"Um mouse é um dispositivo que contém um, dois, ou três botões, dependendo da estimativa que os projetistas dão para a capacidade intelectual de seus usuários" (Tanenbaum, Bos; 2016).

Heaps

Paulo Ricardo Lisboa de Almeida

DSBD

UFPR
UNIVERSIDADE FEDERAL DO PARANÁ

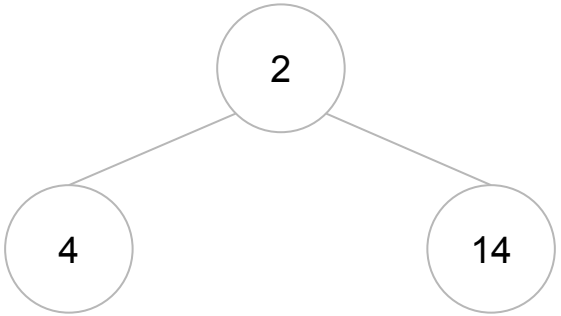
Árvore Binária

De maneira abstrata (e informal), uma **árvore binária** é uma estrutura de dados que possui **nodos ou nós**, e onde cada nó possui 0, 1 ou 2 filhos

Exemplo

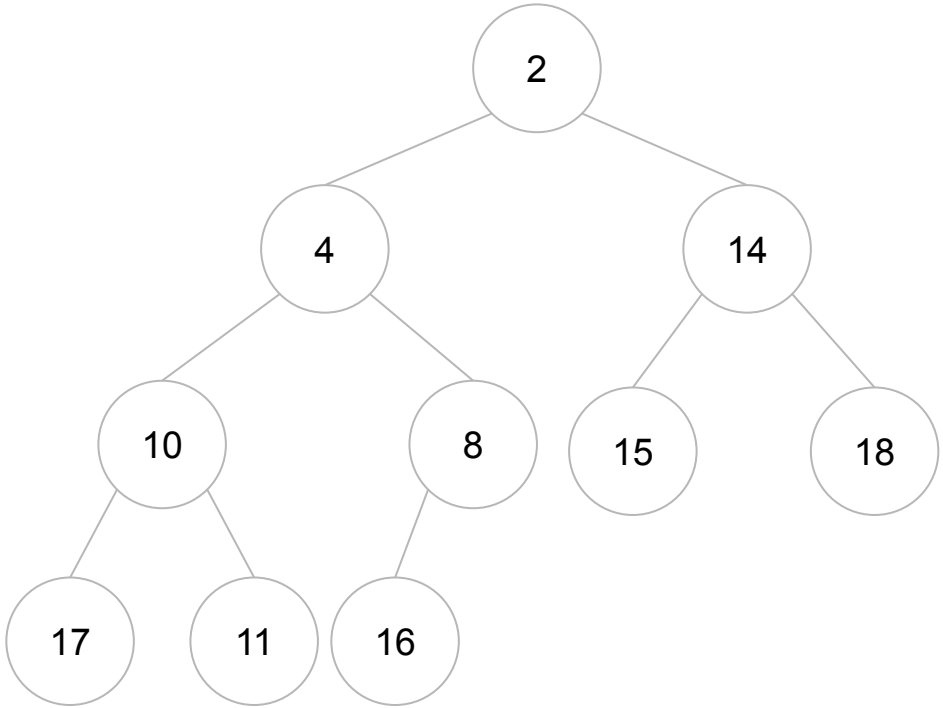
Nó contendo o número 2

Exemplo



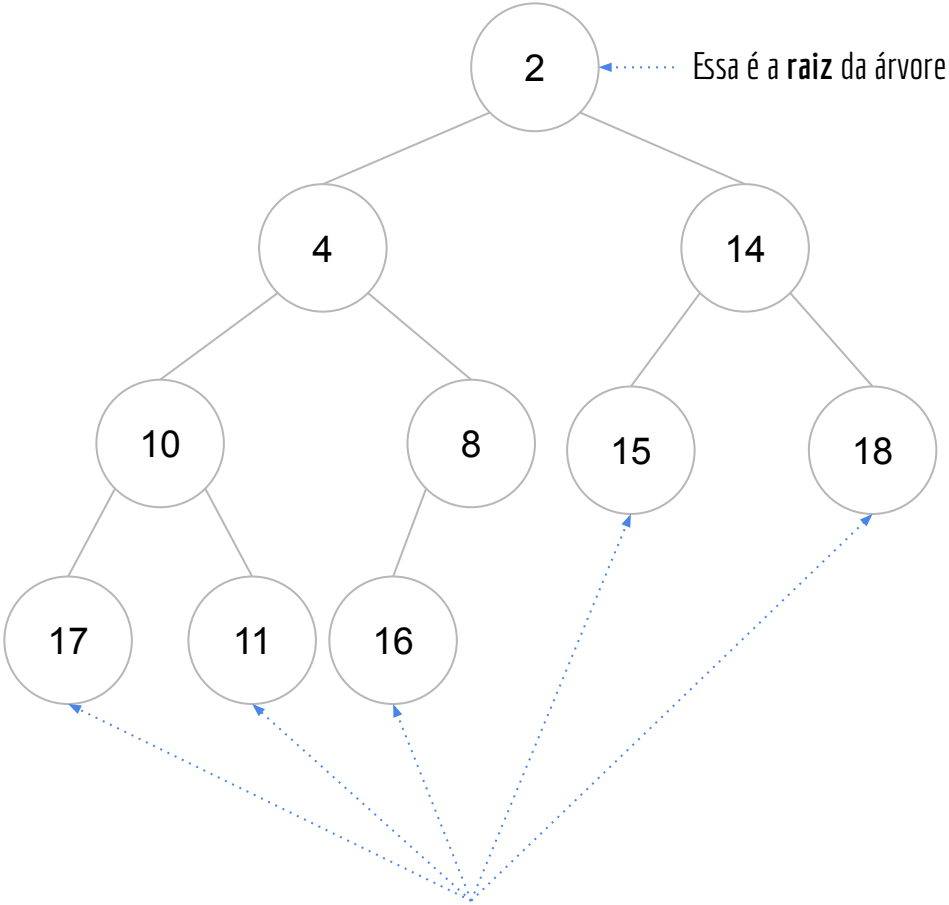
Nó que contém o 2 com dois filhos.
O **pai** do nó que possui o 4 (e o 14) é o nó que possui o 2.

Exemplo



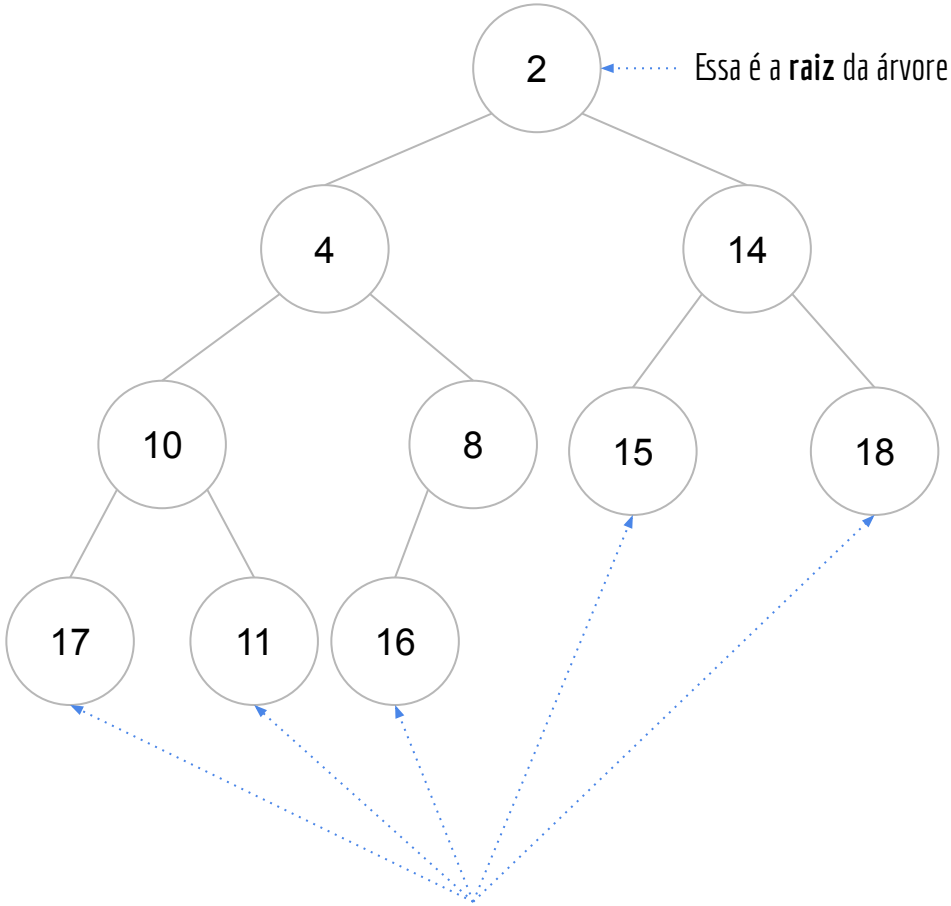
Podemos continuar adicionando filhos, sempre respeitando que cada nó pode possuir 0, 1 ou dois filhos.

Exemplo



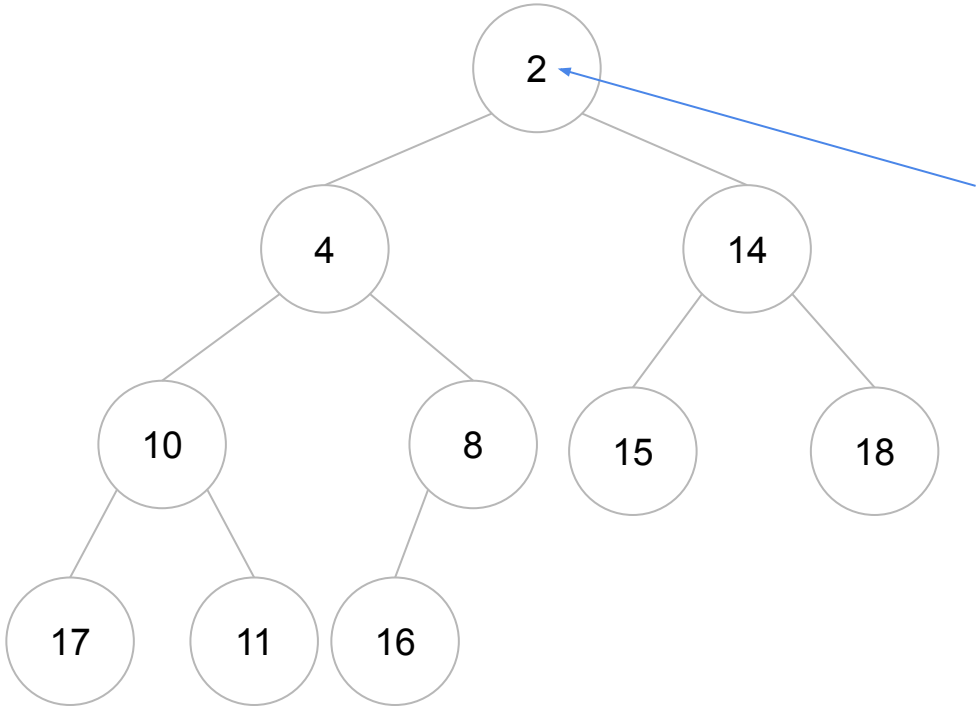
Nós sem filhos são **folhas**.

Exemplo



Nós sem filhos são **folhas**.

Chaves

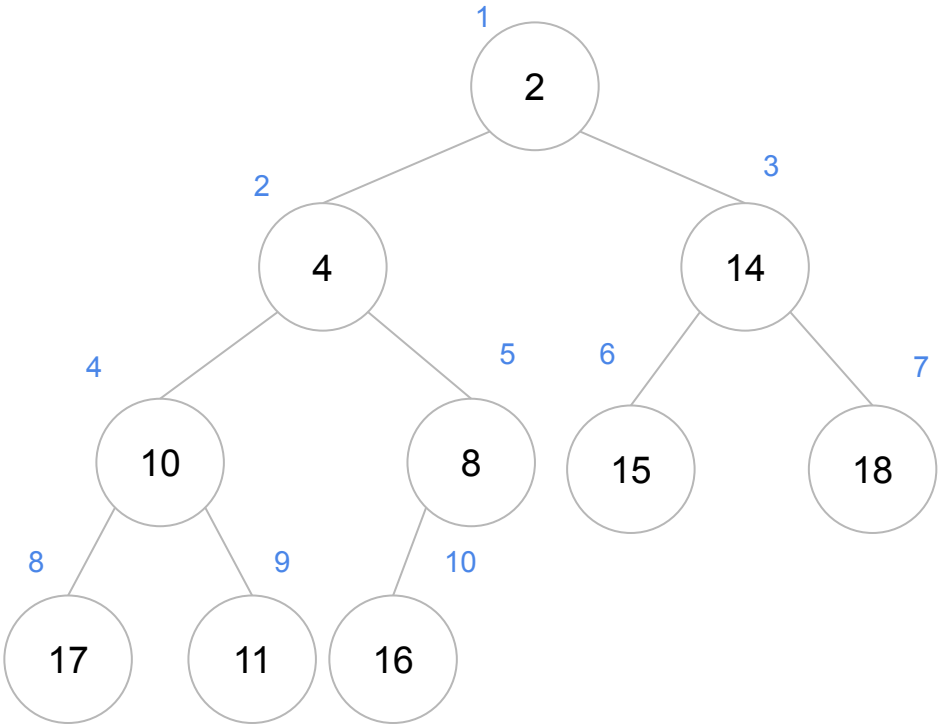


Em uma **heap**, o valor armazenado no nó é a sua **chave**.

Implementação

Vamos realizar a implementação da heap utilizando vetores.

Implementação



1	2	3	4	5	6	7	8	9	10
2	4	14	10	8	15	18	17	11	16

Implementação

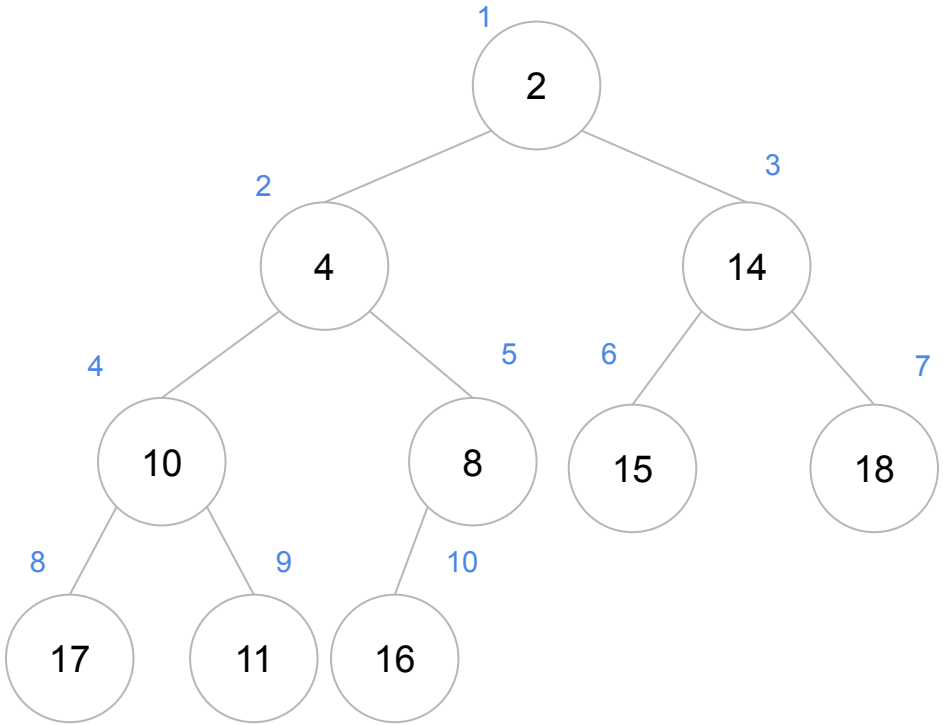
Pergunta

Como devem ser implementadas as funções a seguir?

função indicePai(i)
entrada: índice i do nó
saída: índice do pai desse nó.

função indiceFilhoEsquerdo(i)
entrada: índice i do nó
saída: índice do filho esquerdo desse nó.

função indiceFilhoDireito(i)
entrada: índice i do nó
saída: índice do filho direito desse nó.



1	2	3	4	5	6	7	8	9	10
2	4	14	10	8	15	18	17	11	16

Implementação

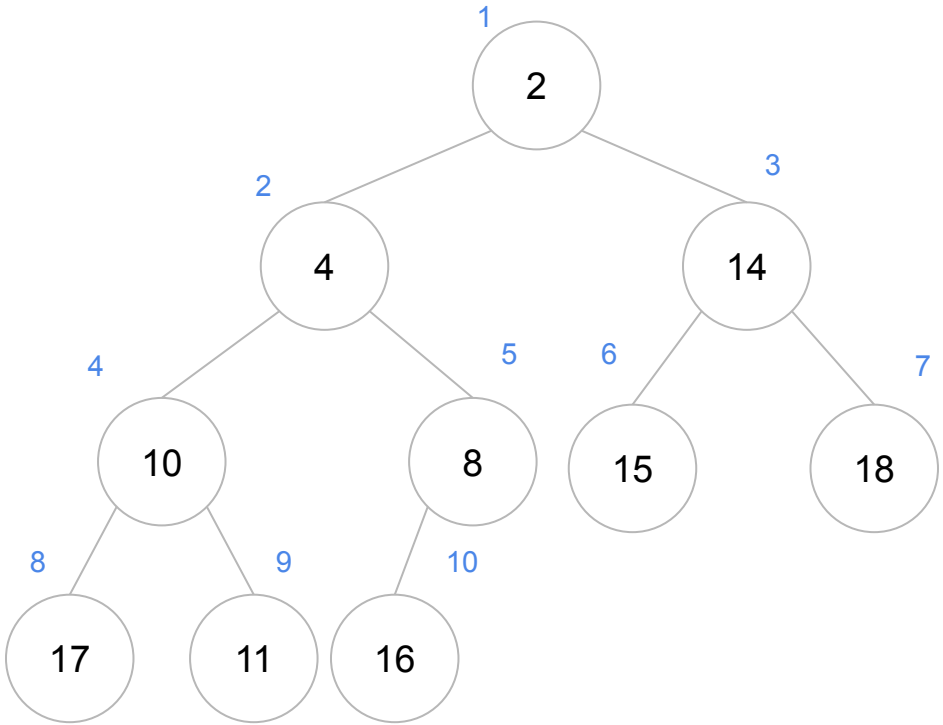
Pergunta

Como devem ser implementadas as funções a seguir?

função indicePai(i)
entrada: índice i do nó
saída: índice do pai desse nó.
retorne $\lfloor i/2 \rfloor$

função indiceFilhoEsquerdo(i)
entrada: índice i do nó
saída: índice do filho esquerdo desse nó.
retorne 2*i

função indiceFilhoDireito(i)
entrada: índice i do nó
saída: índice do filho direito desse nó.
retorne 2*i+1



1	2	3	4	5	6	7	8	9	10
2	4	14	10	8	15	18	17	11	16

Dica de programação

A maioria dos **hardwares** implementa funções de deslocamento de bits.

Devido a forma com que os números inteiros são representados na memória em binário, deslocar um número para a esquerda uma vez e adicionar um zero no final é o mesmo que multiplicá-lo por dois
Analogamente, deslocar para a direita é o mesmo que dividir por dois

Esse é o mesmo atalho que utilizamos para multiplicar ou dividir um número por 10

Dica de programação

Realizar um **deslocamento de bits é mais barato** do que realizar a divisão ou multiplicação por 2

Dica de programação

Realizar um **deslocamento de bits é mais barato** do que realizar a divisão ou multiplicação por 2

Em C, você pode utilizar os operadores » e « para realizar essas operações

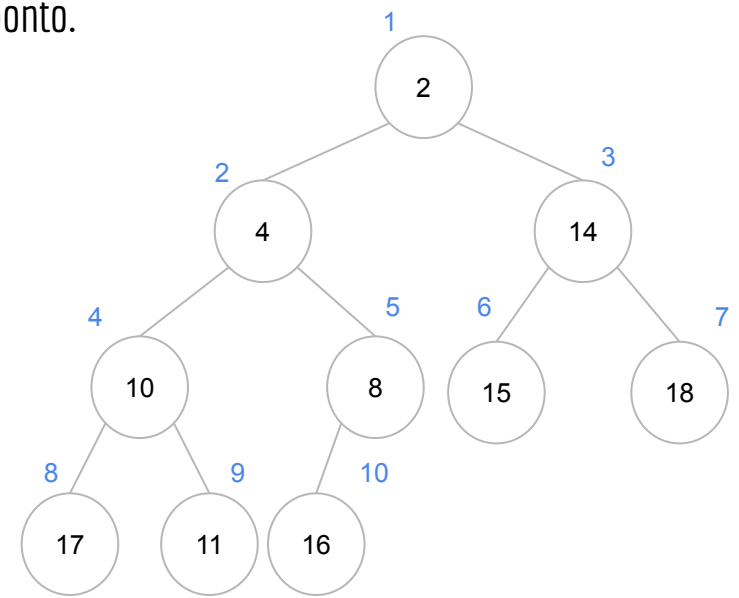
Exemplos

```
int valor1 = auxiliar << 1;//deslocar os bits de auxiliar 1x para a esquerda e armazenar em valor1
int valor2 = auxiliar >> 1;//deslocar os bits de auxiliar 1x para a direita e armazenar em valor2
```

Heaps

Uma heap é então uma árvore (que vamos armazenar em vetores) que possui as seguintes propriedades extras

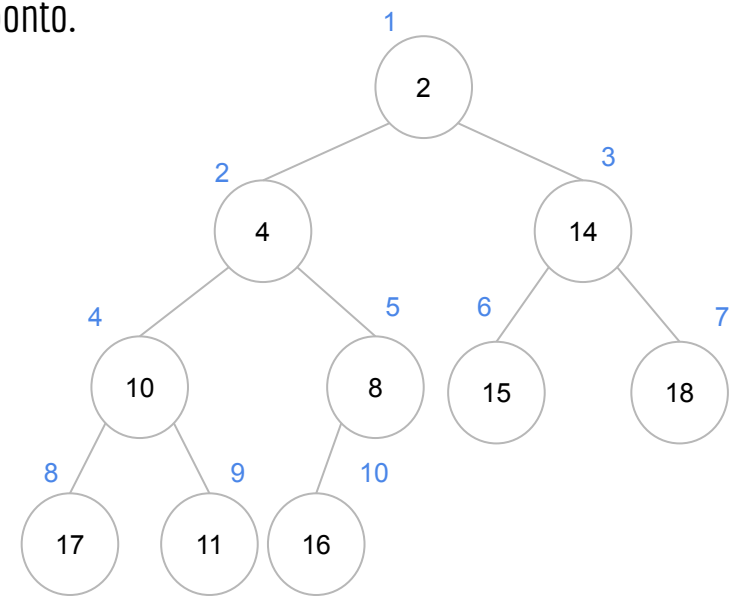
1. A árvore é **quase completa**, ou seja todos os níveis são completos, exceto o último, que pode ser preenchido da esquerda para a direita até um ponto.



Heaps

Uma heap é então uma árvore (que vamos armazenar em vetores) que possui as seguintes propriedades extras

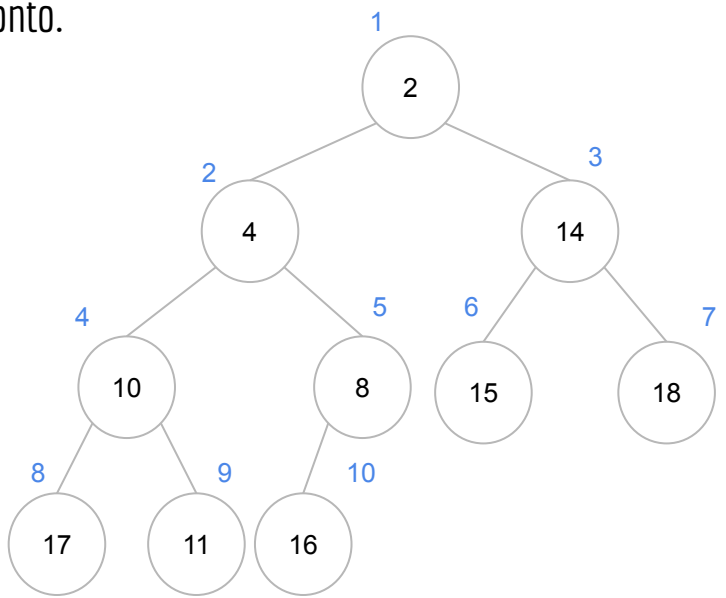
1. A árvore é **quase completa**, ou seja todos os níveis são completos, exceto o último, que pode ser preenchido da esquerda para a direita até um ponto.
2. Uma heap pode ser de máximo ou mínimo. Seja *h* a heap, e *h[i]* o elemento na posição $i \in \mathbb{N}$ da heap
    a. Uma ***heap de mínimo*** (***min-heap***) deve satisfazer a **propriedade da min-heap**
    $$h[pai(i)] \leq h[i], \ \forall i > 1$$



Heaps

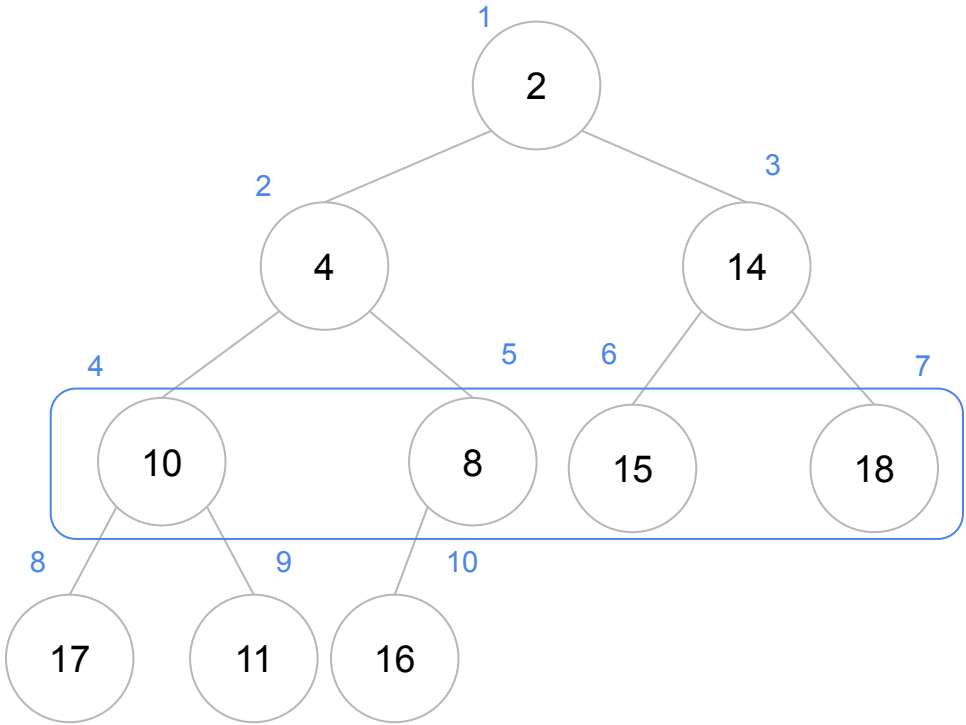
Uma heap é então uma árvore (que vamos armazenar em vetores) que possui as seguintes propriedades extras

1. A árvore é **quase completa**, ou seja todos os níveis são completos, exceto o último, que pode ser preenchido da esquerda para a direita até um ponto.
2. Uma heap pode ser de máximo ou mínimo. Seja *h* a heap, e *h[i]* o elemento na posição $i \in \mathbb{N}$ da heap
   a. Uma ***heap de mínimo*** (***min-heap***) deve satisfazer a **propriedade da min-heap**
   $$h[pai(i)] \leq h[i], \ \forall i > 1$$
   b. Uma ***heap de máximo*** (***max-heap***) deve satisfazer a **propriedade da max-heap**
   $$h[pai(i)] \geq h[i], \ \forall i > 1$$



Heap

Note que a definição nada diz sobre a ordem dos elementos em um mesmo nível da árvore



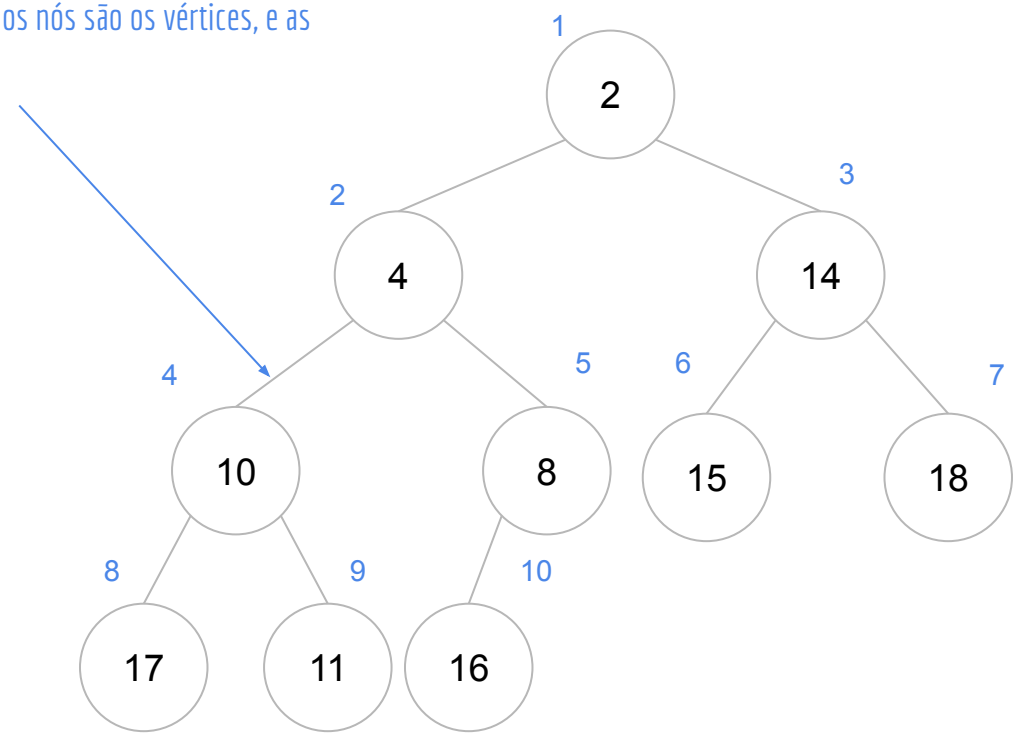
1	2	3	4	5	6	7	8	9	10
2	4	14	10	8	15	18	17	11	16

Altura

Os nós são ligados por **arestas** - na verdade a árvore é um **grafo conexo e acíclico**, onde os nós são os vértices, e as ligações são arestas.

A altura de uma heap é proporcional ao número de arestas percorridas para se chegar da raiz até a folha mais longínqua.

Para um heap de n nós, a altura é $\lfloor log_2 n \rfloor$



1	2	3	4	5	6	7	8	9	10
2	4	14	10	8	15	18	17	11	16

Usos

Heaps tem diversos usos

- Heapsort -› Algoritmo de ordenação (o que estamos interessados na disciplina)
- Filas de prioridade -› Selecione o elemento com a maior prioridade
- Acessar eficientemente o maior/menor elemento de uma coleção (depende se você usar min ou max-heap)
- Algoritmos de Grafos-› e.g., o algoritmo de Dijkstra

Max-Heap

A partir de agora vamos considerar apenas max-heaps

Vamos precisar delas para nossos algoritmos

O que será discutido pode ser trivialmente adaptado para min-heaps

Lembrando que a **propriedade da max-heap** é

$$h[pai(i)] \geq h[i], \ \forall i > 1$$

Max-Heapify

Para criar nossas heaps, vamos começar com o algoritmo Max-Heapify

Assuma que a heap é um vetor *h* de *n* posições, onde a primeira posição é a 1, ou seja, o vetor é indexado na forma *h[1..n]*

Max-Heapify

Para criar nossas heaps, vamos começar com o algoritmo Max-Heapify

Assuma que a heap é um vetor *h* de *n* posições, onde a primeira posição é a 1, ou seja, o vetor é indexado na forma *h[1..n]*

- esquerda(i) é a subárvore que começa no filho esquerdo do nó *i*

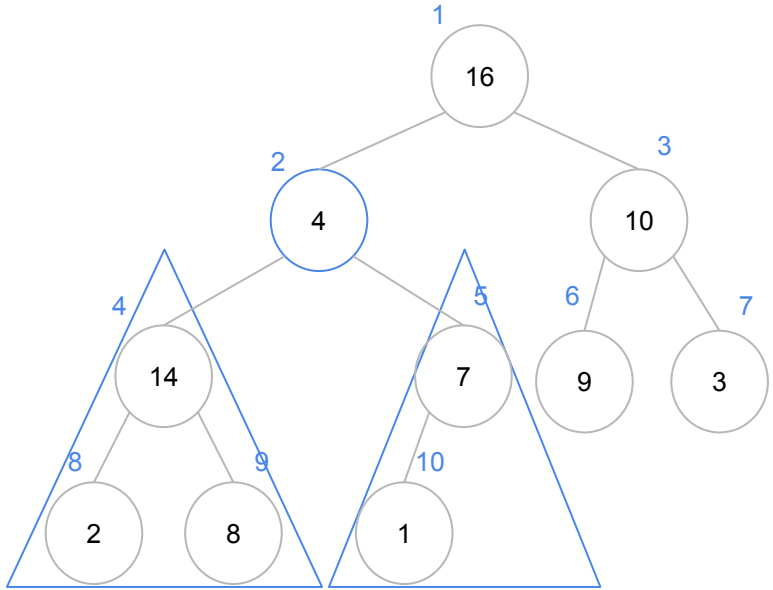
Max-Heapify

Para criar nossas heaps, vamos começar com o algoritmo Max-Heapify

Assuma que a heap é um vetor *h* de *n* posições, onde a primeira posição é a 1, ou seja, o vetor é indexado na forma *h[1..n]*

- esquerda(i) é a subárvore que começa no filho esquerdo do nó *i*
- direita(i) é a subárvore que começa no filho direito do nó *i*

Exemplo considerando i = 2

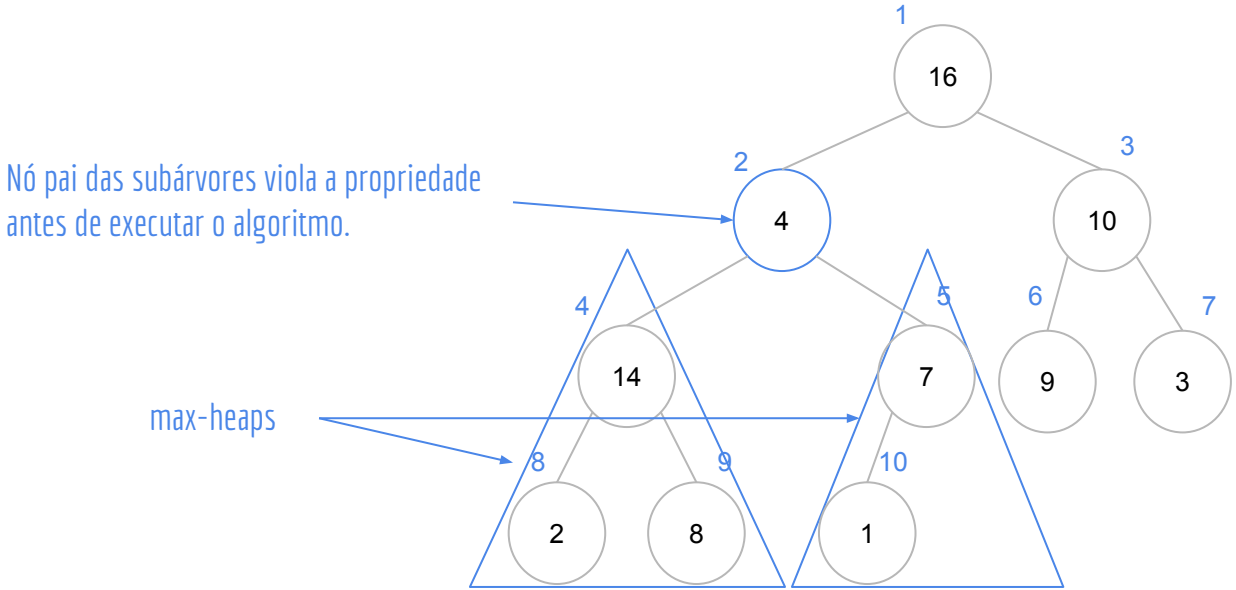


Max-Heapify

função max-heapify(h,i,n)
entrada: um vetor h indexado por h[1..n] e um valor i ∈ [1..n]. A árvore esquerda(i) e direita(i) são max-heaps. h[i] pode ser menor que seus filhos.
saída: a subárvore com raiz em i é modificada de forma que a propriedade da max-heap seja satisfeita

Max-Heapify

função max-heapify(h,i,n)
entrada: um vetor h indexado por h[1..n] e um valor i ∈ [1..n]. A árvore esquerda(i) e direita(i) são max-heaps. h[i] pode ser menor que seus filhos.
saída: a subárvore com raiz em i é modificada de forma que a propriedade da max-heap seja satisfeita



Max-Heapify

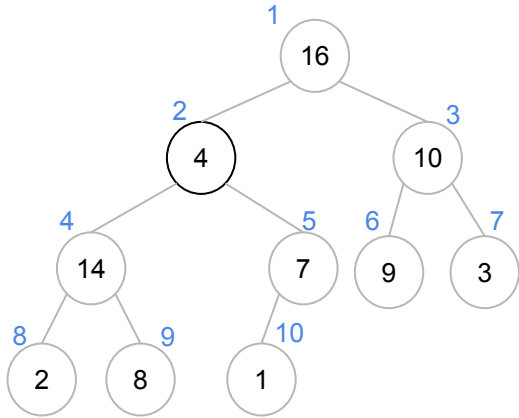
função max-heapify(h,i,n)
entrada: um vetor h indexado por h[1..n] e um valor i ∈ [1..n]. A árvore esquerda(i) e direita(i) são max-heaps. h[i] pode ser menor que seus filhos.
saída: a subárvore com raiz em i é modificada de forma que a propriedade da max-heap seja satisfeita

```
l ← esquerda(i)
r ← direita(i)
se l ≤ n e h[l] > h[i]
     maior ← l
senão
     maior ← i
se r ≤ n e h[r] > h[maior]
     maior ← r
se maior ≠ i
     trocar(h,i,maior)
     max-heapify(h,maior,n)
```

Teste de mesa

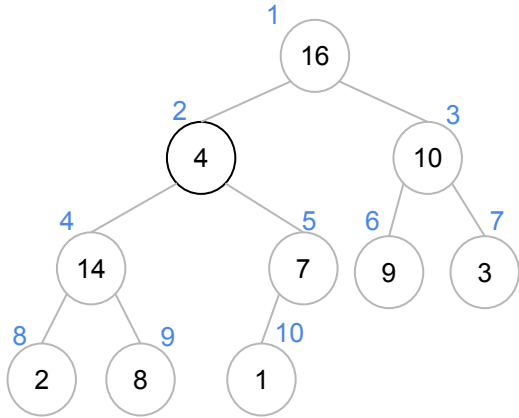
Chamada max-heapfy(h,2,10)

```
função max-heapify(h,i,n)
l ← esquerda(i)
r ← direita(i)
se l ≤ n e h[l] > h[i]
      maior ← l
senão
      maior ← i
se r ≤ n e h[r] > h[maior]
      maior ← r
se maior ≠ i
      trocar(h,i,maior)
      max-heapify(h,maior,n)
```



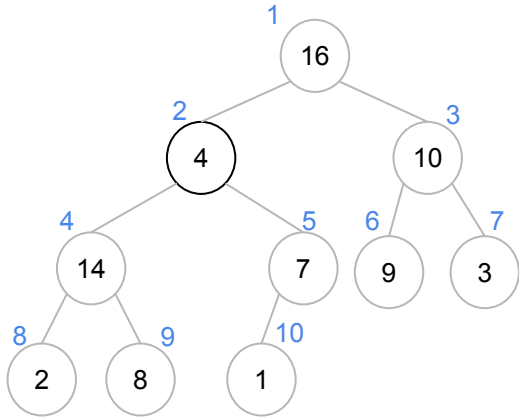
max-heapify				
i	n	l	r	maior
2	10			

```
função max-heapify(h,i,n)
l ← esquerda(i)
r ← direita(i)
se l ≤ n e h[l] > h[i]
      maior ← l
senão
      maior ← i
se r ≤ n e h[r] > h[maior]
      maior ← r
se maior ≠ i
      trocar(h,i,maior)
      max-heapify(h,maior,n)
```



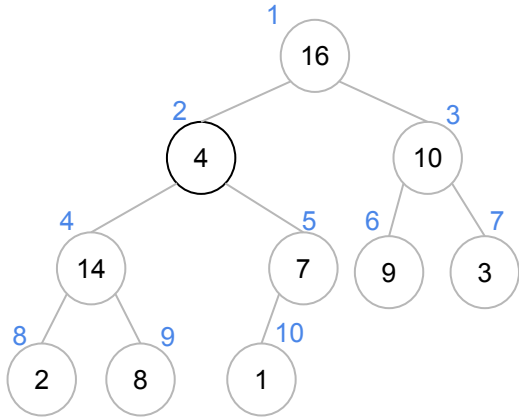
max-heapify				
i	n	l	r	maior
2	10	4		

```
função max-heapify(h,i,n)
l ← esquerda(i)
r ← direita(i)
se l ≤ n e h[l] > h[i]
      maior ← l
senão
      maior ← i
se r ≤ n e h[r] > h[maior]
      maior ← r
se maior ≠ i
      trocar(h,i,maior)
      max-heapify(h,maior,n)
```



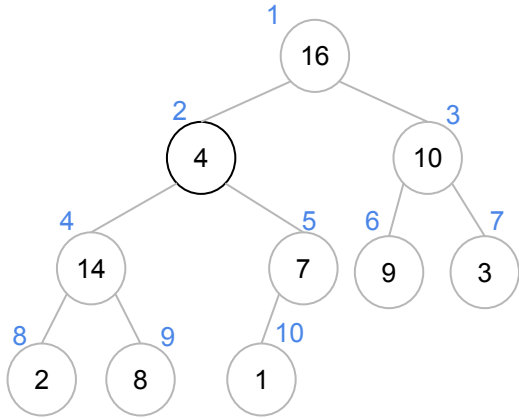
max-heapify				
i	n	l	r	maior
2	10	4	5	

```
função max-heapify(h,i,n)
l ← esquerda(i)
r ← direita(i)
se l ≤ n e h[l] > h[i]
      maior ← l
senão
      maior ← i
se r ≤ n e h[r] > h[maior]
      maior ← r
se maior ≠ i
      trocar(h,i,maior)
      max-heapify(h,maior,n)
```



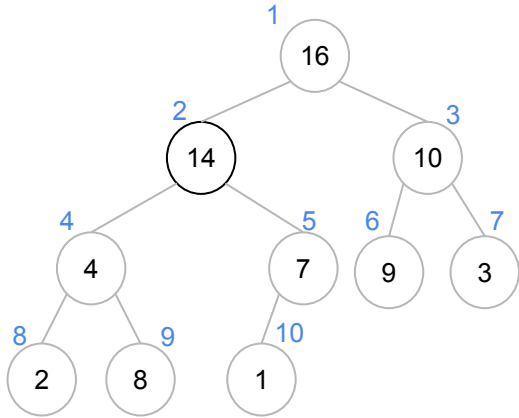
max-heapify				
i	n	l	r	maior
2	10	4	5	4

```
função max-heapify(h,i,n)
l ← esquerda(i)
r ← direita(i)
se l ≤ n e h[l] > h[i]
      maior ← l
senão
      maior ← i
se r ≤ n e h[r] > h[maior]
      maior ← r
se maior ≠ i
      trocar(h,i,maior)
      max-heapify(h,maior,n)
```



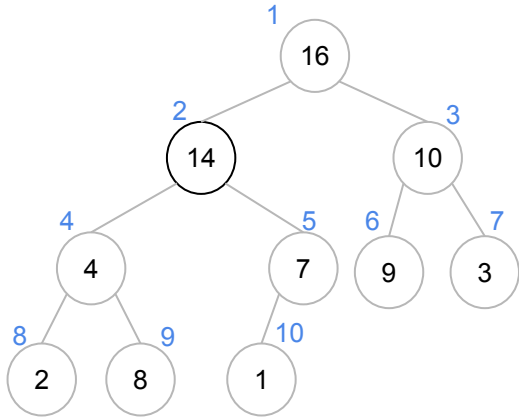
max-heapify				
i	n	l	r	maior
2	10	4	5	4

```
função max-heapify(h,i,n)
l ← esquerda(i)
r ← direita(i)
se l ≤ n e h[l] > h[i]
      maior ← l
senão
      maior ← i
se r ≤ n e h[r] > h[maior]
      maior ← r
se maior ≠ i
      trocar(h,i,maior)
      max-heapify(h,maior,n)
```



max-heapify				
i	n	l	r	maior
2	10	4	5	4

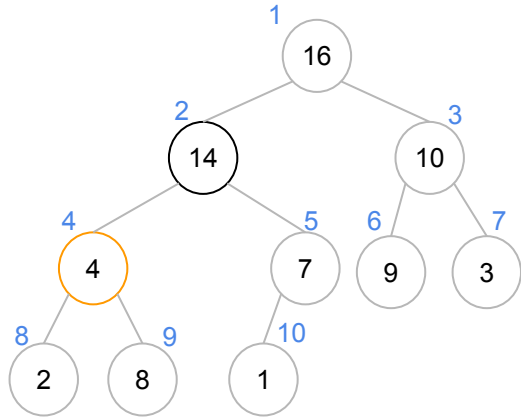
```
função max-heapify(h,i,n)
l ← esquerda(i)
r ← direita(i)
se l ≤ n e h[l] > h[i]
      maior ← l
senão
      maior ← i
se r ≤ n e h[r] > h[maior]
      maior ← r
se maior ≠ i
      trocar(h,i,maior)
      max-heapify(h,maior,n)
```



max-heapify				
i	n	l	r	maior
2	10	4	5	4

max-heapify				
i	n	l	r	maior
4	10			

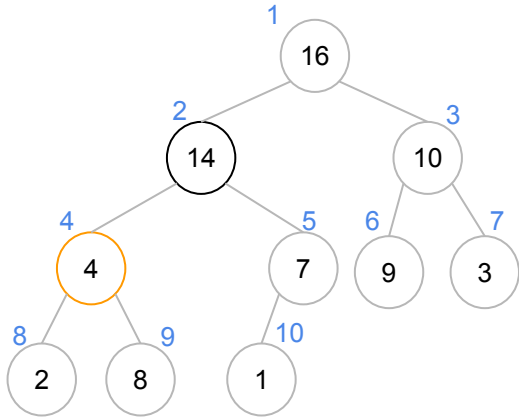
```
função max-heapify(h,i,n)
l ← esquerda(i)
r ← direita(i)
se l ≤ n e h[l] > h[i]
      maior ← l
senão
      maior ← i
se r ≤ n e h[r] > h[maior]
      maior ← r
se maior ≠ i
      trocar(h,i,maior)
      max-heapify(h,maior,n)
```



max-heapify				
i	n	l	r	maior
2	10	4	5	4

max-heapify				
i	n	l	r	maior
4	10	8		

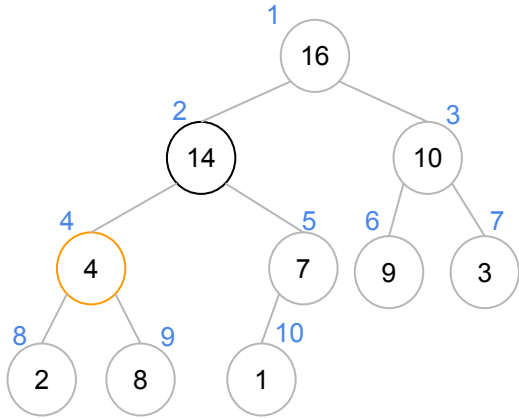
```
função max-heapify(h,i,n)
l ← esquerda(i)
r ← direita(i)
se l ≤ n e h[l] > h[i]
      maior ← l
senão
      maior ← i
se r ≤ n e h[r] > h[maior]
      maior ← r
se maior ≠ i
      trocar(h,i,maior)
      max-heapify(h,maior,n)
```



max-heapify				
i	n	l	r	maior
2	10	4	5	4

max-heapify				
i	n	l	r	maior
4	10	8	9	

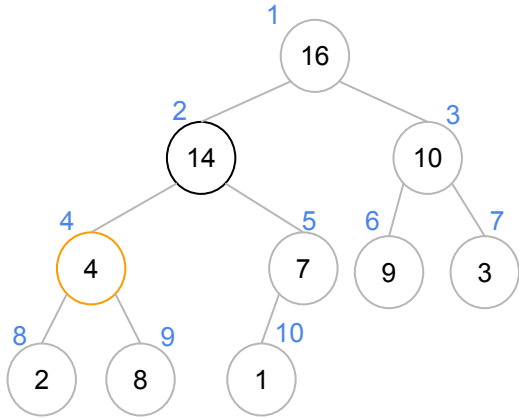
```
função max-heapify(h,i,n)
l ← esquerda(i)
r ← direita(i)
se l ≤ n e h[l] > h[i]
      maior ← l
senão
      maior ← i
se r ≤ n e h[r] > h[maior]
      maior ← r
se maior ≠ i
      trocar(h,i,maior)
      max-heapify(h,maior,n)
```



max-heapify				
i	n	l	r	maior
2	10	4	5	4

max-heapify				
i	n	l	r	maior
4	10	8	9	4

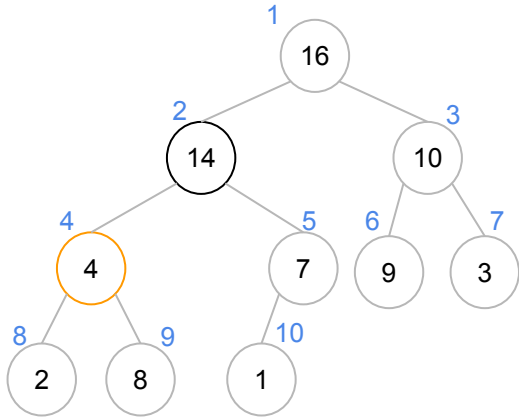
```
função max-heapify(h,i,n)
l ← esquerda(i)
r ← direita(i)
se l ≤ n e h[l] > h[i]
      maior ← l
senão
      maior ← i
se r ≤ n e h[r] > h[maior]
      maior ← r
se maior ≠ i
      trocar(h,i,maior)
      max-heapify(h,maior,n)
```



max-heapify				
i	n	l	r	maior
2	10	4	5	4

max-heapify				
i	n	l	r	maior
4	10	8	9	9

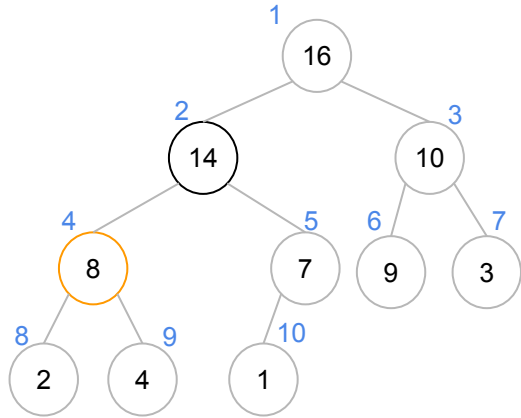
```
função max-heapify(h,i,n)
l ← esquerda(i)
r ← direita(i)
se l ≤ n e h[l] > h[i]
      maior ← l
senão
      maior ← i
se r ≤ n e h[r] > h[maior]
      maior ← r
se maior ≠ i
      trocar(h,i,maior)
      max-heapify(h,maior,n)
```



max-heapify				
i	n	l	r	maior
2	10	4	5	4

max-heapify				
i	n	l	r	maior
4	10	8	9	9

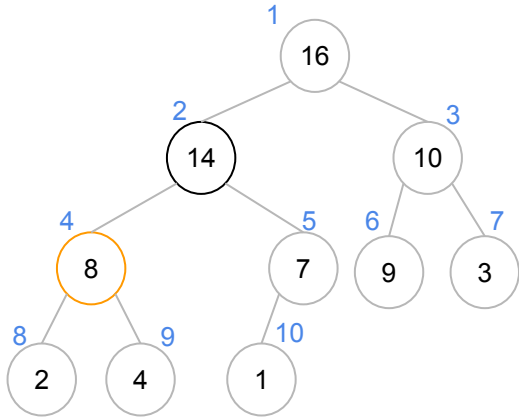
```
função max-heapify(h,i,n)
l ← esquerda(i)
r ← direita(i)
se l ≤ n e h[l] > h[i]
      maior ← l
senão
      maior ← i
se r ≤ n e h[r] > h[maior]
      maior ← r
se maior ≠ i
      trocar(h,i,maior)
      max-heapify(h,maior,n)
```



max-heapify				
i	n	l	r	maior
2	10	4	5	4

max-heapify				
i	n	l	r	maior
4	10	8	9	9

```
função max-heapify(h,i,n)
l ← esquerda(i)
r ← direita(i)
se l ≤ n e h[l] > h[i]
      maior ← l
senão
      maior ← i
se r ≤ n e h[r] > h[maior]
      maior ← r
se maior ≠ i
      trocar(h,i,maior)
      max-heapify(h,maior,n)
```

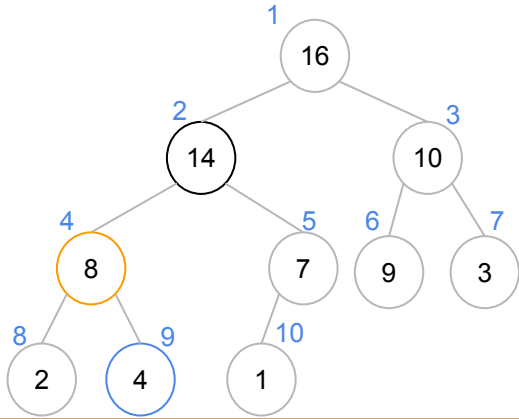


max-heapify				
i	n	l	r	maior
2	10	4	5	4

max-heapify				
i	n	l	r	maior
4	10	8	9	9

max-heapify				
i	n	l	r	maior
9	10			

```
função max-heapify(h,i,n)
l ← esquerda(i)
r ← direita(i)
se l ≤ n e h[l] > h[i]
      maior ← l
senão
      maior ← i
se r ≤ n e h[r] > h[maior]
      maior ← r
se maior ≠ i
      trocar(h,i,maior)
      max-heapify(h,maior,n)
```

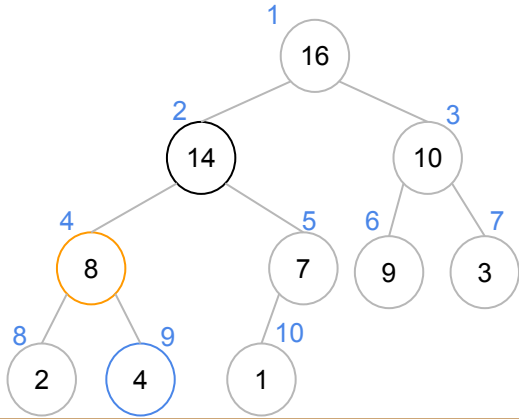


max-heapify				
i	n	l	r	maior
2	10	4	5	4

max-heapify				
i	n	l	r	maior
4	10	8	9	9

max-heapify				
i	n	l	r	maior
9	10	18		

```
função max-heapify(h,i,n)
l ← esquerda(i)
r ← direita(i)
se l ≤ n e h[l] > h[i]
      maior ← l
senão
      maior ← i
se r ≤ n e h[r] > h[maior]
      maior ← r
se maior ≠ i
      trocar(h,i,maior)
      max-heapify(h,maior,n)
```

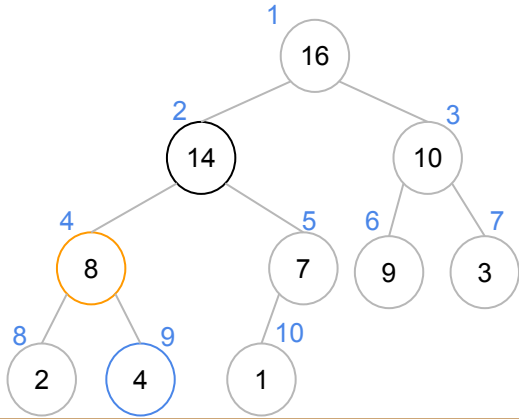


max-heapify				
i	n	l	r	maior
2	10	4	5	4

max-heapify				
i	n	l	r	maior
4	10	8	9	9

max-heapify				
i	n	l	r	maior
9	10	18	19	

```
função max-heapify(h,i,n)
l ← esquerda(i)
r ← direita(i)
se l ≤ n e h[l] > h[i]
      maior ← l
senão
      maior ← i
se r ≤ n e h[r] > h[maior]
      maior ← r
se maior ≠ i
      trocar(h,i,maior)
      max-heapify(h,maior,n)
```

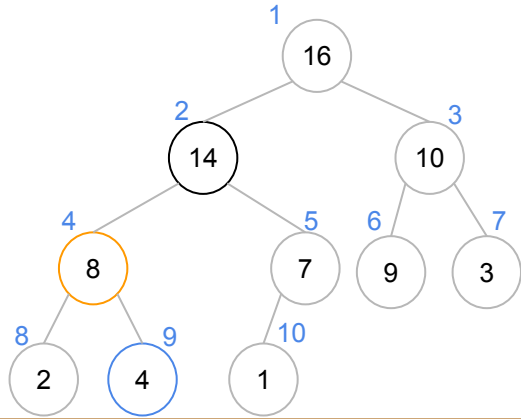


max-heapify				
i	n	l	r	maior
2	10	4	5	4

max-heapify				
i	n	l	r	maior
4	10	8	9	9

max-heapify				
i	n	l	r	maior
9	10	18	19	9

```
função max-heapify(h,i,n)
l ← esquerda(i)
r ← direita(i)
se l ≤ n e h[l] > h[i]
      maior ← l
senão
      maior ← i
se r ≤ n e h[r] > h[maior]
      maior ← r
se maior ≠ i
      trocar(h,i,maior)
      max-heapify(h,maior,n)
```

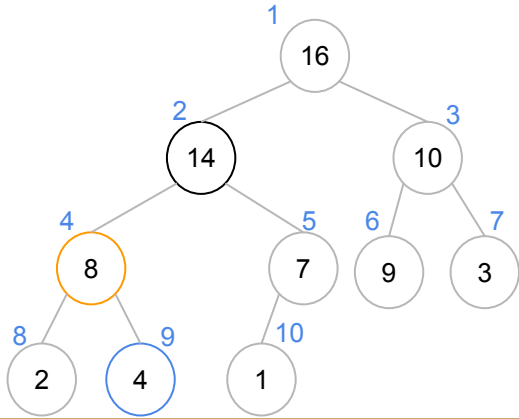


max-heapify				
i	n	l	r	maior
2	10	4	5	4

max-heapify				
i	n	l	r	maior
4	10	8	9	9

max-heapify				
i	n	l	r	maior
9	10	18	19	9

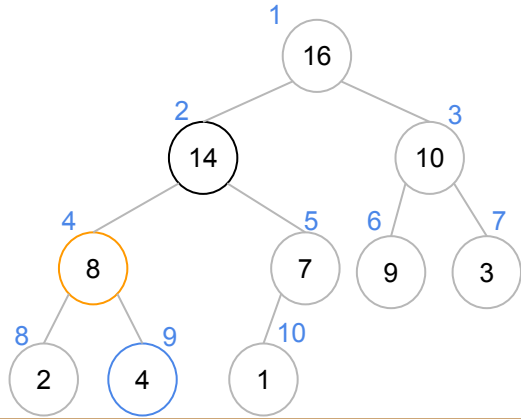
```
função max-heapify(h,i,n)
l ← esquerda(i)
r ← direita(i)
se l ≤ n e h[l] > h[i]
      maior ← l
senão
      maior ← i
se r ≤ n e h[r] > h[maior]
      maior ← r
se maior ≠ i
      trocar(h,i,maior)
      max-heapify(h,maior,n)
```



max-heapify				
i	n	l	r	maior
2	10	4	5	4

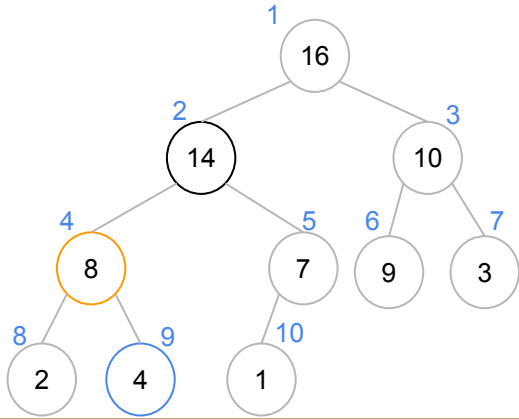
max-heapify				
i	n	l	r	maior
4	10	8	9	9

```
função max-heapify(h,i,n)
l ← esquerda(i)
r ← direita(i)
se l ≤ n e h[l] > h[i]
      maior ← l
senão
      maior ← i
se r ≤ n e h[r] > h[maior]
      maior ← r
se maior ≠ i
      trocar(h,i,maior)
      max-heapify(h,maior,n)
```



max-heapify				
i	n	l	r	maior
2	10	4	5	4

```
função max-heapify(h,i,n)
l ← esquerda(i)
r ← direita(i)
se l ≤ n e h[l] > h[i]
      maior ← l
senão
      maior ← i
se r ≤ n e h[r] > h[maior]
      maior ← r
se maior ≠ i
      trocar(h,i,maior)
      max-heapify(h,maior,n)
```



Max-Heapify

Note que a cada chamada recursiva, o valor “desliza” um nível de cada vez pela árvore até sua posição correta

Análise

Considerando o número de comparações entre elementos do vetor

```
função max-heapify(h,i,n)
l ← esquerda(i)
r ← direita(i)
se l ≤ n e h[l] > h[i]
      maior ← l
senão
      maior ← i
se r ≤ n e h[r] > h[maior]
      maior ← r
se maior ≠ i
      trocar(h,i,maior)
      max-heapify(h,maior,n)
```

Análise

Considerando o número de comparações entre elementos do vetor

No melhor caso, nenhuma chamada recursiva é feita, nenhuma comparação é realizada

$$C^{-}(n) = 0$$

```
função max-heapify(h,i,n)
l ← esquerda(i)
r ← direita(i)
se l ≤ n e h[l] > h[i]
      maior ← l
senão
      maior ← i
se r ≤ n e h[r] > h[maior]
      maior ← r
se maior ≠ i
      trocar(h,i,maior)
      max-heapify(h,maior,n)
```

Análise

Considerando o número de comparações entre elementos do vetor

No melhor caso, nenhuma chamada recursiva é feita, nenhuma comparação é realizada

$$C^{-}(n) = 0$$

```
função max-heapify(h,i,n)
l ← esquerda(i)
r ← direita(i)
se l ≤ n e h[l] > h[i]
      maior ← l
senão
      maior ← i
se r ≤ n e h[r] > h[maior]
      maior ← r
se maior ≠ i
      trocar(h,i,maior)
      max-heapify(h,maior,n)
```

Importante: estamos assumindo um curto circuito nos ifs, onde a caso a primeira comparação seja falsa, a segunda não é realizada. Caso contrário, teríamos duas comparações.

Análise

Considerando o número de comparações entre elementos do vetor

No pior caso, para facilitar, vamos definir o custo de acordo com a altura *a* da árvore

No caso base, a altura da árvore é 0

Chamamos max-heapify para uma folha, ou *maior = i*

```
função max-heapify(h,i,n)
l ← esquerda(i)
r ← direita(i)
se l ≤ n e h[l] > h[i]
      maior ← l
senão
      maior ← i
se r ≤ n e h[r] > h[maior]
      maior ← r
se maior ≠ i
      trocar(h,i,maior)
      max-heapify(h,maior,n)
```

Análise

$$C^+(a) = \begin{cases} 0, & \text{se } a = 0, \\ \ldots, & \text{se } a \geq 1 \end{cases}$$

```
função max-heapify(h,i,n)
l ← esquerda(i)
r ← direita(i)
se l ≤ n e h[l] > h[i]
      maior ← l
senão
      maior ← i
se r ≤ n e h[r] > h[maior]
      maior ← r
se maior ≠ i
      trocar(h,i,maior)
      max-heapify(h,maior,n)
```

Análise

$$C^+(a) = \begin{cases} 0, & \text{se } a = 0, \\ C^+(a-1) + 2, & \text{se } a \geq 1 \end{cases}$$

```
função max-heapify(h,i,n)
l ← esquerda(i)
r ← direita(i)
se l ≤ n e h[l] > h[i]
      maior ← l
senão
      maior ← i
se r ≤ n e h[r] > h[maior]
      maior ← r
se maior ≠ i
      trocar(h,i,maior)
      max-heapify(h,maior,n)
```

Note que a chamada recursiva, *maior* sempre está exatamente um nível abaixo de *i*. Logo, cada chamada recursiva opera em *h-1*.

Análise

$$C^+(a) = \begin{cases} 0, & \text{se } a = 0, \\ C^+(a-1) + 2, & \text{se } a \geq 1 \end{cases}$$

Resolvendo, temos que

$$C^+(a) = 2a$$

Análise

$$C^+(a) = \begin{cases} 0, & \text{se } a = 0, \\ C^+(a-1) + 2, & \text{se } a \geq 1 \end{cases}$$

Resolvendo, temos que

$$C^+(a) = 2a$$

Sabendo-se que a altura é definida como

$$a = \lfloor \log_2 n \rfloor$$

Vamos assumir que a árvore é completa. Isso vai facilitar as contas, e a diferença no resultado é desprezível.

Substituindo, o custo de max-heapify para uma heap de tamanho n é

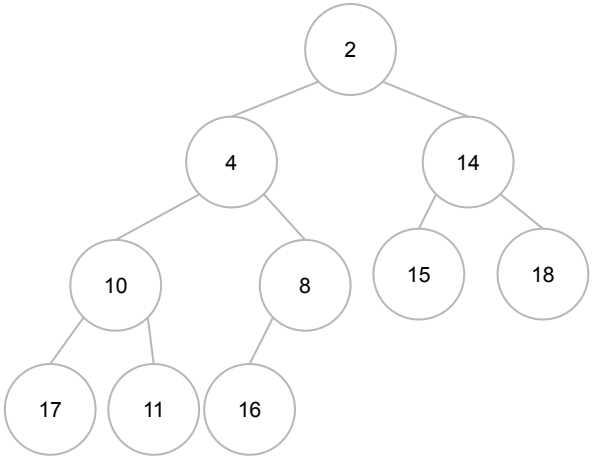
$$C^+(n) = 2\lfloor \log_2 n \rfloor$$

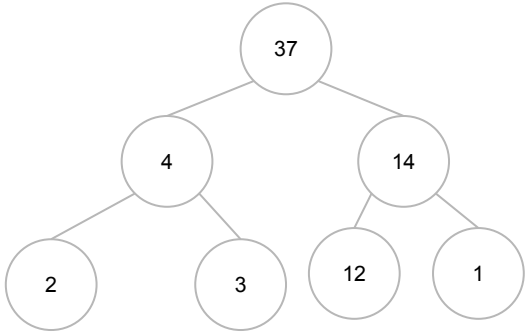
Próxima aula

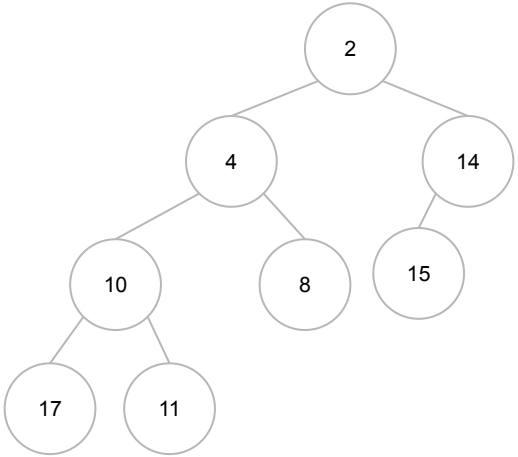
Na próxima aula o max-heapify será utilizado para construir uma heap, e como base para o heapsort.

Exercícios

1. Marque as árvores a seguir como max-heap, min-heap ou nenhum caso a árvore não possa se uma heap. Para árvores não heaps, informe o motivo da árvore não ser uma heap. Para uma heap, informe a altura.







Exercícios

2. Em uma max-heap, onde está o seu maior elemento (chave)? E em uma min-heap?

3. Represente a árvore onde foi feito o teste de mesa em um vetor, e execute o teste de mesa novamente, mas dessa vez usando o vetor.

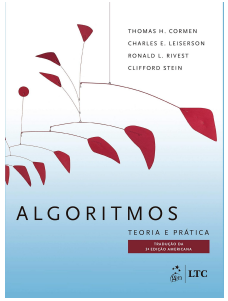
4. Resolva a recorrência dada em aula.

5. Implemente o max-heapify em C.

Referências

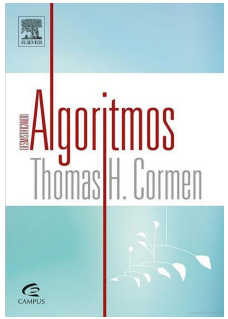
Renato Carmo. Algoritmos e Estruturas de Dados. www.inf.ufpr.br/renato

T. Cormen, C. Leiserson, R. Rivest, C. Stein. Algoritmos: Teoria e Prática. 3a ed. 2012

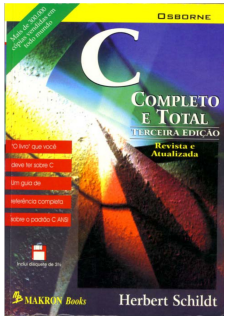


R. Sedgewick, K. Wayne. Algorithms Part I. 4a ed. 2014

T. Cormen. Desmistificando algoritmos. 2017.



H. Schildt. C completo e total. 1996



Licença