Estrutura de Dados Heap 2

Usando uma Fila de Prioridades para Ordenação

Prof. Mateus M. Luna[1] **Prof. André L. Moura**[2]

1. mateus_m_luna@ufg.br
2. andre_moura@ufg.br

Material elaborado com base do trabalho das professoras Telma e Juliana Félix (INF-UFG) e do professor João Luís Garcia Rosa (ICMC-USP)

2022

INF
INSTITUTO DE INFORMÁTICA

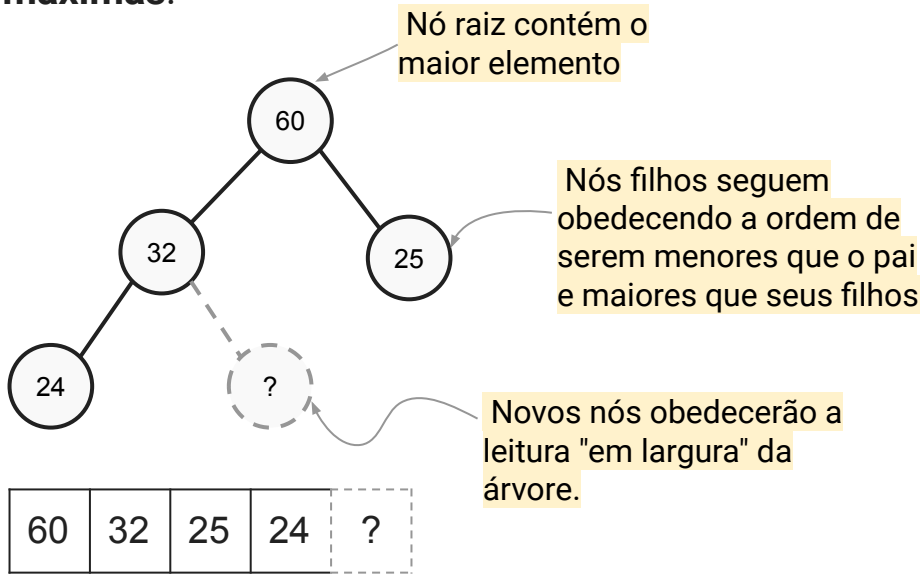
UFG
UNIVERSIDADE FEDERAL DE GOIÁS

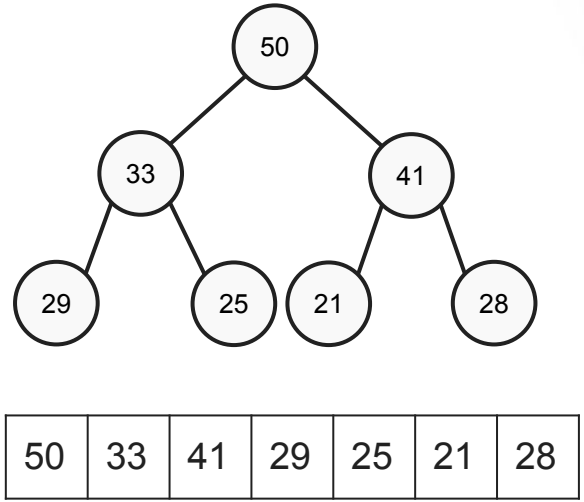
Sumário

Revisando o Conceito de *Heaps*

Estrutura da Dados Heap

A Heap é uma estrutura de dados implementada como um ***array*** *visualizado em* ***árvores binárias quase completas***. Cada nó desta árvore precisa obedecer uma propriedade max-heap ou min-heap (um conjunto de regras). A seguir temos dois exemplos de **heaps máximas**:





Estrutura da Dados Heap

Para codificarmos uma heap, usamos de funções ou de macros que nos permitam navegar em um vetor como se fosse uma busca em largura na árvore:

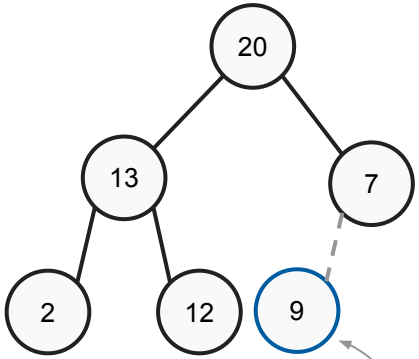
```
PAI ← ( (i - 1) / 2 )       //… Considerando um vetor de base 0
ESQ ← ( i*2 + 1 )                  //… Considerando um vetor de base 0
DIR ← ( (i*2 + 1) + 1 )     //… Considerando um vetor de base 0


//… Em algum lugar do código, com as funções ou macros definidas:
i ← 2
vetor[i]        //… Acesso ao terceiro elemento da heap (o filho esquerdo da raiz)
vetor[PAI]      //… Acesso ao pai do terceiro elemento (a raiz da heap)
vetor[ESQ]      //… Acesso ao quinto elemento (o filho esquerdo do terceiro elemento)
```

Revisando a Inserção e Remoção

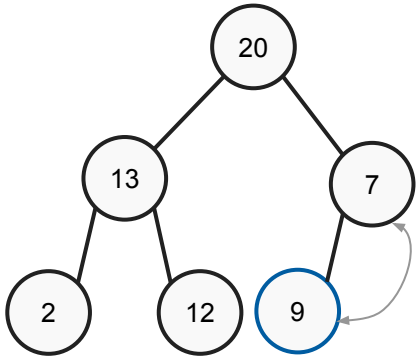
Inserção em Heaps

Para se inserir elementos na heap, colocamos o novo elemento **no final do vetor** e então *subimos* ele por um caminho até a raiz **enquanto ele não obedecer à propriedade da heap** (enquanto for maior que o pai, para a heap máxima).

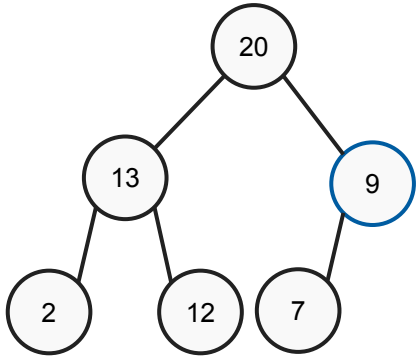
É uma operação **O(logn)**, visto que em seu pior caso realizamos uma quantidade de trocas proporcional à altura da árvore.



Elemento inserido no final



Troca para manter a max-heap



Inserção concluída

Inserção em Heaps

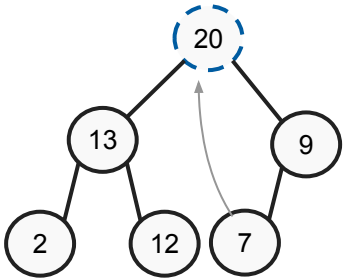
Inserção de um item chamado `elemento` na heap, dado um `vetor` de tamanho `tam`:

```
tam++                                  // Começamos incrementando o tamanho do vetor
vetor ← aumenta_vetor(vetor, tam);  // Alocamos no vetor o novo espaço
i ← tam - 1                          // Começamos a olhar pelo último elemento...
vetor[i] ← elemento                  // ...que de início será colocado no final do vetor

enquanto PAI ≥ 0 faça                // Neste loop mudamos i, portanto o "PAI" também muda
    maior ← PAI                      // Começamos supondo que PAI já é o maior (max-heap)

    se vetor[maior] ≥ vetor[i]  // Caso o atual seja menor que pai, tudo certo!
        sairenquanto
    fimse

    troca(vetor[i], vetor[maior])  // Caso contrário, trocamos para "subir"
    i ← maior

fimenquanto
```

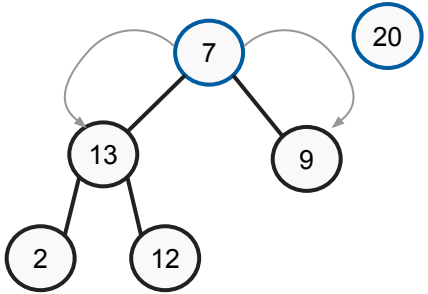
Remoção em Heaps

Para se remover um elemento do **topo da heap**, retiramos ele e então colocamos o último elemento **no lugar da raiz** e então *descemos* este por um caminho até o final **enquanto ele não obedecer à propriedade da heap**.

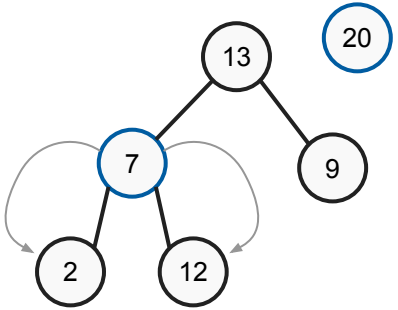
É uma operação **O(logn)**, visto que em seu pior caso realizamos uma quantidade de trocas proporcional à altura da árvore.

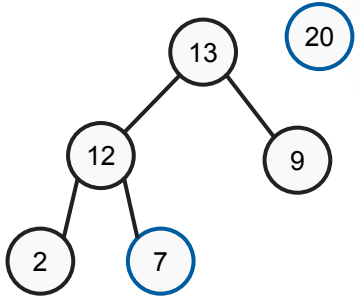


O elemento é removido e o último vira a raiz



Compara-se os filhos para ver se algum é maior. Se sim, troca!





Estado final pós remoção.

Remoção em Heaps

Remoção de um item no topo da heap, dado um `vetor` de tamanho `tam`:

```
removido ← vetor[0]                    // Guardamos o elemento a ser removido, que está na raiz
troca(vetor[0], vetor[tam-1])          // Colocamos no final do vetor, trazemos o último para seu lugar
vetor ← reduz_vetor(vetor, tam--)      // Reduzimos o tamanho do vetor e alocamos no vetor o novo espaço
i ← 0                                  // Começamos a olhar pelo elemento que foi colocado no topo

enquanto i < tam-1 faça                // Neste loop, vamos do início ao fim do vetor
      maior ← i                        // Começamos supondo que PAI já é o maior (max-heap)
      se ESQ < tam && vetor[ESQ] > vetor[i]  // Verificamos se o filho esquerdo é maior que o atual...
            maior = ESQ
      fimse
      se DIR < tam && vetor[DIR] > vetor[i]  // OU se o filho direito é que é maior que o atual...
            maior = DIR
      fimse
      se maior == i                    // Se o maior continua sendo o atual, então já é max-heap!
            sairenquanto  // break;
      fimse
      troca(vetor[i], vetor[maior])    // Caso contrário, trocamos para "descer" o elemento
      i ← maior
fimenquanto
```

"Heapificando" um vetor

"Heapificando" um vetor

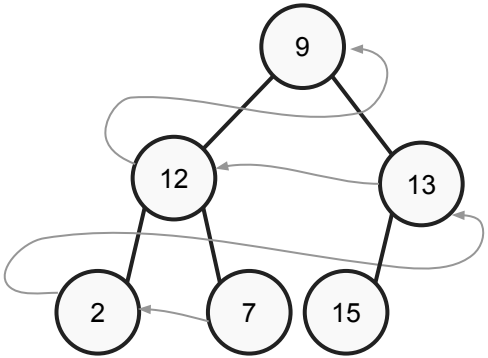
Os cenários de inserção e remoção que estudamos consideram a pré-existência de um vetor que já seja um heap máximo ou mínimo, mesmo que de tamanho 0.

Porém há situações em que queremos *"heapificar"* (*heapify*) um vetor já existente, garantindo que os elementos continuem os mesmos, mas obedecendo à ordem estabelecida pela lista de prioridades.

Consideremos o exemplo, ao lado, que **não representa uma heap máxima.**

Começamos por verificar se a heap é respeitada, olhando o vetor em **sentido reverso**, como indicado nas setas.

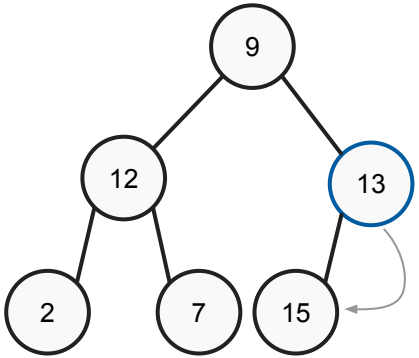
*Olhando os filhos dos elementos **15, 7** e **2**, podemos afirmar que sim, eles são - isoladamente - heaps!*



09	12	13	02	07	15

"Heapificando" um vetor

Considerando o elemento **13** porém, sua sub-árvore não é um heap máximo. Precisamos **eleger o maior** dentre ele e seus filhos para colocá-lo como novo pai:

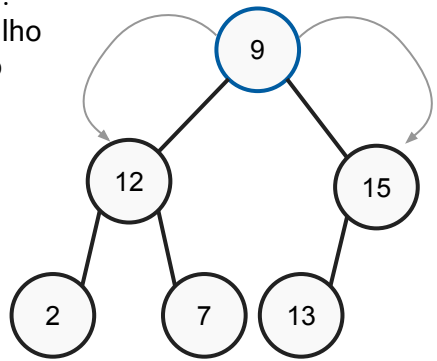


Quem é maior?
O nó atual, o filho esquerdo ou o direito?

09	12	**13**	02	07	15

Seguimos na leitura reversa, com número **12**, que já é uma heap máxima, assim como seus filhos. Por fim, olhamos para o número **9**, que não é uma heap máxima e elegemos o maior dentre os três para a troca:

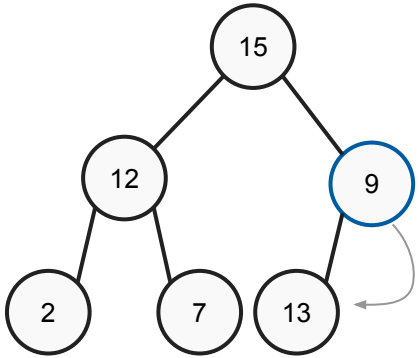
Quem é maior?
O nó atual, o filho esquerdo ou o direito?



09	12	15	02	07	13

"Heapificando" um vetor

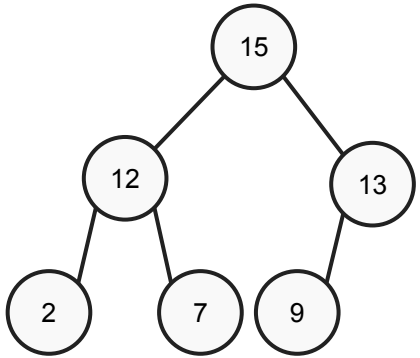
Mesmo após a troca, precisamos ainda olhar para os filhos do novo elemento. 9 está em uma sub-árvore inválida, portanto repetimos a análise:



Quem é maior? O nó atual, o filho esquerdo ou o direito?

15	12	09	02	07	13

Por fim, todos os elementos obedecem à regra de max-heap:



15	12	13	02	07	09

"Heapificando" um vetor

Implementação recursiva para fazer um `vetor` virar uma heap máxima de tamanho `tam`:

```
i ← tam-1        // Aqui associamos i ao último elemento para começar o loop no pai deste elemento
para i ← PAI até i ≥ 0, decrescentemente faça
     heapifica(vetor, i, tam); // Esta função certifica se o nó atual cumpre a regra max-heap
fimpara
define heapifica(vetor, i, tam)
     maior ← i                  // Começamos supondo que o atual já é o maior (max-heap)
     se ESQ < tam && vetor[ESQ] > vetor[maior] // Verificamos se o filho esquerdo é maior
          maior = ESQ
     fimse
     se DIR < tam && vetor[DIR] > vetor[maior] // OU se o filho direito é maior que o atual...
          maior = DIR
     fimse

     se maior != i                          // Se um dos filhos é o maior dos três...,
          troca(vetor[i], vetor[maior])     // trocamos para "descer" o elemento
          heapifica(vetor, maior, tam)      // e heapificamos a sub-àrvore deste filho
     fimse
     retorna
fimdefine
```

"Heapificando" um vetor

Qual o custo desta lógica ajuste do vetor para se tornar uma heap?

Como visto no algoritmo, temos que realizar a conferência e possivelmente troca de elementos para cada nó, o que implica em custo **O(n)**.

Há sim situações em que é preciso descer a altura das sub-árvores, verificando se a propriedade está válida do mesmo modo que fazemos a remoção, o que no pior caso implicaria em custos extras **O(logn)** (variando de sub-árvore este n). Seguindo a simplificação da notação assintótica, porém, podemos dizer que o fator dominante nesta lógica continua sendo **O(n)**, ou seja, *custo linear.*

E se eu quisesse criar um heap "sob-demanda" recebendo elementos sequencialmente?

Neste caso, realizamos a operação de Inserção *n vezes*, o que implica em custo **O(nlogn)**.

Ordenação por Heap

Ordenação por Heap

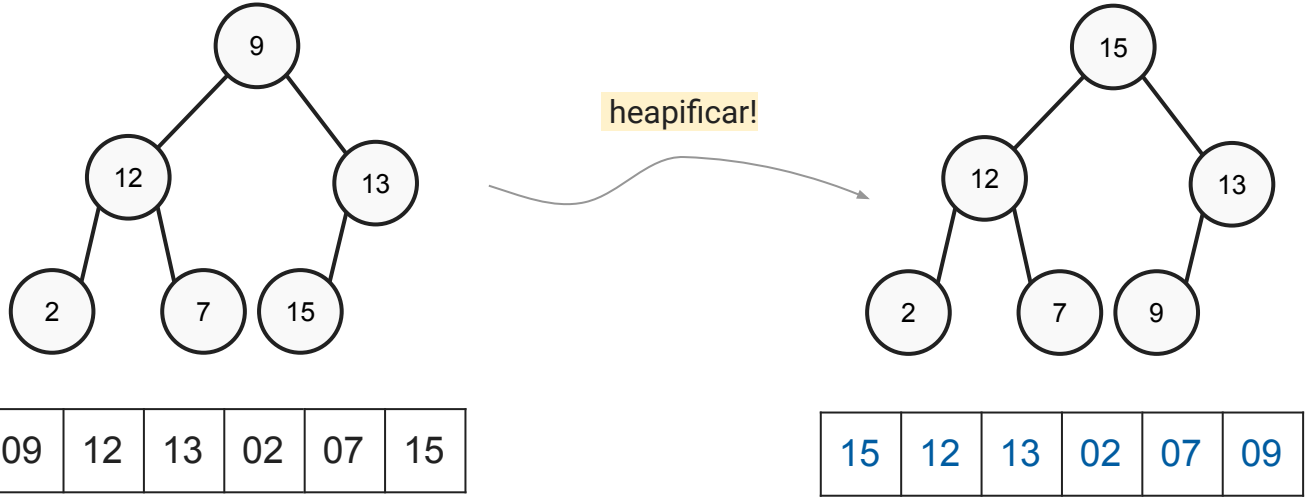
Como já antecipado, podemos tirar vantagem da facilidade com que se extrai o maior (ou menor) elemento de uma heap para montar uma lógica de ordenação:

1. Transforma-se o vetor de `n` elementos em uma heap (no nosso exemplo, heap máxima) - ou seja, "*heapificamos*" o vetor;
2. *"Extrai-se"* o elemento máximo do topo da heap e coloca-o no final do vetor na posição `n-1`);
3. Repete-se o passo anterior, considerando o vetor *até o elemento anterior ao último ordenado* - ou seja, consideramos os `n-1` elementos restantes.

Na chamada Ordenação por Heap ("Heap sort"), não removemos de verdade o elemento máximo, apenas colocamos ele no final do vetor e trabalhamos com o restante que sabemos que pode estar desordenado.

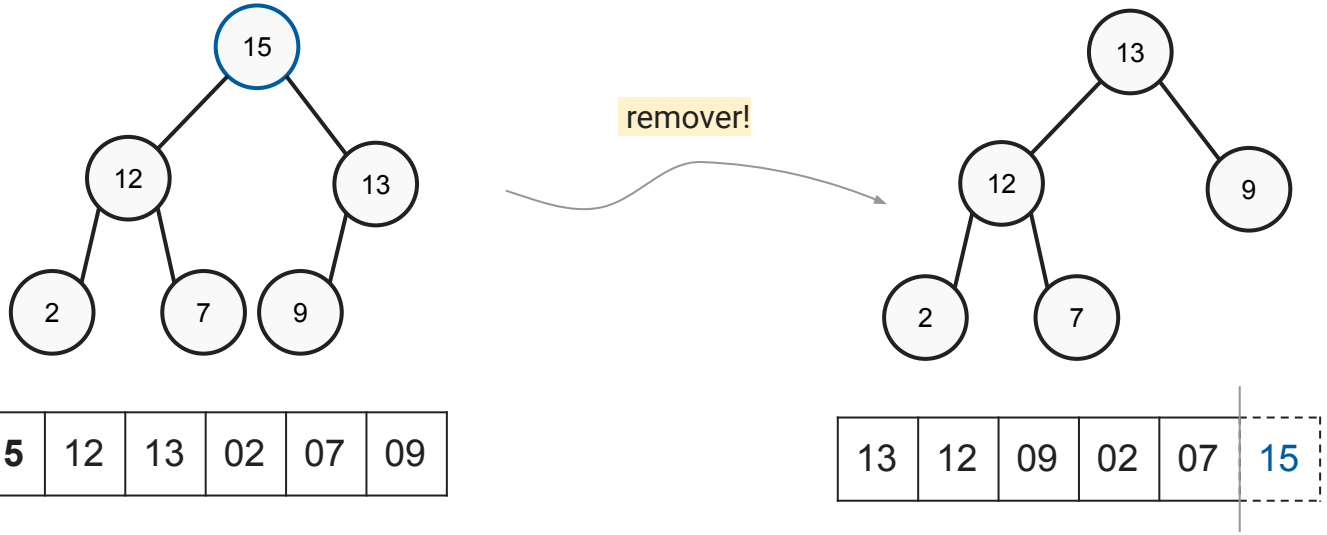
Ordenação por Heap

Começamos com um vetor que não é heap e em seguida, heapificamos ele:



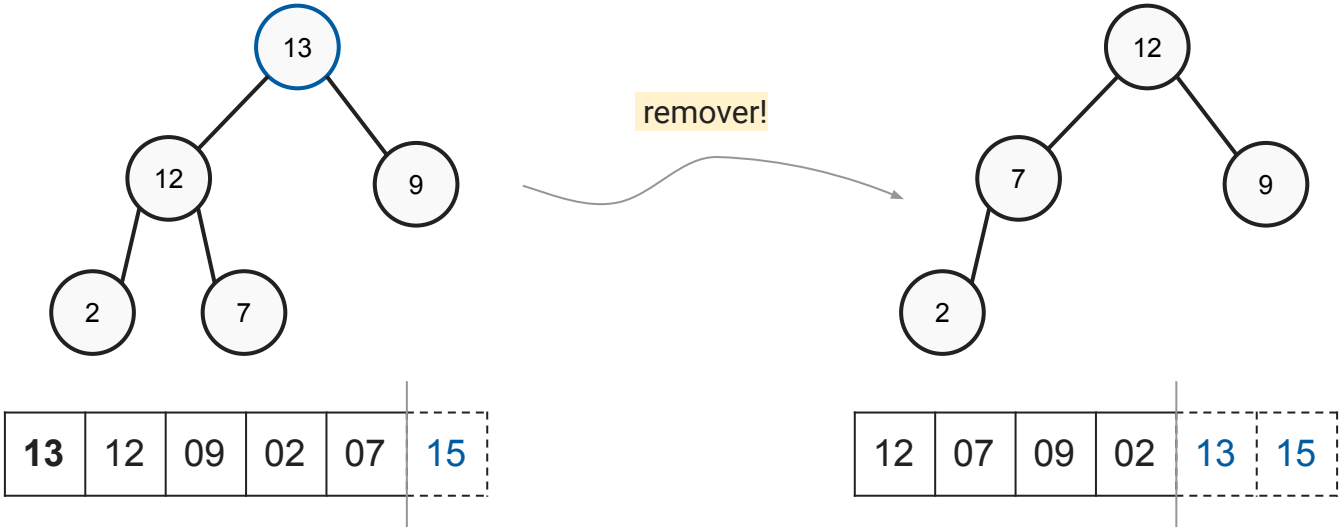
Ordenação por Heap

"Remove-se" então o maior elemento, colocando no final do vetor



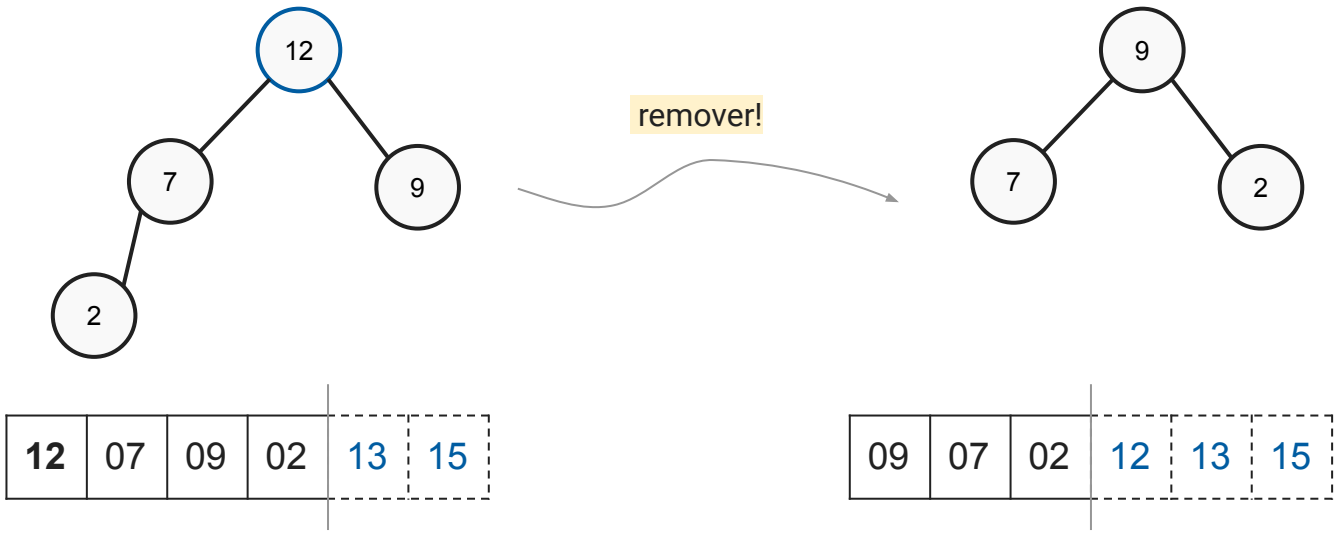
Ordenação por Heap

"Remove-se" então o maior elemento, colocando no final do vetor



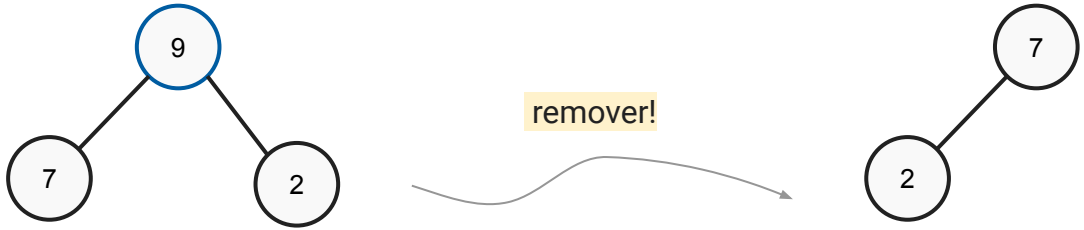
Ordenação por Heap

"Remove-se" então o maior elemento, colocando no final do vetor



Ordenação por Heap

"Remove-se" então o maior elemento, colocando no final do vetor

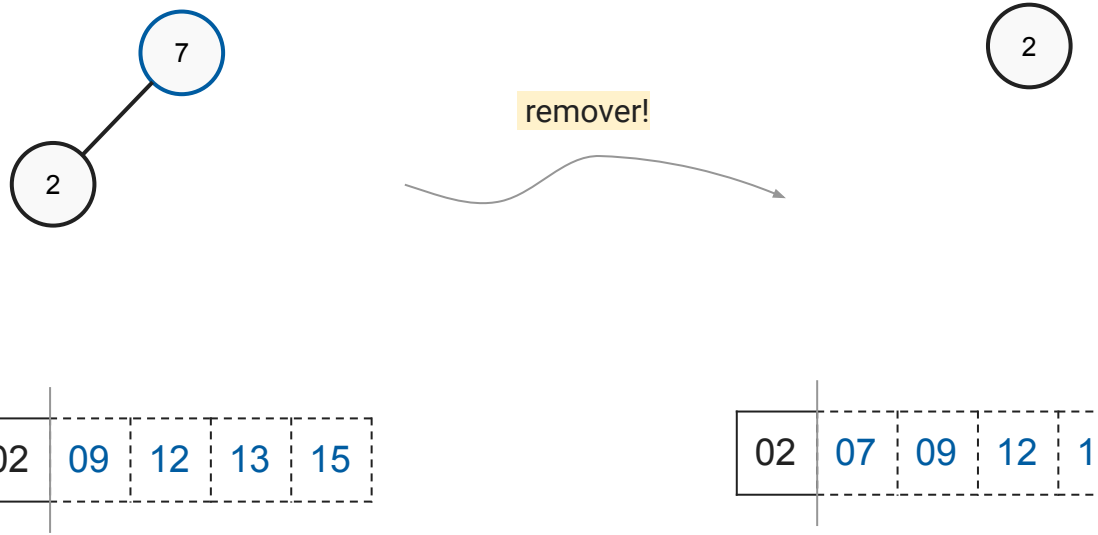


09	07	02	12	13	15

07	02	09	12	13	15

Ordenação por Heap

"Remove-se" então o maior elemento, colocando no final do vetor



Ordenação por Heap

A análise de complexidade da ordenação por heap é bem intuitiva:

De início, há o custo **O(n)** para *heapificação* inicial, mas este logo é sobreposto pelo custo da ordenação. Precisamos realizar a extração do maior elemento n *vezes*, sendo que esta operação custa **O(logn)**, portanto o custo total é **O(nlogn)**.

Tabela 1: Análise de Complexidade (tempo) para a Ordenação por Heap

Caso	BigO
Melhor caso	O(nlogn)
Caso médio	O(nlogn)
Pior caso	O(nlogn)

Ordenação por Heap

E a análise de complexidade do espaço?

Usa-se apenas o espaço já necessário para o vetor *(in-place)*, logo, podemos dizer que o espaço **auxiliar** necessário é constante: **O(1)**.

Ela é estável?

Não! A ordem relativa dos elementos não é preservada. Para se viabilizar uma implementação estável precisaríamos de mais espaço para guardar os índices originais

*Como se compara com o **Quick** e o **Merge**?*

Apesar da complexidade similar, o *quick* tende a ganhar se houver boa escolha de pivôs. Ademais, há situações em que manter uma heap pronta é útil para se trabalhar com filas de prioridades.

E como ficaria uma implementação em C?

Material de apoio

- https://www.cs.usfca.edu/~galles/visualization/Heap.html
- http://wiki.icmc.usp.br/images/b/b3/SCC501Cap4.pdf
- https://imsouza.github.io/algoritmos-de-ordenacao-em-c-2
- https://www.youtube.com/watch?v=HqPJF2L5h9U

Obrigado!

Dúvidas ou sugestões:

mateus_m_luna@ufg.br ou
andre_moura@ufg.br

INF
INSTITUTO DE INFORMÁTICA

UFG
UNIVERSIDADE FEDERAL DE GOIÁS

Estrutura de Dados Heap 2

Usando uma Fila de Prioridades para Ordenação

Prof. Mateus M. Luna[1] **Prof. André L. Moura**[2]

1. mateus_m_luna@ufg.br
2. andre_moura@ufg.br

Material elaborado em parceria com as professoras Telma e Juliana Félix

2022

INF
INSTITUTO DE INFORMÁTICA

UFG
UNIVERSIDADE FEDERAL DE GOIÁS