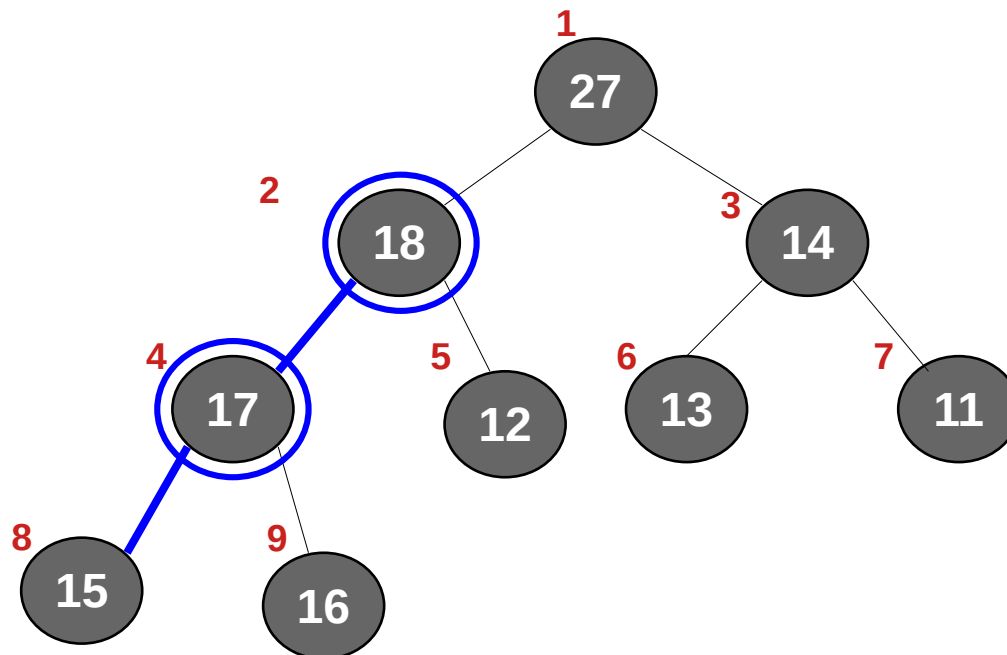
- A **altura** de um nó i é o **maior comprimento** de um caminho de i até uma folha, isto é, o número de arestas no caminho mais longo desde i até uma **folha**.



As folhas têm altura 0;
Qual a altura do nó 2?

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**.



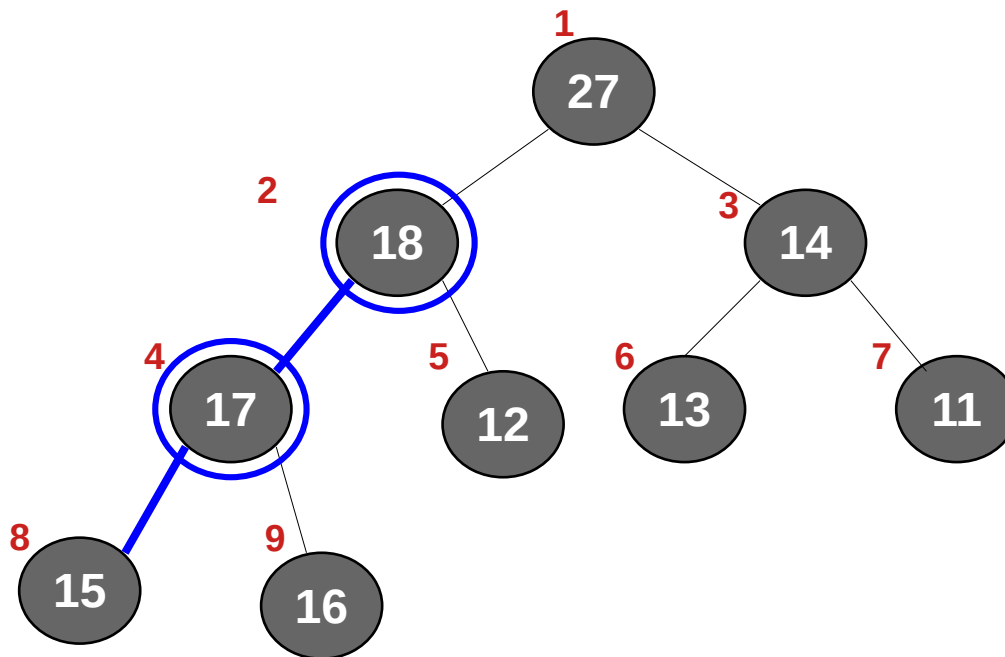
As folhas têm altura 0;

Qual a altura do nó 2?

R.: 2

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**.



Podemos demonstrar
que a altura de um nó i
qualquer é:

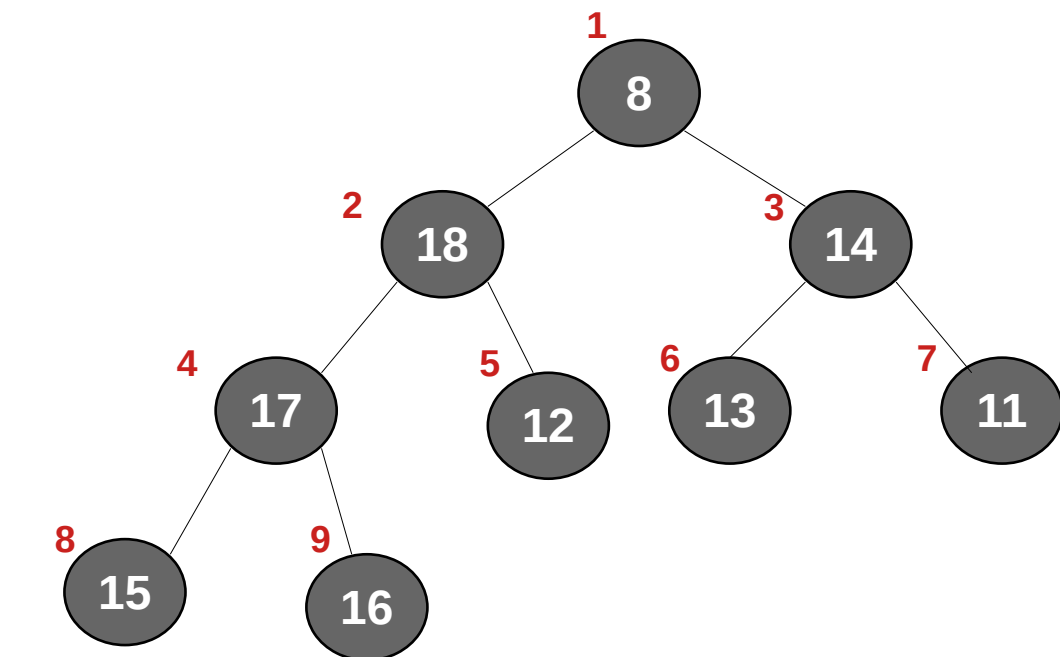
$$h = \lfloor \log(n/i) \rfloor$$

A altura da raiz é:

$$h = \lfloor \log(n) \rfloor$$

Altura da árvore

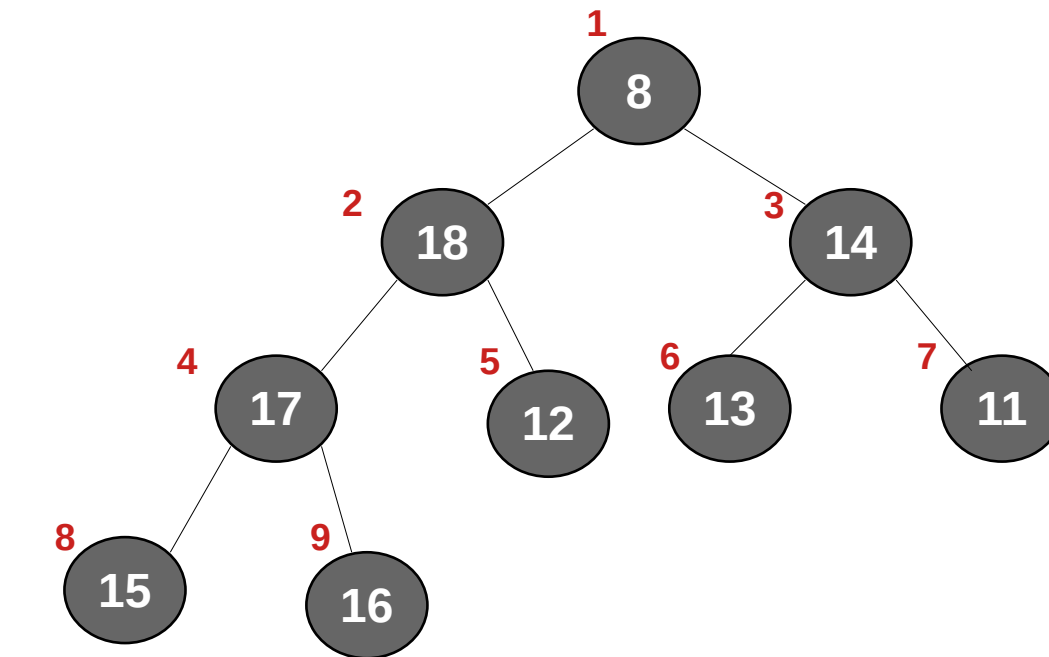
Heap – Manutenção das propriedades



Esta árvore é um heap?

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

Heap – Manutenção das propriedades

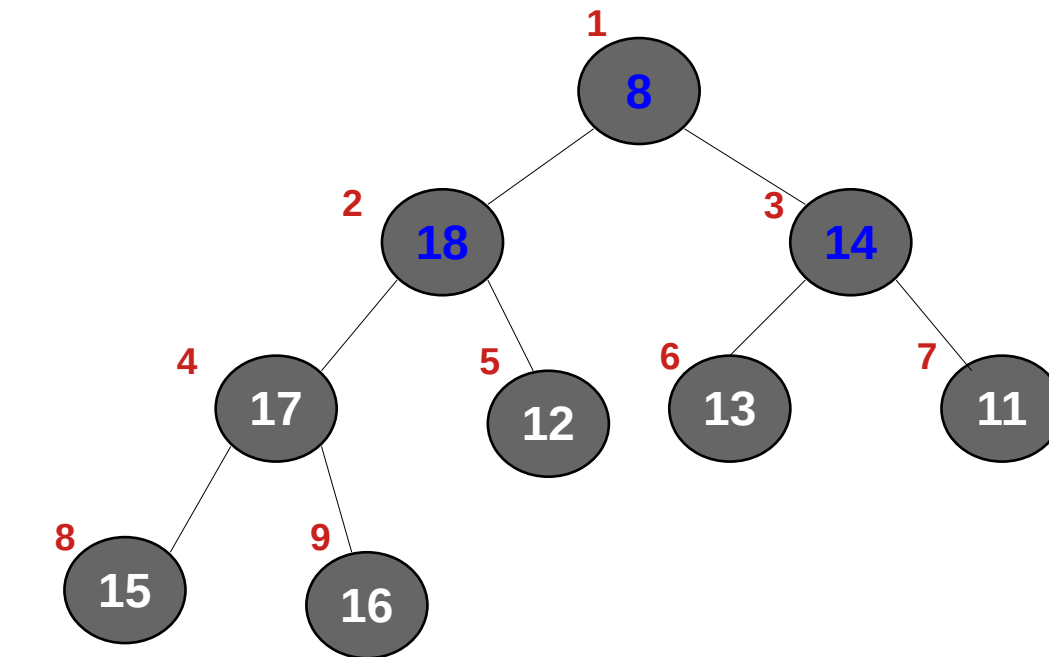


É quase um heap.

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

Heap – Manutenção das propriedades



Pega o maior dos filhos

$2i > 2i+1?$

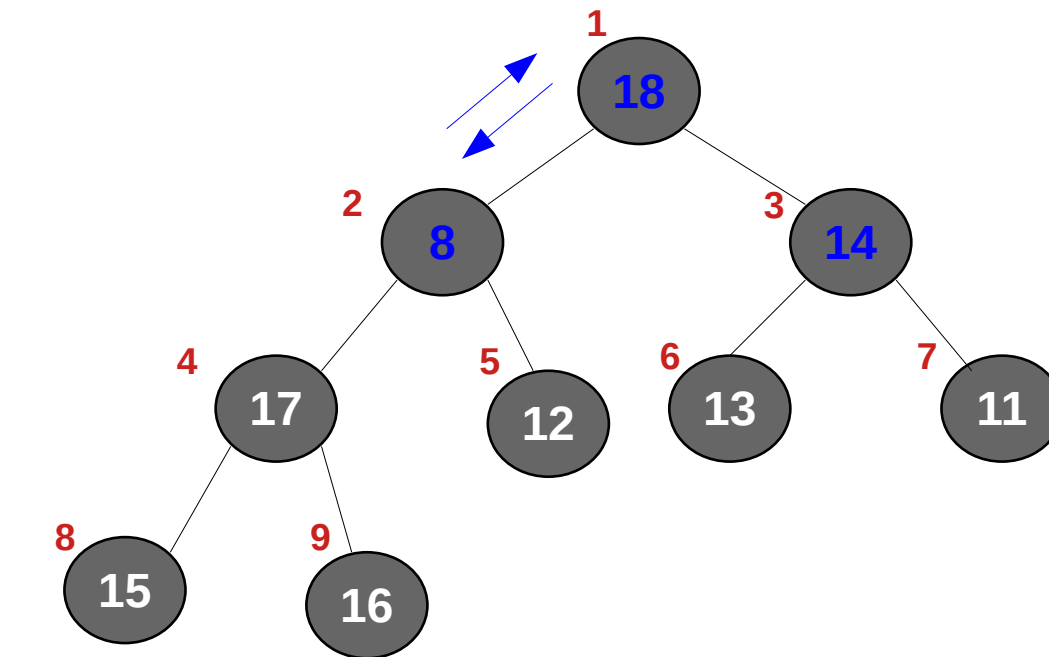
Sim

$i < 2i$

Sim, troca

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

Heap – Manutenção das propriedades



Pega o maior dos filhos

$2i > 2i+1?$

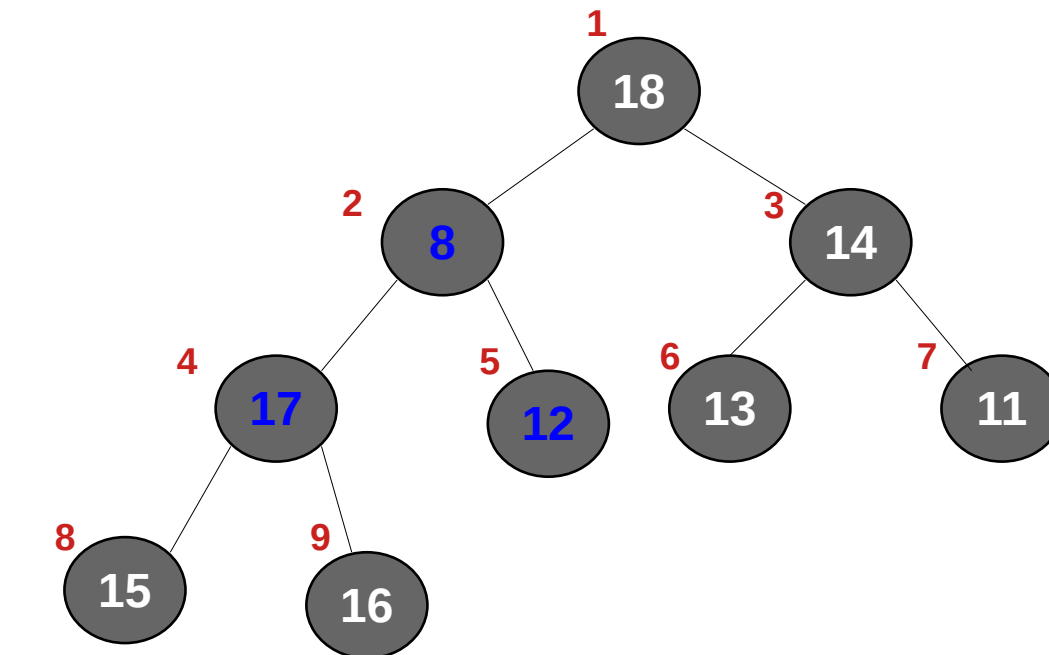
Sim

$i < 2i$

Sim, troca

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

Heap – Manutenção das propriedades



Pega o maior dos filho

$2i > 2i+1?$

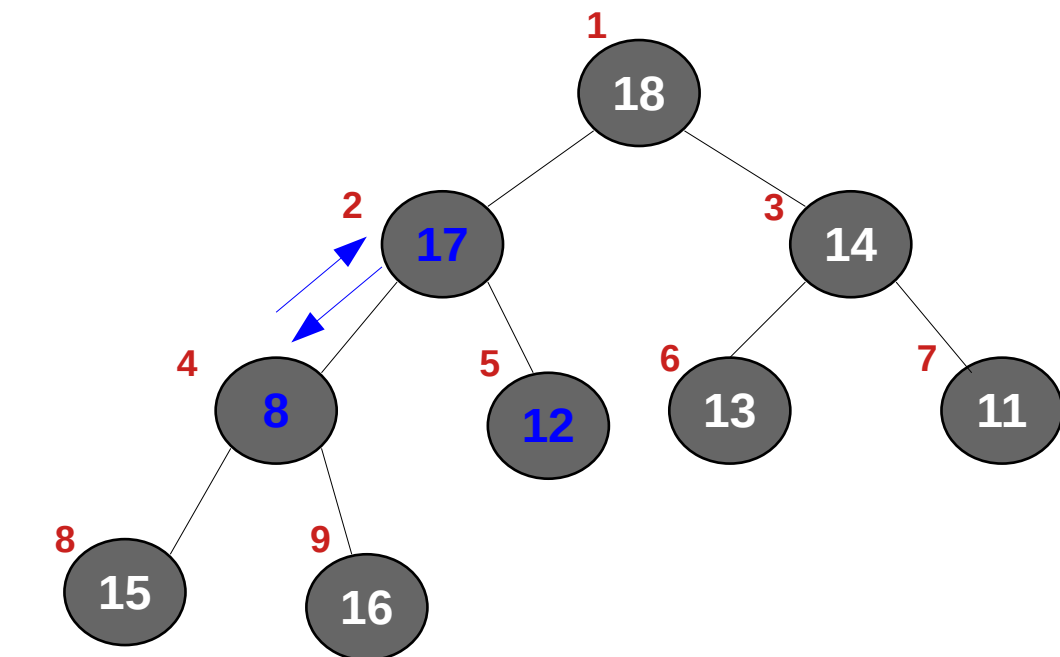
Sim

$i < 2i$

Sim, troca

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

Heap – Manutenção das propriedades



Pega o maior dos filho

$2i > 2i+1?$

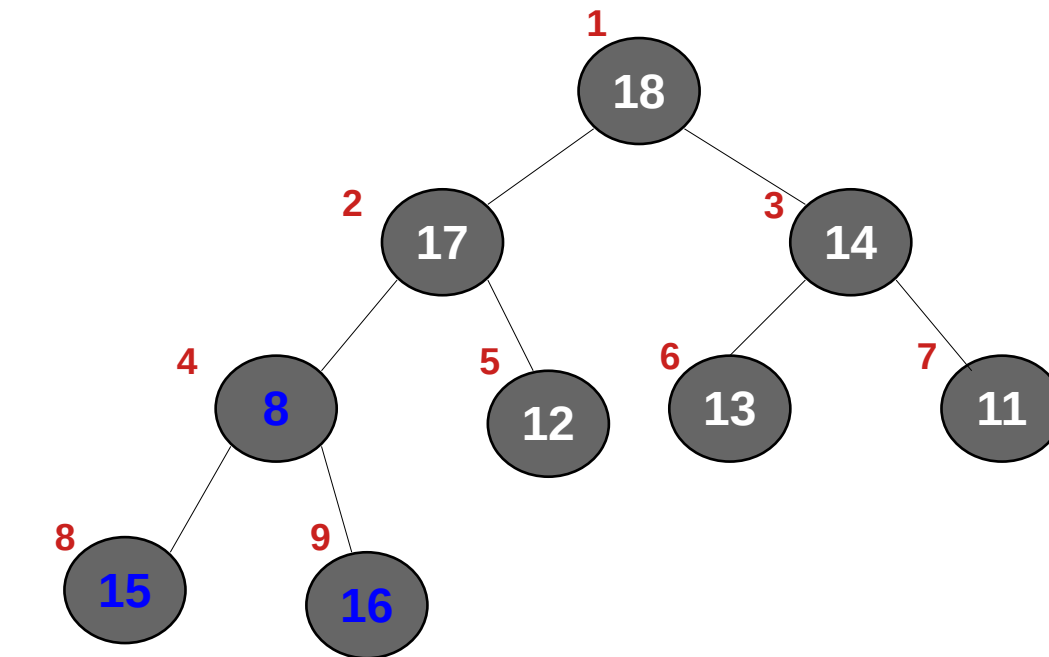
Sim

$i < 2i$

Sim, troca

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

Heap – Manutenção das propriedades



Pega o maior dos filho
 $2i > 2i+1?$

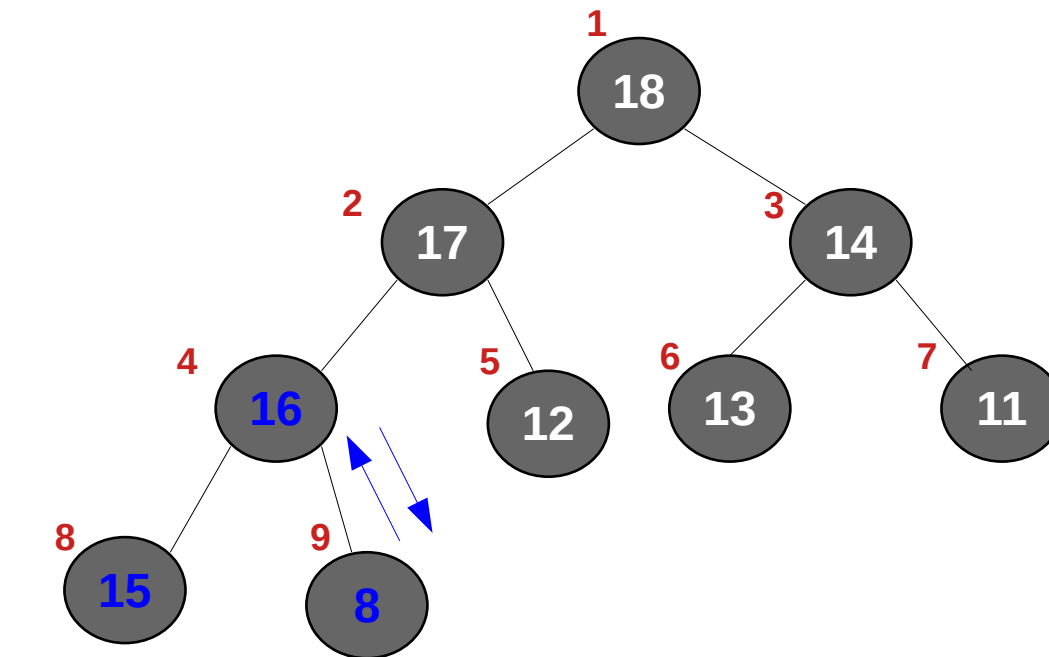
Não

$i < 2i+1$

Sim, troca

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

Heap – Manutenção das propriedades



Pega o maior dos filho
 $2i > 2i+1?$

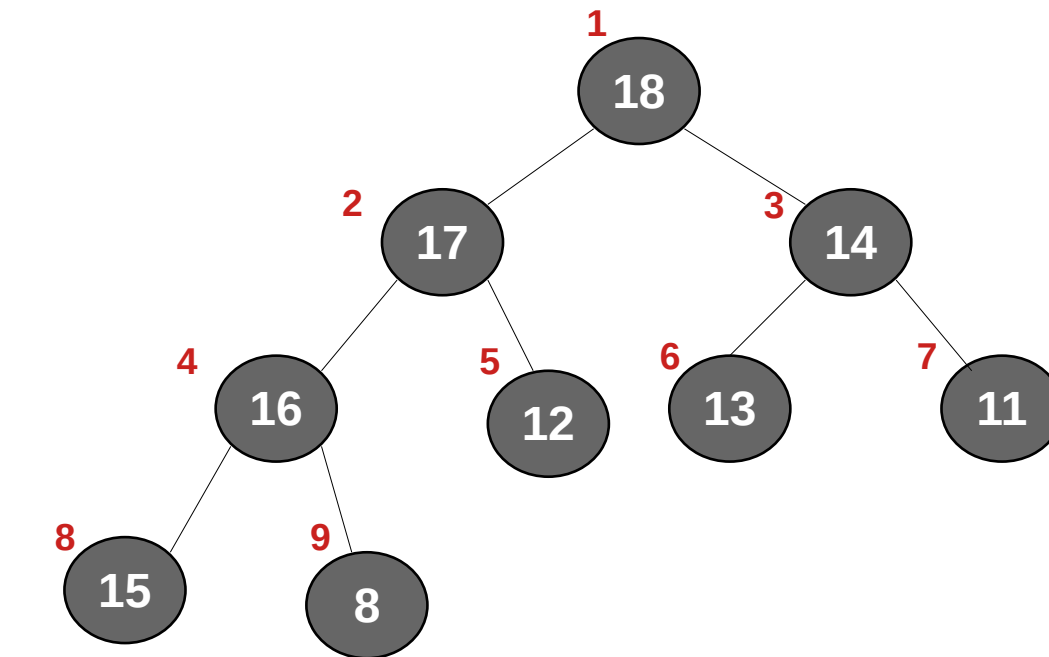
Não

$i < 2i+1$

Sim, troca

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

Heap – Manutenção das propriedades



Para transformar o vetor num heap bastou rebaixar o 8 até ele satisfazer a propriedade de heap.

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



Heap – Algoritmos

MAX-HEAPIFY(A, n, i)

```
1  e ← 2*i
2  d ← 2*i + 1
3  se e ≤ n and A[e] > A[i]
4      então maior ← e
5  senão maior ← i
6  se d ≤ n and A[d] > A[maior]
7      então maior ← d
8  se maior ≠ i
9      então A[i] ↔ A[maior]
10      MAX-HEAPIFY (A, n, maior)
```

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

Outra referência

```
HEAPIFY(A, i)
1  l ← LEFT(i)
2  r ← RIGHT(i)
3  if l ≤ heap-size[A] and A[l] > A[i]
4      then largest ← l
5  else largest ← i
6  if r ≤ heap-size[A] and A[r] > A[largest]
7      then largest ← r
8  if largest ≠ i
9      then exchange A[i] ↔ A[largest]
10     HEAPIFY(A, largest)
```

Heap – Algoritmos

MAX-HEAPIFY(A, n, i)

```
1  e ← 2*i
2  d ← 2*i + 1
3  se e ≤ n and A[e] > A[i]
4      então maior ← e
5  senão maior ← i
6  se d ≤ n and A[d] > A[maior]
7      então maior ← d
8  se maior ≠ i
9      então A[i] ↔ A[maior]
10     MAX-HEAPIFY (A, n, maior)
```

Complexidade

$$T(n) = T(n/2) + O(1)$$



$$O(\log n)$$

Outra referência

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

HEAPIFY(A, i)

```
1  l ← LEFT(i)
2  r ← RIGHT(i)
3  if l ≤ heap-size[A] and A[l] > A[i]
4      then largest ← l
5  else largest ← i
6  if r ≤ heap-size[A] and A[r] > A[largest]
7      then largest ← r
8  if largest ≠ i
9      then exchange A[i] ↔ A[largest]
10     HEAPIFY(A, largest)
```


Heap - Algoritmos

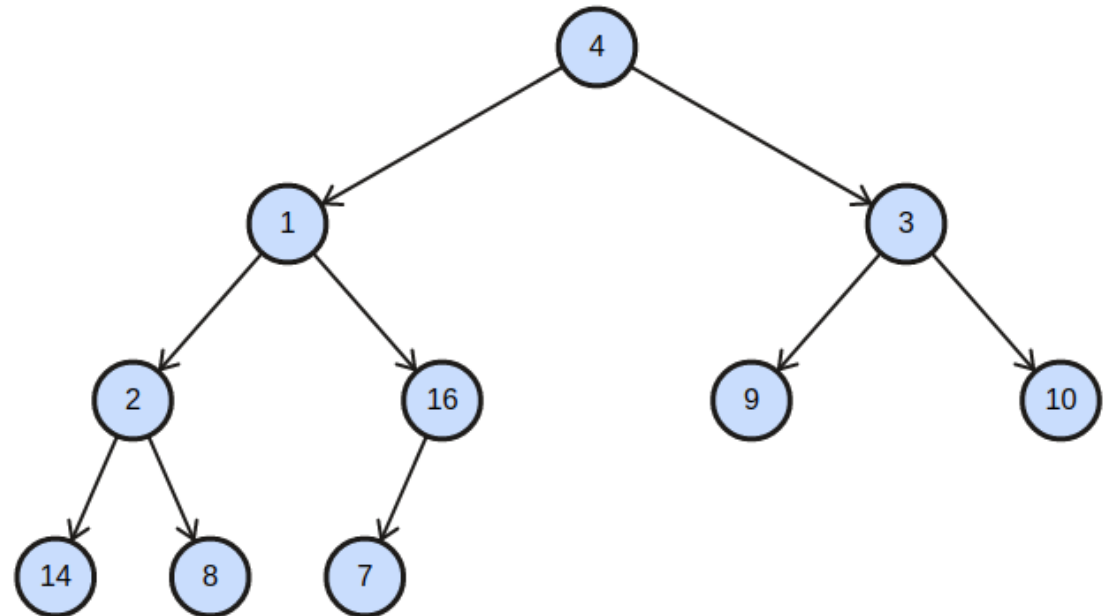
Build Heap

Constrói a heap a partir de um array

[4, 1, 3, 2, 16, 9, 10, 14, 8, 7]

Simulação

<http://btv.melezinek.cz/binary-heap.html>



Heap - Algoritmos

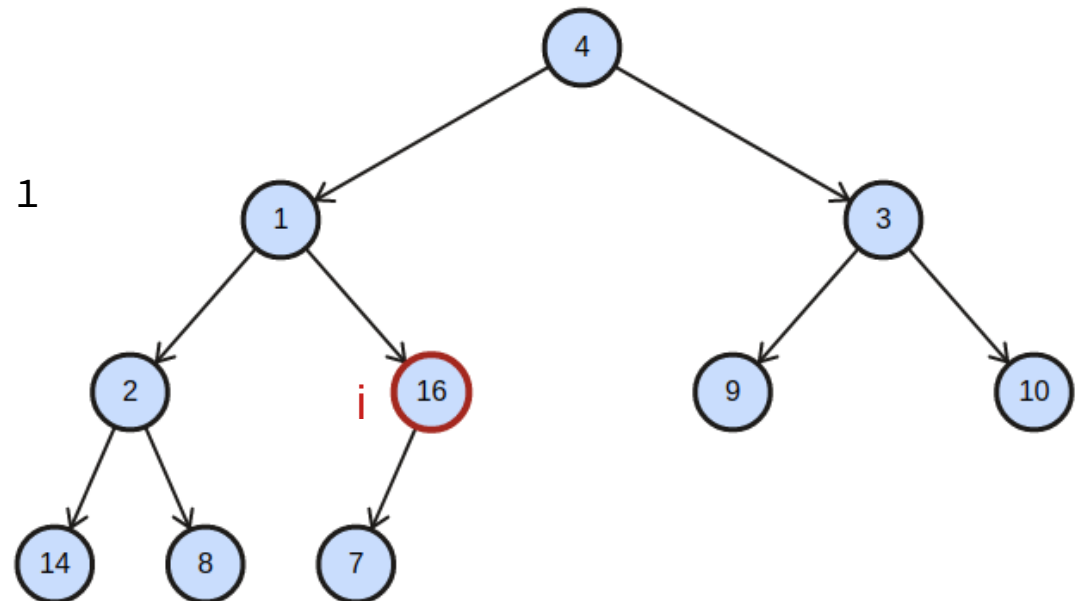
Build Heap

Constrói a heap a partir de um array

[4, 1, 3, 2, 16, 9, 10, 14, 8, 7]

BUILD-HEAP(A)

```
1  heap-size[A] ← length[A]
2  for i ← [length[A]/2] downto 1
3      do Heapify(A, i)
```



Heap - Algoritmos

Build Heap

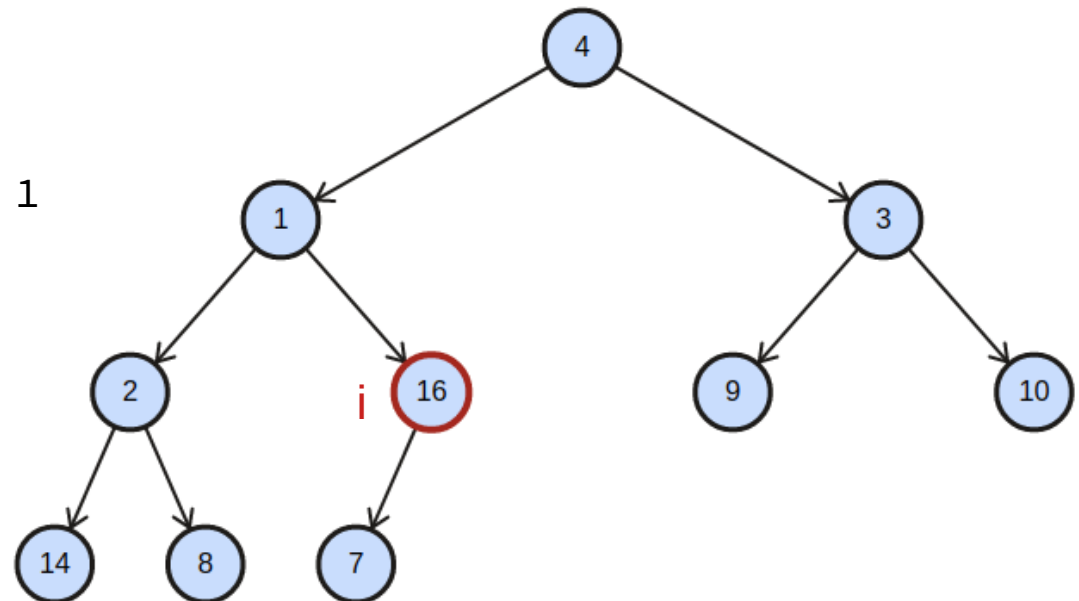
Constrói a heap a partir de um array

[4, 1, 3, 2, 16, 9, 10, 14, 8, 7]

BUILD-HEAP(A)

```
1 heap-size[A] ← length[A]
2 for i ← length[A]/2 downto 1
3   do Heapify(A, i)
```

?



Heap - Algoritmos

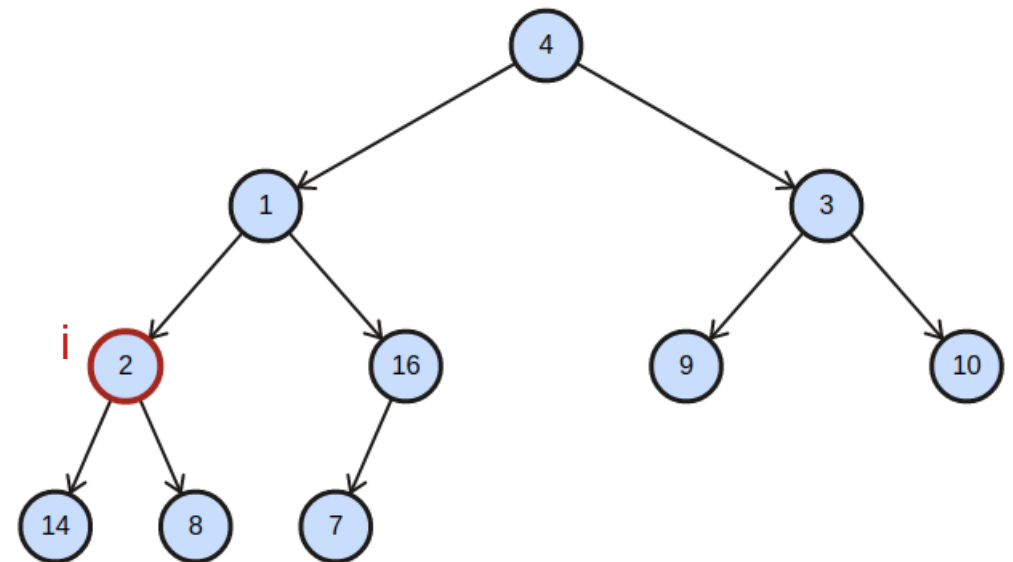
Build Heap

Constrói a heap a partir de um array

[4, 1, 3, 2, 16, 9, 10, 14, 8, 7]

BUILD-HEAP(A)

```
1 heap-size[A]  $\leftarrow$  length[A]
2 for  $i \leftarrow \lfloor \text{length}[A]/2 \rfloor$  downto 1
3   do Heapify(A,  $i$ )
```



Heap - Algoritmos

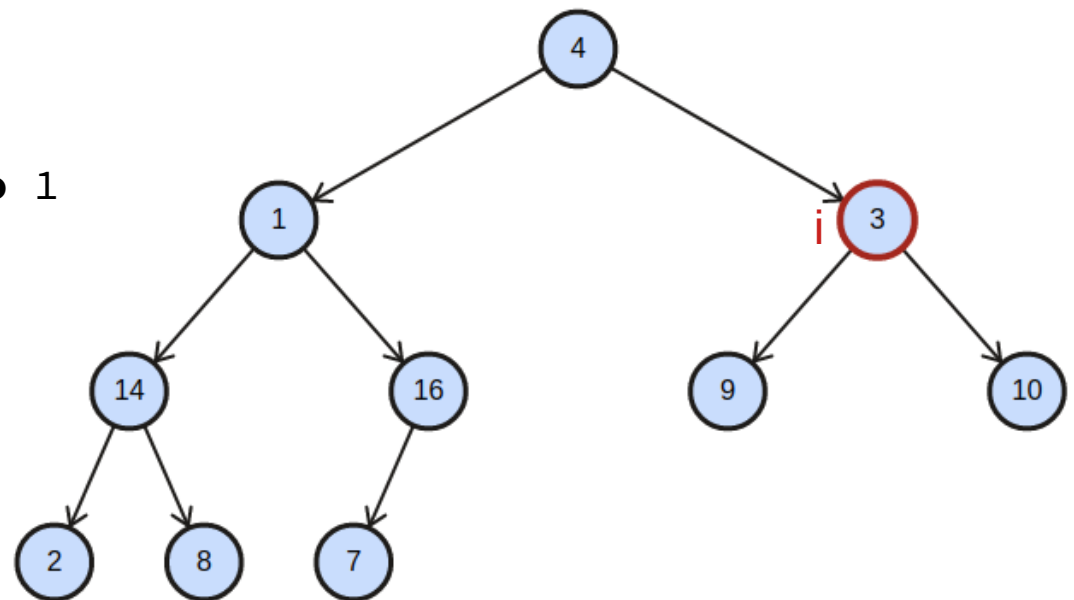
Build Heap

Constrói a heap a partir de um array

[4, 1, 3, 14, 16, 9, 10, 2, 8, 7]

BUILD-HEAP(A)

```
1 heap-size[A]  $\leftarrow$  length[A]
2 for  $i \leftarrow \lfloor \text{length}[A]/2 \rfloor$  downto 1
3   do Heapify(A,  $i$ )
```



Heap - Algoritmos

Build Heap

Constrói a heap a partir de um array

BUILD-HEAP(A)

```
1 heap-size[A] ← length[A]
2 for i ← [length[A]/2] downto 1
3   do Heapify(A, i)
```

Complexidade

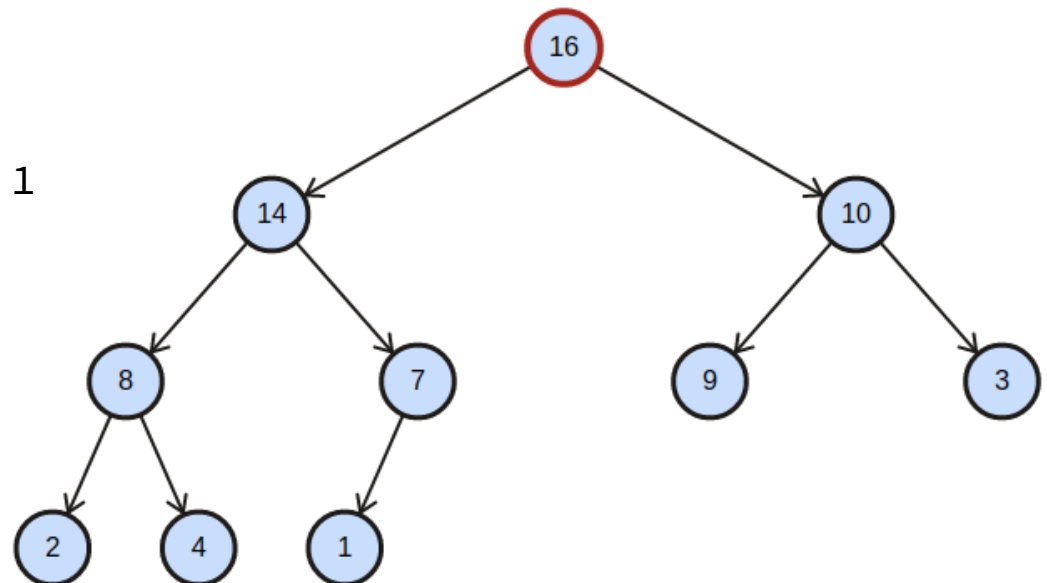
$O(n/2 \cdot \log n)$

Maioria dos casos

$O(n)$

Versão final da Heap

[16, 14, 10, 8, 7, 9, 3, 2, 4, 1]



Heap

Inserção

Inserir na heap e mantém as propriedades (invariante)

- Inserir na última posição
- Sobe o elemento se necessário até manter o invariante

Simulação

<http://btv.melezinek.cz/binary-heap.html>

Heap

Inserção

HEAP-INSERT(A, key)

```
1 heap-size[A]  $\leftarrow$  heap-size[A] + 1
2 i  $\leftarrow$  heap-size[A]
3 while i > 1 and A[Parent(i)] < key
4     do A[i]  $\leftarrow$  A[Parent(i)]
5     i  $\leftarrow$  Parent(i)
6 A[i]  $\leftarrow$  key
```

Qual a diferença básica para o Heapify?

Qual a complexidade?

Inserção

HEAP-INSERT(A, key)

```
1 heap-size[A] ← heap-size[A] + 1
2 i ← heap-size[A]
3 while i > 1 and A[Parent(i)] < key
4     do A[i] ← A[Parent(i)]
5     i ← Parent(i)
6 A[i] ← key
```

Qual a diferença básica para o Heapify?

R.: No Heapify acontece um 'conserto' do heap de cima para baixo. Na inserção o nó sempre é inserido numa folha e caso necessário ele vai subindo para se adequar as propriedades do heap.

Qual a complexidade?

Heap

Inserção

HEAP-INSERT(A, key)

```
1 heap-size[A] ← heap-size[A] + 1
2 i ← heap-size[A]
3 while i > 1 and A[Parent(i)] < key
4     do A[i] ← A[Parent(i)]
5     i ← Parent(i)
6 A[i] ← key
```

Qual a diferença básica para o Heapify?

R.: No Heapify acontece um 'conserto' do heap de cima para baixo. Na inserção o nó sempre é inserido numa folha e caso necessário ele vai subindo para se adequar as propriedades do heap.

Qual a complexidade?

R.: **$O(\log n)$**

Remoção

Remove da heap e mantêm o invariante

- Remove sempre o elemento raiz
- Move o último elemento da heap para a raiz
- Aplica o heapify

Simulação

<http://btv.melezinek.cz/binary-heap.html>

Remoção

HEAP-EXTRACT-MAX(A)

```
1  if heap-size[A] < 1
2      then error "heap underflow"
3  max ← A[1]
4  A[1] ← A[heap-size[A]]
5  heap-size[A] ← heap-size[A] - 1
6  Heapify(A, 1)
7  return max
```

Qual a complexidade?

R.: **$O(\log n)$**

Exercícios

- 1) Implemente uma **fila de prioridade** não ordenada usando elementos dinâmicos (células). Proponha as estruturas de dados necessárias e implemente as funções para inserção de um novo elemento na fila e a remoção do elemento de mais alta prioridade. Utilize desenhos esquemáticos para exemplificar o funcionamento da estrutura de dados;
- 2) Implemente a estrutura de dados **Heap**, tendo como base os algoritmos de heap vistos nesse slide.