Técnicas e Análise de Algoritmos

Heaps e Filas de Prioridade

Professor: **Jeremias Moreira Gomes**
E-mail: jeremias.gomes@idp.edu.br

Introdução

Heaps

Heap Binária

- A heap binária é uma estrutura de dados que mantém um conjunto de elementos, organizados de forma a permitir a identificação eficiente do menor dentre todos estes elementos
- Uma variante comum da heap binária é a troca da identificação do menor elemento para a identificação do maior dentre seus elementos
- As duas operações principais de uma heap binária são a inserção de novos elementos ou a identificação (e extração) do menor elemento

Heap Binária

- Outra operação importante é a construção de uma heap a partir de uma sequência de elementos dados
- Uma heap binária pode ser implementada a partir de uma árvore binária ou de um vetor
- A primeira alternativa tem como vantagem a visualização mais natural da propriedade fundamental das heaps
- A segunda permite uma implementação eficiente em termos de memória

Propriedade fundamental da heap binária (de mínimo)

Para qualquer elemento x contido na heap, a chave de x é menor ou igual do que as chaves de todos os seus descendentes.

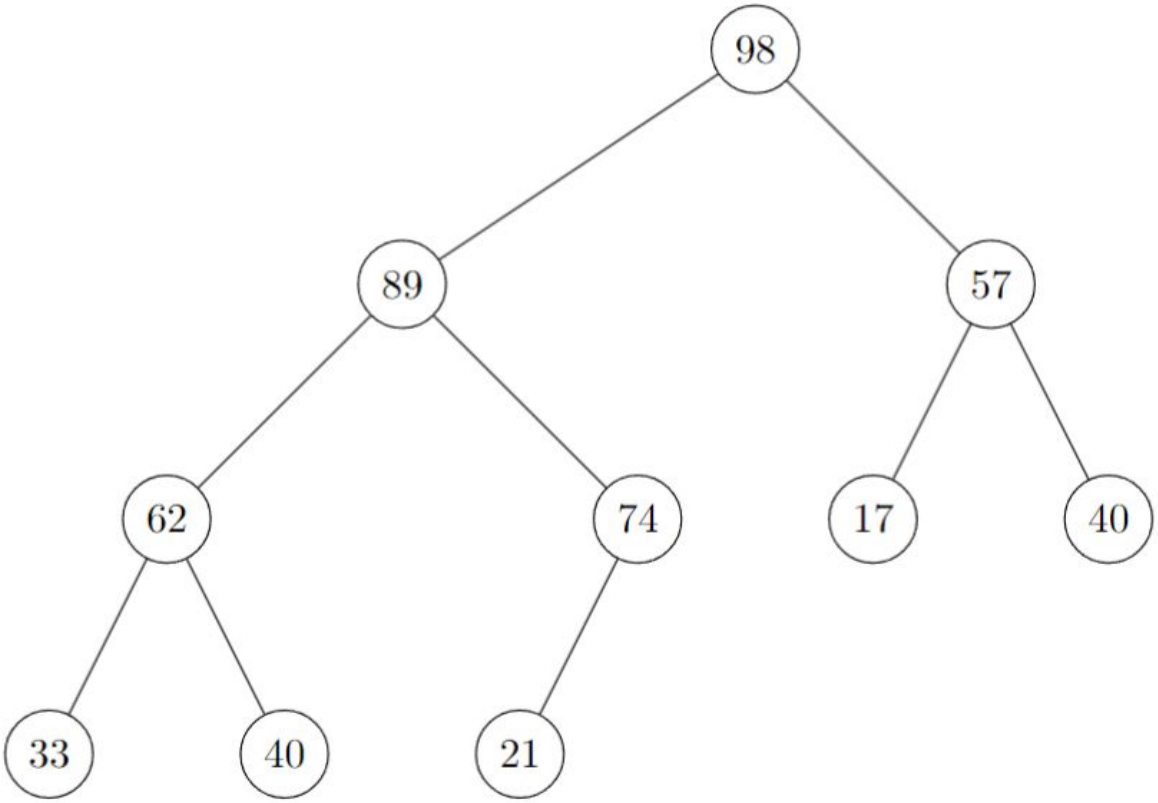
Heap como Árvores

- A visualização de uma heap como uma árvore binária permite a identificação de propriedades consequentes da propriedade fundamental
- Primeiramente, a raiz da árvore será o menor dentre todos os elementos
- Em segundo lugar, a representação da heap não é única

Heap como Árvores

- Além disso, a travessia de uma folha até a raiz leva a um caminho cujas chaves estão em ordem decrescente
- Esta propriedade é fundamental para a implementação das operações de inserção e remoção de elementos de um elemento

Heap como Árvores (max heap)



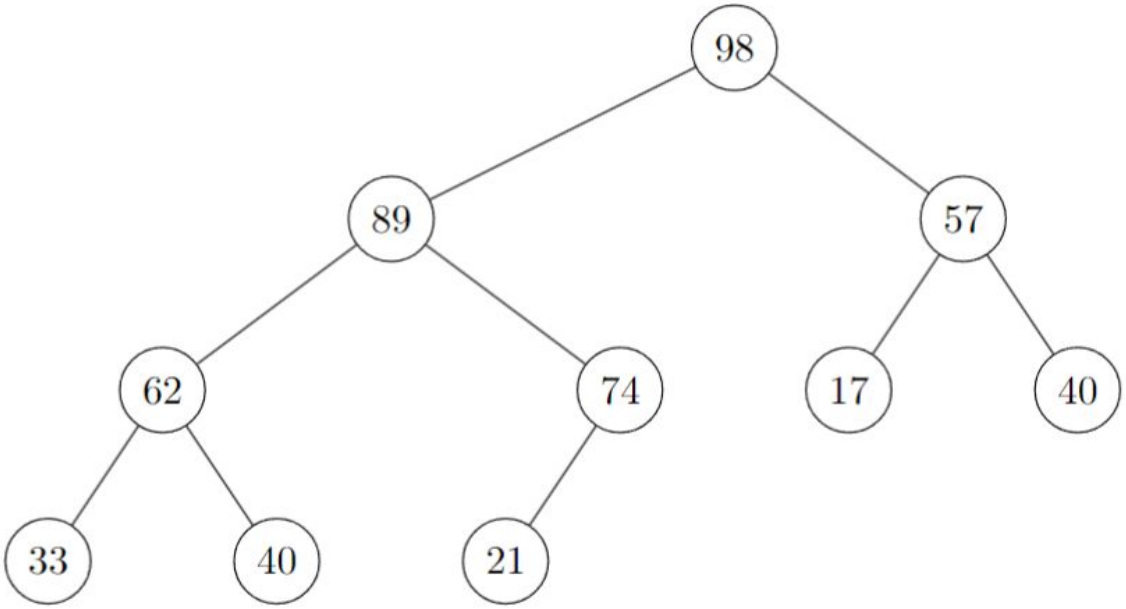
Heap como Vetores

- Uma heap binária pode ser armazenada como uma árvore balanceada
- Esta propriedade permite o armazenamento em um vetor
- Se a raiz for armazenada no índice 1 (e não no zero), as relações de parentesco ficam simplificadas, através de operações simples
  - O pai de um elemento x que ocupa em i está localizado em ⌊i/2⌋
  - O filho à esquerda de x tem índice l = 2i
  - O filho à direita de x tem índice r = 2i + 1

Heap como Vetores

- Mesmo que o índice zero fique inutilizado, a economia de memória em relação à representação com árvore é notável: com os parentes podem ser localizados a partir de seus índices, não é necessário armazenar ponteiros
- Observe a relação entre a representação como vetor e uma travessia por largura da árvore binária

Heap como Vetores



1	2	3	4	5	6	7	8	9	10
98	89	57	62	74	17	40	33	40	21

Inserção em Heaps

- A inserção de um elemento em um heap binária pode ser feita em O(logN), onde N é o número de elementos armazenados na heap
- A inserção deve preservar a árvore cheia e a propriedade fundamental
- Manter a árvore cheia é simples: basta inserir o elemento na primeira posição desocupada do vetor, isto é, na posição do filho nulo mais à esquerda no último nível
    - Porém, essa inserção pode violar a propriedade da heap

Inserção em Heaps

- Para restaurar a propriedade da heap, basta trocar as informações do novo nó com o seu pai
- Esta troca pode levar a uma nova violação, entre o pai e o avô
- A violação sobe um nível por vez, de modo que são necessárias, no máximo, O(log N) correções

Extração do Elemento Mínimo

- O elemento mínimo de uma min heap está localizado na raiz da árvore
- Portanto a identificação deste elemento pode ser feita em O(1)
- A extração deste elemento é o processo reverso da inserção
- A raiz deve ser substituída pela folha mais à direita do último nível
- Esta substituição pode gerar uma violação da propriedade fundamental do nó em relação aos seus filhos

Extração do Elemento Mínimo

- As possíveis violações devem ser resolvidas da raiz para as folhas
- Se a violação ocorrem com ambos filhos, deve-se escolher o que possui a maior informação para prosseguir com as trocas
- Esta extração tem complexidade O(log N)

Construção de Heap em Tempo Linear

- Dado um vetor de elementos, uma heap binária (de mínimo ou de máximo) pode ser construída em O(N log N) através de N inserções
- Porém é possível construir a mesma heap em O(N)
    - Esta rotina, denominada **heapify**, foi proposta por Floyd

Construção de Heap em Tempo Linear

- Para construir em tempo linear, primeiramente preenche-se uma árvore binária com os elementos na ordem em que foram informados
- Em seguida, em ordem reversa (da última folha para a raiz), as violações da propriedade fundamental devem ser corrigidas, utilizado a mesma estratégia da extração do elemento mínimo
    - É garantido (provado) que a aplicação dessa rotina irá corrigir a heap em tempo no máximo linear

Heap Binária

- Apesar da sua forma simplificada, em comparação com árvores binárias de busca, heaps tem aplicações essenciais na computação
  - Heap Sort
    - Algoritmo que utiliza a operação de heapify como base
  - Filas de Prioridade
    - Discutidas a seguir

Filas de Prioridade

Filas de Prioridade

- As filas com prioridades são variações da fila onde os elementos são acessados ou inseridos de acordo com a prioridade estabelecida
- Assim, a estratégia FIFO não se mantém
    - **O primeiro elemento a sair não é mais o primeiro a entrar, e sim o elemento com maior prioridade presente na fila**

Filas de Prioridade

- O desafio é encontrar uma implementação eficiente
- Se os elementos são inseridos ordenadamente na fila, de acordo com a prioridade, a complexidade do método push() é O(N), e do método pop() é O(1)
- Se os elementos são inseridos no final da fila, e a prioridade é avaliada no momento do acesso, a complexidade do método push() é O(1), e do método pop() é O(N)

Filas de Prioridade - STL

- A STL do C++ oferece um contêiner que implementa uma fila com prioridades:
  - A `priority_queue`, que faz parte da biblioteca queue
- Esta fila com prioridades é implementada através de uma heap binária
- Esta estratégia permite que os métodos push() e pop() sejam implementados com complexidade O(log N)

Filas de Prioridade - STL

- Diferente da fila, para acessar o próximo elemento, segundo a prioridade estabelecida, é utilizado o método top()
- Por padrão o maior elemento, segundo o critério de comparação, é o de maior prioridade
- Este comportamento pode ser modificado através da reescrita do operador de comparação

Filas de Prioridade - STL

```
priority_queue<int> fila; // max-heap
fila.push(30);
fila.push(8);
fila.push(45);
fila.push(2);
fila.push(18);

while (fila.size() > 0) {
    auto v = fila.top();
    fila.pop();
    cout << v << endl;
}
```

Filas de Prioridade - STL

- A priority_queue da STL é uma max heap
- Ela pode ser transformada em uma min heap de duas maneiras:
  - A primeira dela é útil quando a fila armazena tipos numéricos
    - Neste caso, basta inserir o simétrico de cada elemento na fila, o que inverterá o critério de comparação
  - A segunda maneira é utilizar a estrutura greater() da STL, a qual faz com que o operator > seja utilizado nas comparações de ordenação

Filas de Prioridade - STL

```
priority_queue<int, vector<int>, greater<int>> fila; // min-heap
fila.push(30);
fila.push(8);
fila.push(45);
fila.push(2);
fila.push(18);

while (fila.size() > 0) {
    auto v = fila.top();
    fila.pop();
    cout << v << endl;
}
```

Filas de Prioridade - STL

```
struct ClienteBanco {
    string nome;
    int idade;
    ClienteBanco(string nome, int idade) : nome(nome), idade(idade) {}

    bool operator<(const ClienteBanco& other) const {
        if ((idade > 64) && (other.idade > 64)) {
            return idade < other.idade;
        } else if ((idade > 64) && (other.idade < 65)) { return false;
        } else if ((idade < 65) && (other.idade > 64)) { return true;
        } else {
            return idade > other.idade;
        }
    }
};
```

Filas de Prioridade - STL

```
priority_queue<ClienteBanco> fila;

fila.push(ClienteBanco("Joao", 20));
fila.push(ClienteBanco("Paulo", 70));
fila.push(ClienteBanco("Maria", 30));
fila.push(ClienteBanco("Carla", 80));
fila.push(ClienteBanco("Pedro", 23));
fila.push(ClienteBanco("Carlos", 38));

while (fila.size() > 0) {
    auto cliente = fila.top();
    fila.pop();
    cout << cliente.nome << " " << cliente.idade << endl;
}
```

Heap Sort

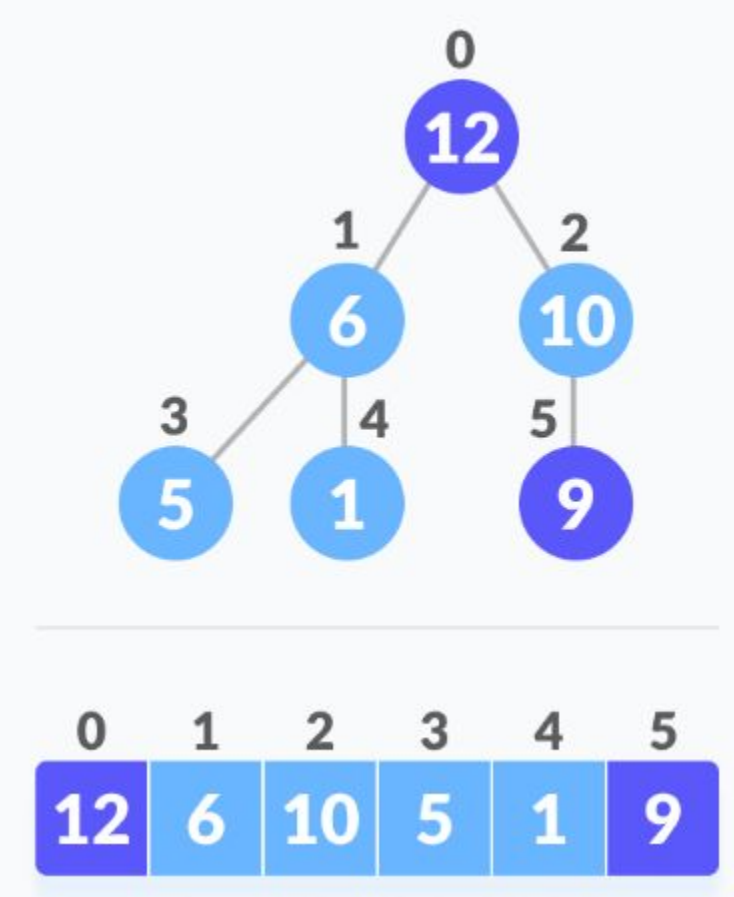
Heap Sort

- É um algoritmo de ordenação linearítmico (O(n lg n))
- Aproveita-se da representação de árvores utilizando vetores
- Utiliza heaps a partir de uma árvore binária completa
    - Folhas do último nível estão mais à esquerda
- É um algoritmo instável
    - Não preserva a ordem

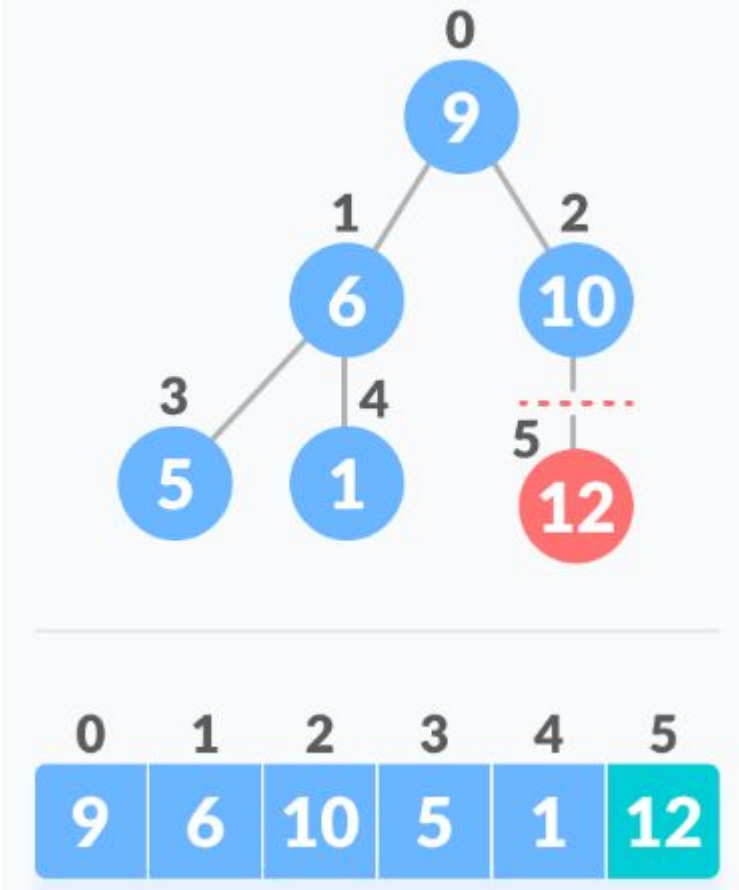
Heap Sort

- Funcionamento
  a. Constrói-se uma max-heap com os elementos
  b. Extrái-se o maior elemento, que está na raiz, movendo-o para o final do vetor (toca-se a raiz com o último elemento)
  c. Reconstrói-se a max-heap, sem o elemento movido para o final
  d. Retorna-se ao passo b, até não existirem mais elementos

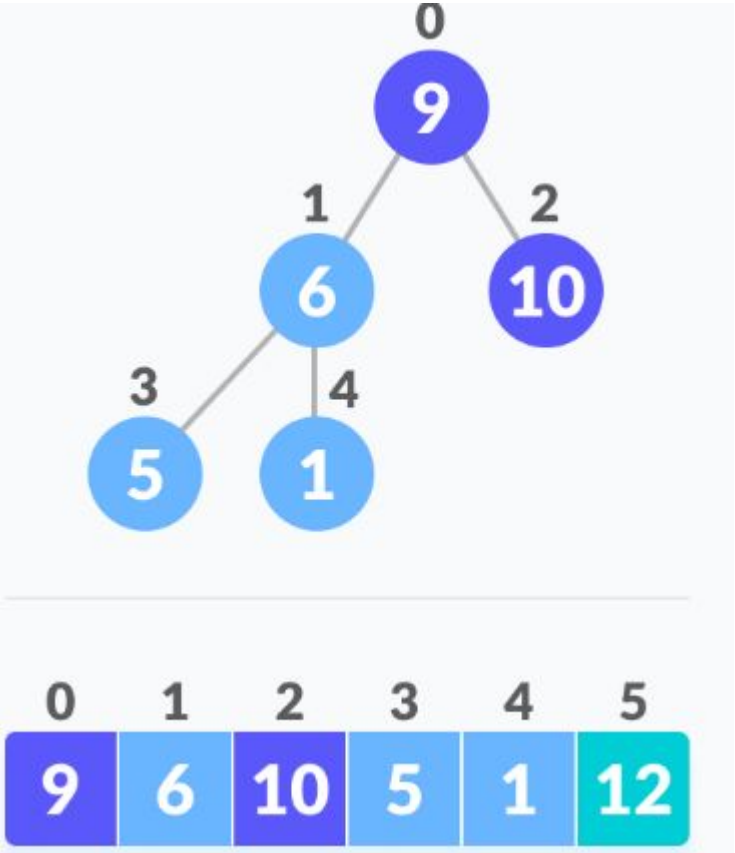
Heap Sort



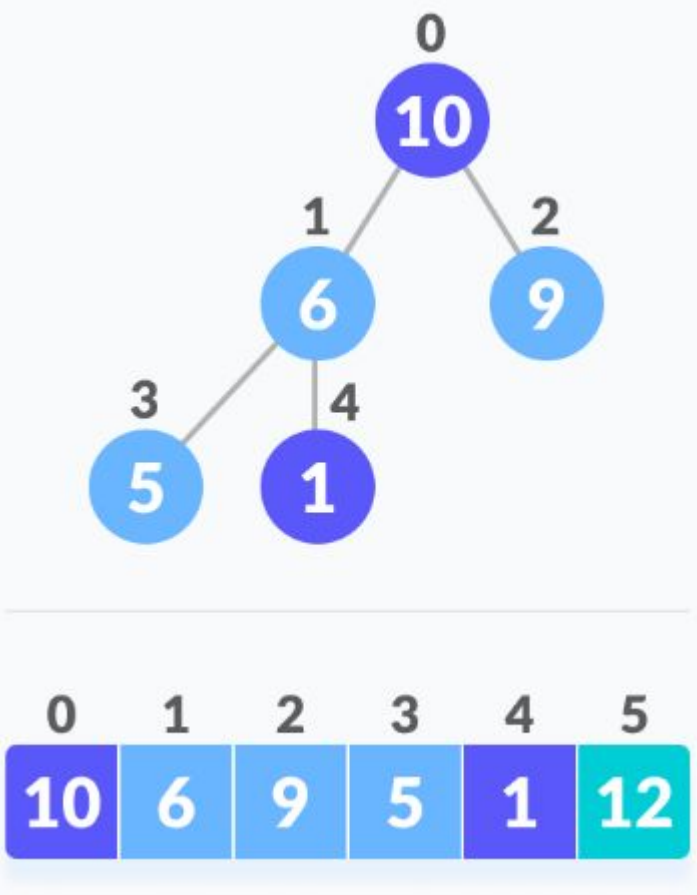
Heap Sort



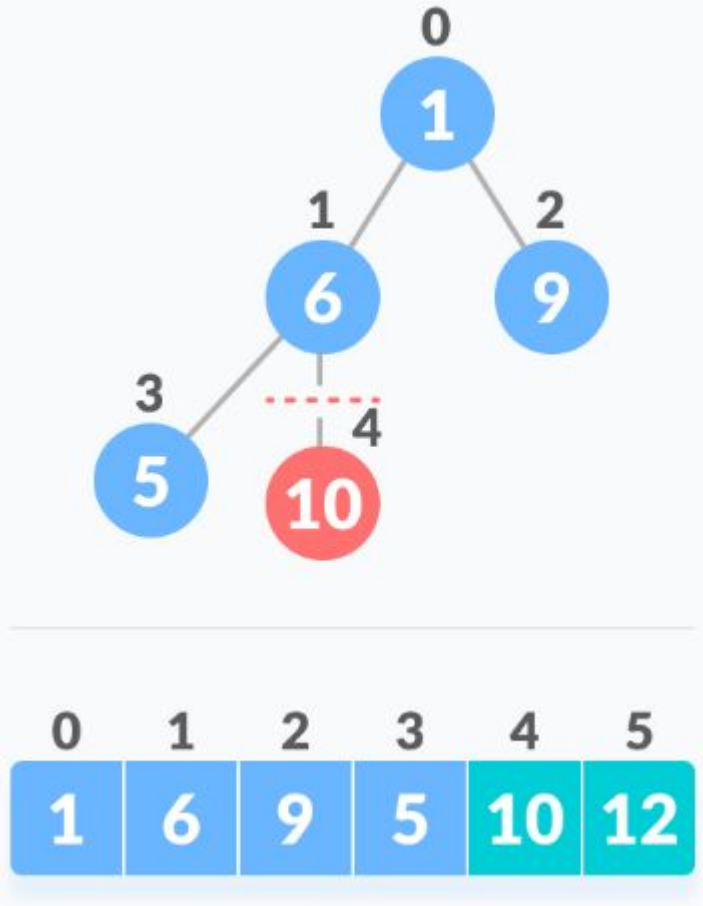
Heap Sort



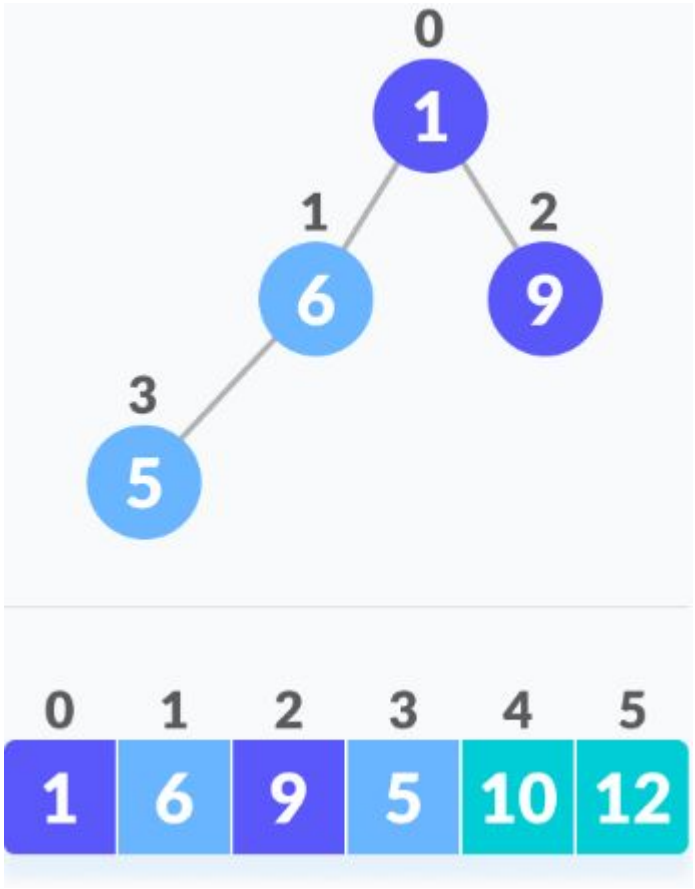
Heap Sort



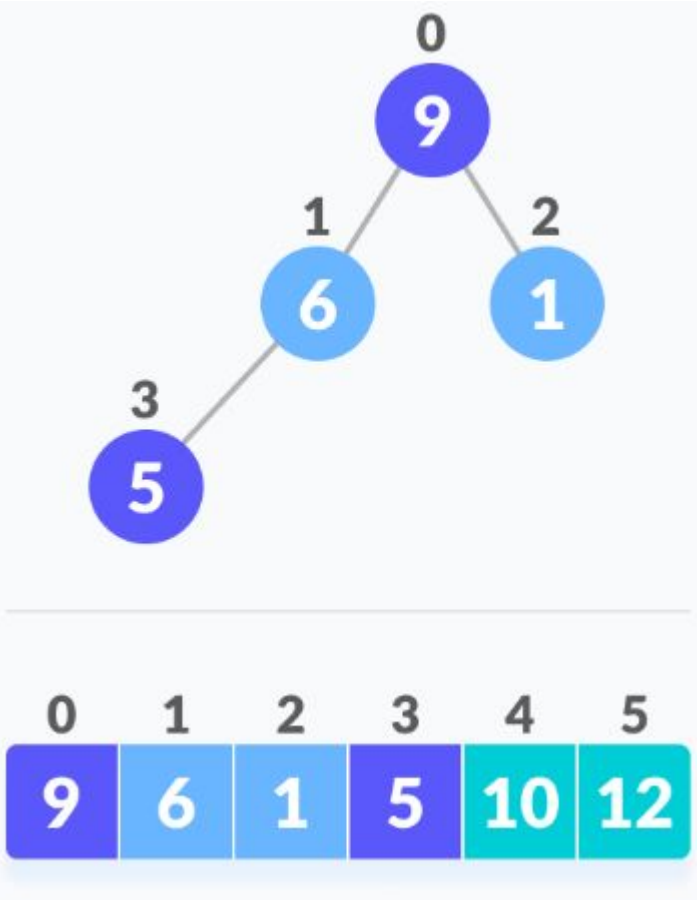
Heap Sort



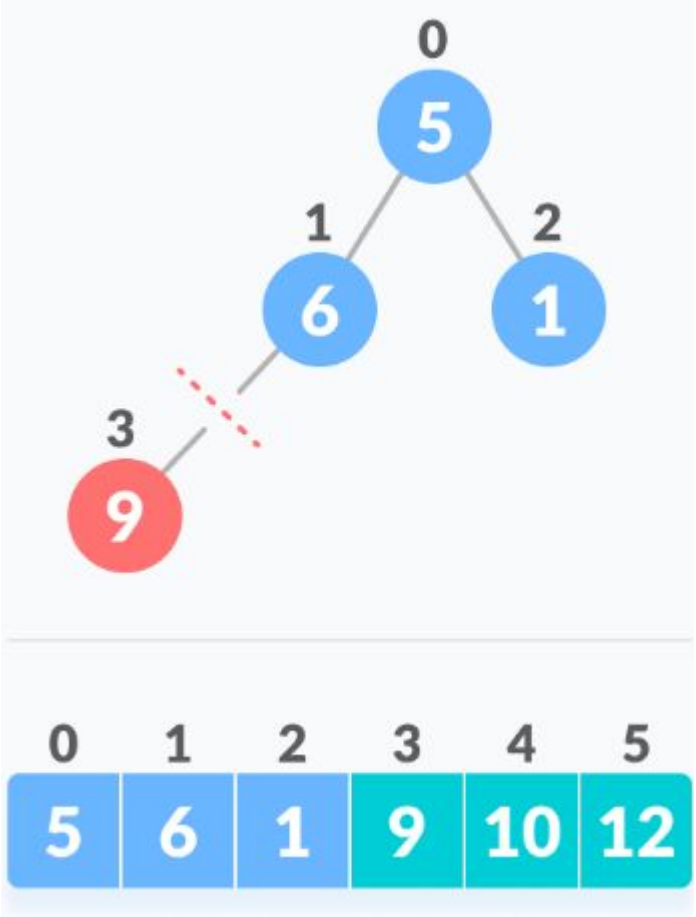
Heap Sort



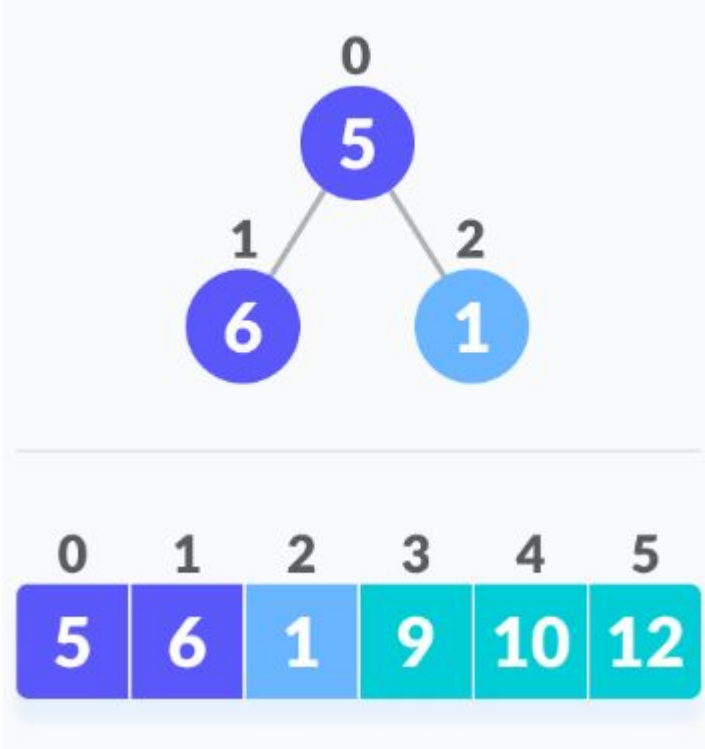
Heap Sort



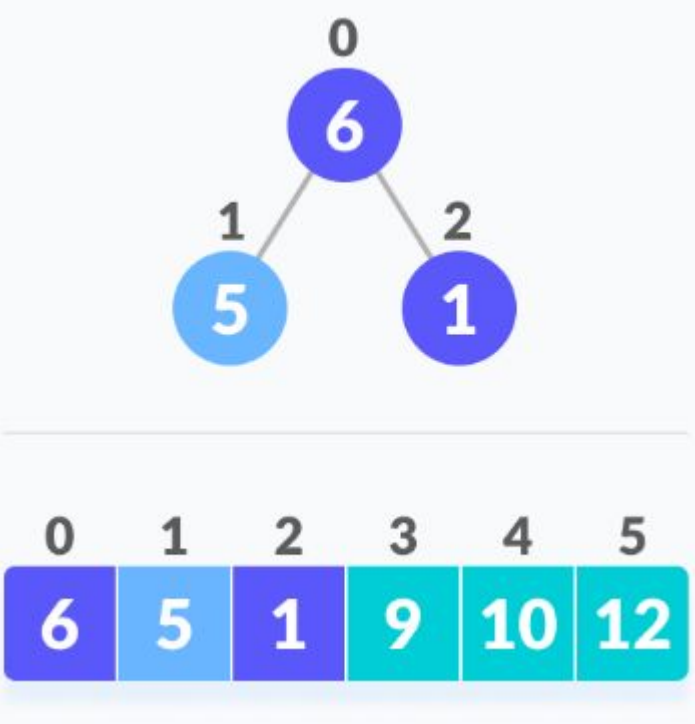
Heap Sort



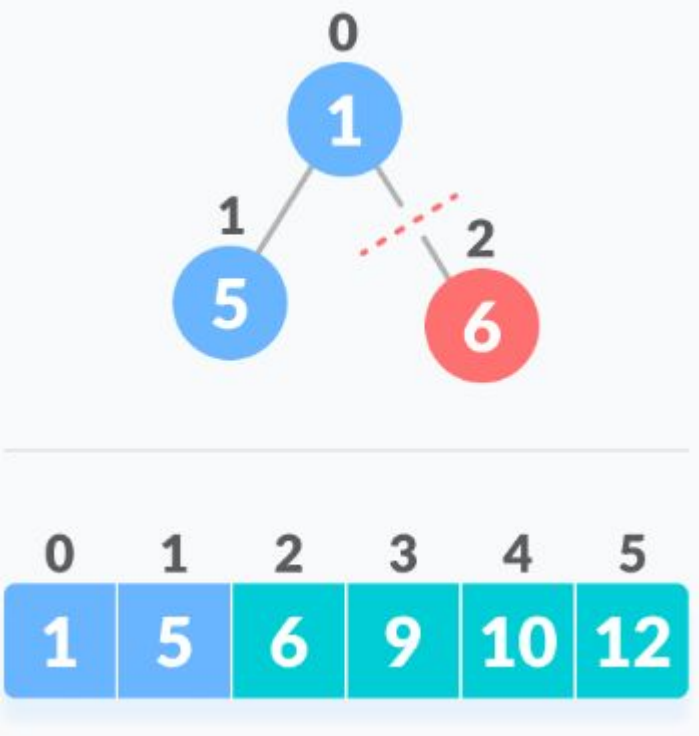
Heap Sort



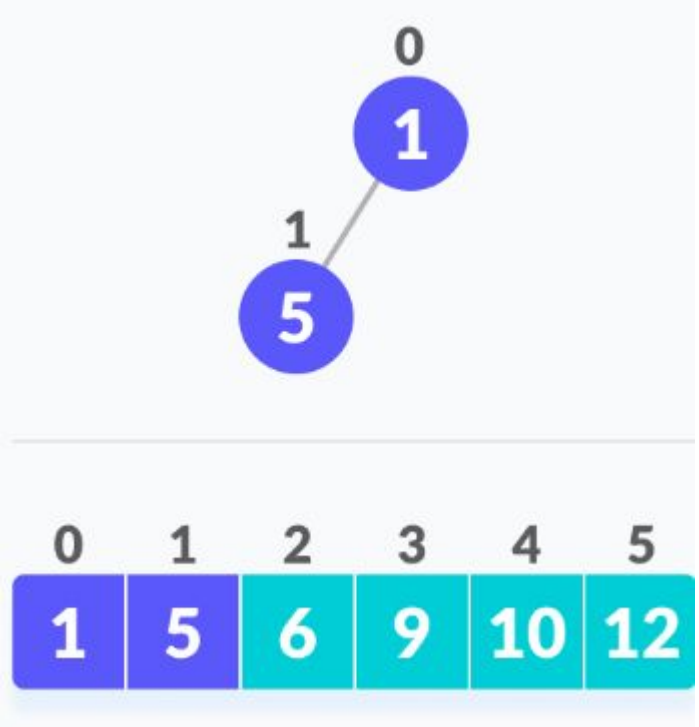
Heap Sort



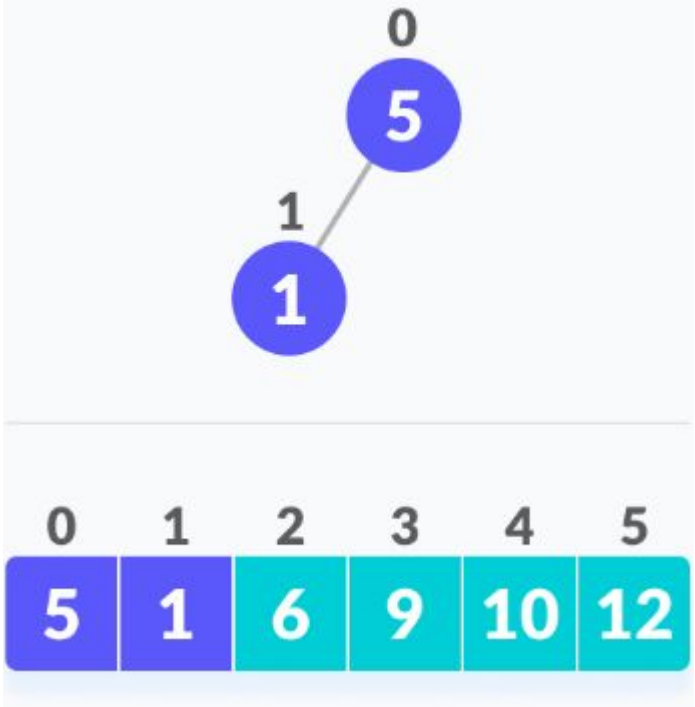
Heap Sort



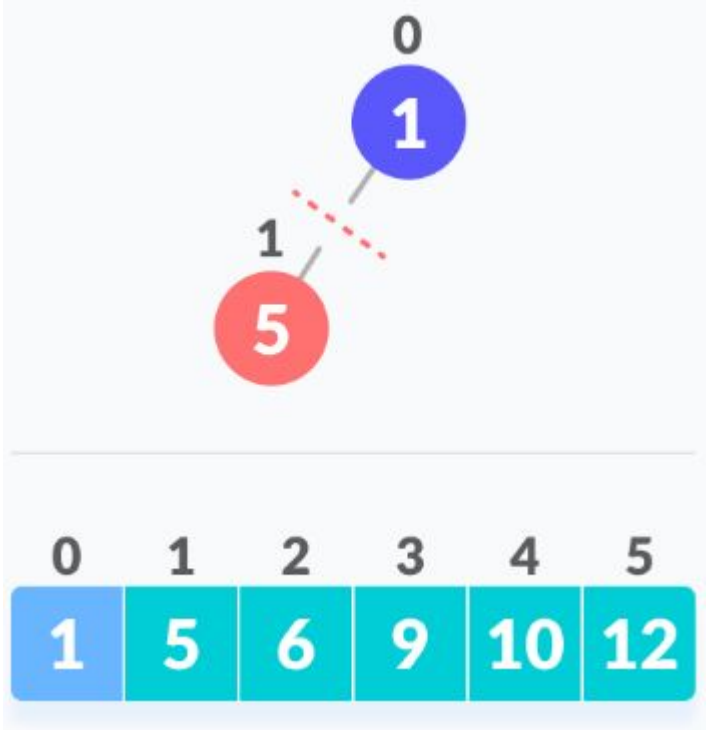
Heap Sort



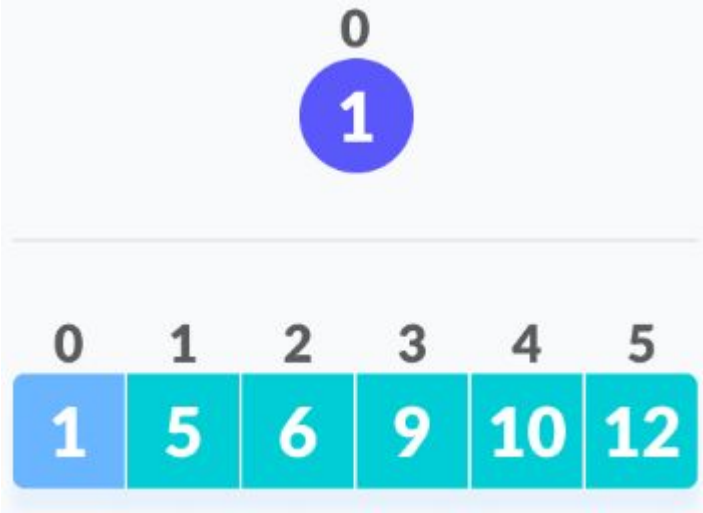
Heap Sort



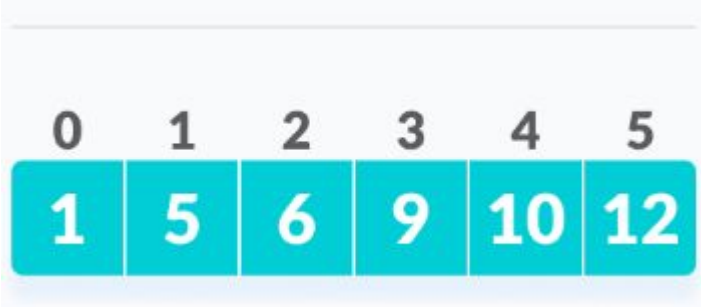
Heap Sort



Heap Sort



Heap Sort



Ordenação e Heaps

- A biblioteca algorithm da linguagem C++ contém três rotinas de ordenação, a saber:
  - sort(), stable_sort() e partial_sort()
- A função sort() é implementada através de uma estratégia mista
- Já o partial_sort() é implementada por meio do Heapsort: é mantida uma max heap com exatamente k elementos, removendo o (k + 1)-ésimo elemento sempre que o tamanho da heap for maior do que k

Ordenação e Heaps

```
vector<int> xs = { 30, 8, 45, 2, 18, 3, 1, 9, 10 };

// Somente os 4 menores elementos ordenados
partial_sort(xs.begin(), xs.begin() + 4, xs.end());

for (auto x : xs) {
    cout << x << " ";
}
cout << endl;
```

Conclusão