Universidade Federal de Ouro Preto
Campus João Monlevade

CSI 488 – Algoritmos e Estruturas de Dados I

TAD – Fila de Prioridade, Heap e HeapSort

Prof. Mateus Ferreira Satler

Índice

1. Fila de Prioridade

- Como visto anteriormente, as **Filas** são estruturas de dados do tipo *FIFO – First In, First Out*, ou seja:
  - Um novo elemento da fila somente pode ser inserido na última posição (fim da fila).
  - Um elemento só pode ser removido da primeira posição (inicio da fila).

1. Fila de Prioridade

- Em algumas aplicações, é necessário remover da Fila o elemento de **maior** prioridade.

- Uma Fila de Prioridade é uma estrutura de dado que mantém uma coleção de elementos, cada um com uma prioridade associada.

  - Elemento: par (chave, informação)
    - A chave especifica o nível de prioridade.

1. Fila de Prioridade

- Uma fila de pacientes esperando transplante de fígado em geral é uma fila de prioridade.

- Em Sistemas Operacionais, um exemplo é a fila de prioridade de processos aguardando o processador para execução.
  - Os processos mais prioritários são executados antes dos outros.

- Sistemas de gerência de memória usam a técnica de substituir a página menos utilizada na memória principal por uma nova página.

1. Fila de Prioridade

- Principais Operações:

1. **Inserir** um elemento novo na fila de prioridade.
2. **Remover** o elemento de <u>maior prioridade</u> da fila de prioridade.

1. Fila de Prioridade

- Uma maneira de representar uma fila de prioridade é manter uma lista linear ligada ou encadeada em que os elementos estão **ordenados por prioridades decrescentes**.

- Assim:
    - Para remover um elemento da fila de prioridade, basta remover o primeiro elemento: tempo constante O(1).
    - Para inserir um novo elemento, o pior caso é O(n), onde n é o número de elementos na fila de prioridade.

1. Fila de Prioridade

- Outra maneira muito simples é armazenar de forma aleatória os elementos em uma lista linear sequencial, sem nenhuma ordem.

    - Para inserir um novo elemento, basta inserir em qualquer lugar, por exemplo no final da lista: Tempo O(1).

    - Para remover um elemento da fila de prioridade é preciso percorrer a lista para obter o elemento com a maior prioridade.
        - Remove-se este elemento, colocando no seu lugar um outro qualquer, por exemplo aquele no final da lista.
        - Tempo O(n), onde n é o número de elementos na fila.

1. Fila de Prioridade

- Em resumo:
  - Usando uma fila ordenada:
    - Inserção é O(n)
    - Remoção é O(1)
  - Usando uma fila não-ordenada:
    - Inserção é O(1)
    - Remoção é O(n)
- Portanto uma abordagem mais rápida precisa ser pensada quando grandes conjuntos de dados são considerados.

2. Heap

- **Heap** é uma estrutura de **árvore binária completa** em que cada nó terminal ou não-folha tem uma **prioridade maior ou igual** à prioridade de seus filhos.
  - Uma árvore binária diz-se **completa** quando os seus níveis estão cheios, com possível exceção do último, o qual está preenchido da esquerda para a direita até um certo ponto.
  - É importante ressaltar que **maior prioridade** pode significar **menor valor da chave** ou **maior valor de chave**.
    - Quanto menor a chave, maior a prioridade; ou
    - Quanto maior a chave, maior a prioridade.

2. Heap

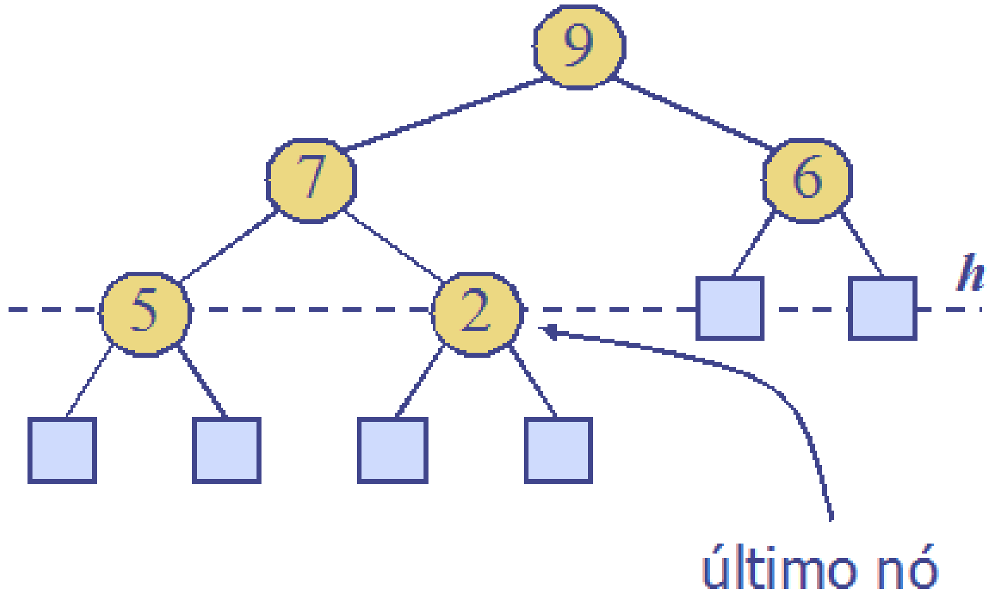
- Em outras palavras, um **Heap** é uma árvore binária que satisfaz as propriedades

  - **Ordem:** para cada nó **v**, exceto o nó raiz, tem-se que:
    - `prioridade(v) ≤ prioridade(pai(v))`

  - **Completude:** é completa, isto é, se `h` é a altura:
    - `Todo nó folha está no nível h ou h-1`
    - `O nível h-1 está totalmente preenchido`
    - `As folhas do nível h estão todas mais a esquerda`

2. Heap

- Exemplo:

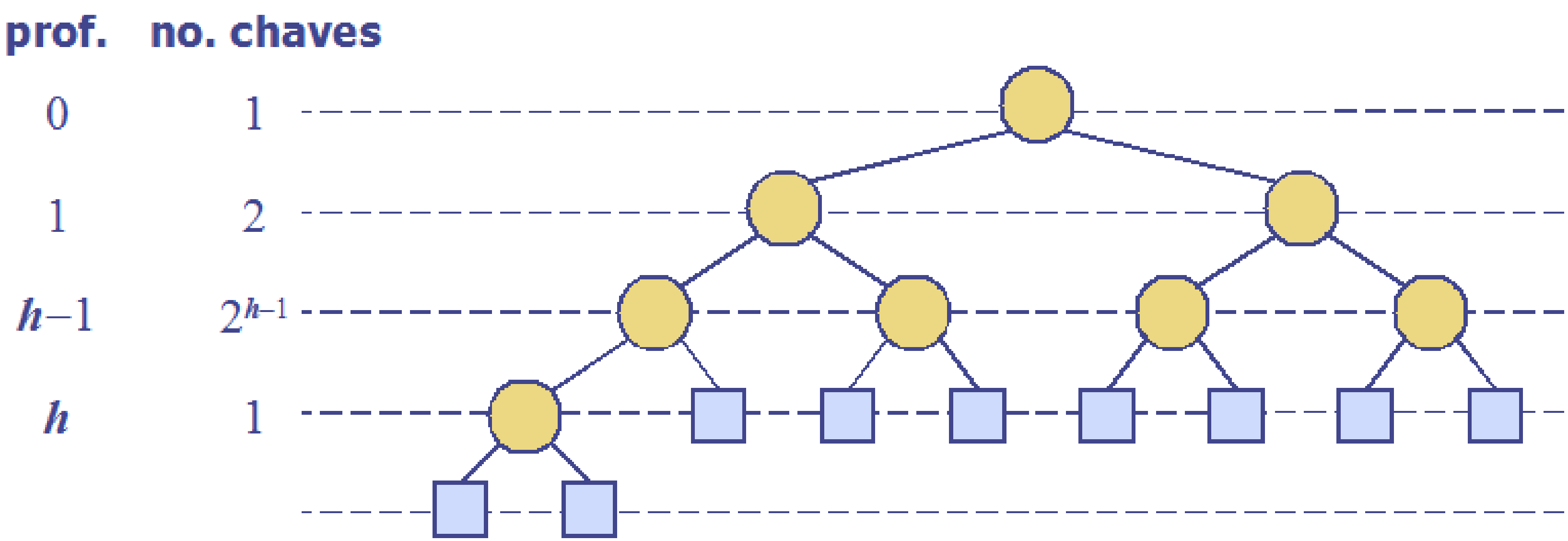


  - **Último nó:** nó interno mais à direita de profundidade **h**

  - Importante notar que **NÃO É** uma Árvore Binária de Busca:
    - Os dados estão bem menos estruturados, pois há o interesse apenas no elemento de **maior prioridade.**

2. Heap

- Um **Heap** armazenando **n** nós possui altura **h** de ordem **O(log n)**.

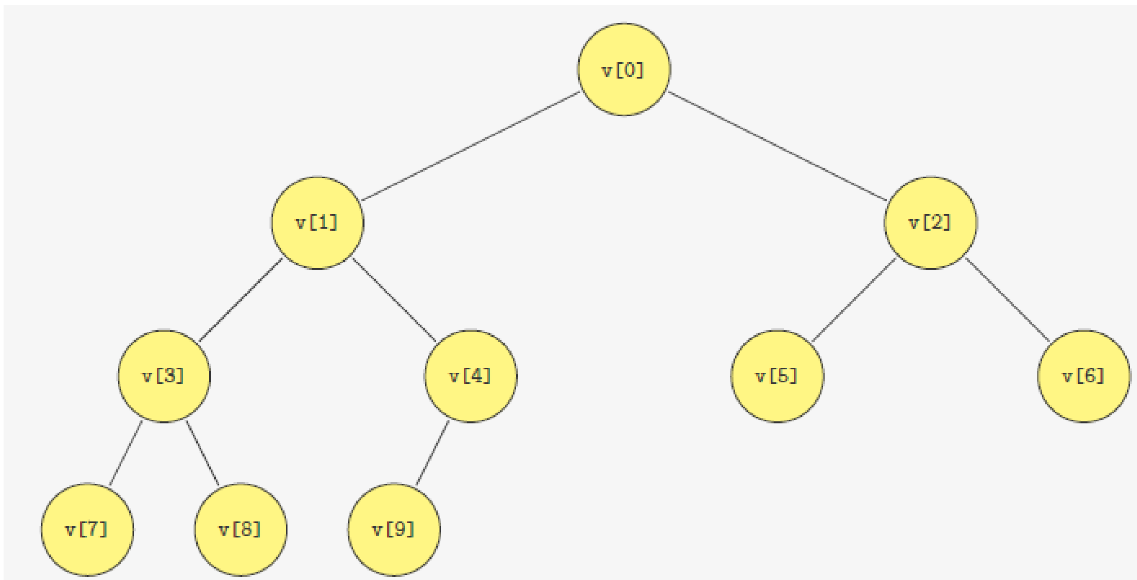


2. Heap

- Filas de Prioridade com Heaps:
  - Armazena-se um elemento (chave, informação) em cada nó.
  - Mantém-se o controle sobre a localização do último nó.
  - Remove-se sempre o Item armazenado na raiz, devido à propriedade de ordem do heap.
    - Heap mínimo: menor chave na raiz do heap
    - Heap máximo: maior chave na raiz do heap

2. Heap

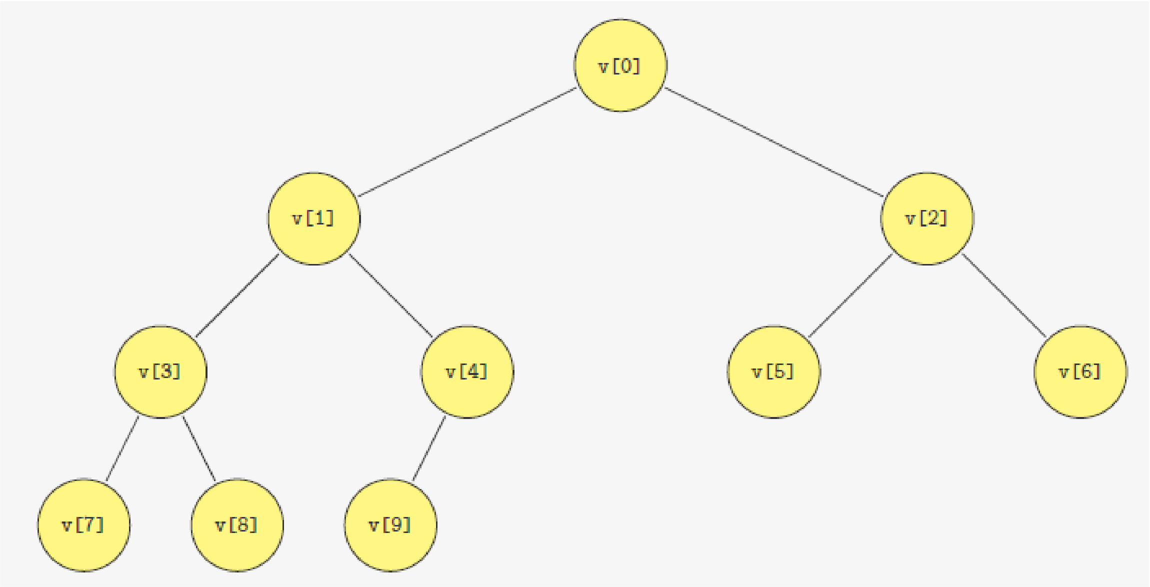
- Um **Heap** tem uma representação muito simples e interessante utilizando vetores.



	0	1	2	3	4	5	6	7	8	9	...
V											

2. Heap

- Um **Heap** tem uma representação muito simples e interessante utilizando vetores.



  - Em relação a **`v[i]`**:
    - O filho esquerdo é **`v[2*i+1]`** e o filho direito é **`v[2*i+2]`**
    - O pai é **`v[(i-1)/2]`**.

2. Heap

- **Implementação:**

```c
typedef struct Item_Est {
   char nome [20];
   int chave;
} Item;

typedef struct FP_Est {
   Item * v;
   int n, tamanho;
} FP;

typedef FP * p_fp;
```

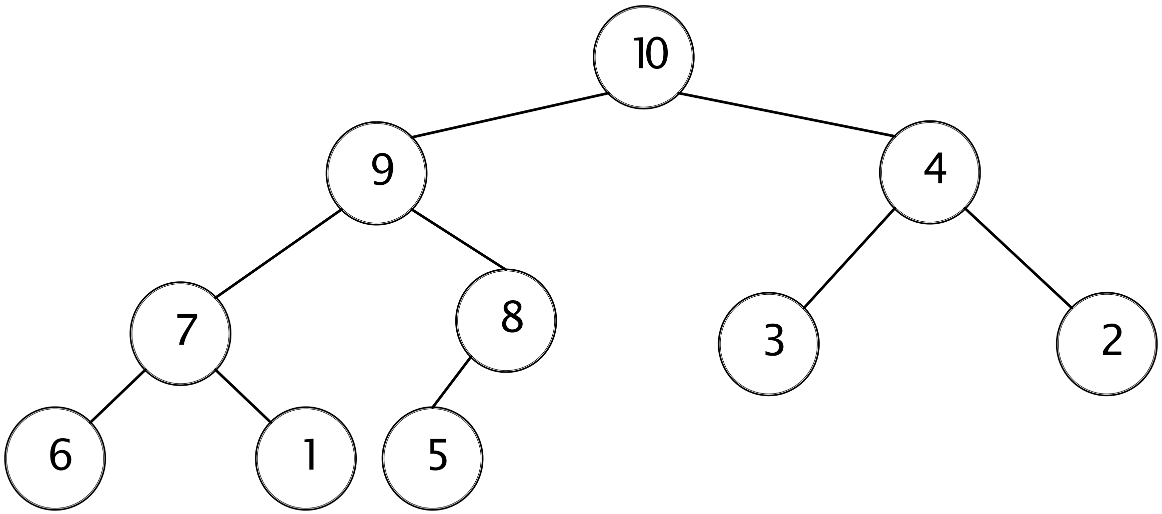
2. Heap

- **Implementação:**

```
p_fp criar_filaprio (int tam) {
    p_fp fprio = malloc ( sizeof(FP) );
    fprio->v = malloc (tam * sizeof(Item));
    fprio->n = 0;
    fprio->tamanho = tam;
    return fprio;
}
```

2. Heap

- Em um **Heap** (de máximo):
  - As chaves dos filhos são menores ou iguais à do pai, ou seja, a raiz possui a maior chave.



	0	1	2	3	4	5	6	7	8	9	...
V	10	9	4	7	8	3	2	6	1	5	

2.1. Heap – Inserção

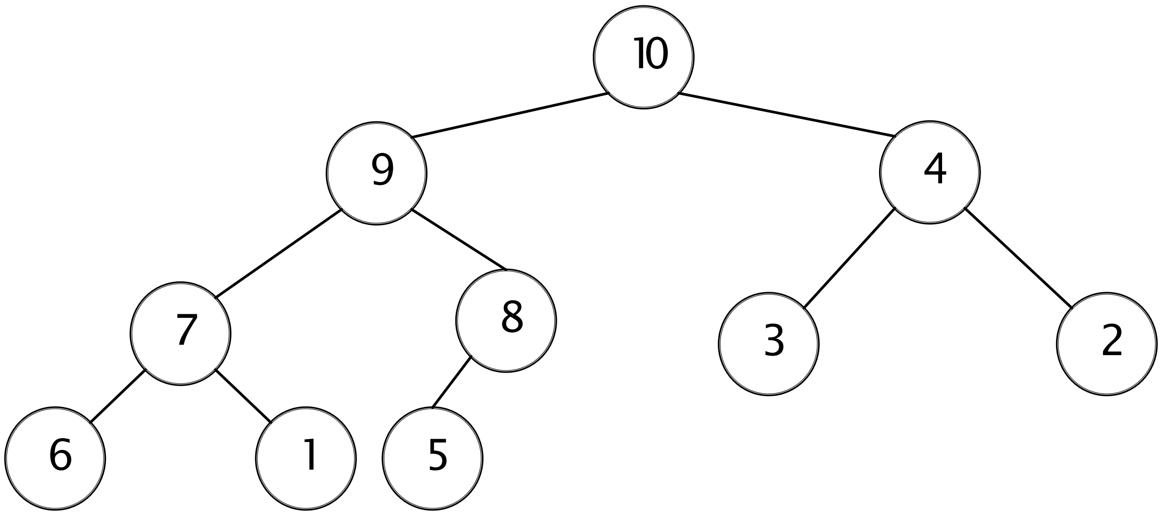
- Corresponde à inserção de um Item no Heap.

- O algoritmo consiste de 3 passos:

1. Encontrar e criar nó de inserção (que passará a ser o último nó).
2. Armazenar o Item com a chave neste nó.
3. Restaurar ordem do Heap.
   - Após a inserção de um novo Item, a propriedade de ordem do Heap pode ser violada.
   - A ordem do Heap é restaurada trocando os itens caminho acima a partir do nó de inserção.
   - Se o Item tiver maior prioridade que seu pai, então os dois trocam de lugar. Essa operação é repetida até que o Item encontre o seu lugar correto na árvore.

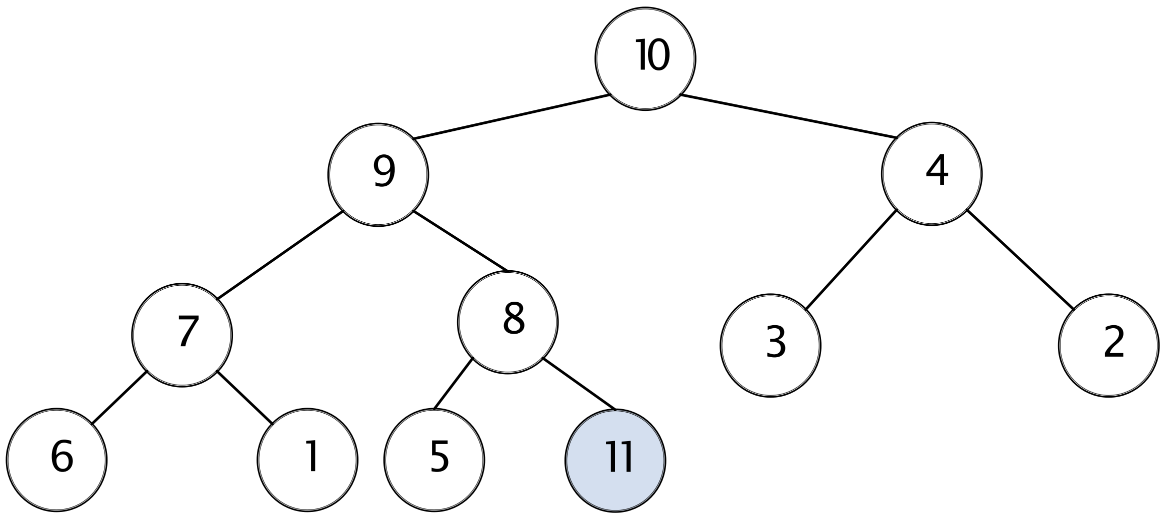
2.1. Heap – Inserção

- Exemplo: inserir o valor 11



2.1. Heap – Inserção

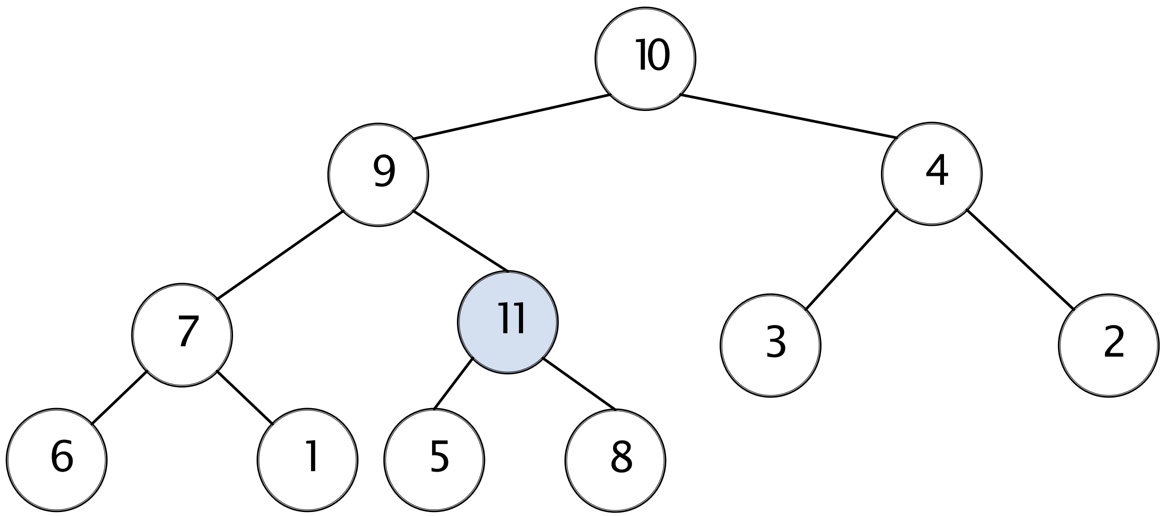
- Exemplo: inserir o valor 11



  - Se o Item tiver prioridade maior que o pai, troca de posição.

2.1. Heap – Inserção

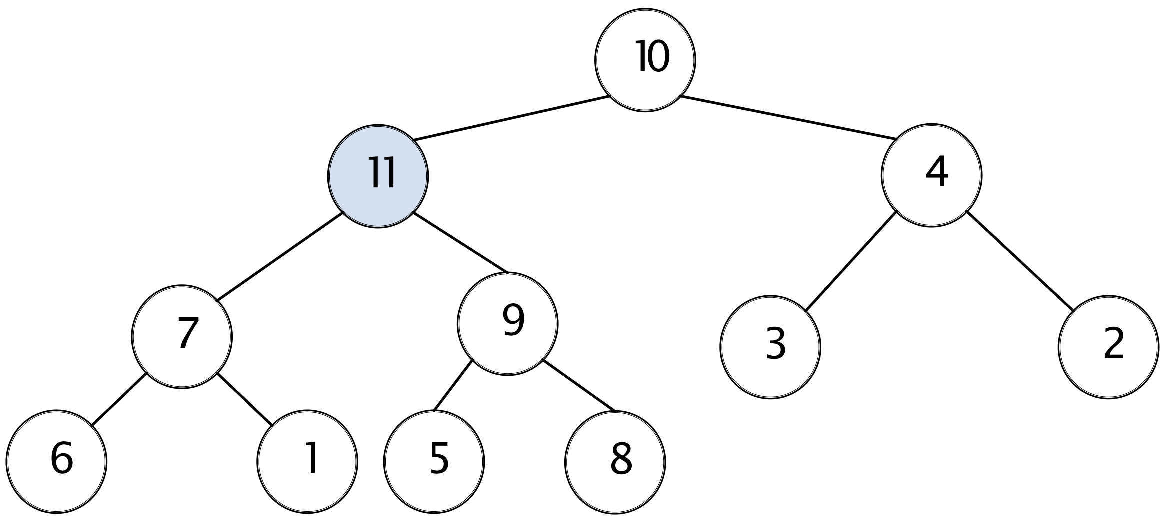
- Exemplo: inserir o valor 11



  - Se o Item tiver prioridade maior que o pai, troca de posição.

2.1. Heap – Inserção

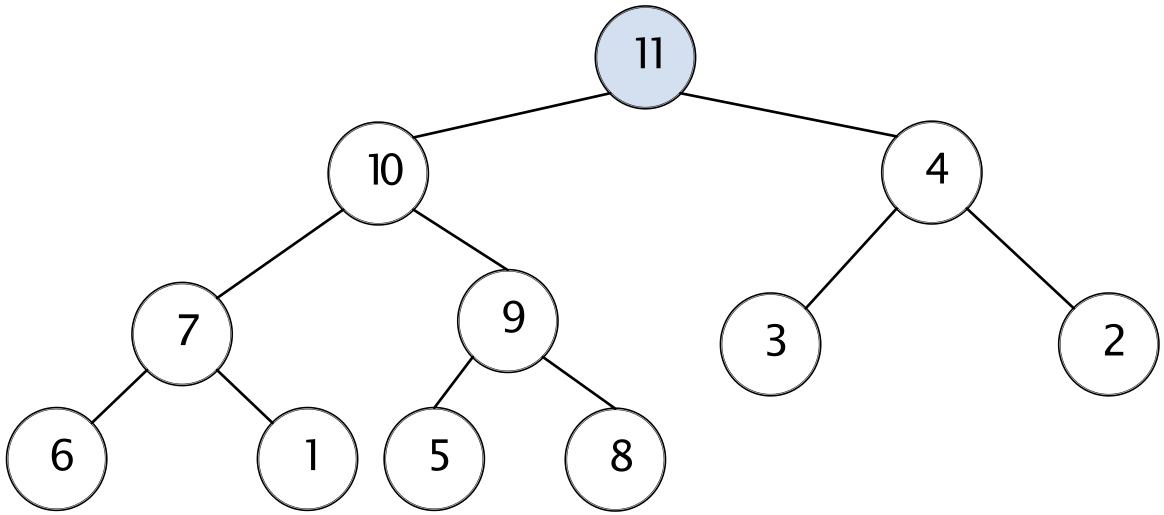
- Exemplo: inserir o valor 11



  - Se o Item tiver prioridade maior que o pai, troca de posição.

2.1. Heap – Inserção

- Exemplo: inserir o valor 11



  - Se o Item tiver prioridade maior que o pai, troca de posição.

2.1. Heap – Inserção

- Implementação:

```
void insere (p_fp fprio , Item item) {
   fprio->v[fprio->n] = item;
   fprio->n++;
   sobe_no_heap (fprio , fprio->n-1);
}

#define PAI(i) ((i-1)/2)

void sobe_no_heap (p_fp fprio , int k) {
   if (k > 0 && fprio->v[PAI(k)].chave < fprio->v[k].chave) {
      Item temp;
      temp = fprio->v[PAI(k)];
      fprio->v[PAI(k)] = fprio->v[k];
      fprio->v[k] = temp;
      sobe_no_heap (fprio , PAI(k));
   }
}
```

2.2. Heap – Remoção

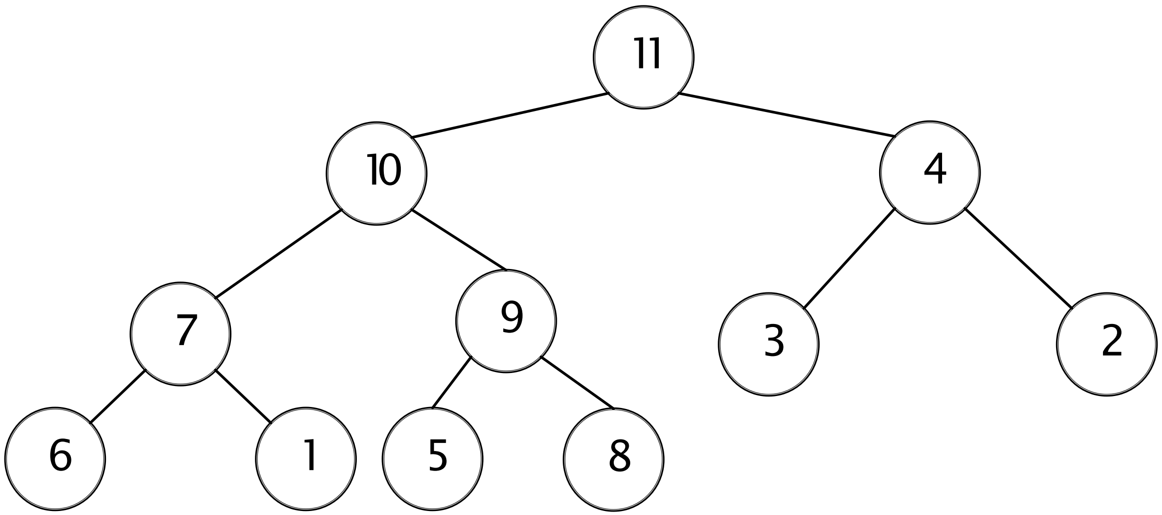
- Corresponde à remoção do Item da raiz (maior prioridade).

- O algoritmo consiste de 3 passos:

1. Armazenar o conteúdo do nó raiz do Heap (para retorno).
2. Copiar o conteúdo do último nó para raiz, e posteriormente remover o último nó.
3. Restaurar ordem do Heap.
   - Utilizar o último nó para substituição da raiz na remoção possui várias vantagens, entre elas, completude garantida e implementação em tempo constante.
   - Após a remoção, a propriedade de ordem do Heap pode ser violada.
   - A ordem do Heap é restaurada trocando os itens caminho abaixo a partir da raiz.

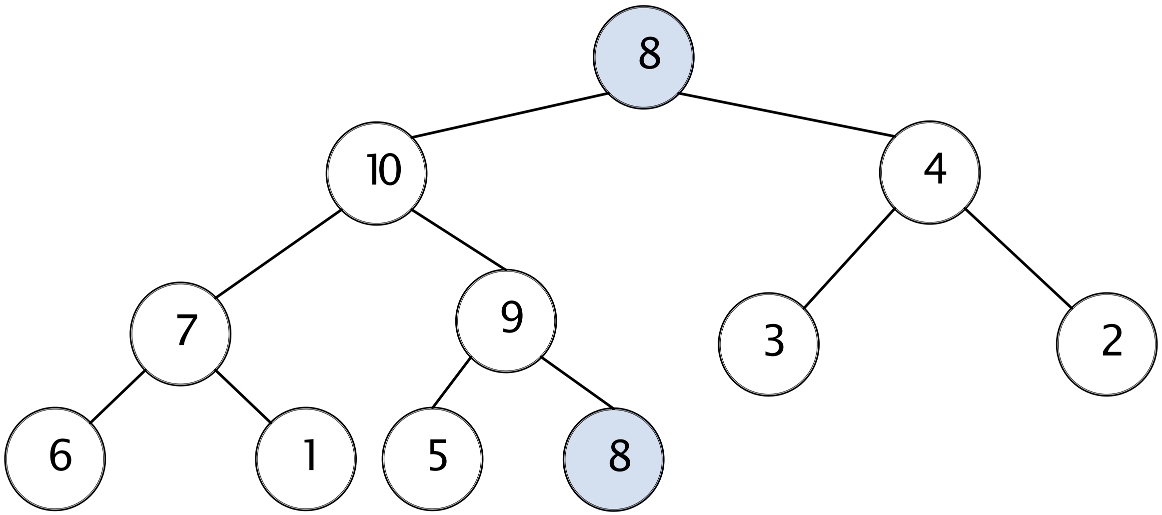
2.2. Heap – Remoção

- Exemplo: remover o Item de maior prioridade 11



2.2. Heap – Remoção

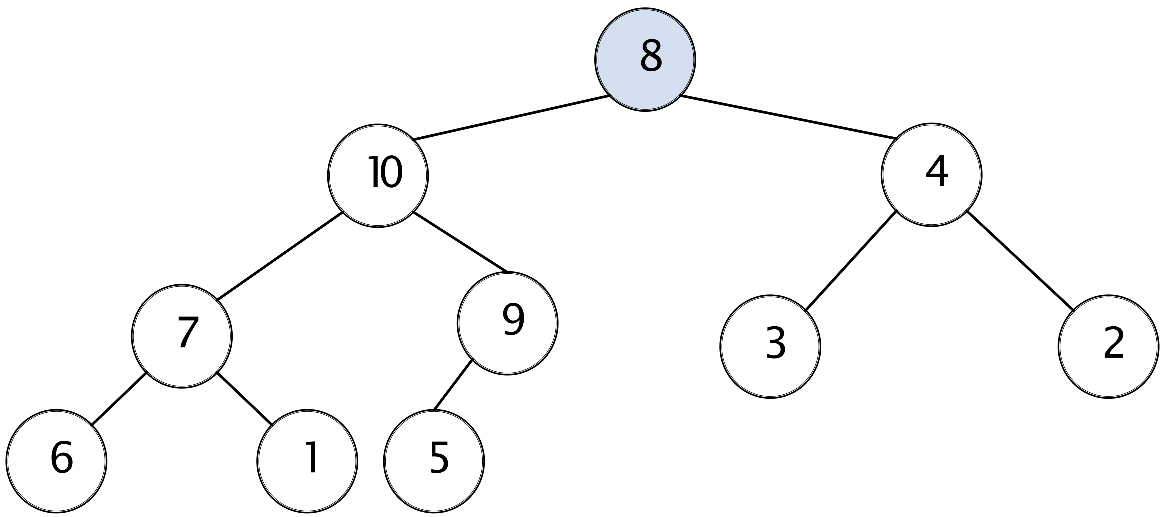
- Exemplo: remover o Item de maior prioridade 11



  - Armazena o Item da raiz e copiar o conteúdo do último nó para raiz.

2.2. Heap – Remoção

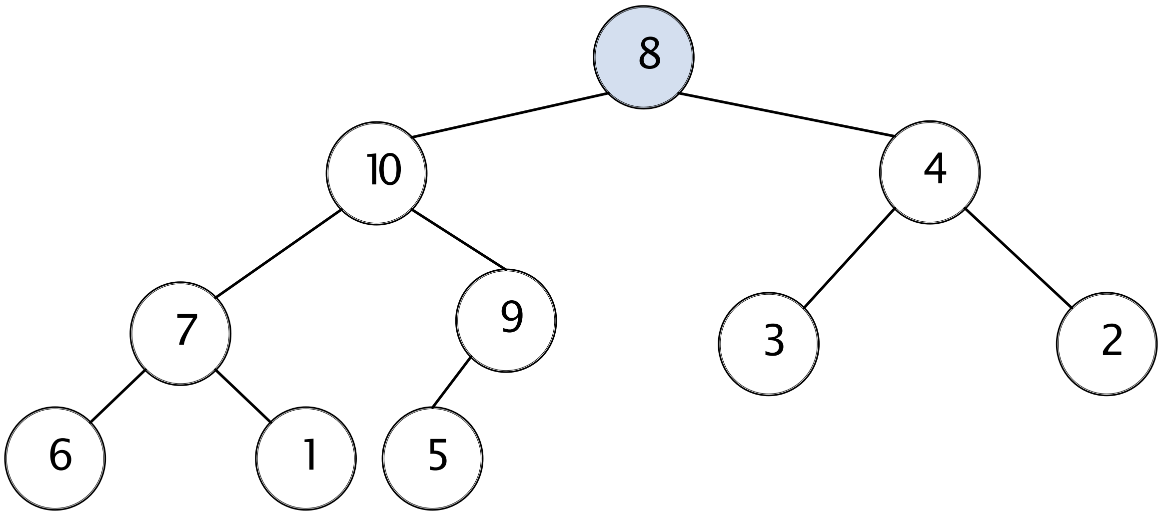
- Exemplo: remover o Item de maior prioridade 11



  - Remover o último nó.

2.2. Heap – Remoção

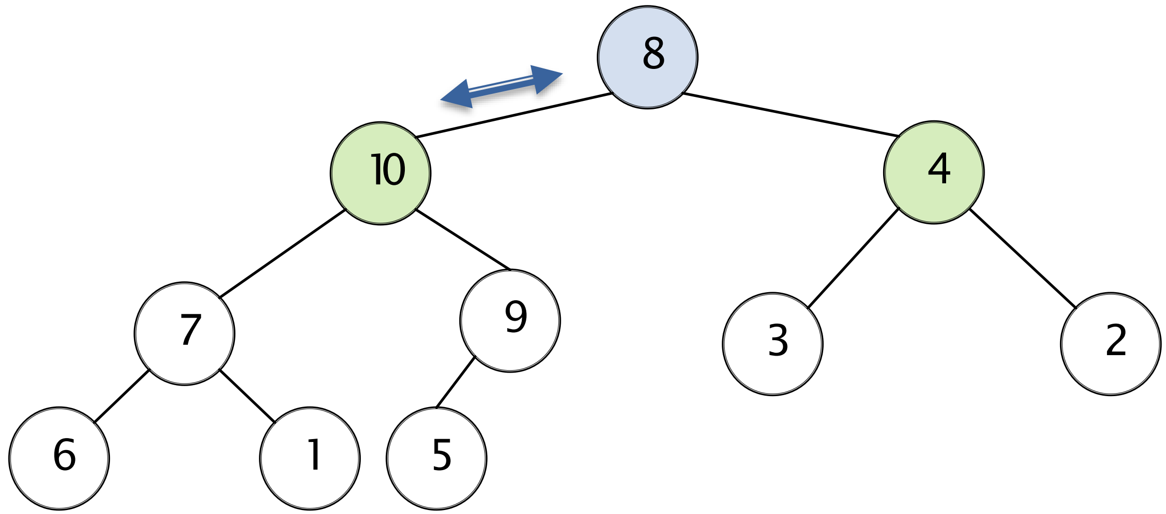
- Exemplo: remover o Item de maior prioridade 11



  - Descer o Heap, trocando o pai com o filho de maior prioridade (caso necessário).

2.2. Heap – Remoção

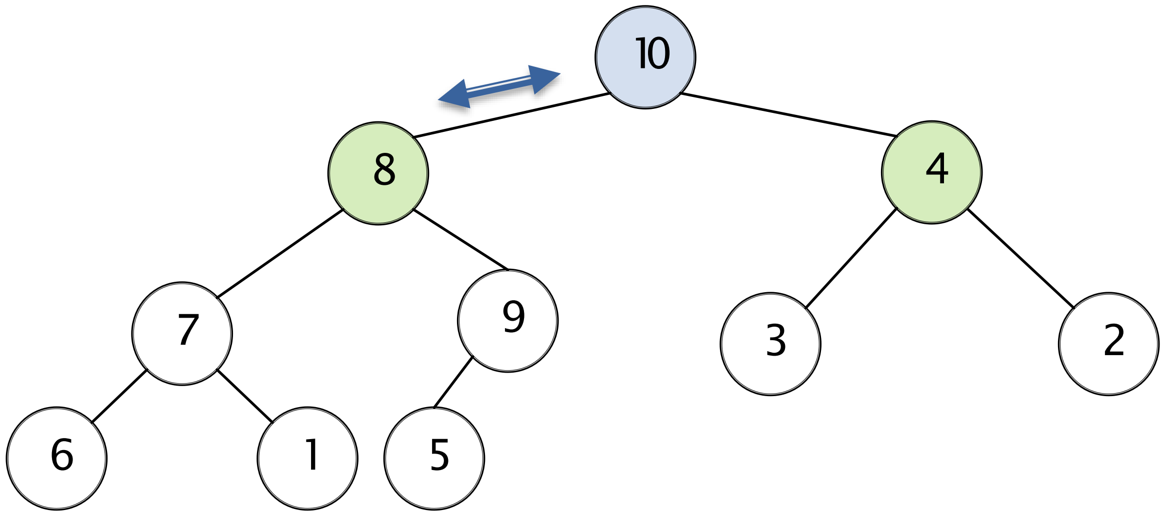
- Exemplo: remover o Item de maior prioridade 11



  - Descer o Heap, trocando o pai com o filho de maior prioridade (caso necessário).

2.2. Heap – Remoção

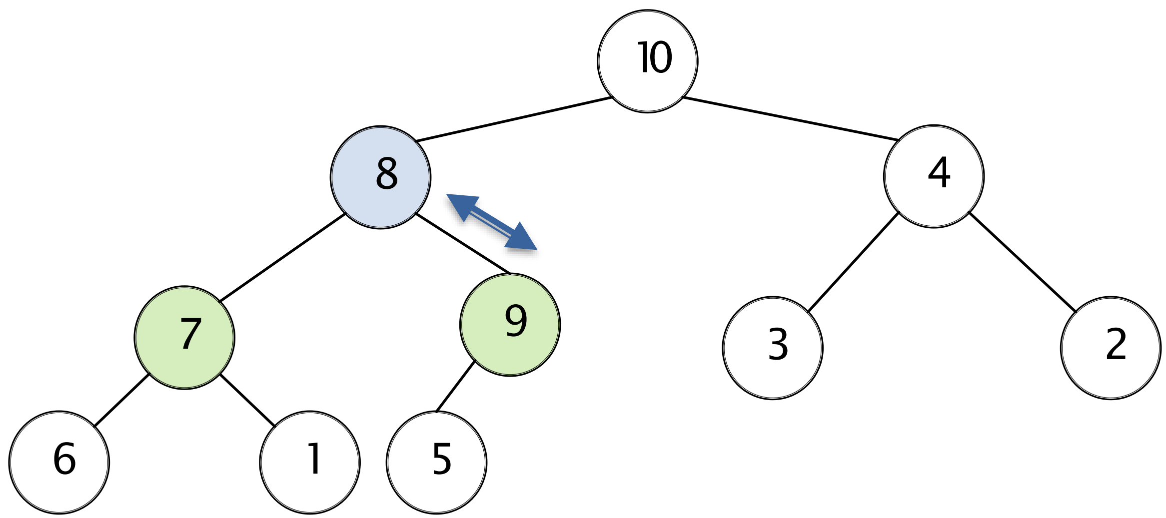
- Exemplo: remover o Item de maior prioridade 11



  - Descer o Heap, trocando o pai com o filho de maior prioridade (caso necessário).

2.2. Heap – Remoção

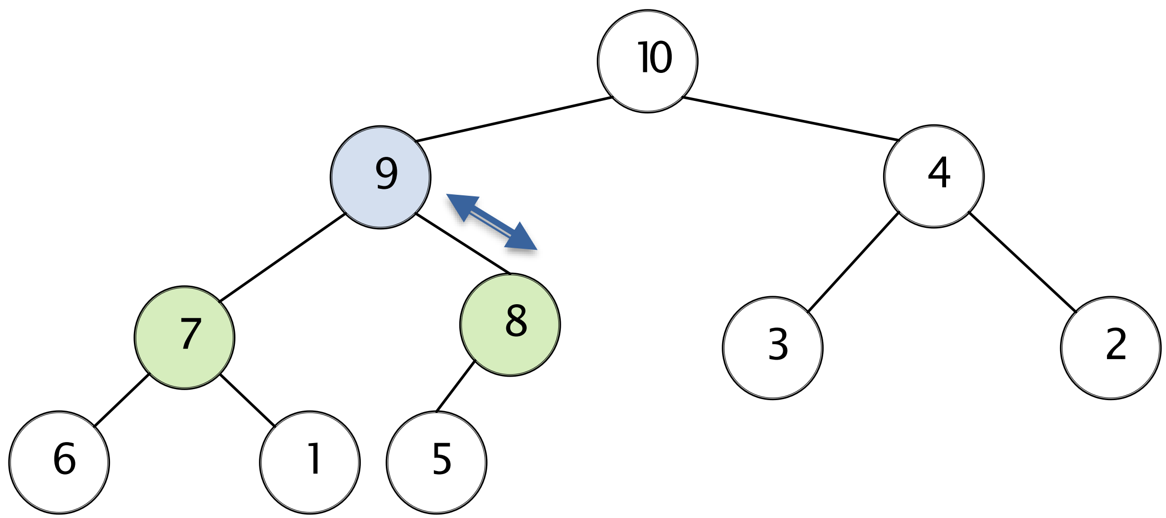
- Exemplo: remover o Item de maior prioridade 11



  - Descer o Heap, trocando o pai com o filho de maior prioridade (caso necessário).

2.2. Heap – Remoção

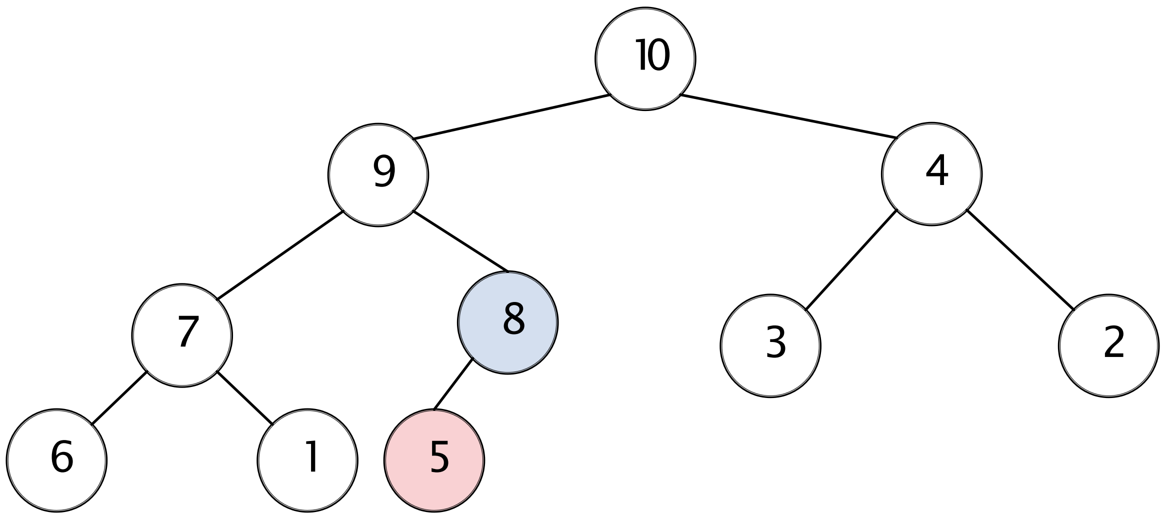
- Exemplo: remover o Item de maior prioridade 11



  - Descer o Heap, trocando o pai com o filho de maior prioridade (caso necessário).

2.2. Heap – Remoção

- Exemplo: remover o Item de maior prioridade 11



  - Descer o Heap, trocando o pai com o filho de maior prioridade (caso necessário).

2.2. Heap – Remoção

- **Implementação:**

```
Item extrai_máximo (p_fp fprio) {
    Item item = fprio->v[0];
    fprio->v[0] = fprio->v[fprio->n-1]);
    fprio->n--;
    desce_no_heap (fprio , 0);
    return item;
}

#define F_ESQ(i) (2*i+1) /*Filho esquerdo de i*/
#define F_DIR(i) (2*i+2) /*Filho direito de i*/
```

2.2. Heap – Remoção

- Implementação:

```
void desce_no_heap (p_fp fprio , int k) {
   int maior_filho;
   if (F_ESQ(k) < fprio->n) {
      maior_filho = F_ESQ(k);
      if (F_DIR(k) < fprio->n &&
          fprio->v[F_ESQ(k)].chave < fprio->v[F_DIR(k)].chave)
          maior_filho = F_DIR(k);
      if (fprio->v[k].chave < fprio->v[maior_filho].chave) {
         Item temp;
         temp = fprio->v[k];
         fprio->v[k] = fprio->v[maior_filho];
         fprio->v[maior_filho] = temp;
         desce_no_heap (fprio , maior_filho);
      }
   }
}
```

2.3. Heap – Análise

- Custo da função de **Inserção**:
  - No máximo, sobe até a raiz.
  - Ou seja, **`O(log n)`**.

- Custo da função de **Remoção**:
  - No máximo, desce até o último nível da árvore.
  - Ou seja, **`O(log n)`**.

2.3. Heap – Análise

- Comparação de Filas de Prioridade:
  - Lista Não-Ordenada:
    - Função de Inserção: **O(1)**
    - Função de Remoção: **O(n)**
  - Lista Ordenada:
    - Função de Inserção: **O(n)**
    - Função de Remoção: **O(1)**
  - Heap:
    - Função de Inserção: **O(log n)**
    - Função de Remoção: **O(log n)**

3. HeapSort

- As operações das filas de prioridades podem ser utilizadas para implementar algoritmos de ordenação.
  - Basta utilizar repetidamente a operação de **Inserção** para construir a fila de prioridades.
  - Em seguida, utilizar repetidamente a operação de **Remoção** para receber os itens na ordem correta.
  - O uso de heap para fazer a ordenação origina o método **HeapSort**.

3. HeapSort

- **Algoritmo:**

1. Receber um vetor desordenado.
2. Transformar o vetor em um Heap.
3. Trocar o item na posição 1 do vetor (raiz do Heap) com o item da posição n-1.
4. Usar o procedimento **desce_no_heap** para reconstituir o Heap para os itens `V[0], V[1], ..., V[n - 2]`.
5. Repita os passos 3 e 4 com os n – 1 itens restantes, depois com os n – 2, até que reste apenas um item.

3. HeapSort

- **Algoritmo:**
  1. Receber um vetor desordenado.

	0	1	2	3	4	5	6	7	8	9	...
V	4	8	2	7	6	3	5	1	9	10	

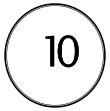
3. HeapSort

- **Algoritmo:**
  2. Transformar o vetor em um Heap.

	0	1	2	3	4	5	6	7	8	9	...
V	4	8	2	7	6	3	5	1	9	10	

3. HeapSort

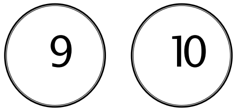
- **Algoritmo:**
  2. Transformar o vetor em um Heap.



	0	1	2	3	4	5	6	7	8	9	...
V	4	8	2	7	6	3	5	1	9	10	

3. HeapSort

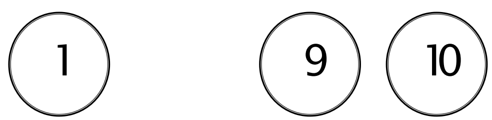
- **Algoritmo:**
  2. Transformar o vetor em um Heap.



	0	1	2	3	4	5	6	7	8	9	...
V	4	8	2	7	6	3	5	1	9	10	

3. HeapSort

- **Algoritmo:**
  2. Transformar o vetor em um Heap.



	0	1	2	3	4	5	6	7	8	9	...
V	4	8	2	7	6	3	5	1	9	10	

3. HeapSort

- **Algoritmo:**
  2. Transformar o vetor em um Heap.

	0	1	2	3	4	5	6	7	8	9	...
V	4	8	2	7	6	3	5	1	9	10	

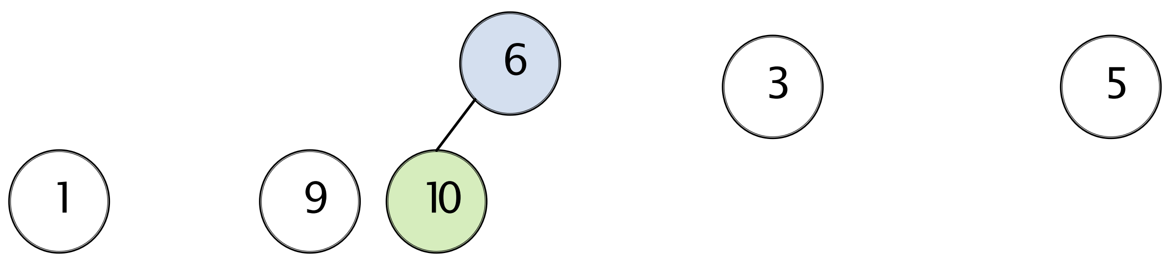
3. HeapSort

- **Algoritmo:**
  2. Transformar o vetor em um Heap.

	0	1	2	3	4	5	6	7	8	9	...
V	4	8	2	7	6	3	5	1	9	10	

3. HeapSort

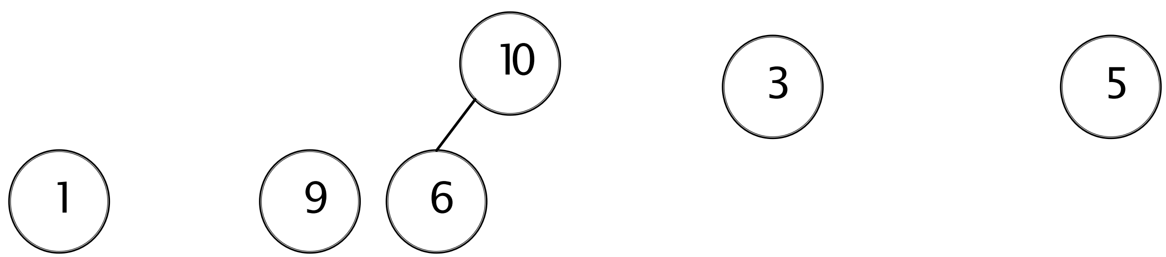
- **Algoritmo:**
  2. Transformar o vetor em um Heap.



	0	1	2	3	4	5	6	7	8	9	...
V	4	8	2	7	6	3	5	1	9	10	

3. HeapSort

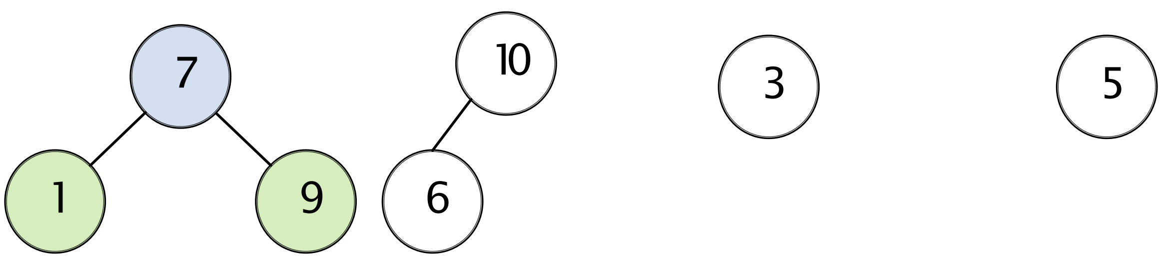
- **Algoritmo:**
  2. Transformar o vetor em um Heap.



	0	1	2	3	4	5	6	7	8	9	...
V	4	8	2	7	10	3	5	1	9	6	

3. HeapSort

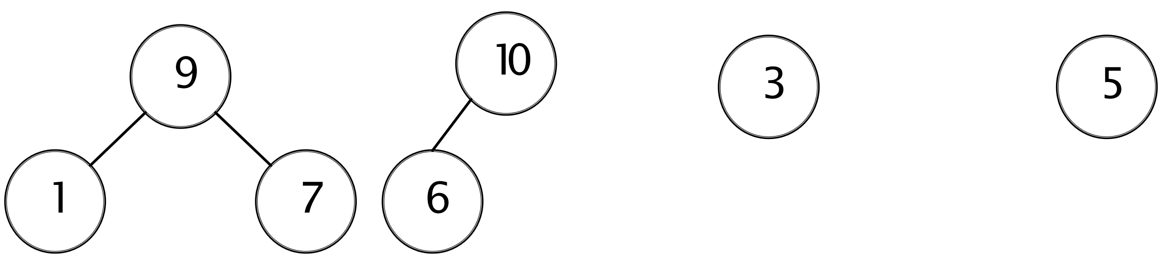
- **Algoritmo:**
  2. Transformar o vetor em um Heap.



	0	1	2	3	4	5	6	7	8	9	...
V	4	8	2	7	10	3	5	1	9	6	

3. HeapSort

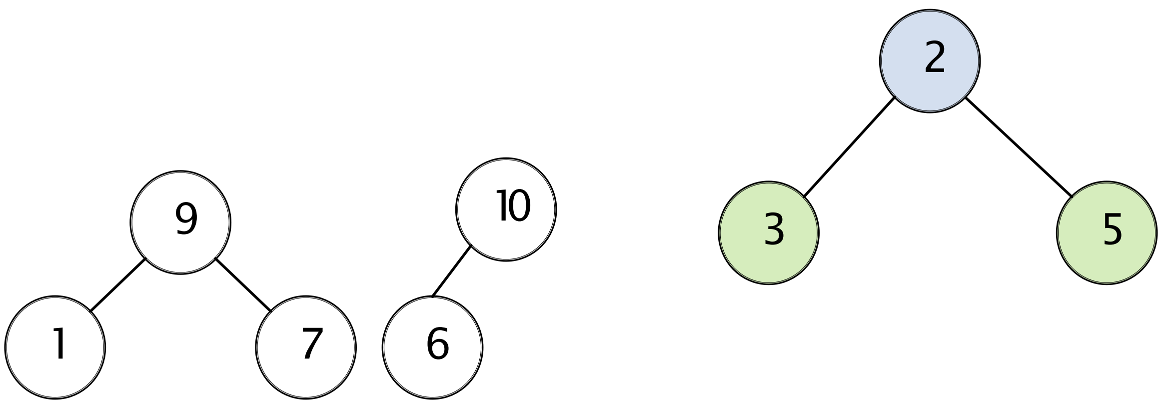
- **Algoritmo:**
  2. Transformar o vetor em um Heap.



	0	1	2	3	4	5	6	7	8	9	...
V	4	8	2	9	10	3	5	1	7	6	

3. HeapSort

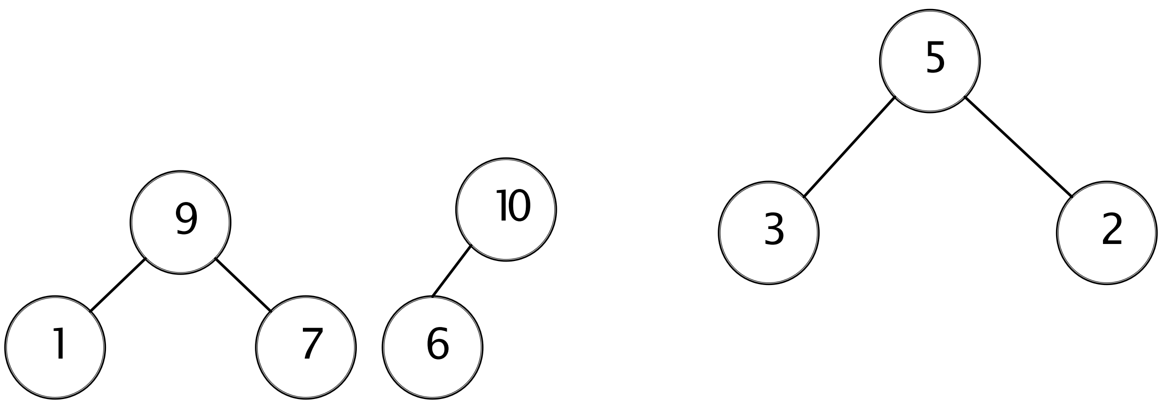
- **Algoritmo:**
  2. Transformar o vetor em um Heap.



	0	1	2	3	4	5	6	7	8	9	...
V	4	8	2	9	10	3	5	1	7	6	

3. HeapSort

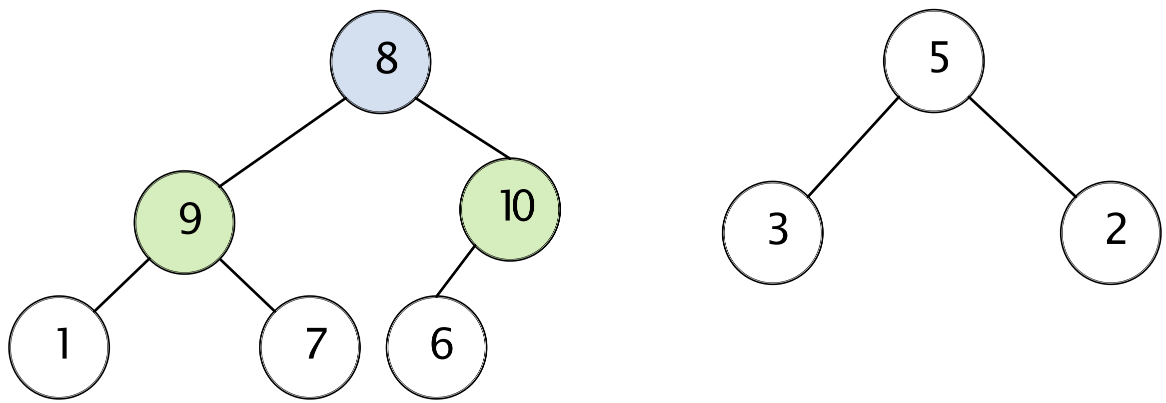
- **Algoritmo:**
  2. Transformar o vetor em um Heap.



	0	1	2	3	4	5	6	7	8	9	...
V	4	8	5	9	10	3	2	1	7	6	

3. HeapSort

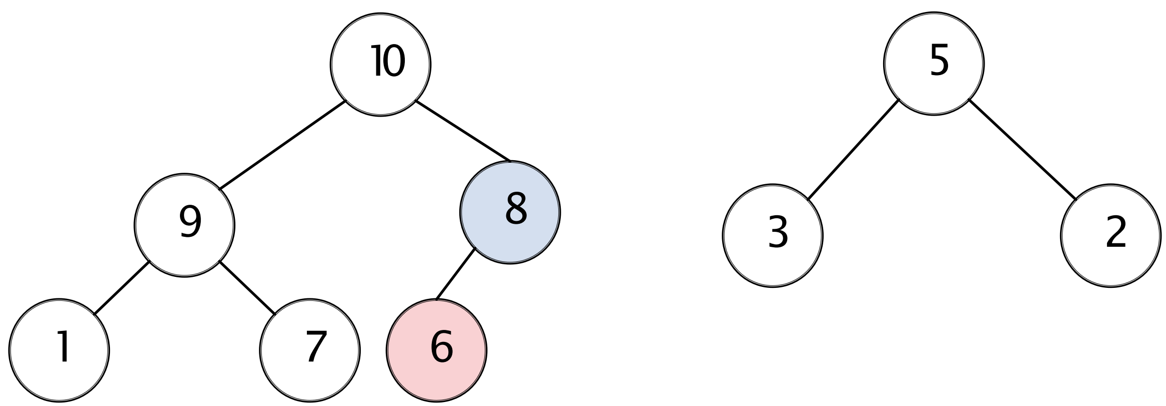
- **Algoritmo:**
  2. Transformar o vetor em um Heap.



	0	1	2	3	4	5	6	7	8	9	...
V	4	8	5	9	10	3	2	1	7	6	

3. HeapSort

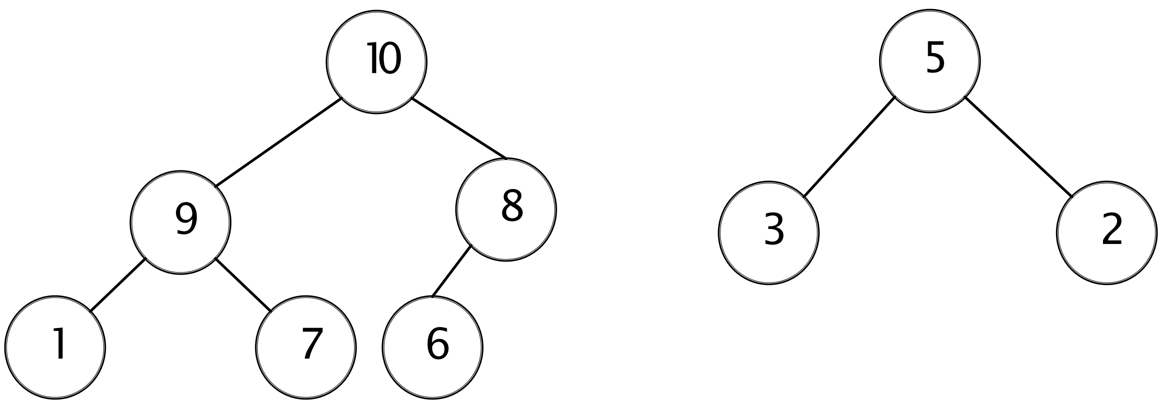
- **Algoritmo:**
  2. Transformar o vetor em um Heap.



	0	1	2	3	4	5	6	7	8	9	...
V	4	10	5	9	8	3	2	1	7	6	

3. HeapSort

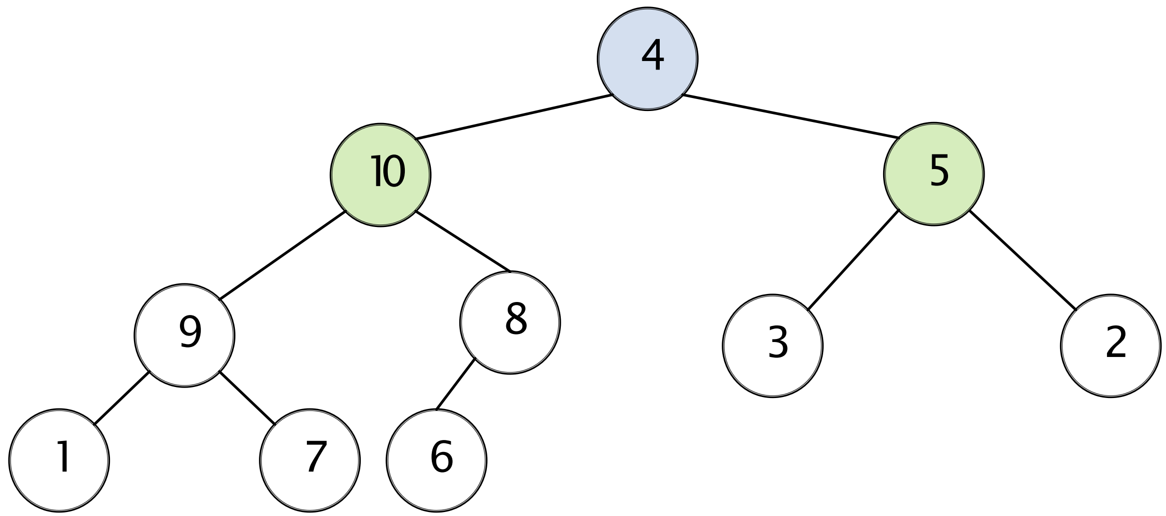
- **Algoritmo:**
  2. Transformar o vetor em um Heap.



	0	1	2	3	4	5	6	7	8	9	...
V	4	10	5	9	8	3	2	1	7	6	

3. HeapSort

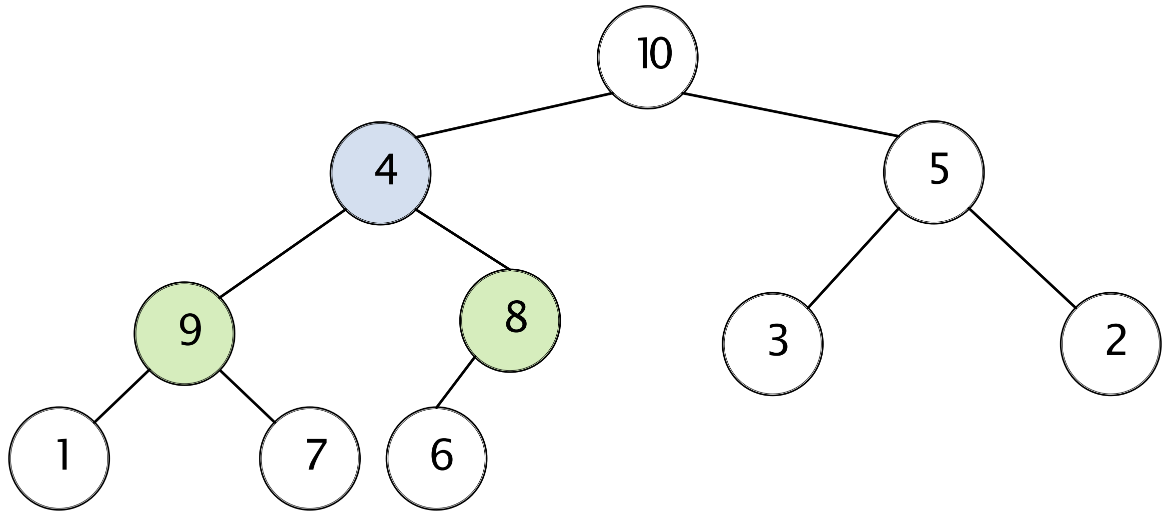
- **Algoritmo:**
  2. Transformar o vetor em um Heap.



	0	1	2	3	4	5	6	7	8	9	...
V	4	10	5	9	8	3	2	1	7	6	

3. HeapSort

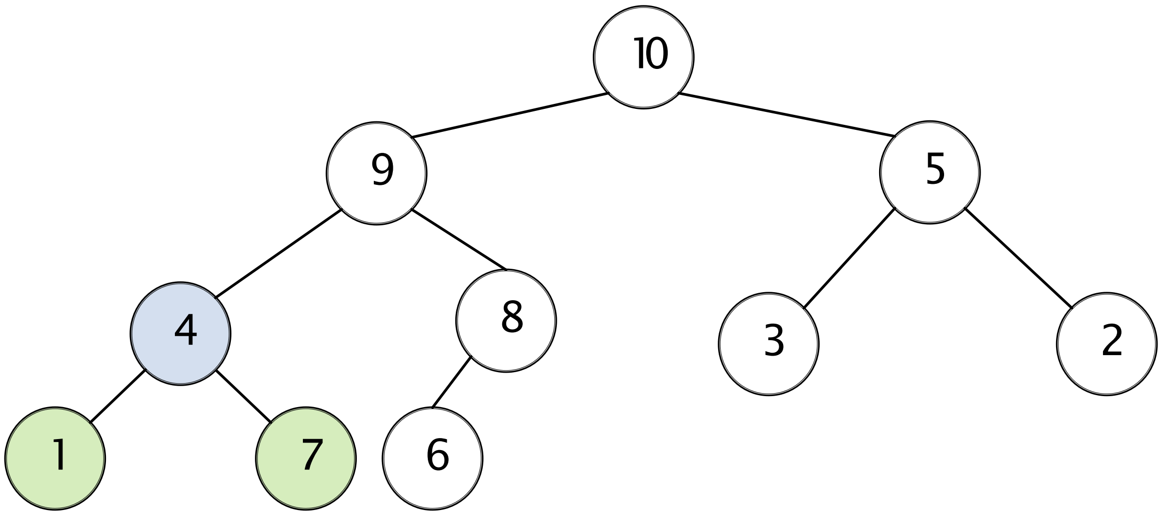
- **Algoritmo:**
  2. Transformar o vetor em um Heap.



	0	1	2	3	4	5	6	7	8	9	...
V	10	4	5	9	8	3	2	1	7	6	

3. HeapSort

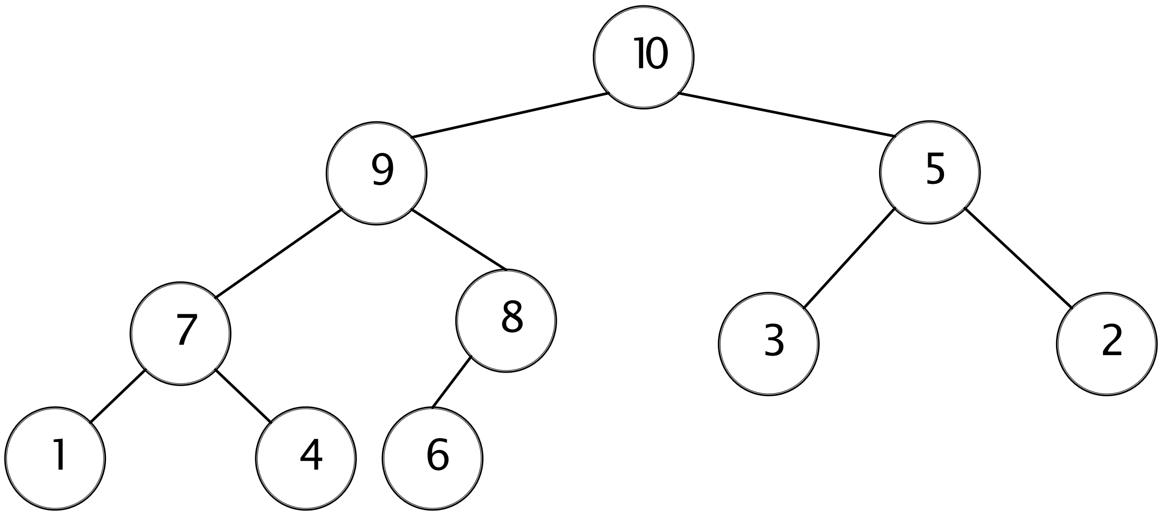
- **Algoritmo:**
  2. Transformar o vetor em um Heap.



	0	1	2	3	4	5	6	7	8	9	...
V	10	9	5	4	8	3	2	1	7	6	

3. HeapSort

- **Algoritmo:**
  2. Transformar o vetor em um Heap.



	0	1	2	3	4	5	6	7	8	9	...
V	10	9	5	7	8	3	2	1	4	6	

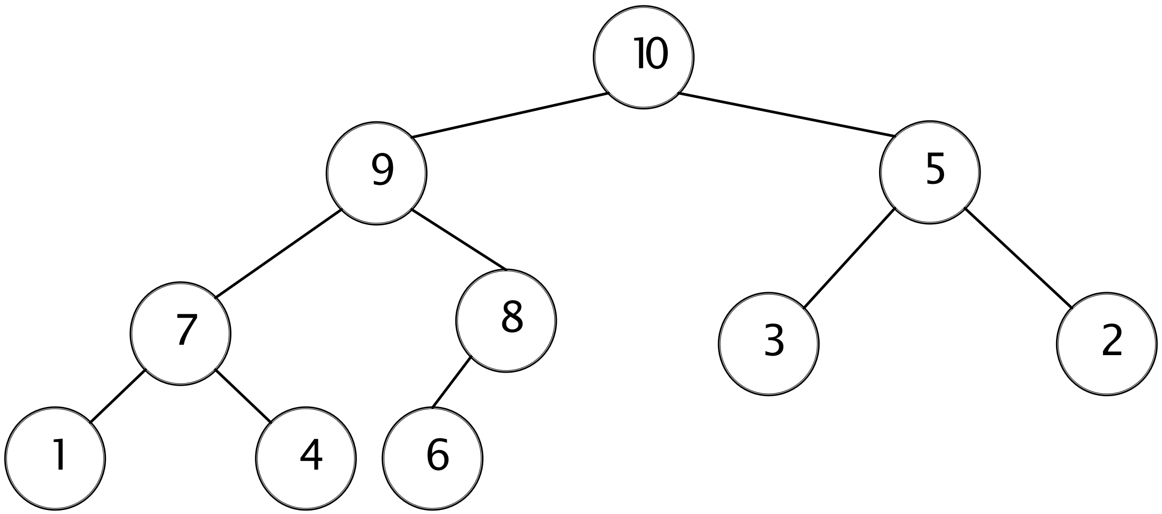
3. HeapSort

- **Algoritmo:**
  - Uma vez que o vetor é um Heap:

3. Trocar o item na posição 1 do vetor (raiz do Heap) com o item da posição n-1.
4. Usar o procedimento **desce_no_heap** para reconstituir o Heap para os itens `V[0], V[1], ..., V[n - 2]`.
5. Repita os passos 3 e 4 com os n – 1 itens restantes, depois com os n – 2, até que reste apenas um item.

3. HeapSort

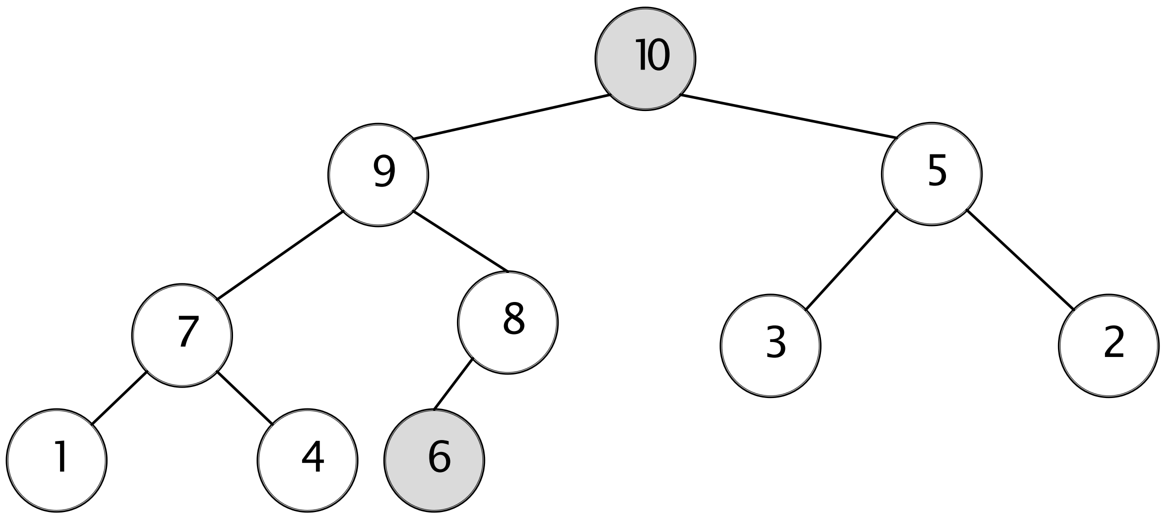
- **Algoritmo:**



	0	1	2	3	4	5	6	7	8	9	...
V	10	9	5	7	8	3	2	1	4	6	

3. HeapSort

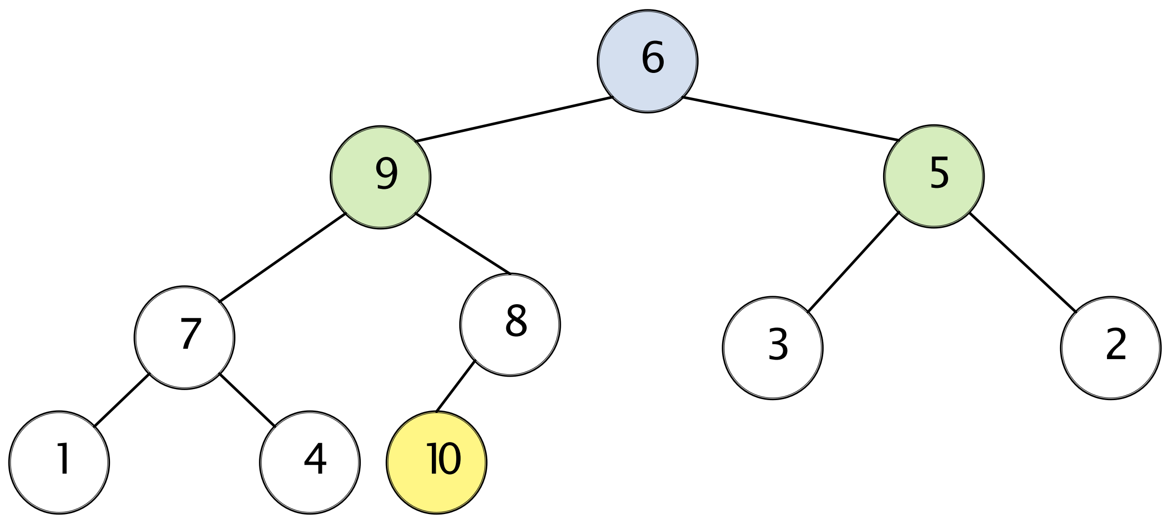
- **Algoritmo:**



	0	1	2	3	4	5	6	7	8	9	...
V	10	9	5	7	8	3	2	1	4	6	

3. HeapSort

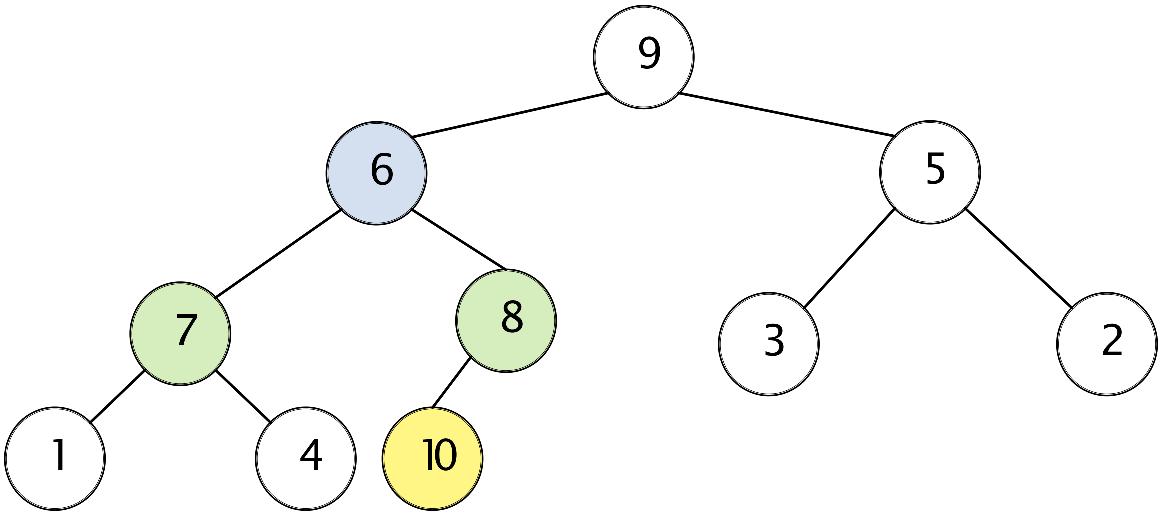
- **Algoritmo:**



	0	1	2	3	4	5	6	7	8	9	...
V	6	9	5	7	8	3	2	1	4	10	

3. HeapSort

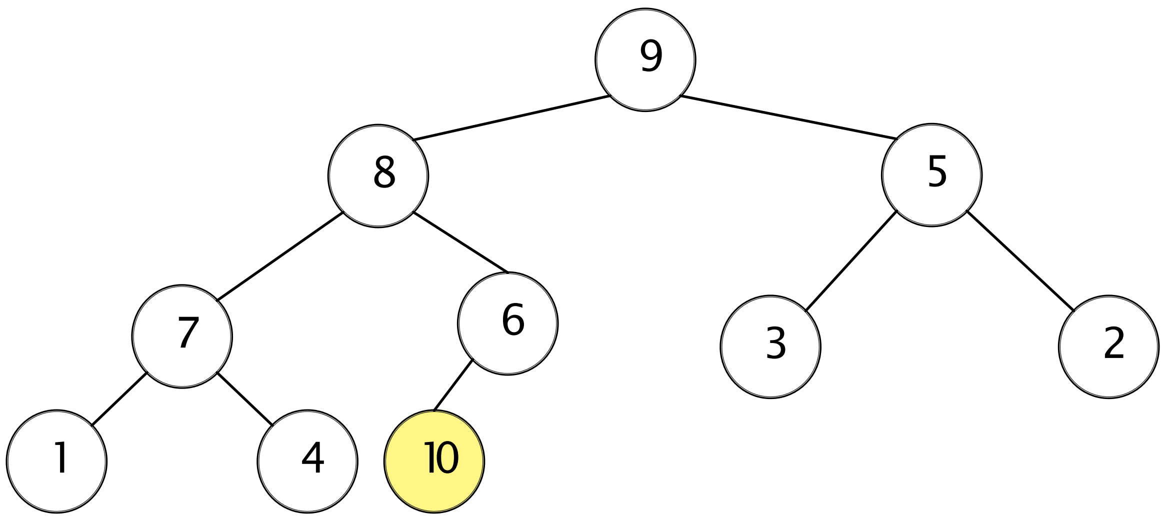
- **Algoritmo:**



	0	1	2	3	4	5	6	7	8	9	...
V	9	6	5	7	8	3	2	1	4	10	

3. HeapSort

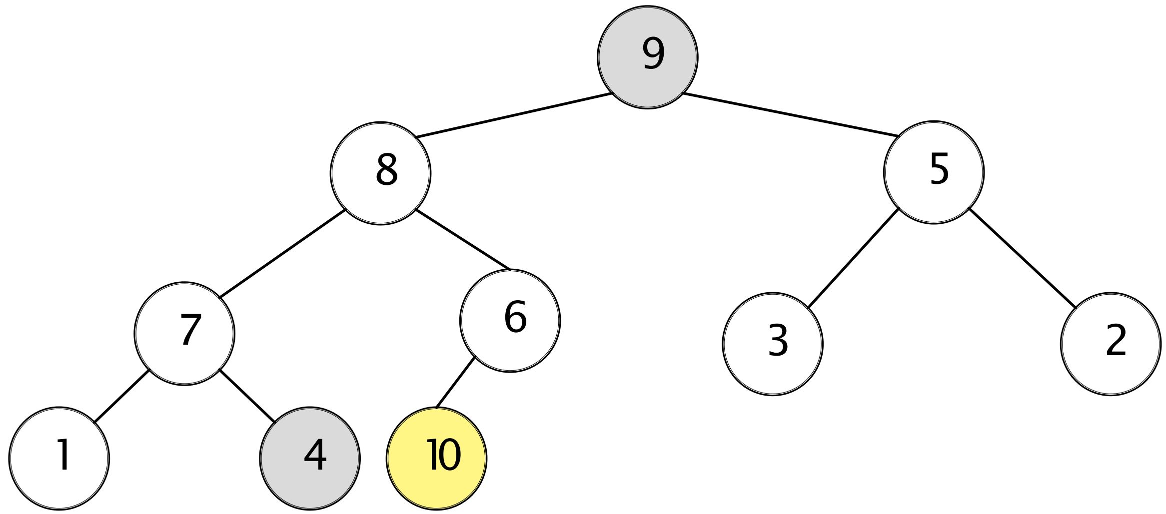
- **Algoritmo:**



	0	1	2	3	4	5	6	7	8	9	...
V	9	8	5	7	6	3	2	1	4	10	

3. HeapSort

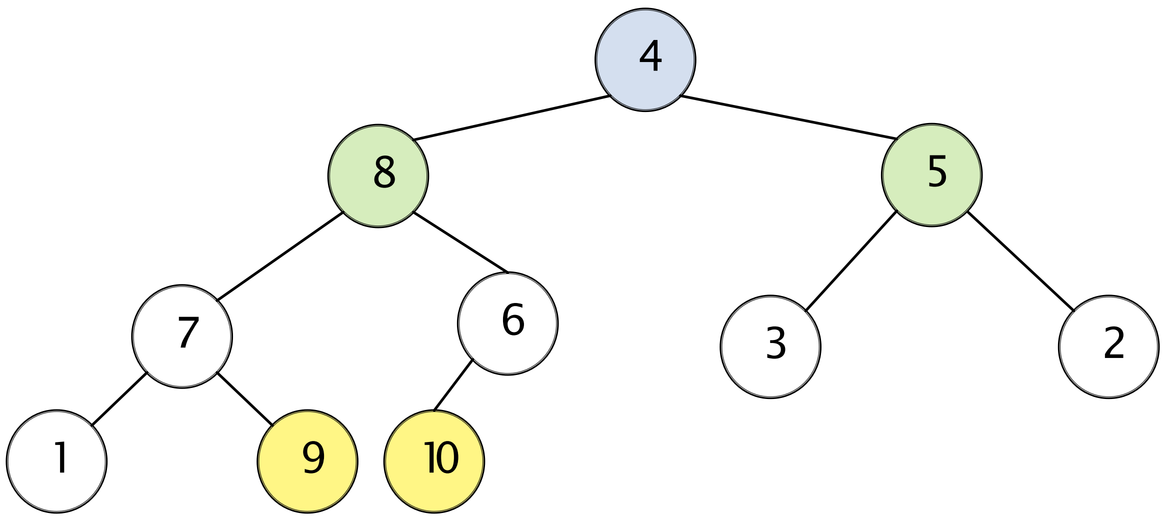
- **Algoritmo:**



	0	1	2	3	4	5	6	7	8	9	...
V	9	8	5	7	6	3	2	1	4	10	

3. HeapSort

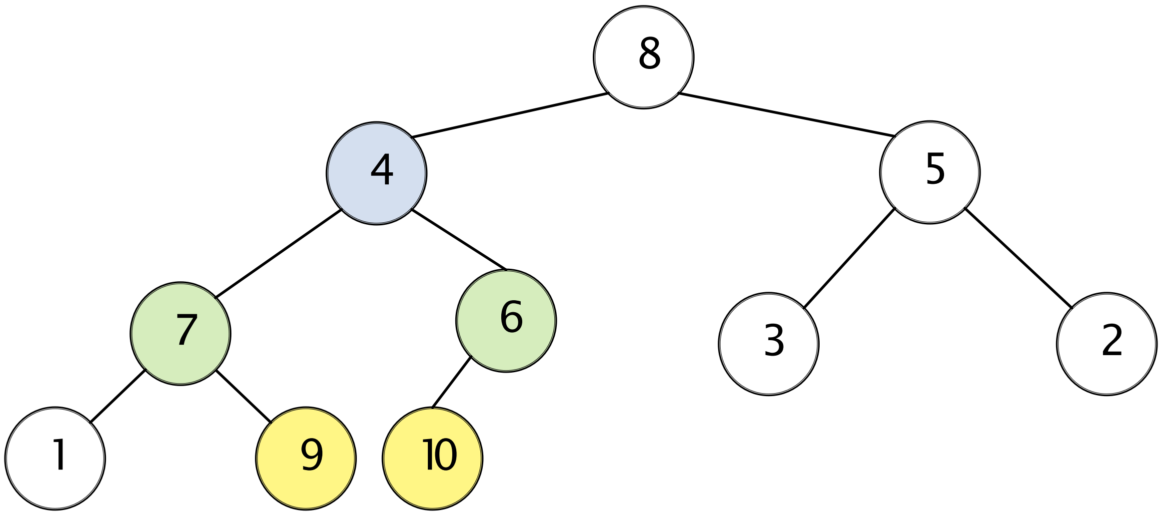
- **Algoritmo:**



	0	1	2	3	4	5	6	7	8	9	...
V	4	8	5	7	6	3	2	1	9	10	

3. HeapSort

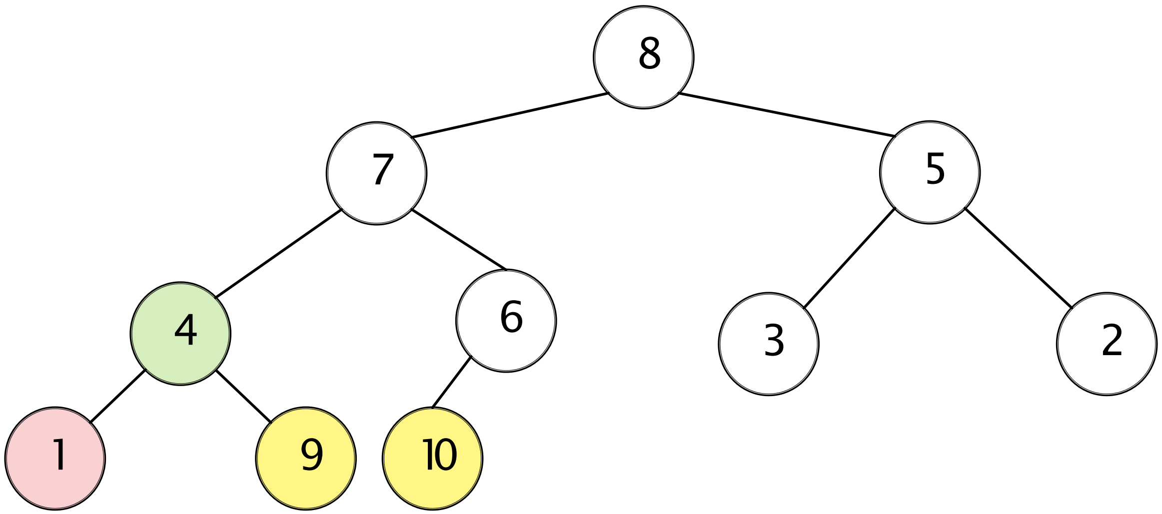
- **Algoritmo:**



	0	1	2	3	4	5	6	7	8	9	...
V	8	4	5	7	6	3	2	1	9	10	

3. HeapSort

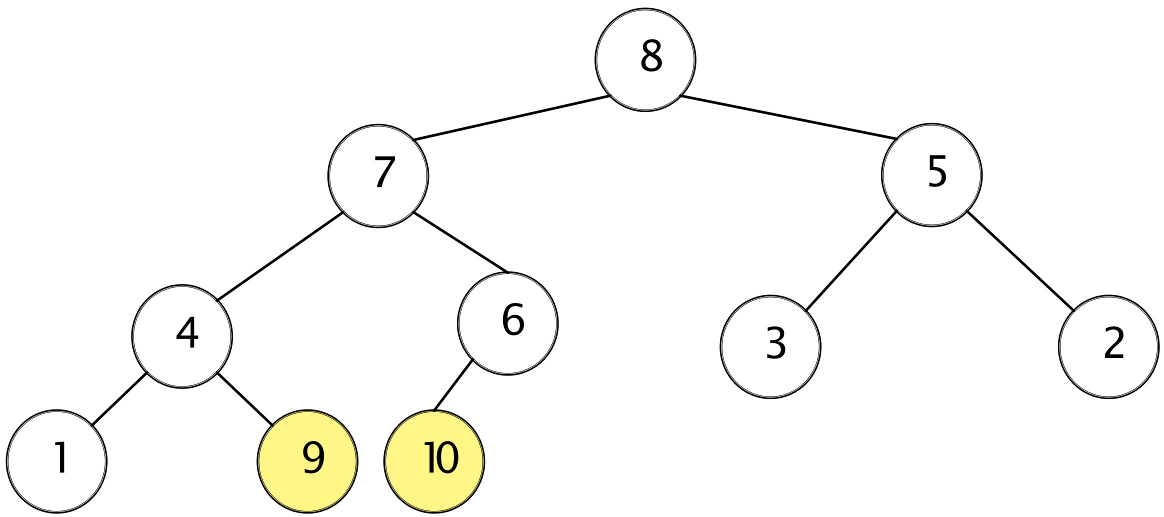
- **Algoritmo:**



	0	1	2	3	4	5	6	7	8	9	...
V	8	7	5	4	6	3	2	1	9	10	

3. HeapSort

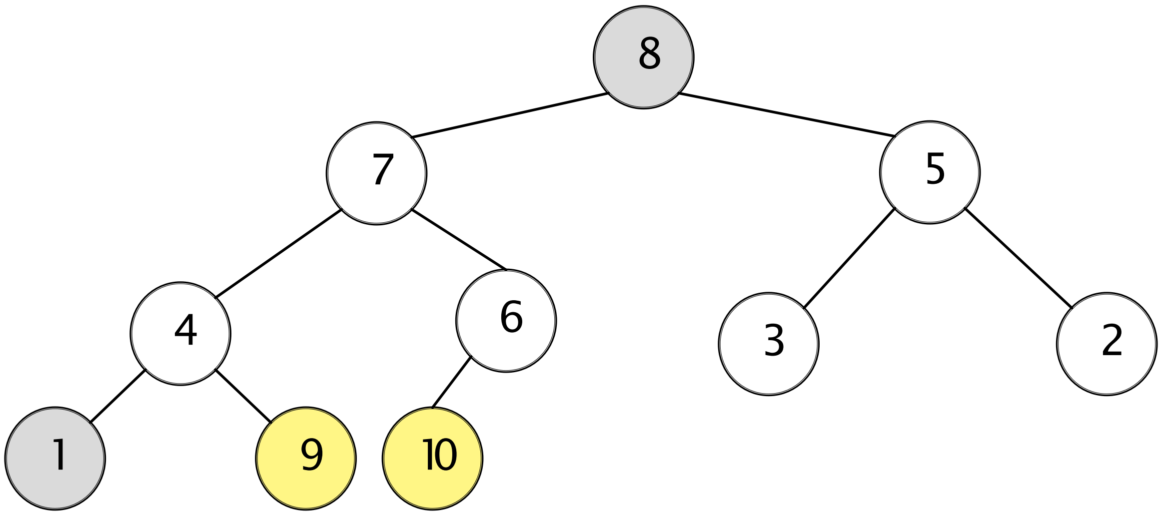
- **Algoritmo:**



	0	1	2	3	4	5	6	7	8	9	...
V	8	7	5	4	6	3	2	1	9	10	

3. HeapSort

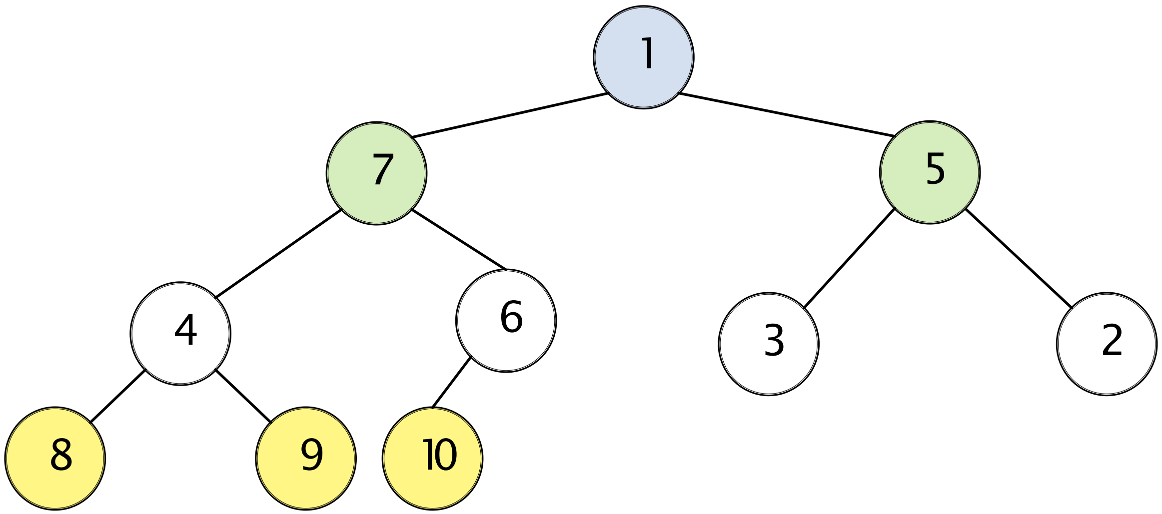
- **Algoritmo:**



	0	1	2	3	4	5	6	7	8	9	...
V	8	7	5	4	6	3	2	1	9	10	

3. HeapSort

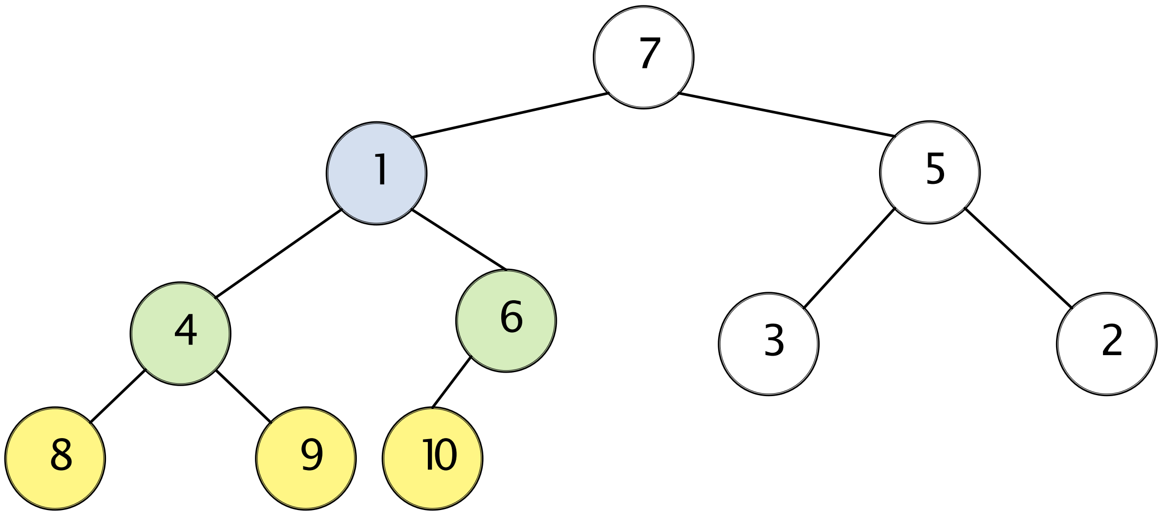
- **Algoritmo:**



	0	1	2	3	4	5	6	7	8	9	...
V	1	7	5	4	6	3	2	8	9	10	

3. HeapSort

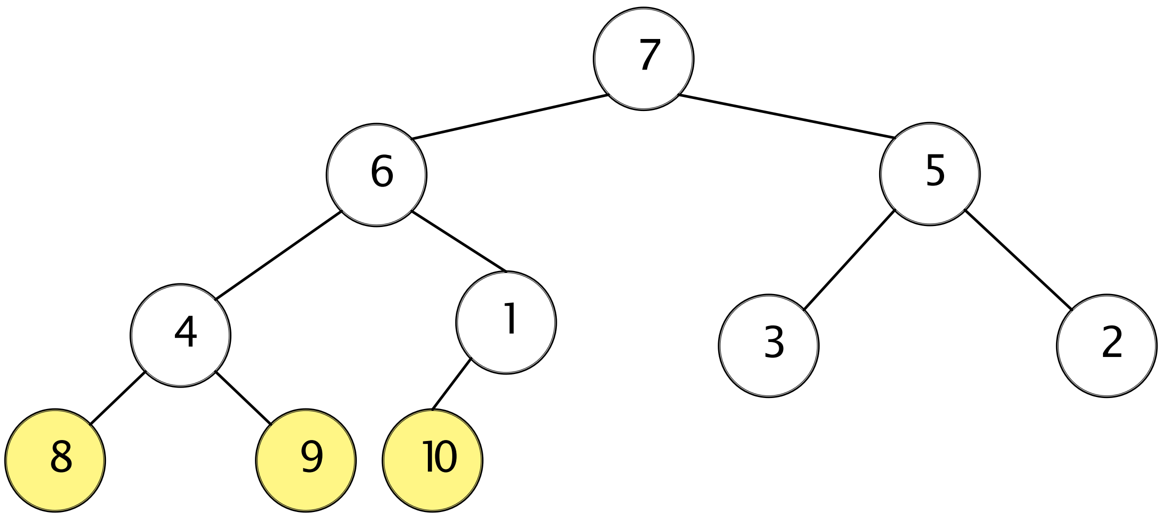
- **Algoritmo:**



	0	1	2	3	4	5	6	7	8	9	...
V	7	1	5	4	6	3	2	8	9	10	

3. HeapSort

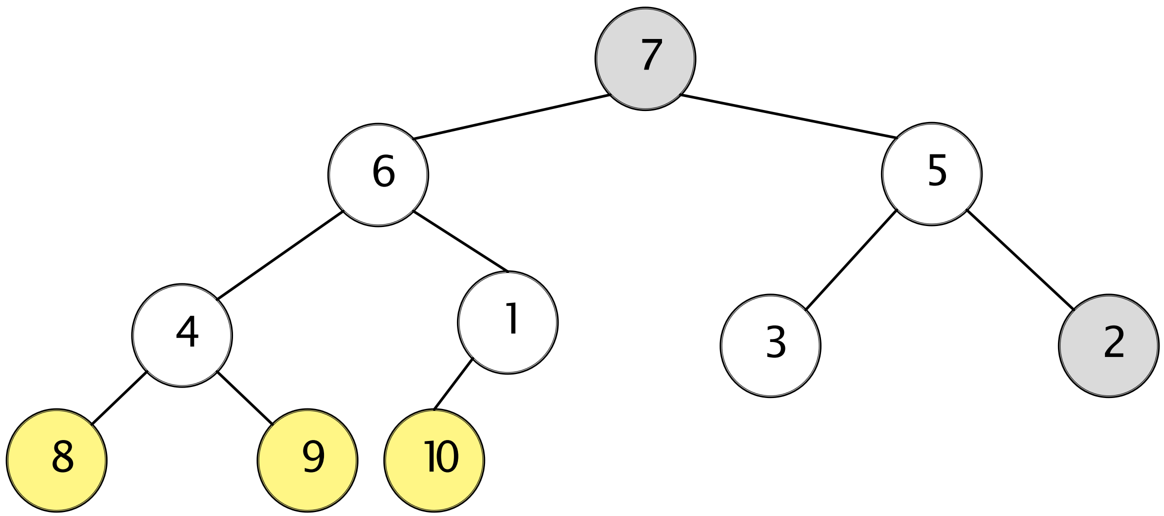
- **Algoritmo:**



	0	1	2	3	4	5	6	7	8	9	...
V	7	6	5	4	1	3	2	8	9	10	

3. HeapSort

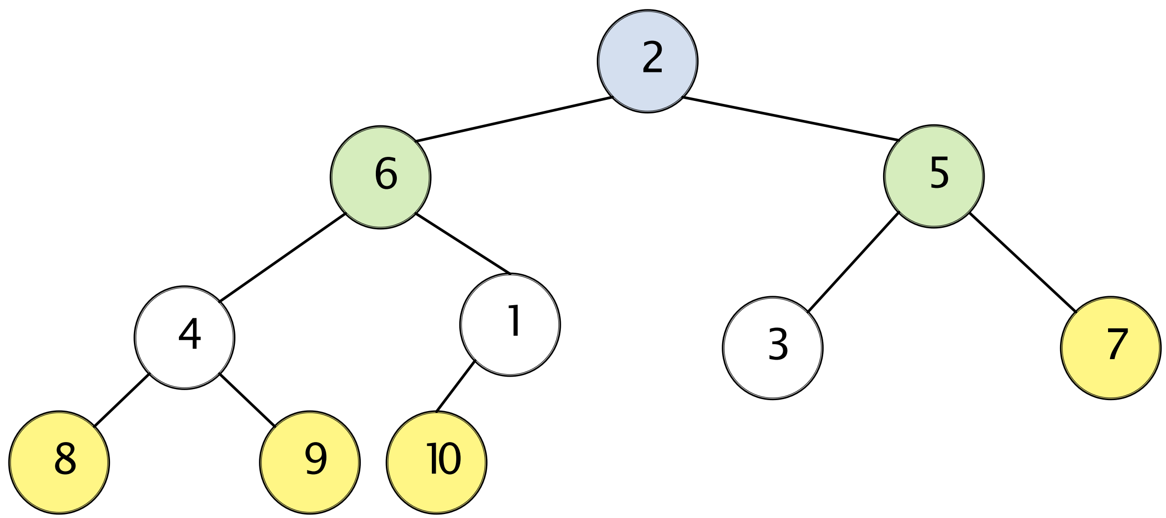
- **Algoritmo:**



	0	1	2	3	4	5	6	7	8	9	...
V	7	6	5	4	1	3	2	8	9	10	

3. HeapSort

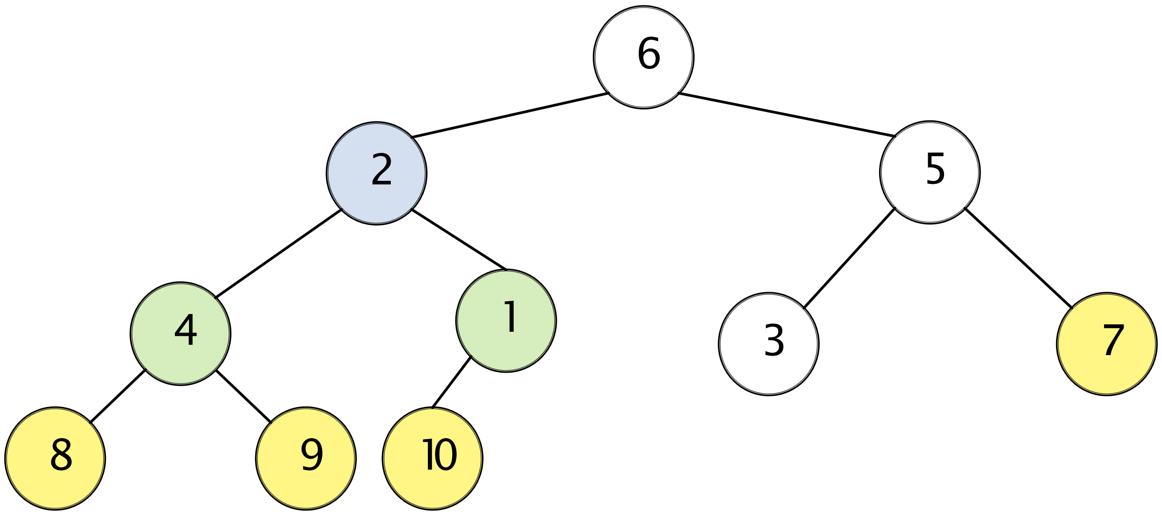
- **Algoritmo:**



	0	1	2	3	4	5	6	7	8	9	...
V	2	6	5	4	1	3	7	8	9	10	

3. HeapSort

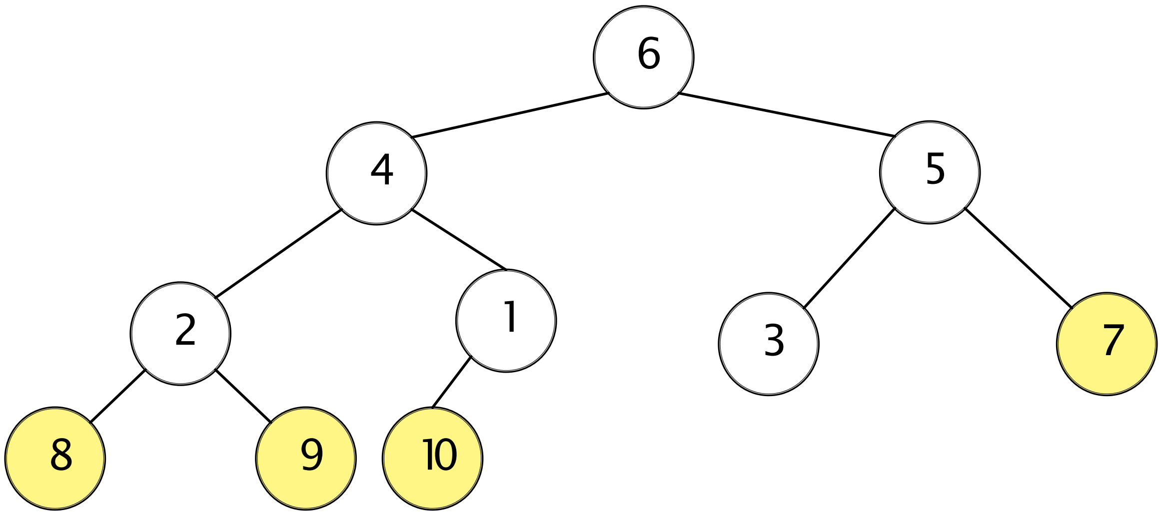
- **Algoritmo:**



	0	1	2	3	4	5	6	7	8	9	...
V	6	2	5	4	1	3	7	8	9	10	

3. HeapSort

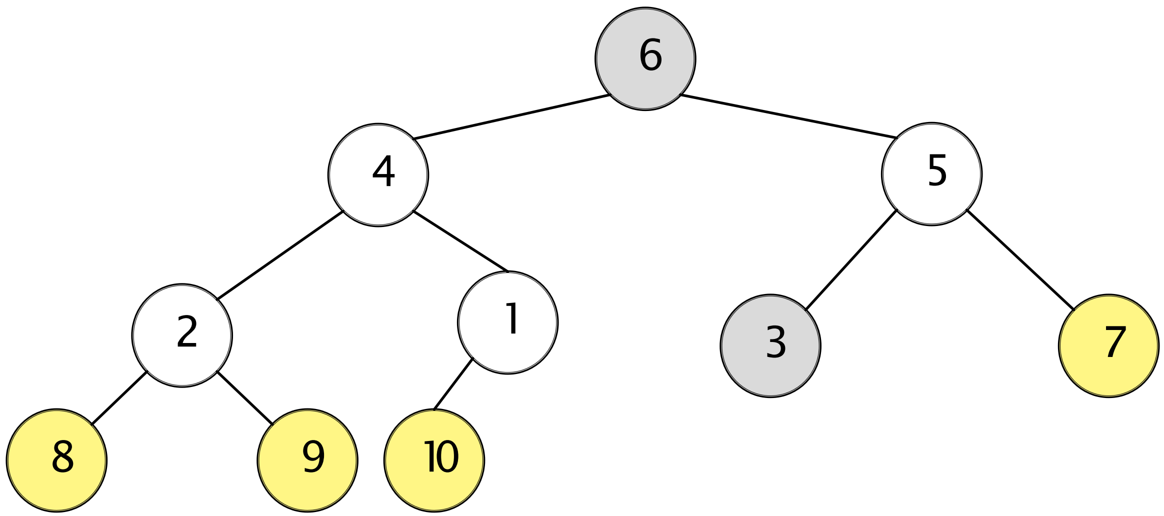
- **Algoritmo:**



	0	1	2	3	4	5	6	7	8	9	...
V	6	4	5	2	1	3	7	8	9	10	

3. HeapSort

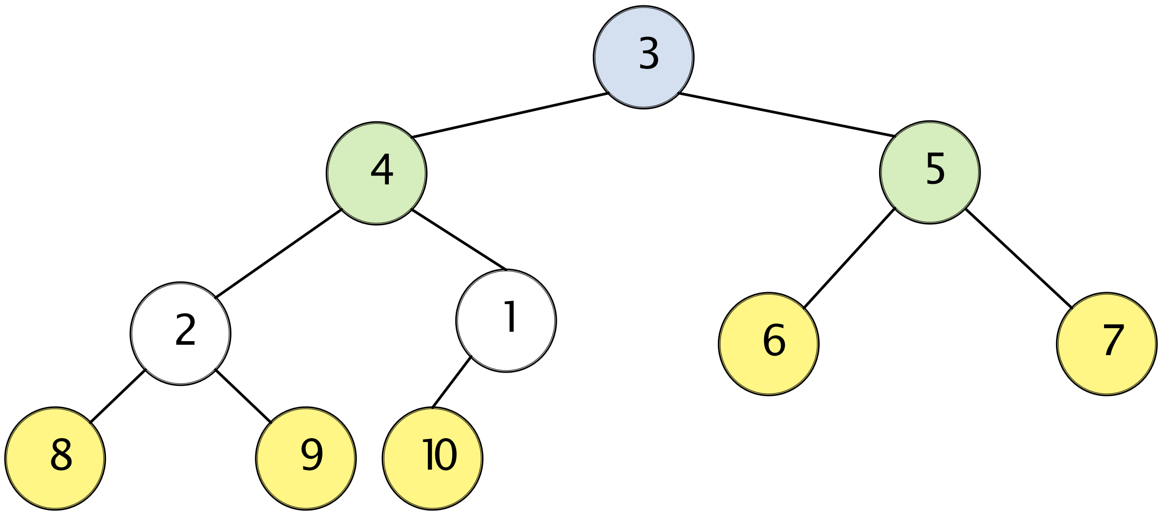
- **Algoritmo:**



	0	1	2	3	4	5	6	7	8	9	...
V	6	4	5	2	1	3	7	8	9	10	

3. HeapSort

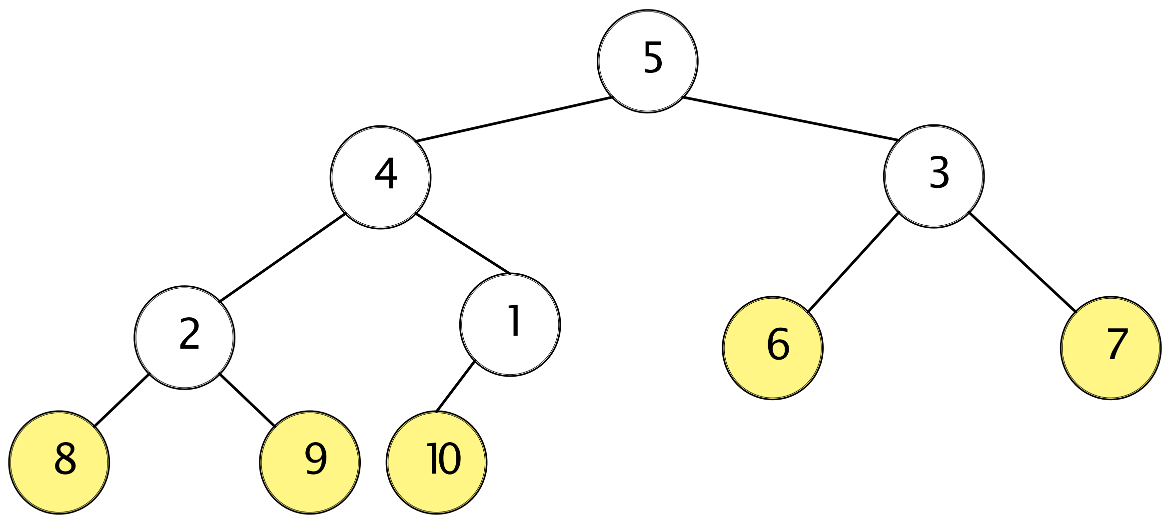
- **Algoritmo:**



	0	1	2	3	4	5	6	7	8	9	...
V	3	4	5	2	1	6	7	8	9	10	

3. HeapSort

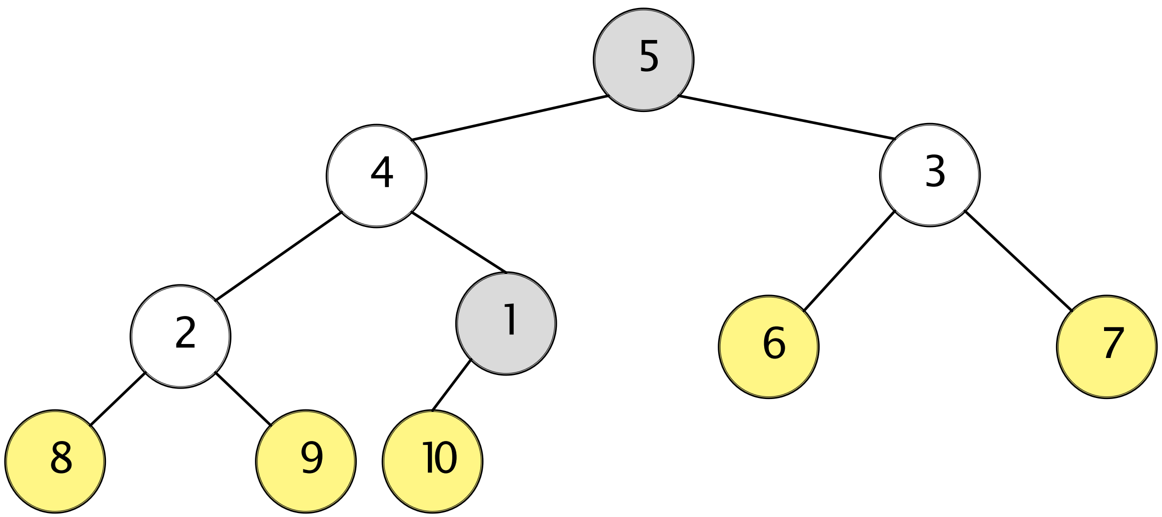
- **Algoritmo:**



	0	1	2	3	4	5	6	7	8	9	...
V	5	4	3	2	1	6	7	8	9	10	

3. HeapSort

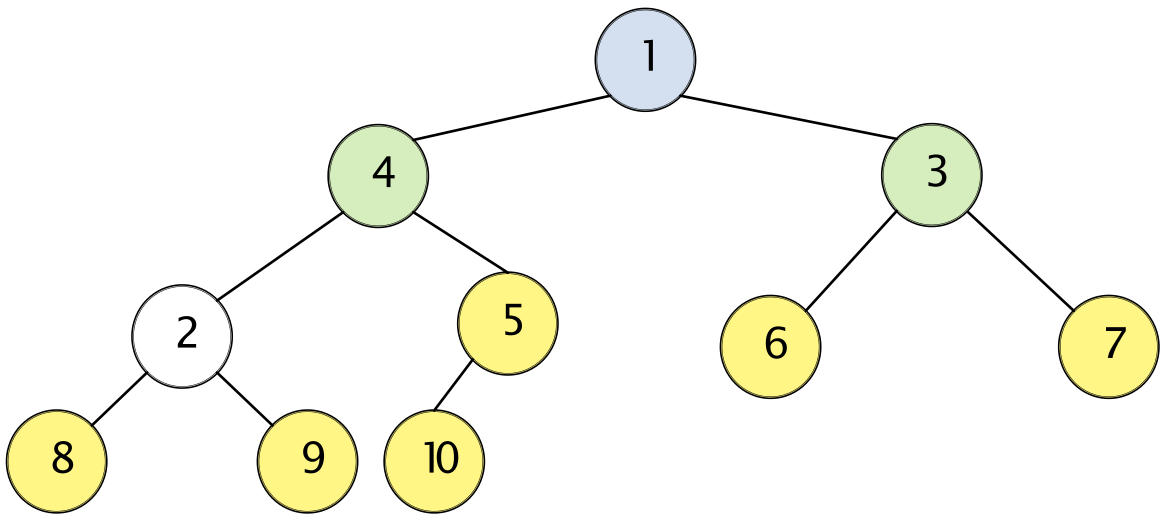
- **Algoritmo:**



	0	1	2	3	4	5	6	7	8	9	...
V	5	4	3	2	1	6	7	8	9	10	

3. HeapSort

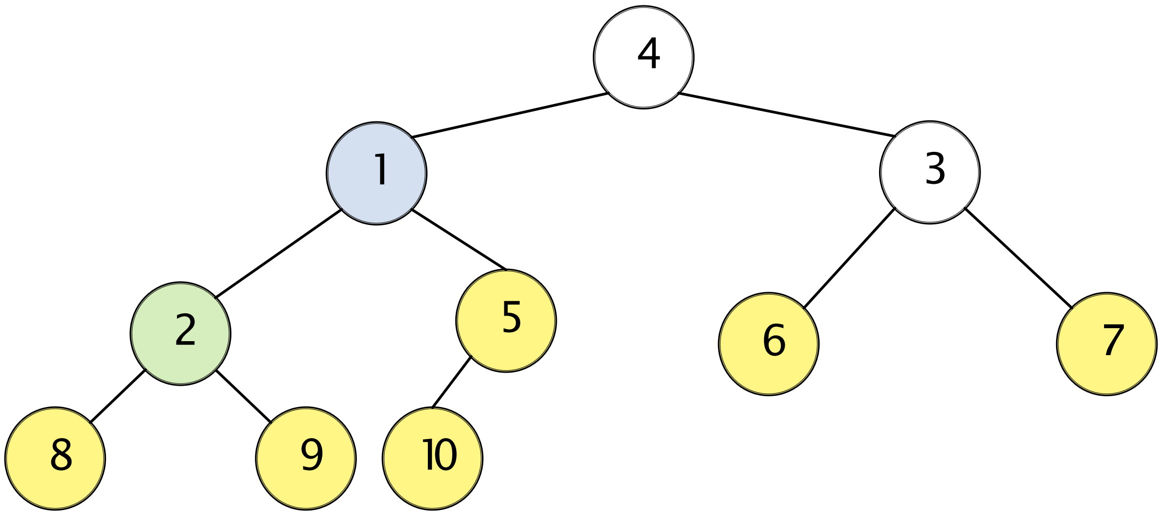
- **Algoritmo:**



	0	1	2	3	4	5	6	7	8	9	...
V	1	4	3	2	5	6	7	8	9	10	

3. HeapSort

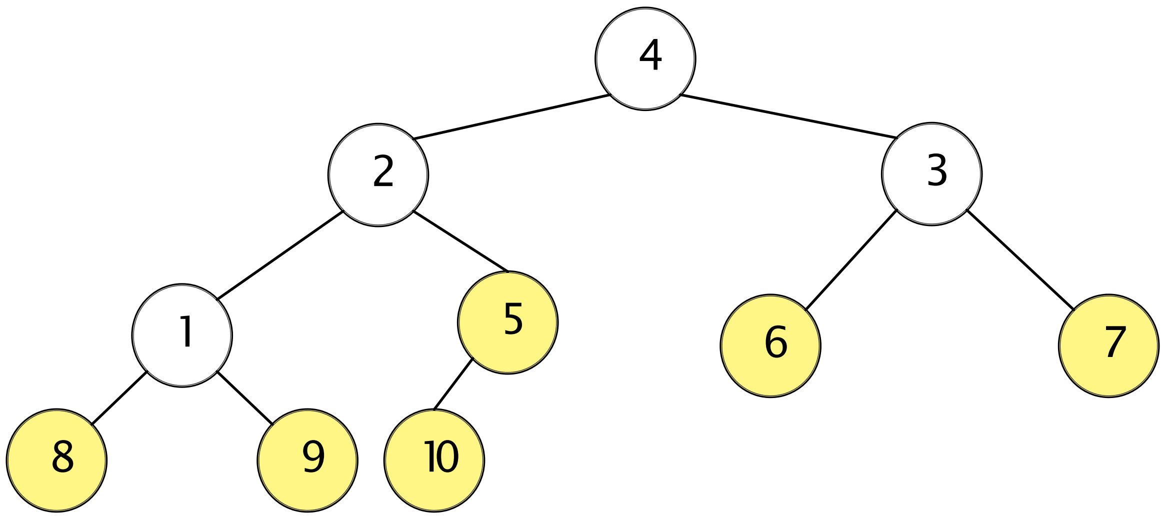
- **Algoritmo:**



	0	1	2	3	4	5	6	7	8	9	...
V	4	1	3	2	5	6	7	8	9	10	

3. HeapSort

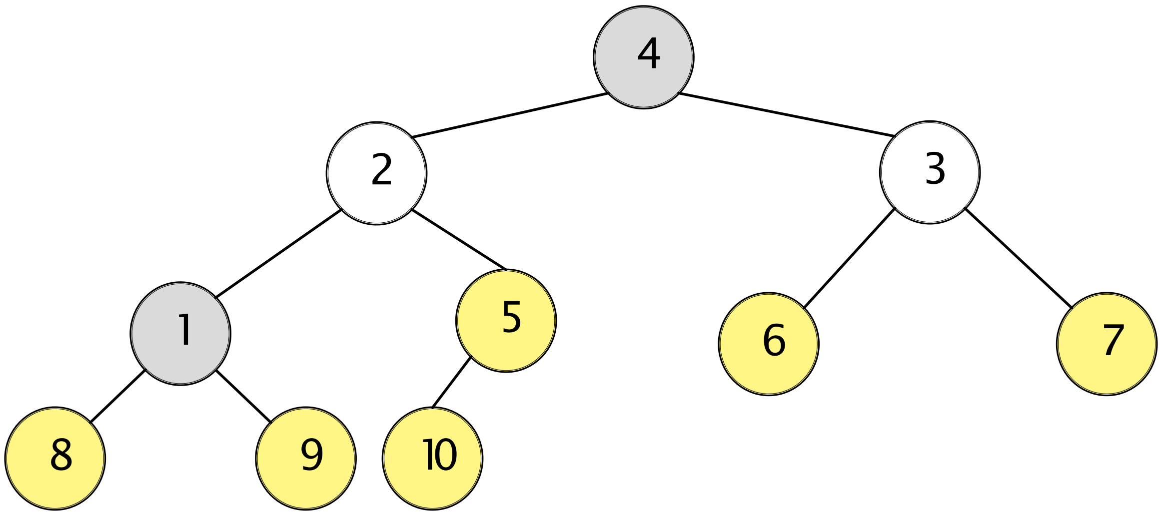
- **Algoritmo:**



	0	1	2	3	4	5	6	7	8	9	...
V	4	2	3	1	5	6	7	8	9	10	

3. HeapSort

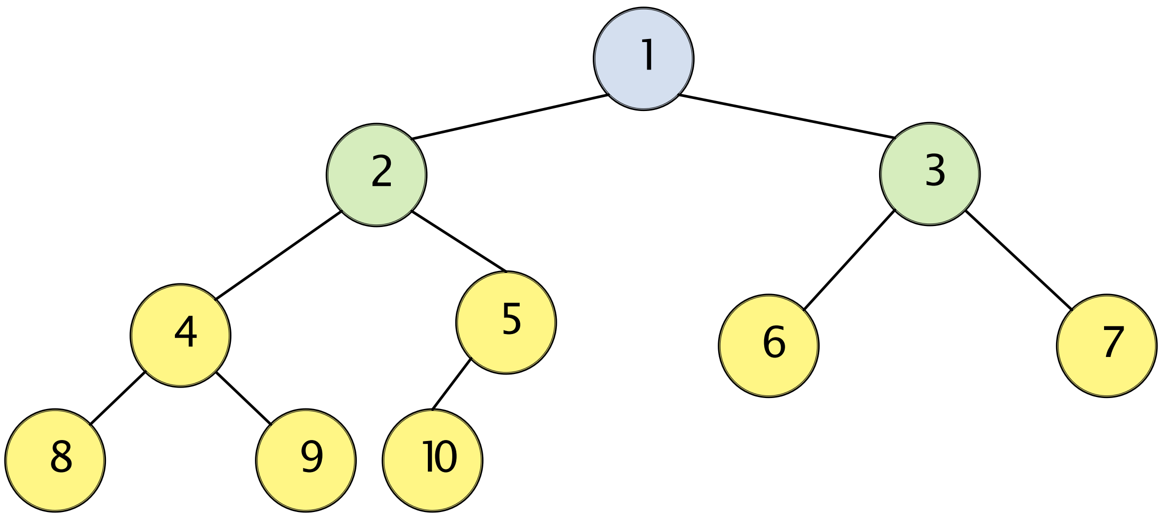
- **Algoritmo:**



	0	1	2	3	4	5	6	7	8	9	...
V	4	2	3	1	5	6	7	8	9	10	

3. HeapSort

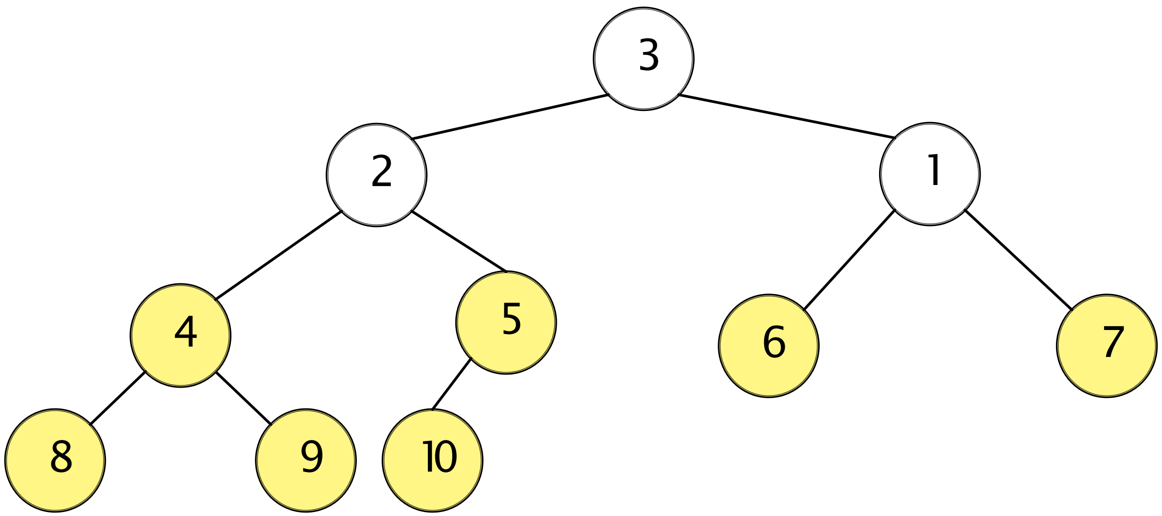
- **Algoritmo:**



	0	1	2	3	4	5	6	7	8	9	...
V	1	2	3	4	5	6	7	8	9	10	

3. HeapSort

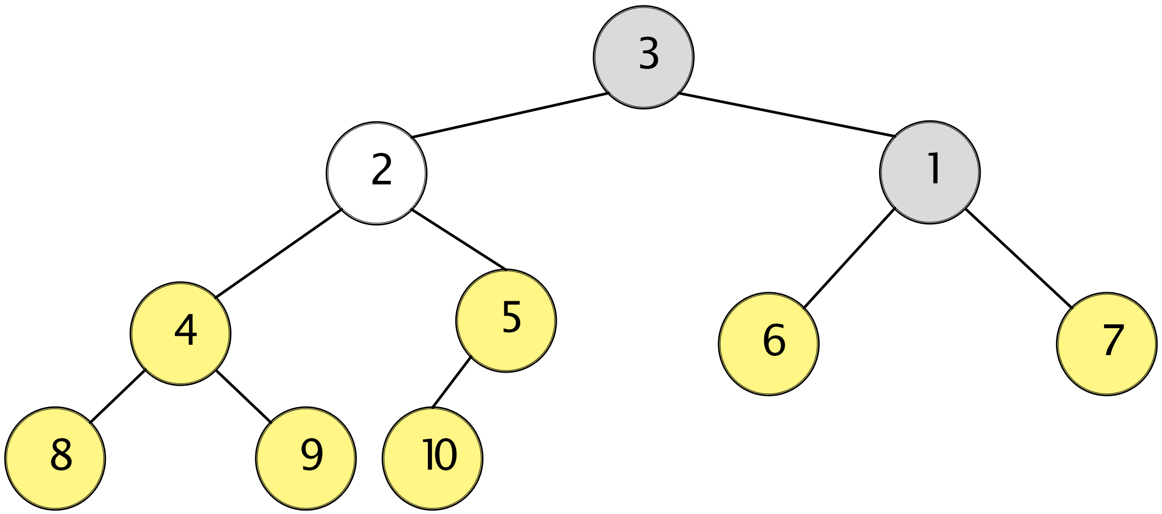
- **Algoritmo:**



	0	1	2	3	4	5	6	7	8	9	...
V	3	2	1	4	5	6	7	8	9	10	

3. HeapSort

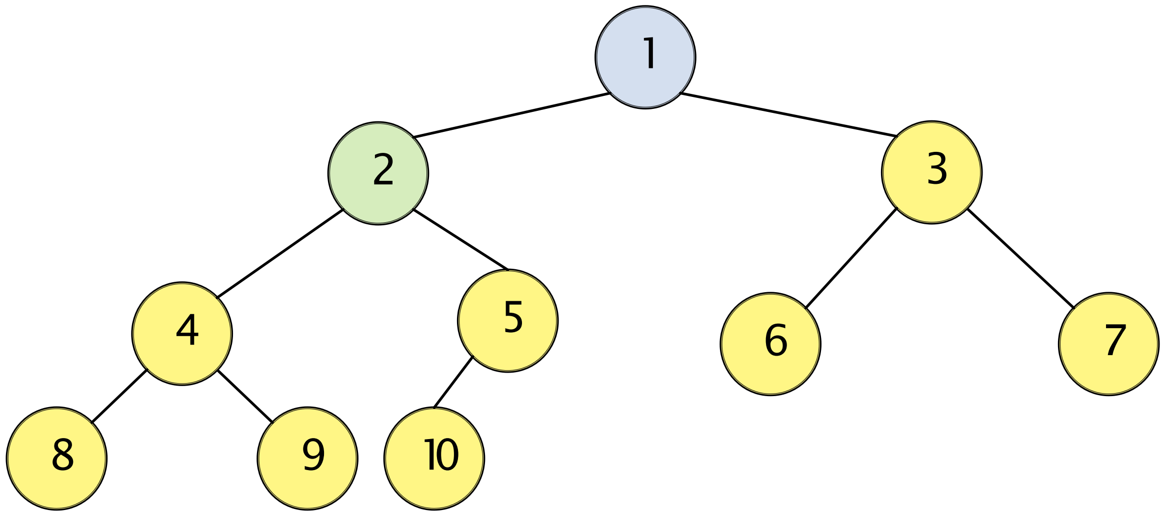
- **Algoritmo:**



	0	1	2	3	4	5	6	7	8	9	...
V	3	2	1	4	5	6	7	8	9	10	

3. HeapSort

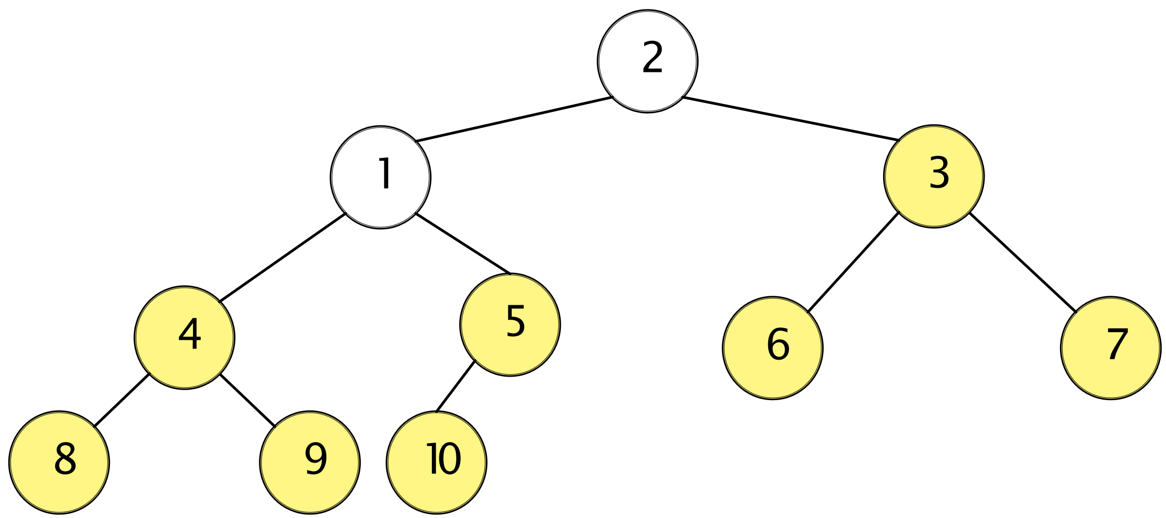
- **Algoritmo:**



	0	1	2	3	4	5	6	7	8	9	...
V	1	2	3	4	5	6	7	8	9	10	

3. HeapSort

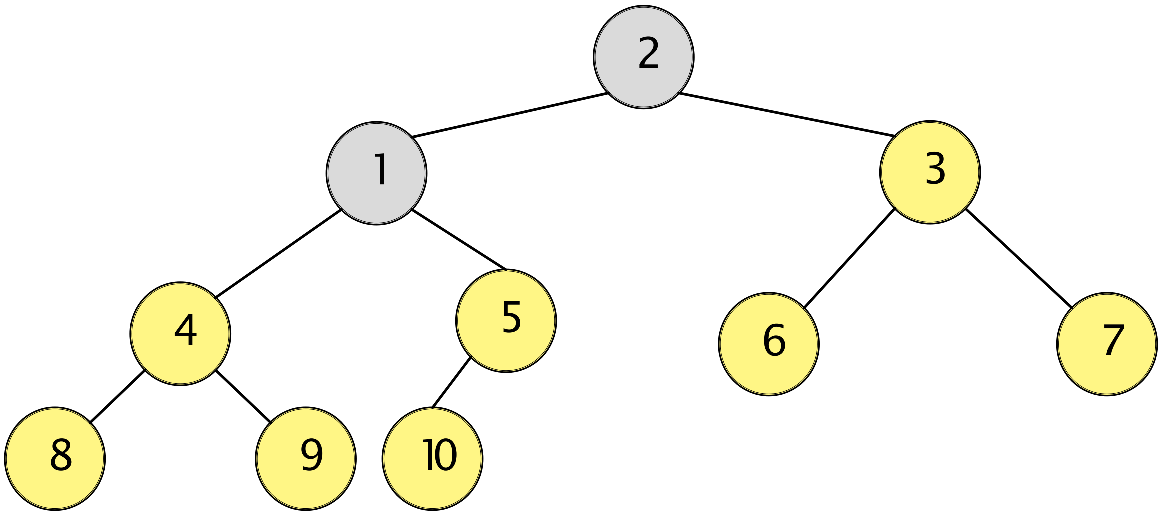
- **Algoritmo:**



	0	1	2	3	4	5	6	7	8	9	...
V	2	1	3	4	5	6	7	8	9	10	

3. HeapSort

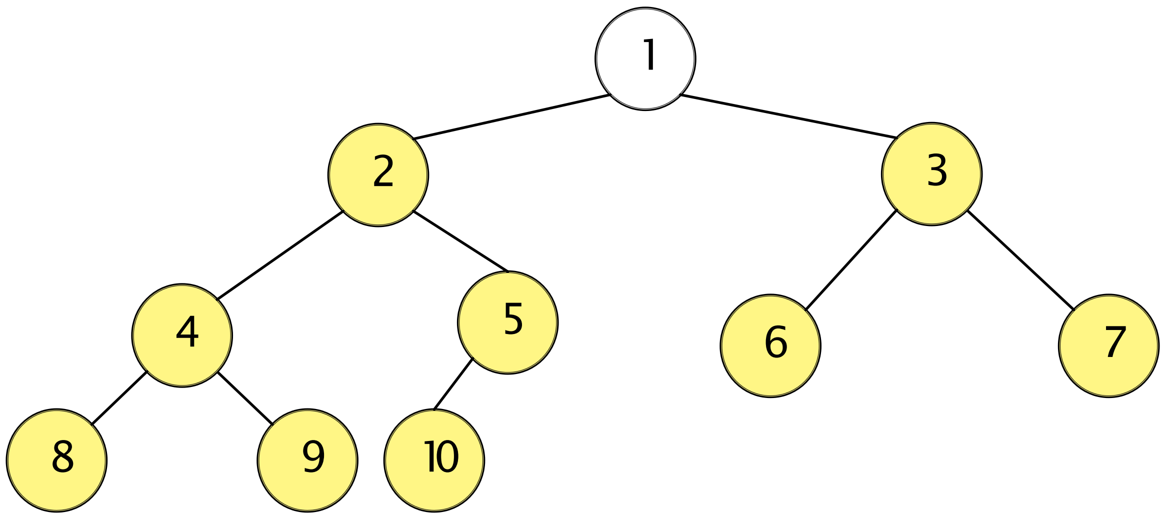
- **Algoritmo:**



	0	1	2	3	4	5	6	7	8	9	...
V	2	1	3	4	5	6	7	8	9	10	

3. HeapSort

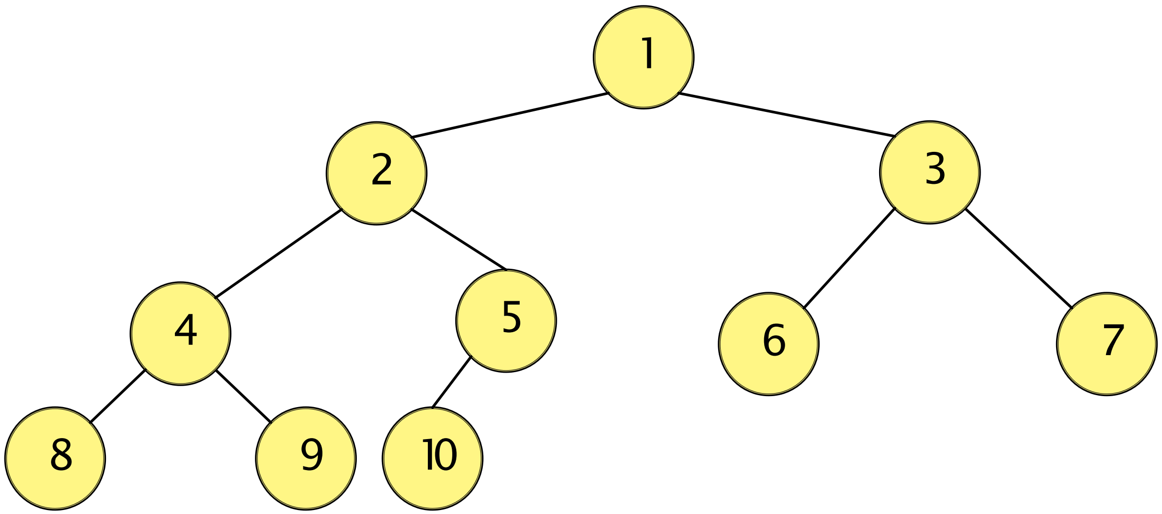
- **Algoritmo:**



	0	1	2	3	4	5	6	7	8	9	...
V	1	2	3	4	5	6	7	8	9	10	

3. HeapSort

- **Algoritmo:**
  - Vetor Ordenado!



	0	1	2	3	4	5	6	7	8	9	...
V	1	2	3	4	5	6	7	8	9	10	

3.1. HeapSort – Implementação

```c
#define F_ESQ(i) (2*i+1) /*Filho esquerdo de i*/
#define F_DIR(i) (2*i+2) /*Filho direito de i*/

void nova_desce_no_heap (int * heap , int n, int k) {
   int maior_filho;
   if (F_ESQ(k) < n) {
      maior_filho = F_ESQ(k);
      if (F_DIR(k) < n && heap[F_ESQ(k)] < heap[F_DIR(k)])
         maior_filho = F_DIR(k);
      if (heap[k] < heap[maior_filho]) {
         int temp;
         temp = heap[k];
         heap[k] = heap[maior_filho];
         heap[maior_filho] = temp;
         nova_desce_no_heap(heap , n, maior_filho);
      }
   }
}
```

3.1. HeapSort – Implementação

```
void heapsort (int * v, int n) {
   int k;
   for (k = n/2; k >= 0; k--) /* transforma em heap */
      nova_desce_no_heap (v, n, k);

   while (n > 1) { /* extrai o máximo */
      int temp;
      temp = v[0];
      v[0] = v[n-1];
      v[n-1] = temp;
      n--;
      nova_desce_no_heap (v, n, 0);
   }
}
```

3.2. HeapSort – Análise

- Como visto anteriormente, o custo do procedimento **nova_desce_no_heap**, no pior caso, é **O(log n)**.
  - No máximo, desce até o último nível da árvore.

- O procedimento **nova_desce_no_heap** é executado **n-1** vezes dentro do laço **while**.

- Logo, a complexidade do algoritmo **HeapSort** é, no pior caso, **O(n log n)**.

3.2. HeapSort – Análise

- **Vantagens:**
  - O comportamento do **HeapSort** é sempre **`O(n log n)`**, qualquer que seja a entrada.

- **Desvantagens:**
  - O anel interno do algoritmo é bastante complexo se comparado com o do **Quicksort.**
  - O **HeapSort** não é estável.

- **Recomendado:**
  - Para aplicações que não podem tolerar eventualmente um caso desfavorável.
  - Não é recomendado para arquivos com poucos registros, por causa do tempo necessário para construir o Heap.

4. Referências

- Material de aula dos Profs. Luiz Chaimowicz e Raquel O. Prates, da UFMG: https://homepages.dcc.ufmg.br/~glpappa/aeds2/AEDS2.1%20Conceitos%20Basicos%20TAD.pdf

- Horowitz, E. & Sahni, S.; Fundamentos de Estruturas de Dados, Editora Campus, 1984.

- Wirth, N.; Algoritmos e Estruturas de Dados, Prentice/Hall do Brasil, 1989.

- Material de aula do Prof. José Augusto Baranauskas, da USP: https://dcm.ffclrp.usp.br/~augusto/teaching.htm

- Material de aula do Prof. Rafael C. S. Schouery, da Unicamp: https://www.ic.unicamp.br/~rafael/cursos/2s2019/mc202/index.html