Heap

Material adicional recomendado para essa anotação: Heaps in 3 minutes - Introduction, e Heaps in 6 minutes - Methods.

Um *heap* binário é uma estrutura de dados usada para implementação de **filas de prioridade**. Em geral é implementado em um arranjo, de forma a ser acessado em uma **árvore binária** (um tipo especial de árvore).

O que são filas de prioridade?

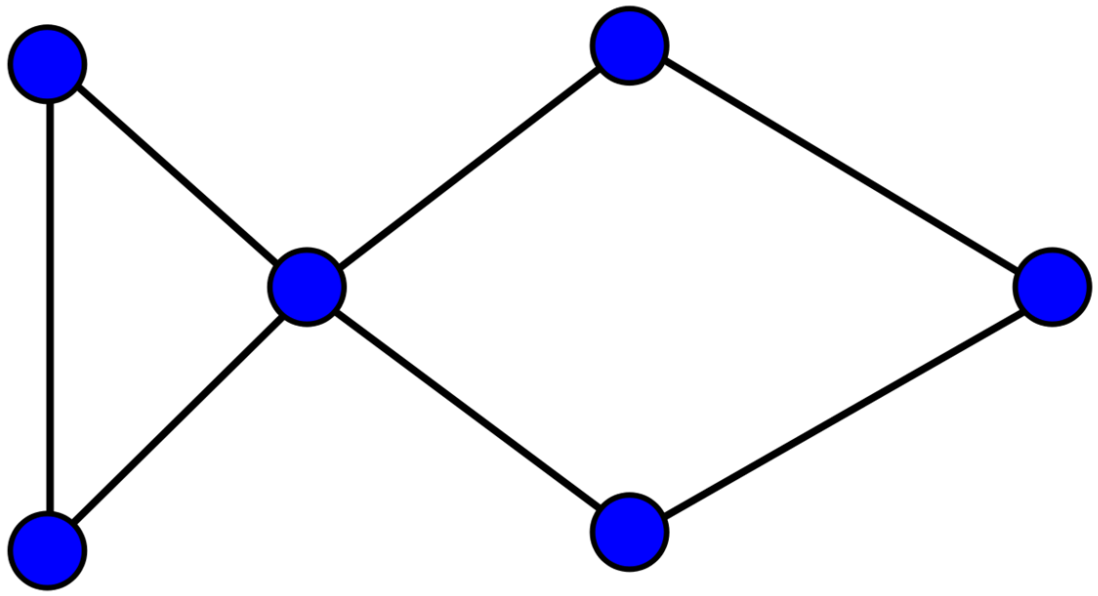
A fila de prioridade é uma abstração importante para *heaps*. Diferente de uma fila normal, onde a ordem é baseada em chegada (FIFO), na fila de prioridade, **os elementos com maior prioridade são removidos primeiro**. É comum usar *heap* para implementar filas de prioridade eficientemente.

O que são grafos?

Um grafo G(V, E) é formado por um conjunto de vértices (V) e um subconjunto de pares de V, chamados arestas (E). Em um grafo, os vértices podem estar conectados diretamente a muitos outros vértices por meio dessas arestas.

Grafos são excelentes estruturas para representar problemas em algoritmos, como o Algoritmo de Dijkstra, o problema do caixeiro-viajante, entre outros. Um exemplo simples de grafos em uma aplicação web é a rede de conexões do LinkedIn, que exibe as pessoas mais próximas de você, com base nas suas conexões.

Na imagem abaixo, os vértices são os círculos azuis, enquanto as arestas são as ligações entre esses círculos.



Termos importantes:

- Um **caminho** é qualquer sequência de vértices que conecta um nó a outro no grafo.
- Se as arestas de um grafo são direcionadas (com uma ordem específica), o grafo é chamado de **dígrafo** (grafo orientado).
- Se **nem todos os vértices** estão conectados entre si, o grafo é considerado **desconexo**, significando que existem vértices que não possuem ligação direta ou indireta com outros.
- Se existe algum nó em um grafo que possui um caminho que volta a ele passando por outros nós, esse grafo é chamado de **cíclico**.

Exemplo de grafo cíclico:

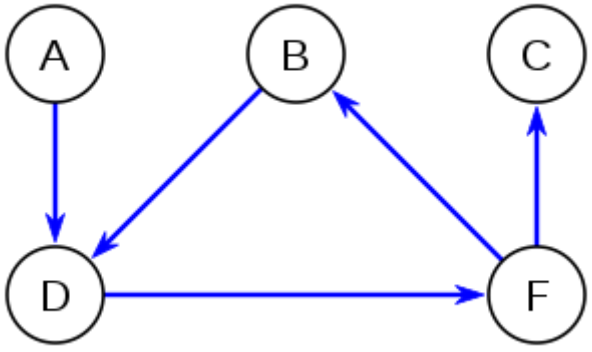


Imagem retirada de uma videoaula do curso ministrado pelo professor Joaquim.

O que é uma árvore?

Uma **árvore** é uma estrutura de dados hierárquica que pode ser vista como um tipo especial de **grafo**. Ela é formada por **nós** (ou vértices) conectados por **arestas**, organizados de modo que cada nó possa ter **zero ou mais filhos**. Árvores são amplamente utilizadas em ciência da computação para representar estruturas hierárquicas, como sistemas de arquivos, organização de dados em memória, e muitos algoritmos de busca e ordenação.

Árvores são grafos, obrigatoriamente: **conexos** (existe caminho entre quaisquer dois de seus vértices) e **acíclicos** (não há caminhos que permitam retornar ao mesmo nó depois de passar por outros vértices).

Árvores Enraizadas

Em computação, utilizamos frequentemente **árvores enraizadas**, que possuem um vértice especial conhecido como **raiz**. Ao representar árvores enraizadas, a raiz é **geralmente destacada na parte superior da estrutura**, refletindo a hierarquia da árvore.

Estrutura de uma Árvore Enraizada

Uma árvore enraizada é composta por:

1. **Raiz**: O nó que serve como **ponto de origem** para toda a árvore. Geralmente há algum significado especial associado a esse nó, que justifica sua escolha como nó mais importante.
2. **Nós Filhos**: Os nós diretamente conectados à raiz ou a outros nós. Cada nó pode ter zero ou mais filhos.
3. **Folhas**: Nós que não têm filhos, ou seja, são os terminais da árvore.

Exemplo de Árvore Enraizada

Vamos considerar uma árvore enraizada simples representada no terminal:

```
      A          (Raiz)
     / \
    B   C        (Filhos de A)
   / \
  D   E          (Filhos de B)
```

Descrição da Estrutura

- **Raiz**: O nó **A** é a raiz da árvore.
- **Filhos**:
  - **B** e **C** são filhos da raiz **A**.
  - **D** e **E** são filhos do nó **B**.
- **Folhas**:
  - **C**, **D** e **E** são folhas, pois não possuem filhos.

Definições Importantes

Na análise de árvores enraizadas, consideramos também algumas definições importantes:

- **Grau**: O grau de um nó é o **número de filhos** que ele possui. Por exemplo:
  - O grau do nó **A** (raiz) é **2**, pois tem **B** e **C** como filhos.
  - O grau do nó **B** é **2**, pois tem **D** e **E** como filhos.
  - O grau dos nós **C**, **D** e **E** é **0**, pois não têm filhos.
- **Profundidade**: A profundidade de um nó é a **distância (número de arestas) até a raiz**. Os nós que estão a uma mesma distância da raiz estão no mesmo nível. Exemplos:
  - A profundidade de **A** (raiz) é **0**.
  - A profundidade de **B** e **C** é **1**.
  - A profundidade de **D** e **E** é **2**.
- **Altura da Árvore**: A altura da árvore é definida como a maior profundidade entre todos os nós. Neste exemplo, a altura da árvore é **2**, pois o nó mais profundo é **D** ou **E**, que estão a **2** arestas da raiz.
- **Grau da Árvore**: O grau da árvore é o maior grau de seus nós. Neste exemplo:
  - O grau da árvore é **2**, uma vez que o nó **B** e a raiz **A** têm o maior número de filhos.

O que é um *Heap*?

Um *heap* é uma árvore **binária balanceada** e **completa**.

- **Árvore Binária**: Um *heap* é uma árvore em que cada nó **pode ter no máximo dois filhos**. Isso garante uma estrutura hierárquica e organizada.
- **Árvore Completa**: Um *heap* deve ser uma árvore binária completa, o que significa que **todos os níveis da árvore, exceto possivelmente o último, estão completamente preenchidos, e todos os nós do último nível estão o mais à esquerda possível**.
- **Árvore Balanceada**: Um *heap* é também balanceado, o que implica que **todos os nós folhas estão no mesmo nível ou, no máximo, com um nível de diferença**.

Isso garante que as operações de inserção e remoção tenham um desempenho eficiente.

Heap Máximo

Em um *heap* máximo, **cada nó é maior ou igual a seus filhos**. Portanto, o **maior elemento da árvore está sempre na raiz**. Essa propriedade permite que a remoção do maior elemento (o topo) seja realizada rapidamente.

```
     10
    /  \
   9    8
  / \  / \
 7  6 5   4
```

Heap Mínimo

Em um *heap* mínimo, **cada nó é menor ou igual a seus filhos**. Assim, **o menor elemento da árvore está sempre na raiz**. Isso é útil quando se deseja acessar rapidamente o menor elemento.

```
     1
    / \
   3   2
  / \ / \
 7  6 5  4
```

Implementando um *heap*

Para maior desempenho nas operações, utilizaremos arranjos para a implementação. Essa escolha só é possível devido às características dessa estrutura de dados (ou seja, o fato de ser uma árvore binária, completa e balanceada).

Para que serve um *heap*?

Um *heap* é uma forma prática e eficiente de implementar filas de prioridade, além de poder ser usado em outros contextos, como na ordenação.

Para utilizarmos um *heap* para construção de filas de prioridades, devemos:

1. Remover a raiz.

2. Substituí-la pelo último elemento da ED.
   Para garantir que o *heap* continue completo e balanceado.
3. Reorganizar o *Heap*.

Operações básicas em um *heap*

Geralmente, utilizamos métodos de reorganização em *heaps* quando o utilizamos para a implementação de filas de prioridade. Esses métodos são:

- `criarHeap()/destruirHeap()`: Criação e destruição da estrutura de dados.
- `arruma()` / `heapify()` / `constroi-heap()`: Método auxiliar utilizado para gerar o *heap*.
- `retiraRaiz()`: Retira a raiz do *heap*, trocando-a de posição com o elemento na última posição da estrutura de dados.
- `corrigeDescendo()`: Caso um elemento seja menor que um de seus filhos, efetua-se a troca de valores e repete-se o processo no nó filho. Esse método é **utilizado na retirada da raiz**.

  Como funciona a `corrigeDescendo()`?

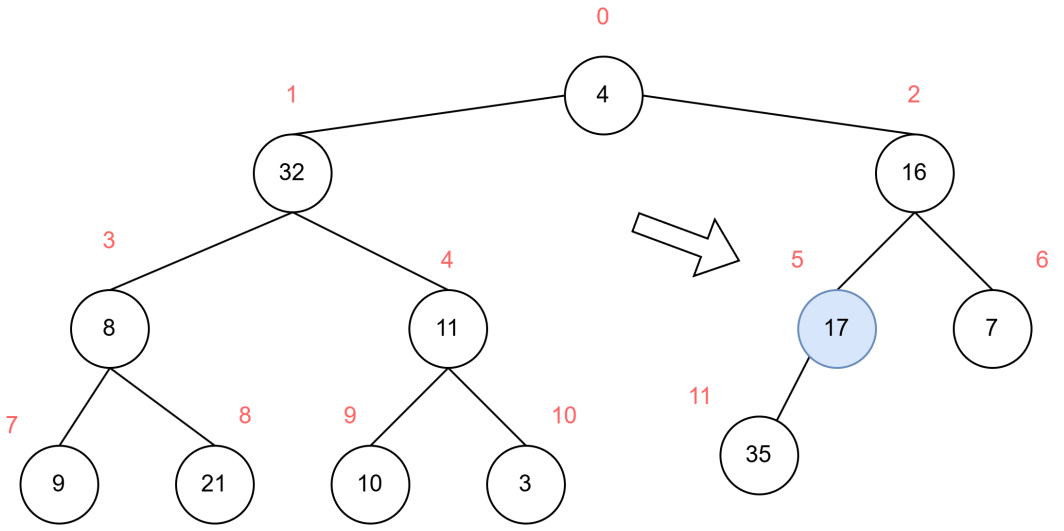
  OBS: Estou tratando aqui de um *max-heap*.
  Como é sabido, utilizamos a `corrigeDescendo()` ao criar um *heap* a partir de um vetor já existente. Normalmente, ao criarmos esse *heap*, procuramos onde está demarcada a **primeira metade** (**parte superior**) dos elementos, indo até a raiz.

  Abaixo, nesse vetor de 12 elementos, temos como posição que marca o início da metade do superior como a posição 5 do vetor (representada pelo elemento "17").

4 | 32 | 16 | 8 | 11 | 17 | 7 | 9 | 21 | 10 | 3 | 35

Primeira metade: (n/2) - 1

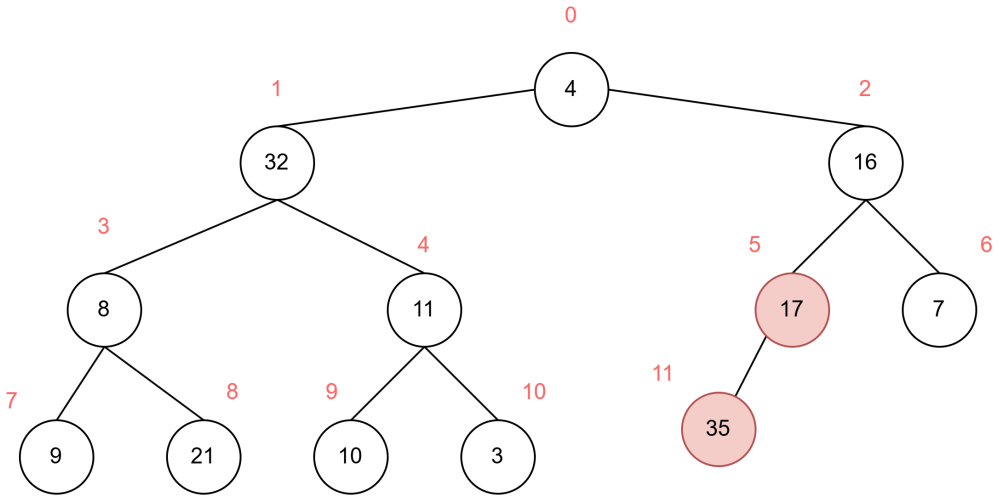
(12/2) - 1 = 5



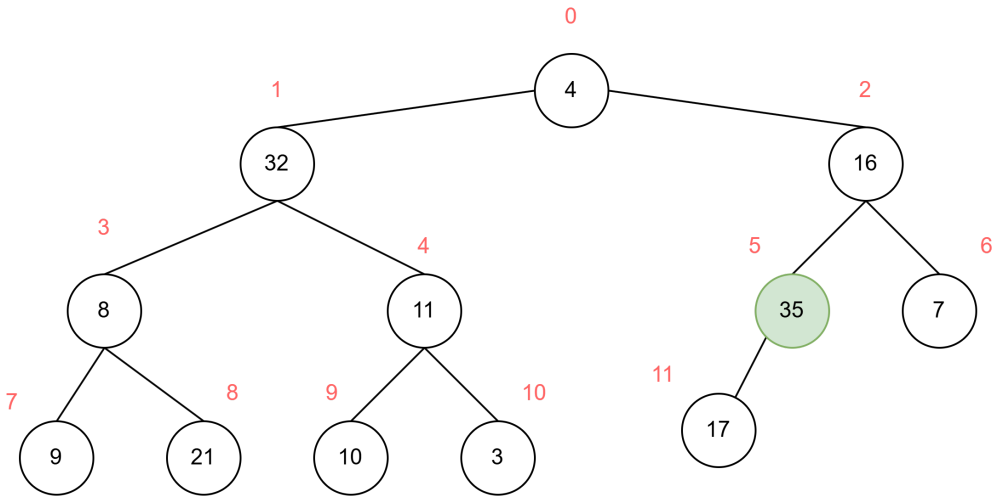
Com a metade superior demarcada, iremos começar um processo de correção a partir dos seus nós filhos. Checaremos se, cada um dos seus filhos, é maior que o pai, e caso haja alguma troca, o processo irá se repetir no filho trocado.

1. O filho "35" de "17" é maior que o pai! Haverá troca!

4 | 32 | 16 | 8 | 11 | 17 | 7 | 9 | 21 | 10 | 3 | 35

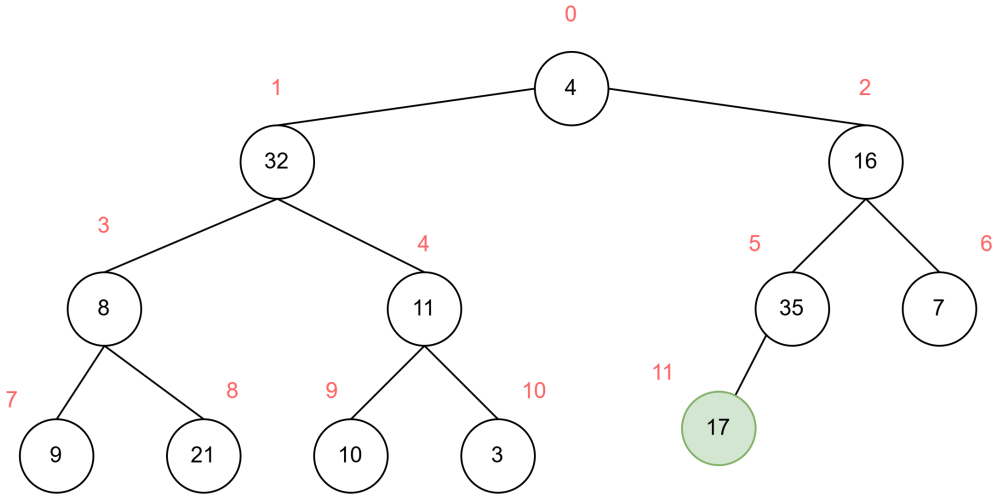


4	32	16	8	11	35	7	9	21	10	3	17



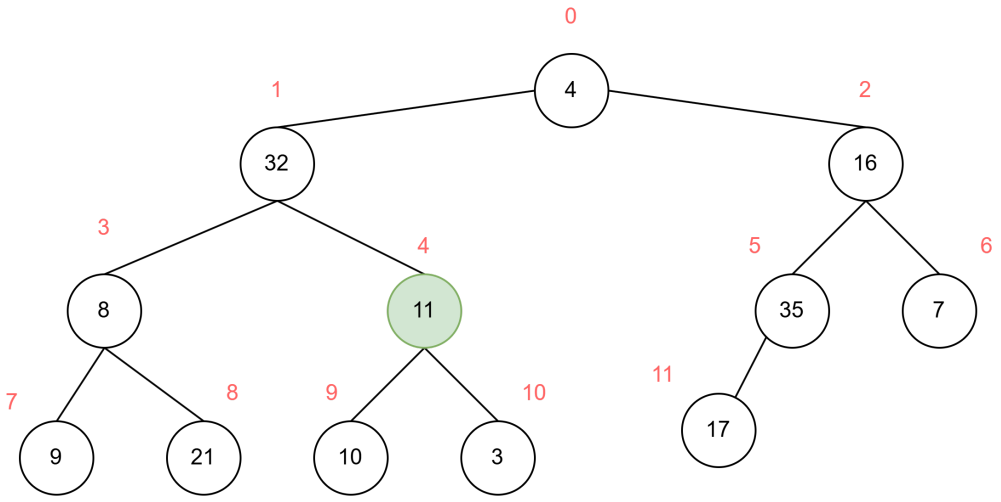
2. Não há nenhum filho de "17", portanto, não haverá troca.

4	32	16	8	11	35	7	9	21	10	3	17



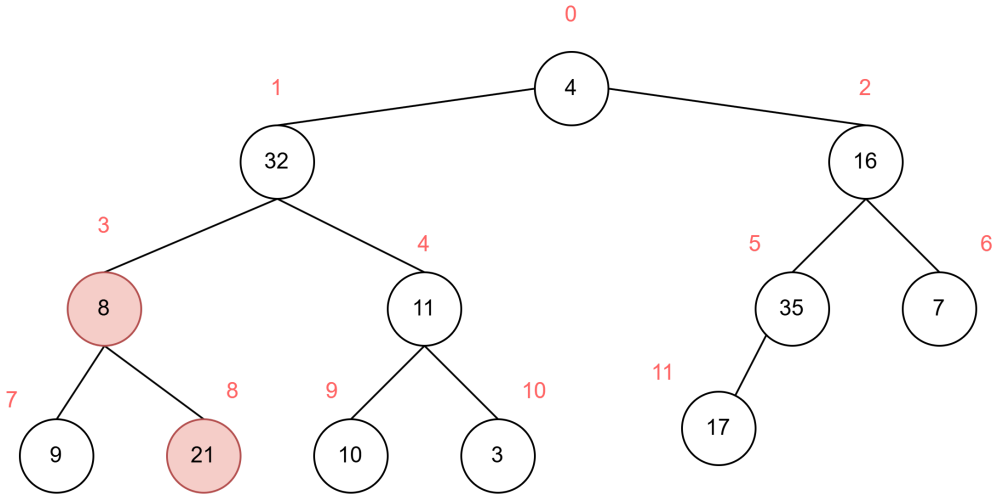
3. Nenhum filho de "11" é maior que ele, portanto, passamos para o próximo nó.

4 | 32 | 16 | 8 | 11 | 35 | 7 | 9 | 21 | 10 | 3 | 17

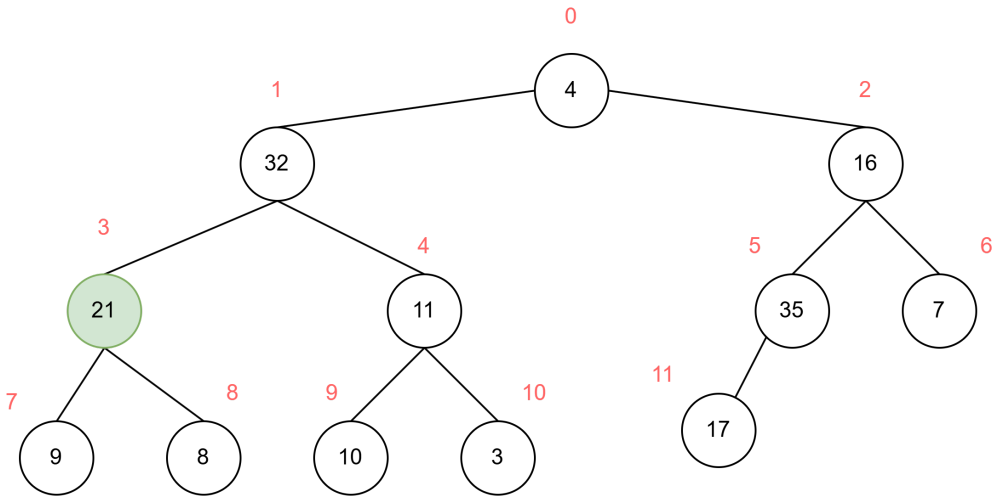


4. O filho "21" de "8" é maior que o pai! Haverá troca!

4 | 32 | 16 | 8 | 11 | 35 | 7 | 9 | 21 | 10 | 3 | 17

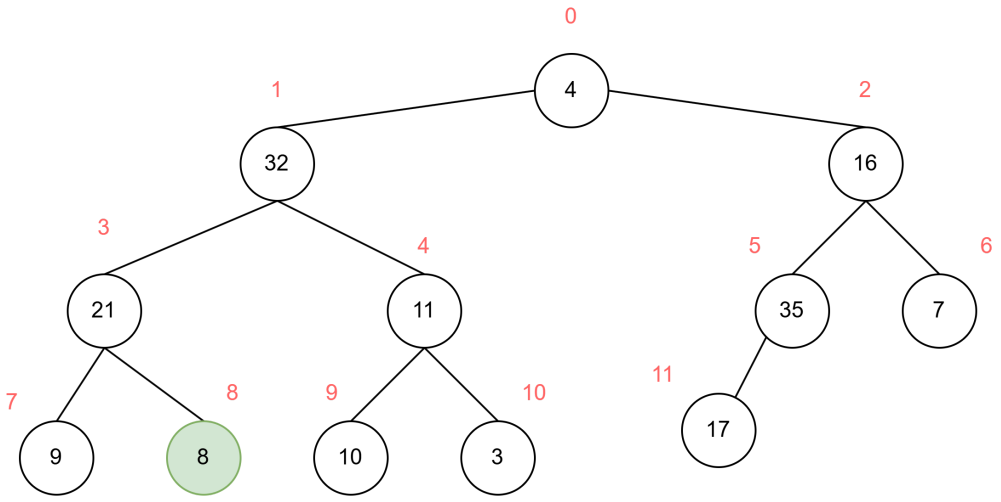


4 | 32 | 16 | 21 | 11 | 35 | 7 | 9 | 8 | 10 | 3 | 17



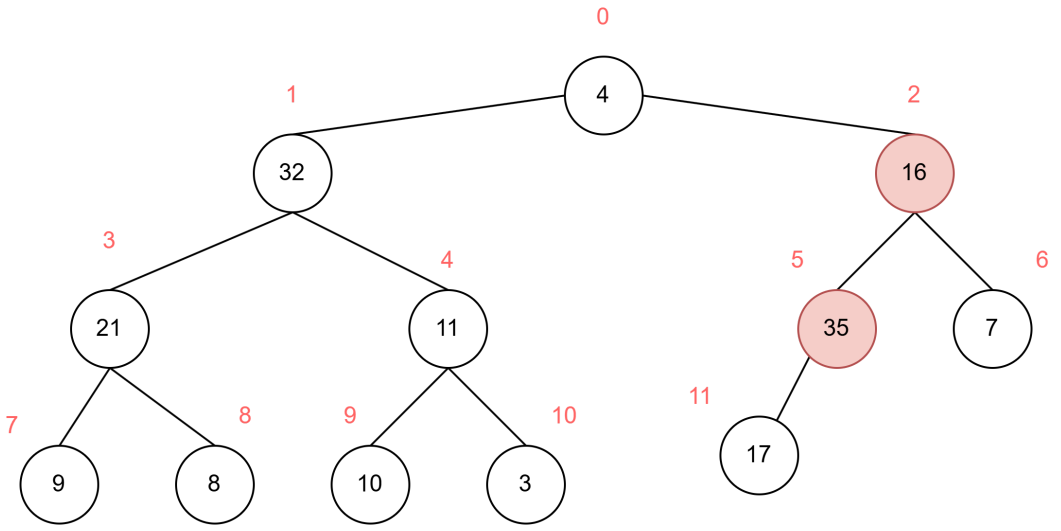
5. Não há nenhum filho de "8", portanto, não haverá troca.

4 | 32 | 16 | 21 | 11 | 35 | 7 | 9 | 8 | 10 | 3 | 17

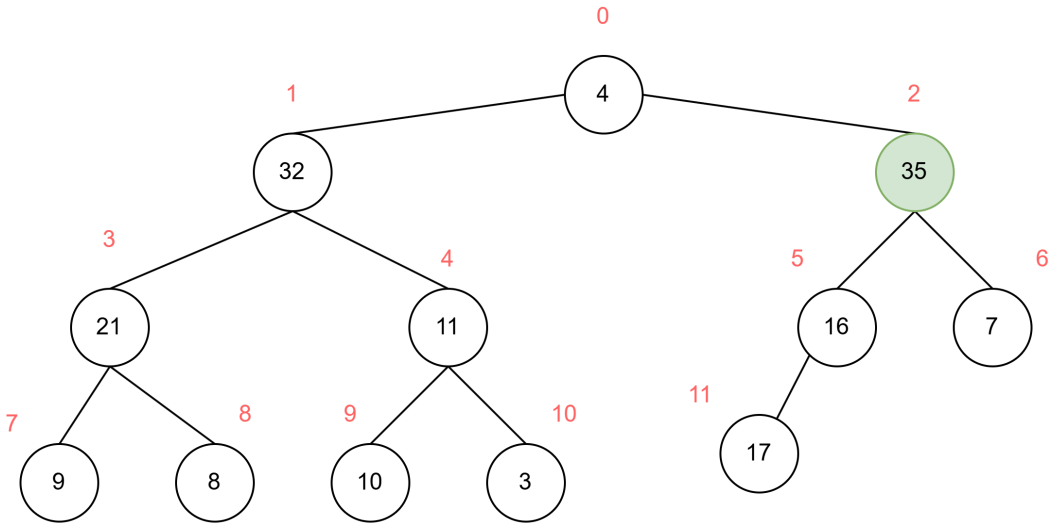


6. O filho "35" de "16" é maior que o pai! Haverá troca!

4 | 32 | 16 | 21 | 11 | 35 | 7 | 9 | 8 | 10 | 3 | 17

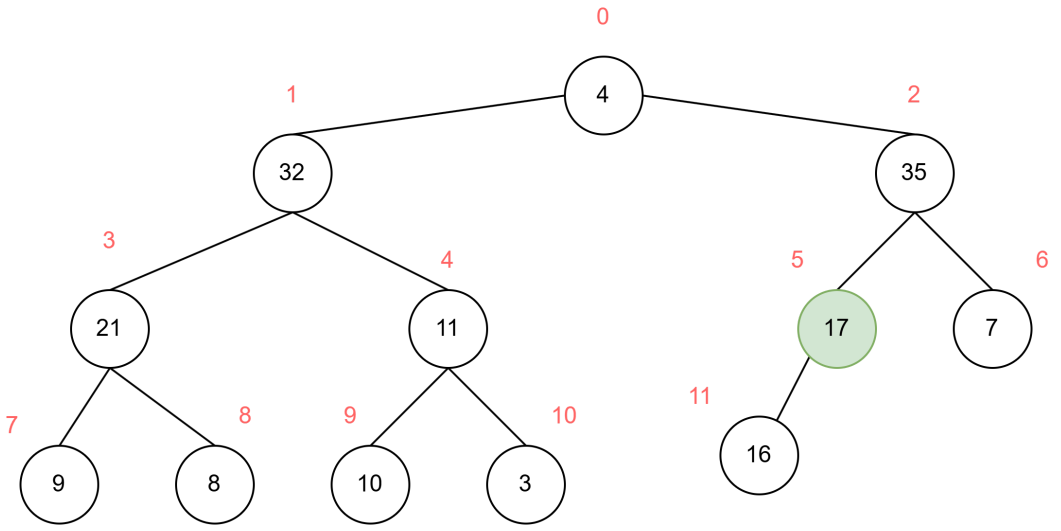


4 | 32 | 35 | 21 | 11 | 16 | 7 | 9 | 8 | 10 | 3 | 17



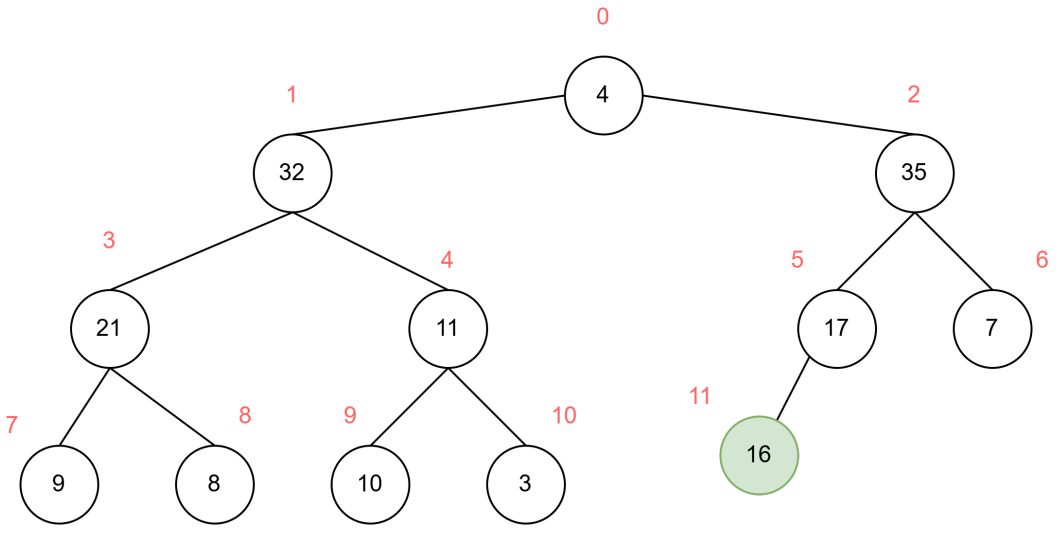
7. O filho "17" de "16" é maior que o pai! Haverá troca!

4 | 32 | 35 | 21 | 11 | 17 | 7 | 9 | 8 | 10 | 3 | 16



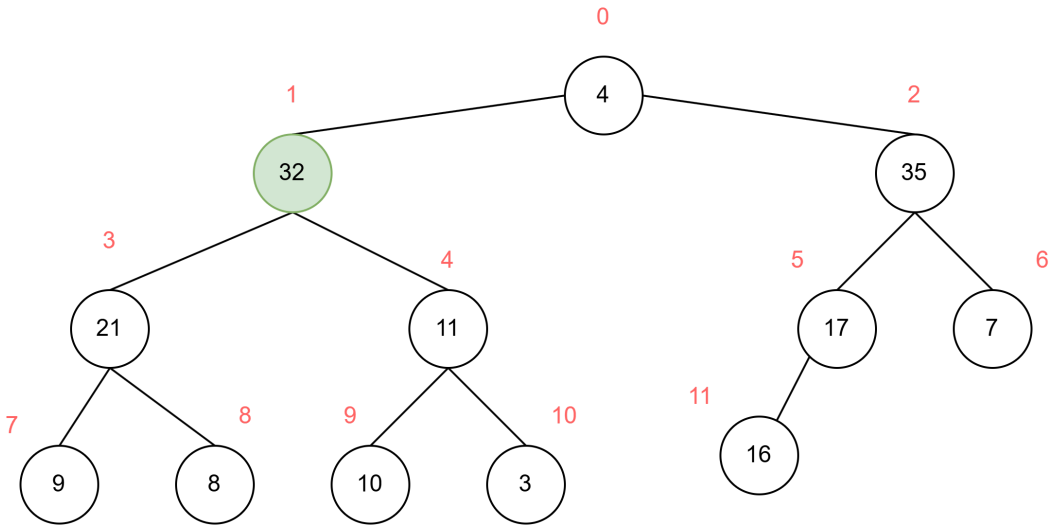
8. Não há nenhum filho de "16" e, portanto, não haverá troca.

4 | 32 | 35 | 21 | 11 | 17 | 7 | 9 | 8 | 10 | 3 | 16



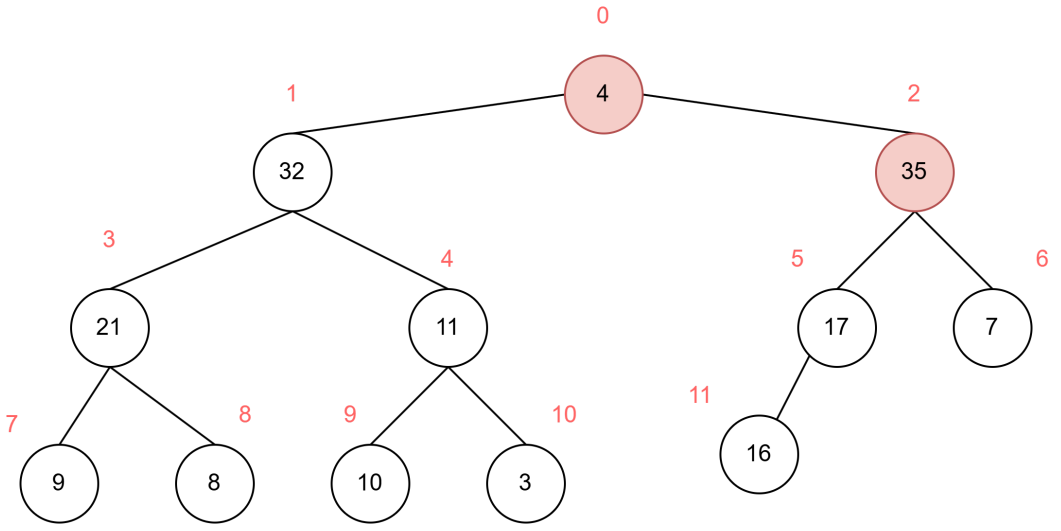
9. Não há nenhum filho de "32" maior que ele, portanto, não haverá troca.

4	32	35	21	11	17	7	9	8	10	3	16

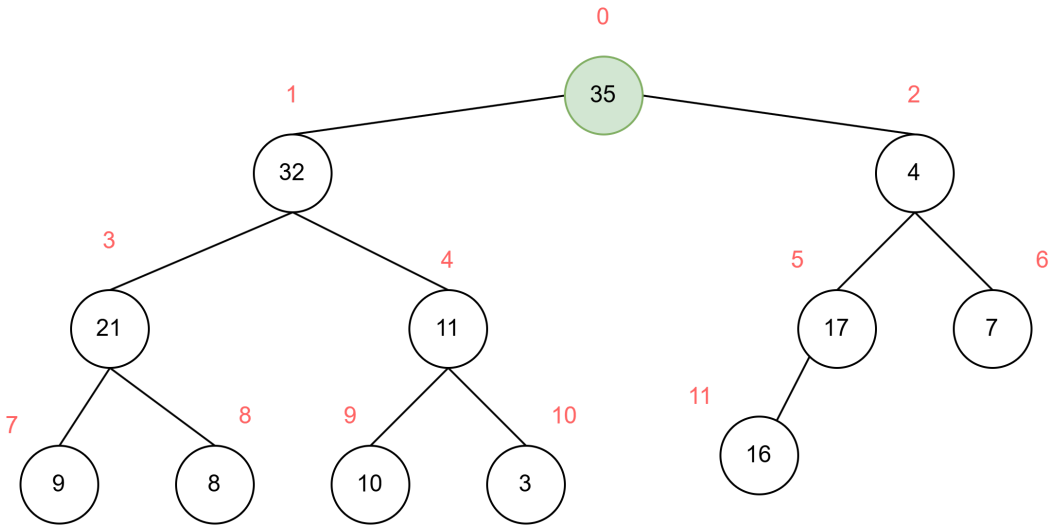


10. Existem dois filhos de "4" maiores que o pai, sendo o filho "35" o maior. Haverá troca!

4	32	35	21	11	17	7	9	8	10	3	16

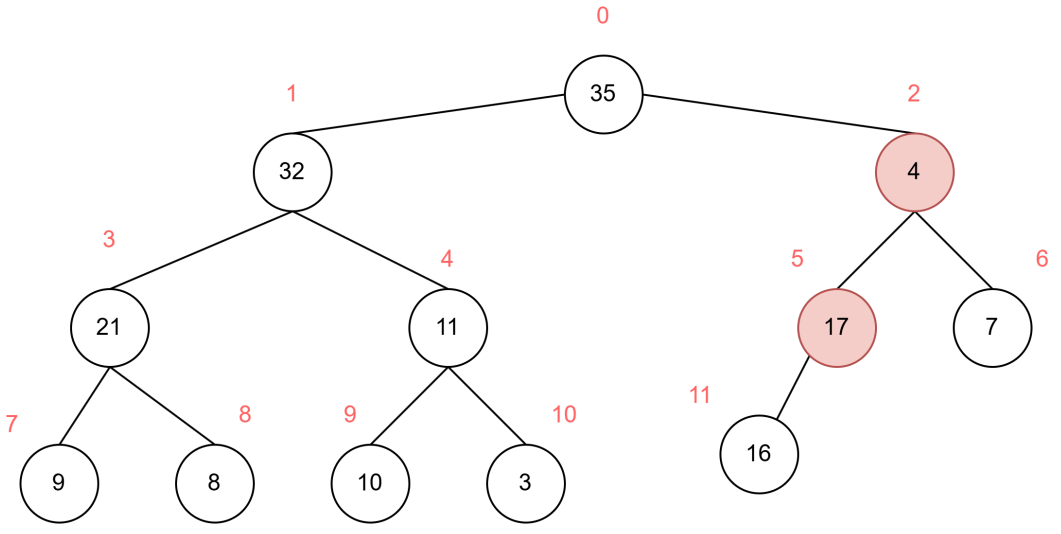


35 | 32 | 4 | 21 | 11 | 17 | 7 | 9 | 8 | 10 | 3 | 16



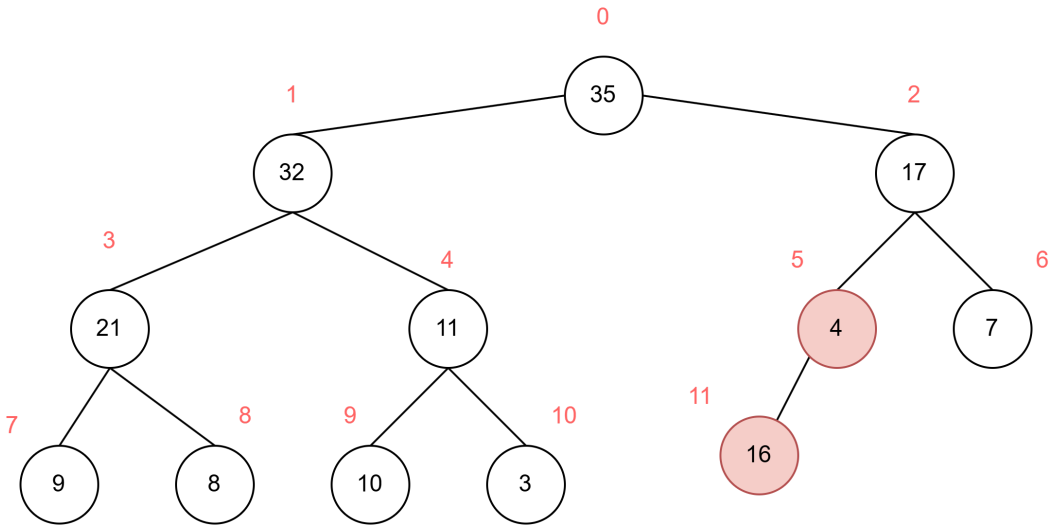
11. Existem dois filhos de "4" maiores que o pai, sendo o filho "17" o maior. Haverá troca!

35 | 32 | 4 | 21 | 11 | 17 | 7 | 9 | 8 | 10 | 3 | 16

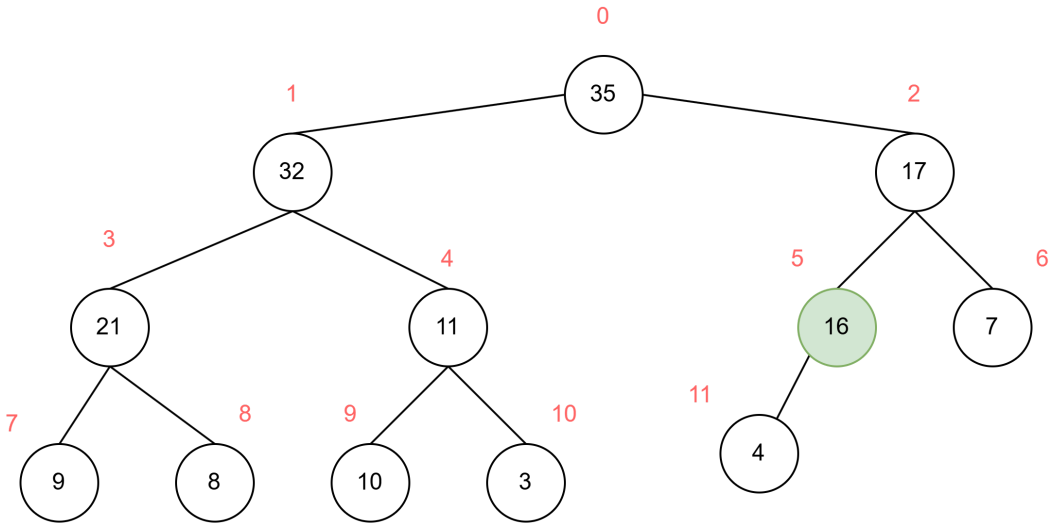


12. O filho "17" de "4" é maior que o pai! Haverá troca!

35 | 32 | 17 | 21 | 11 | 4 | 7 | 9 | 8 | 10 | 3 | 16

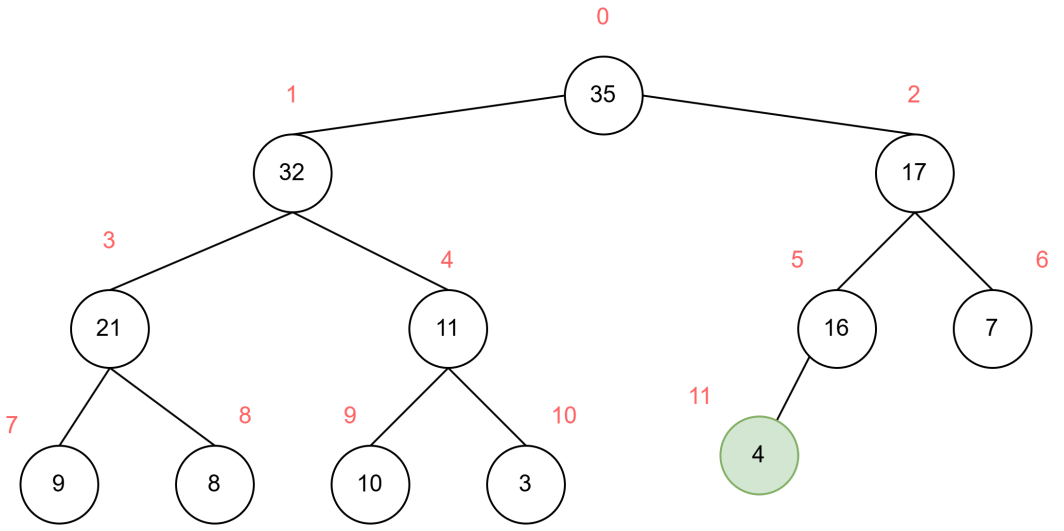


35 | 32 | 17 | 21 | 11 | 16 | 7 | 9 | 8 | 10 | 3 | 4



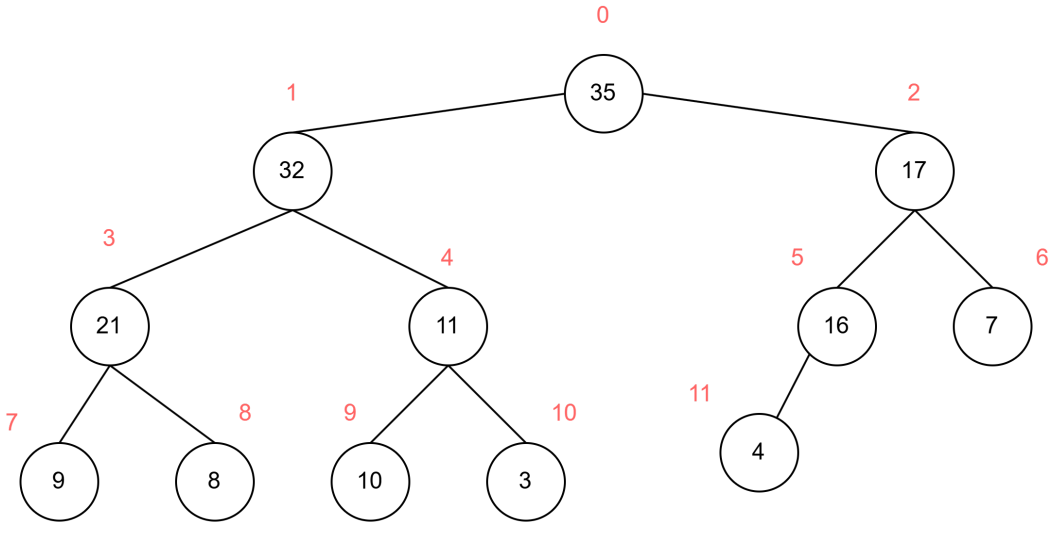
13. Não há nenhum filho de "4" e, portanto, não haverá troca.

35 | 32 | 17 | 21 | 11 | 16 | 7 | 9 | 8 | 10 | 3 | 4



Assim, aplicamos `corrigeDescendo()` em todos os elementos da metade superior do *heap* até a raiz, o organizando corretamente.

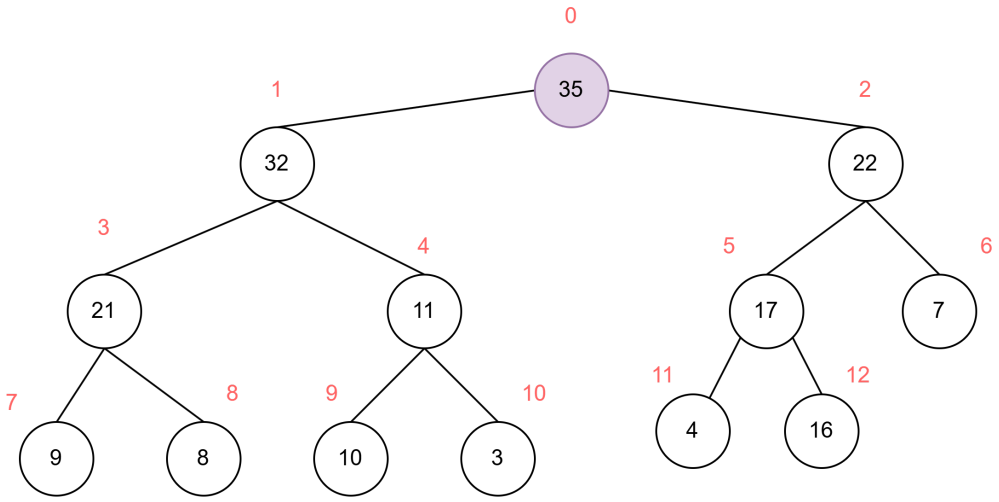
35 | 32 | 17 | 21 | 11 | 16 | 7 | 9 | 8 | 10 | 3 | 4

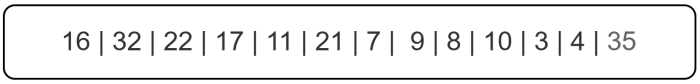


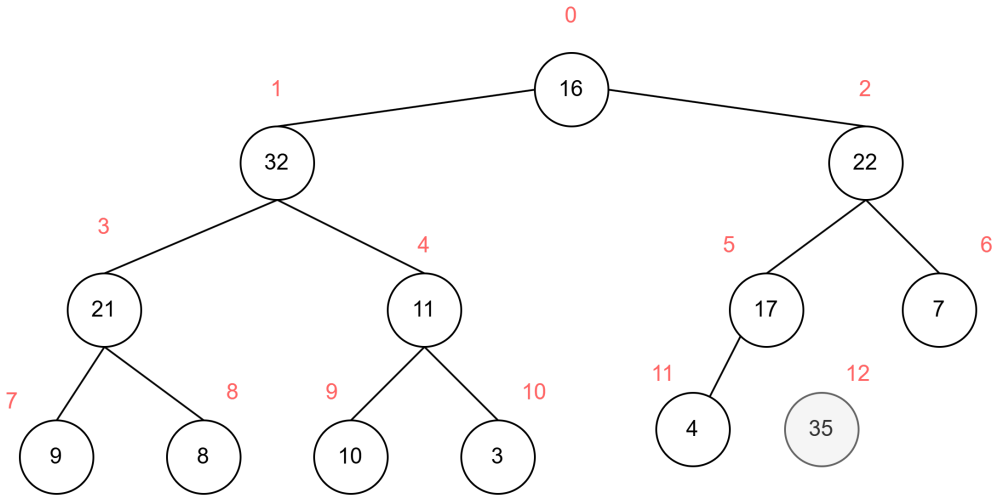
Como usar a `corrigeDescendo()` na retirada da raiz?

OBS: Estou tratando aqui de um *max-heap*.
Usando um *heap* já ordenado, com 13 elementos, iremos retirar sua raiz, o elemento "35".

1. Para isso, iremos substituir a raiz com o elemento na última posição do *heap* ("16").

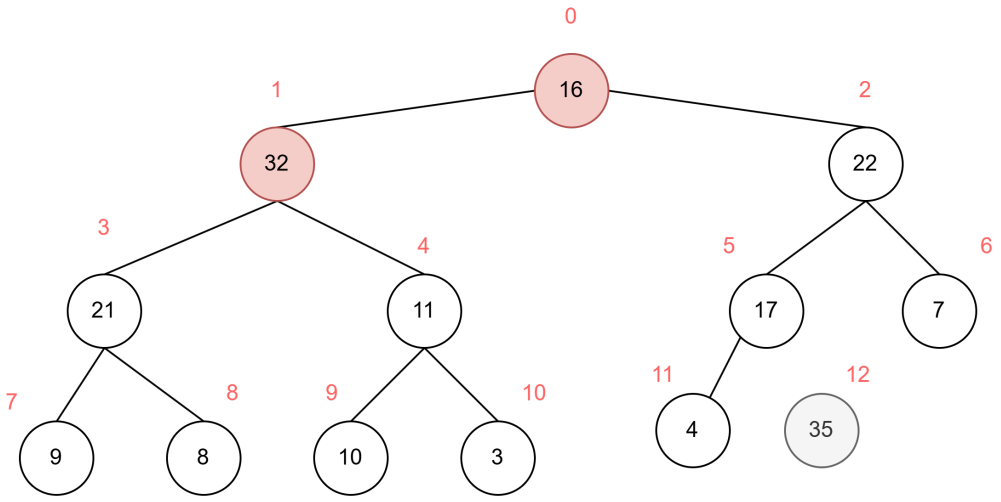




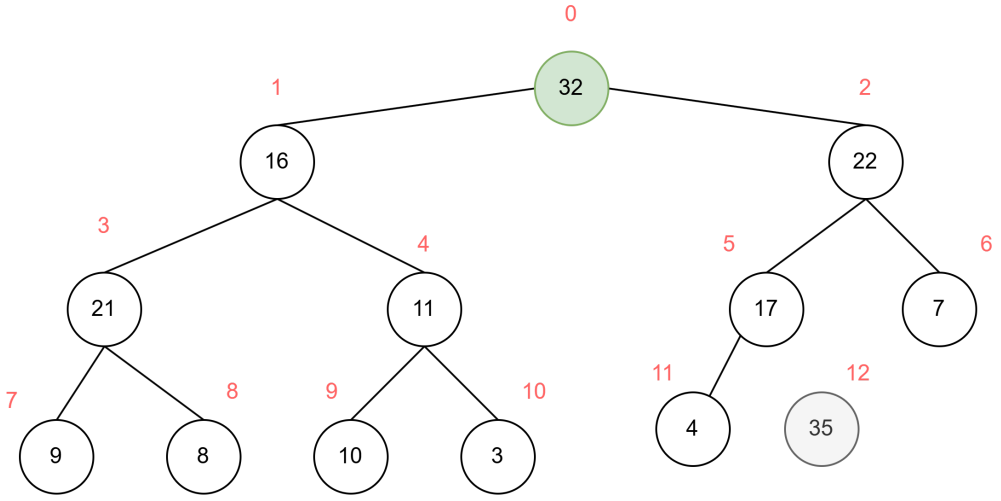


2. Agora, aplicaremos a `corrigeDescendo()`, substituindo o elemento "16" por filhos maiores que ele, até não restar mais nenhum filho menor que ele que possibilite a troca. Iniciaremos esse processo o trocando com seu maior filho atual, o elemento "32".

16 | 32 | 22 | 17 | 11 | 21 | 7 | 9 | 8 | 10 | 3 | 4 | 35

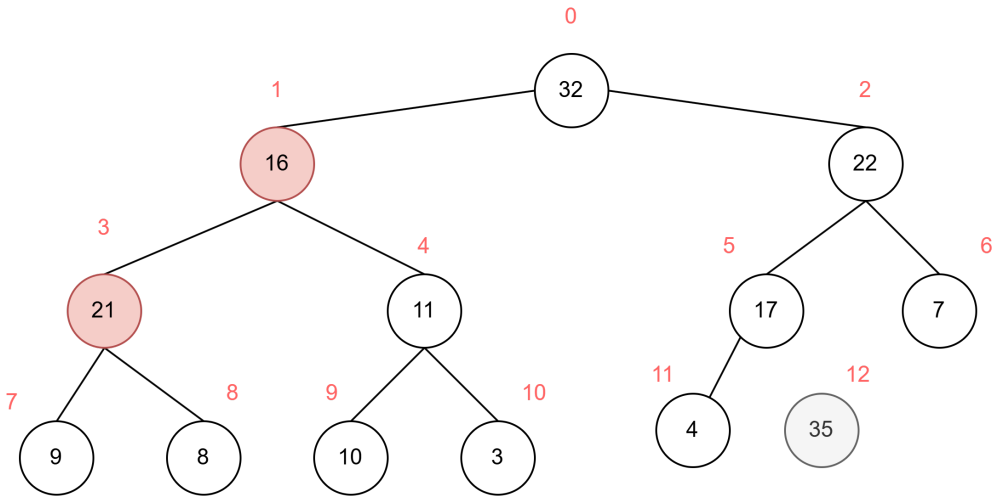


32 | 16 | 22 | 17 | 11 | 21 | 7 | 9 | 8 | 10 | 3 | 4 | 35

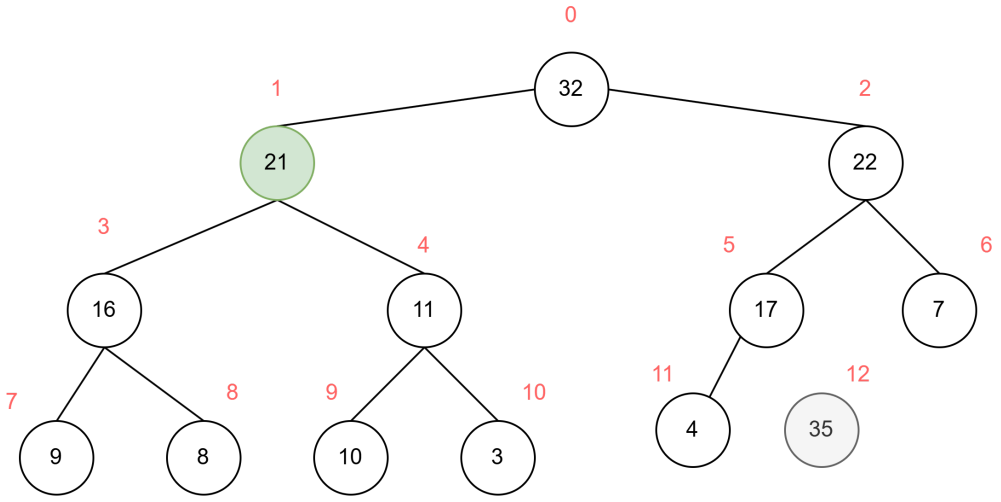


3. O filho "21" de "16" é maior que o pai, portanto, haverá troca.

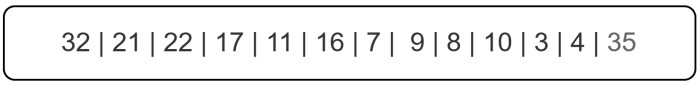
32	16	22	17	11	21	7	9	8	10	3	4	35

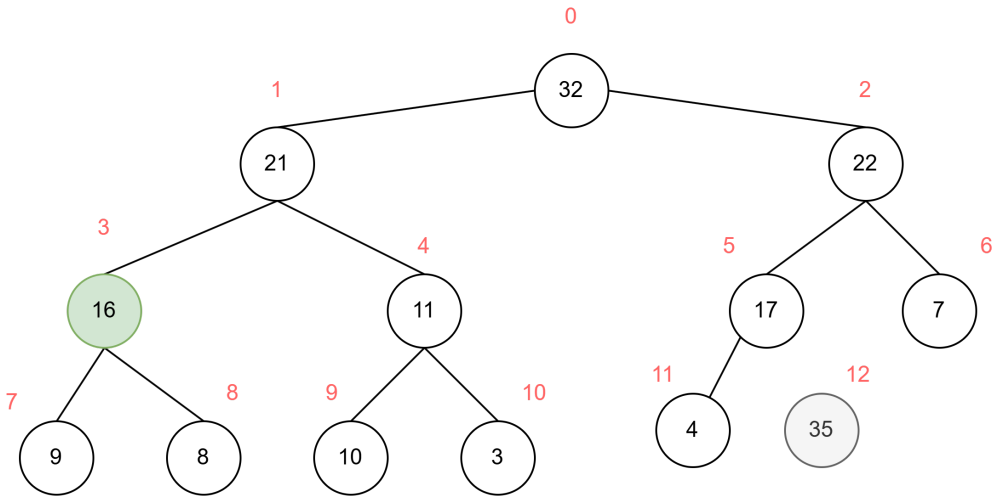


32	21	22	17	11	16	7	9	8	10	3	4	35

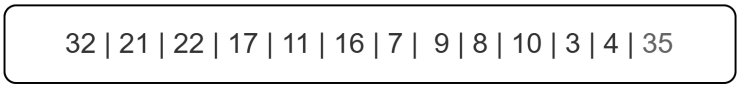


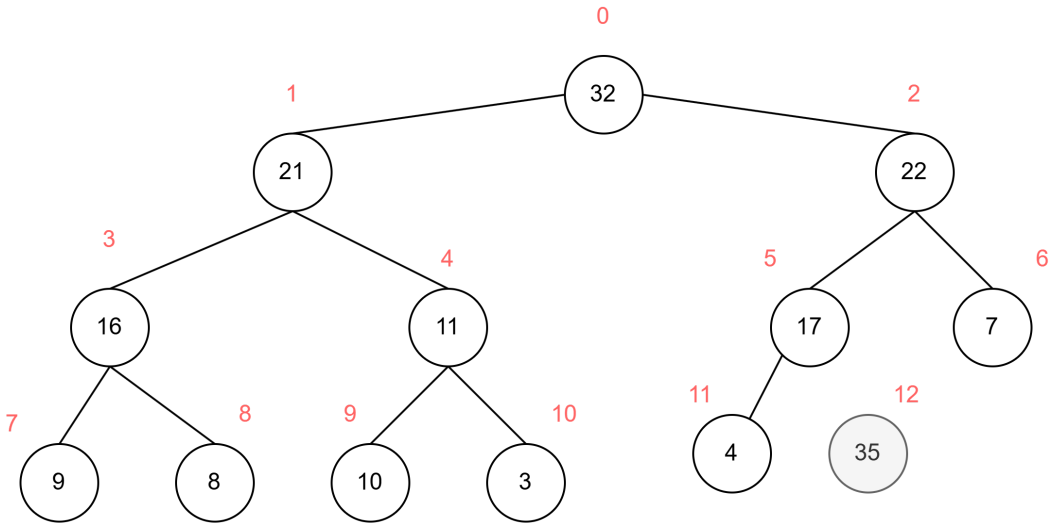
4. Não há nenhum elemento filho de "16" que seja maior que ele, encerramos o `corrigeDescendo()` aqui.





Ao fim desse processo, temos como nova raiz o número "32", tendo sido excluído o elemento "35" (apesar de ainda persistir no vetor, como efeito colateral da troca de posições).



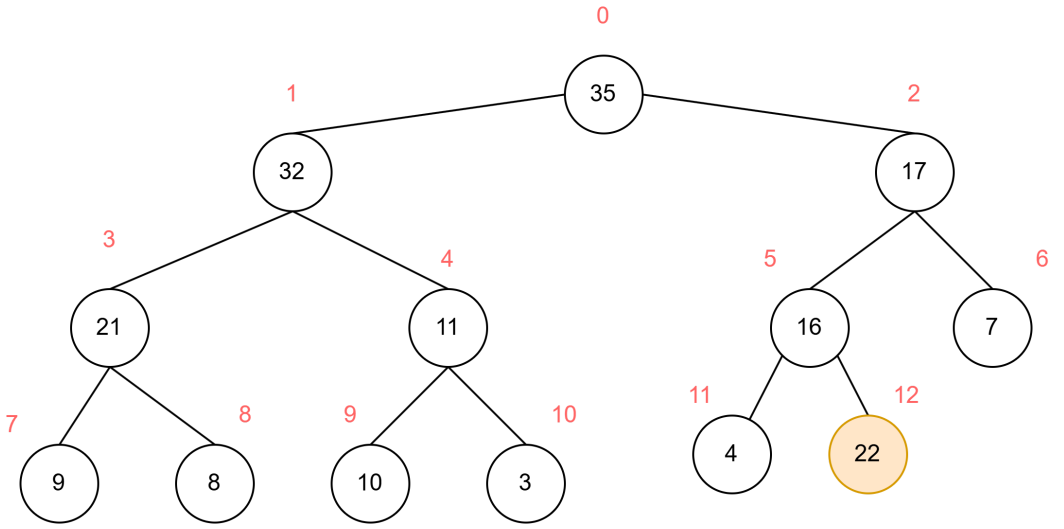


- `insere()`: Insere um novo elemento no *heap*.
- `corrigeSubindo()`: Caso um elemento seja maior que seu pai, efetua-se a troca de valores e repete-se o processo no nó pai. Esse método é utilizado na i**nserção de um novo elemento no *heap***.

Como funciona a `corrigeSubindo()`?

OBS: Estou tratando aqui de um *max-heap*.
Usando, como exemplo, um *heap* já ordenado, digamos que queremos fazer a inserção de um novo elemento ("21"). Para isso, devemos alocá-lo à última posição disponível no *heap*, que seria a décima segunda.

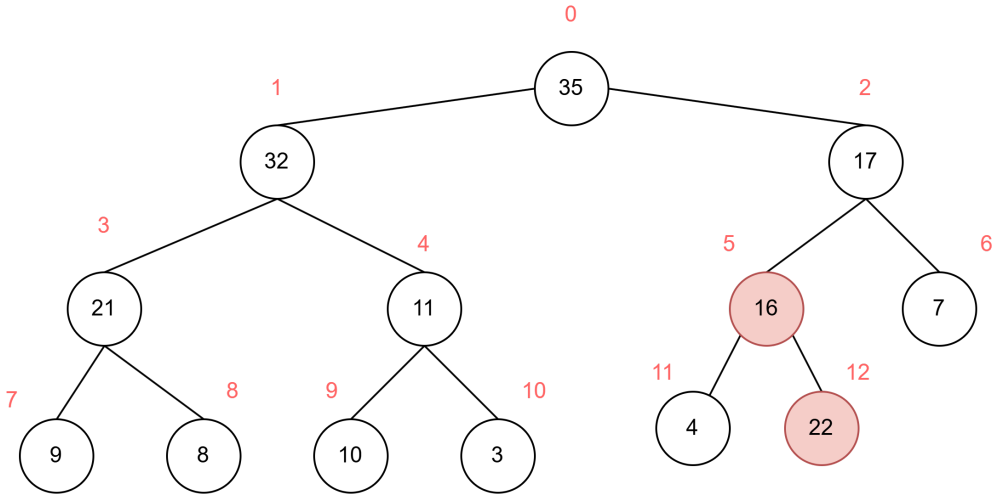
35	32	17	21	11	16	7	9	8	10	3	4	22



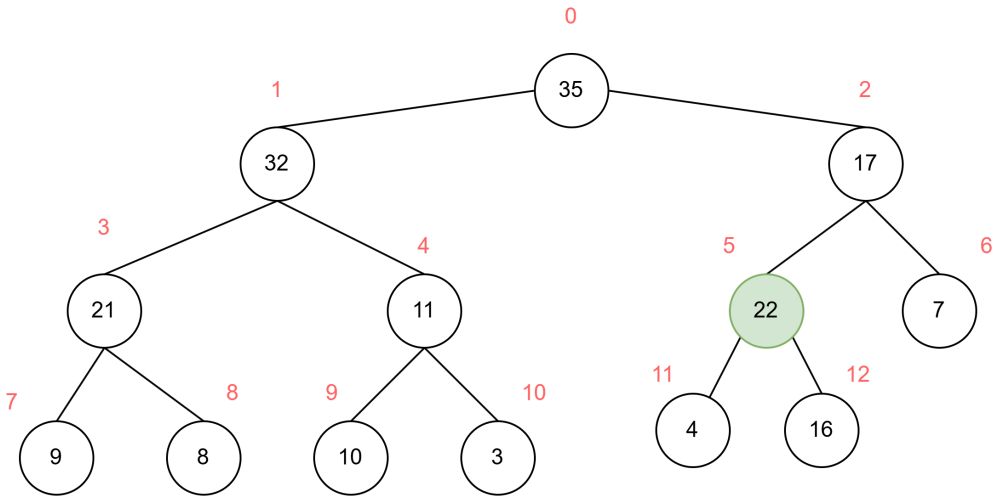
Agora, devemos aplicar o método `corrigeSubindo()` para compará-lo com seus nós pais, trocando-o de posição enquanto ele for maior que o elemento acima dele.

1. O elemento "21" é maior que seu pai "16". Haverá troca!

35	32	17	21	11	16	7	9	8	10	3	4	22

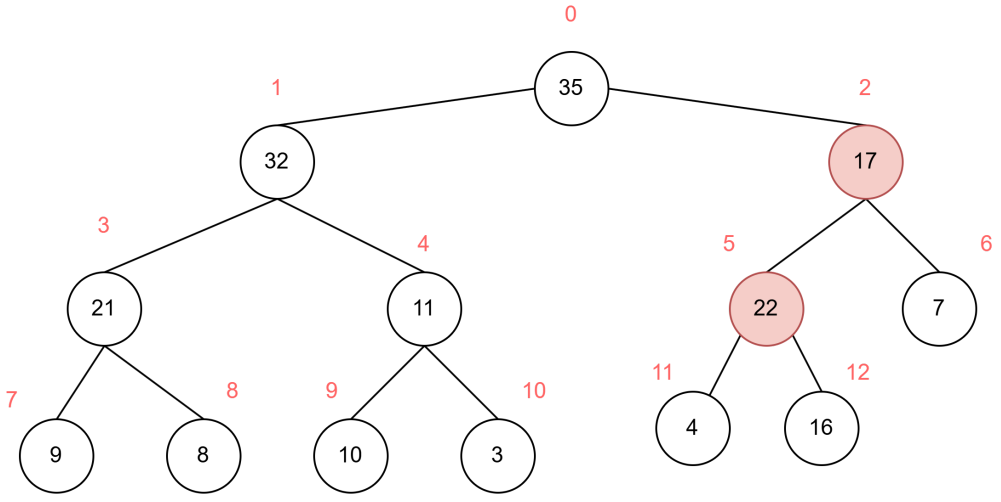


35 | 32 | 17 | 21 | 11 | 22 | 7 | 9 | 8 | 10 | 3 | 4 | 16

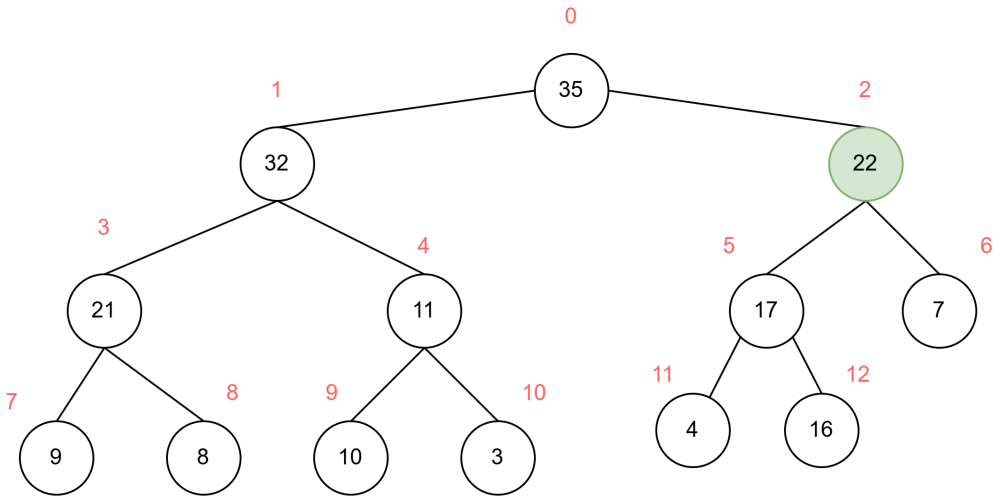


2. O elemento "22" é maior que seu pai "17". Haverá troca!

35 | 32 | 17 | 22 | 11 | 21 | 7 | 9 | 8 | 10 | 3 | 4 | 16

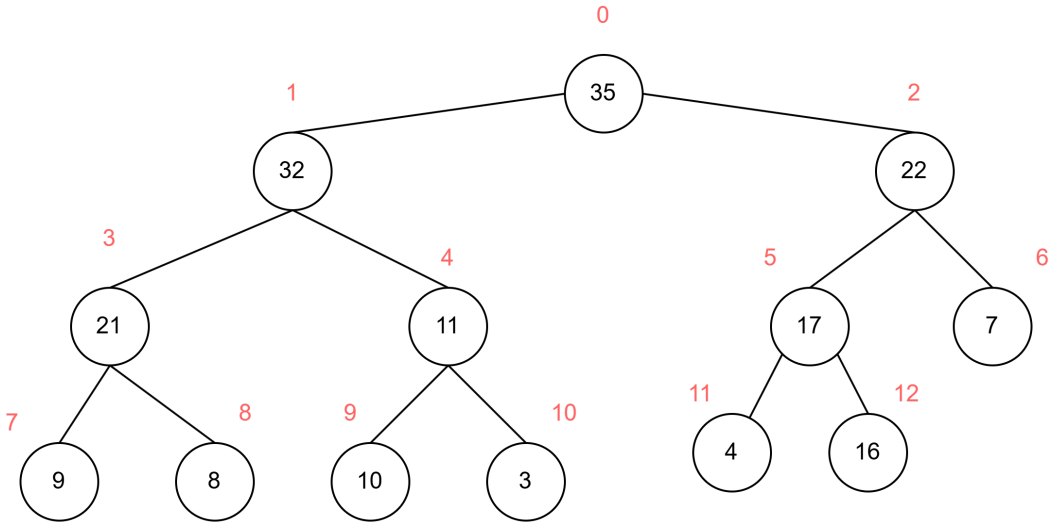


35 | 32 | 22 | 17 | 11 | 21 | 7 | 9 | 8 | 10 | 3 | 4 | 16



E assim chegamos ao fim do método `corrigeSubindo()` para o elemento "22", visto que seu pai agora é um elemento maior que ele.

35 | 32 | 22 | 17 | 11 | 21 | 7 | 9 | 8 | 10 | 3 | 4 | 16



Heap Sort

A ordenação utilizando *heaps* se baseia na construção de um *heap*, seguida da retirada, que resultará em um *subarray* ordenado. Assim, esse método de ordenação é um subproduto da retirada de elementos.

Lógica de mapeamento de nós pais e nós filhos

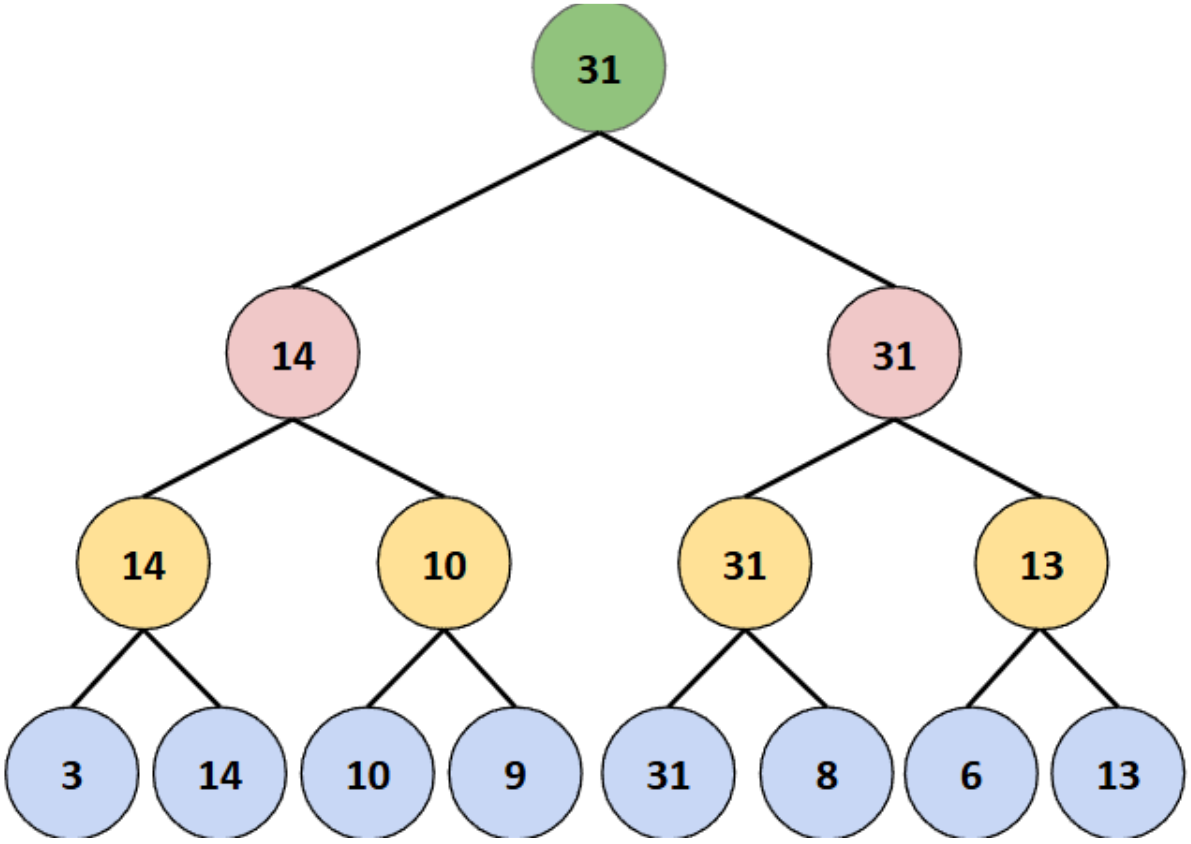
Considerando-se arranjos começando em posição zero, temos que:
nohPai(i) ← (i-1)/2
nohEsquerdo(i) ← 2i + 1
nohDireito(i) ← 2i + 2

Semelhantemente, caso fossemos implementar um *heap* a partir da posição, teríamos as seguintes fórmulas para mapeamento:
pai(i) ← (i)/2
esquerdo(i) ← 2i
direito(i) ← 2i + 1

Nesse caso, seria bom guardarmos o tamanho utilizado do *heap* na posição zero do vetor.

Torneio

Um torneio é uma árvore estritamente binária, em que cada nó não folha (pai) contém uma cópia do maior elemento entre seus dois filhos.



Exemplo de **torneio**. Retirado das videoaulas do Prof. Joaquim Quintero Uchôa, disponíveis em: Aula 06 parte 05 - YouTube.