INSTITUTO FEDERAL DE EDUCAÇÃO, CIÊNCIA E TECNOLOGIA DO RIO GRANDE DO NORTE
Campus Nova Cruz – Código INEP: 24083003
Av. José Rodrigues de Aquino Filho, Nº 640, RN 120, Alto de Santa Luzia, CEP 59215-000, Nova Cruz (RN)
CNPJ: 10.877.412/0015-633 – Telefone: (84) 4005-4107

Texto Complementar:

Listas de Prioridades

(EDL – TADS4M)

1. Introdução

Uma **lista de prioridades** (*priority queue*, do inglês) é uma estrutura de dados abstrata que armazena elementos, cada um associado a uma **prioridade**, de modo que o elemento com maior (ou menor) prioridade possa ser rapidamente acessado ou removido. Diferente de uma fila tradicional, onde os elementos são processados de acordo com a ordem de chegada — FIFO (*First-In*, *First-Out*), na lista de prioridades a ordem de remoção depende da prioridade atribuída a cada item.

Um exemplo prático bastante utilizado é o do atendimento em uma Unidade de Pronto-Atendimento. Em um pronto-socorro hospitalar, as pessoas chegam em momentos diferentes e com graus de urgência variados. Nem sempre o primeiro a chegar é o primeiro a ser atendido — afinal, quem está em risco de vida precisa de atendimento imediato, mesmo que tenha chegado depois que outras pessoas em situações de menor risco de vida. Essa lógica é exatamente a de uma lista de prioridades (*priority queue*), e não a de uma fila comum (*queue*).

Em um pronto-socorro, a ordem de chegada é secundária. O que determina o atendimento é a gravidade do caso, ou seja, a prioridade. Por isso, normalmente, em unidade de pronto-atendimento, o paciente passa por uma triagem, geralmente feita por um enfermeiro, que classifica o estado clínico em níveis de prioridade. Esse cenário é bem diferente da que ocorre em uma fila de supermercado, por exemplo, onde a ordem de chegada é determinante: o primeiro a chegar será o primeiro a ser atendido.

2. Opções de implementação

Uma lista de prioridades pode ser implementada de diferentes maneiras, porém, a forma mais comum é utilizando uma estrutura de dados denominada *heap*. Antes de apresentarmos maiores detalhes sobre essa estrutura em específico, a tabela abaixo lista outras opções e algumas observações em relação ao desempenho.

Estrutura Base	Inserção	Remoção	Observações
Lista não ordenada	$O(1)$	$O(n)$	Inserção rápida, remoção lenta
Lista ordenada	$O(n)$	$O(1)$	Remoção rápida, inserção lenta
Árvore de Busca Binária	$O(\log n)$	$O(\log n)$	Requer balanceamento
Heap (mais usada)	$O(\log n)$	$O(\log n)$	Estrutura eficiente, simples de manter

3. Estrutura de uma *Heap*

Uma *heap* é uma árvore binária completa que obedece a uma prioridade de ordenação:

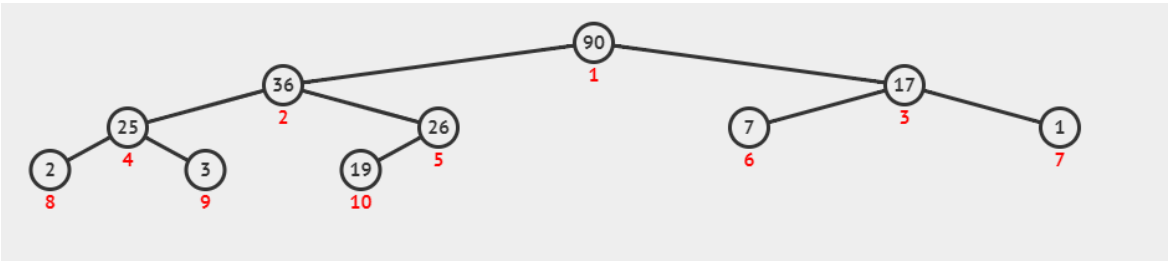
- ***Heap* Máxima** (*Max-Heap*): o valor de cada nó é maior ou igual ao de seus filhos. O maior elemento encontra-se na raiz;
- ***Heap* Mínima** (*Min-Heap*): o valor de cada nó é menor ou igual ao de seus filhos. O menor elemento encontra-se na raiz.

Antes de prosseguirmos é importante definir o que é uma árvore binária completa. Essa estrutura trata-se de uma árvore binária em que todos os níveis da árvore, exceto possivelmente o último, estão completamente preenchidos. Os nós do último nível estão o mais à esquerda possível. Essa característica é de suma importância para a *heap*, pois permite representá-la de forma compacta em um **vetor**, sem desperdiçar memória.

Isso mesmo! A implementação de uma *heap* é comumente realizada em um vetor. Assim, os elementos são armazenados em um vetor e as relações de pai e filho são mantidas por meio do cálculo de índices. Se o índice de um nó no vetor é igual a i, então, considerando que o primeiro índice do vetor é zero:

- $\boldsymbol{Pai}(i) = (i-1)/\,2$. A divisão realizada é a divisão inteira, portanto não há casas decimais no resultado.
- $\boldsymbol{Filho\ esquerdo}(\boldsymbol{i}) = 2 \times i + 1$
- $\boldsymbol{Filho\ direito}(\boldsymbol{i}) = 2 \times i + 2$

As imagens a seguir trazem a *heap* representada por meio de uma estrutura de árvore binária completa e a estrutura correspondente de armazenamento dos elementos em um vetor. Nessas imagens, o primeiro índice do vetor tem valor 1. Assim, $\boldsymbol{Pai}(i) = i/\,2$, $\boldsymbol{Filho\ esquerdo}(\boldsymbol{i}) = 2 \times i$ e $\boldsymbol{Filho\ direito}(\boldsymbol{i}) = 2 \times i + 1$.



Visualizar a estrutura por meio da árvore binária torna-se mais fácil, sendo possível identificar de maneira direta os nós da estrutura, seus pais e seus filhos. Entretanto, como dito anteriormente, na prática, os elementos são armazenados em um vetor e para identificar pais e filhos deve-se aplicar as fórmulas anteriormente apresentadas. Qual seriam o pai e os filhos do elemento de valor 17, armazenado no índice 3?

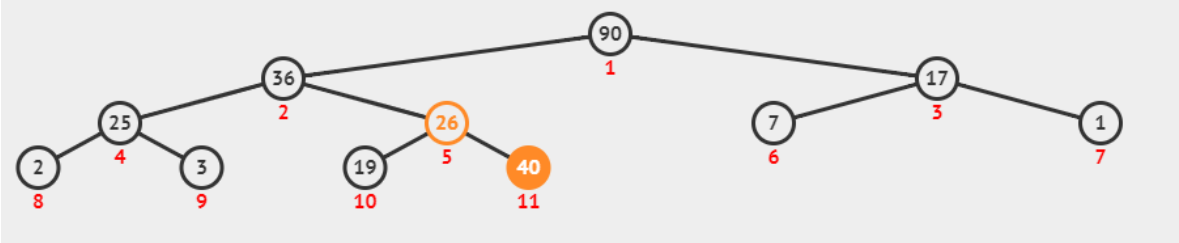
Elemento de índice 3 (valor 17):
Pai: 3/2 = 1,5 = 1 (parte inteira)
Filho esquerdo : 3*2 = 6
Filho direito: 3*2 + 1 = 7

4. Operações comuns em uma *Heap*

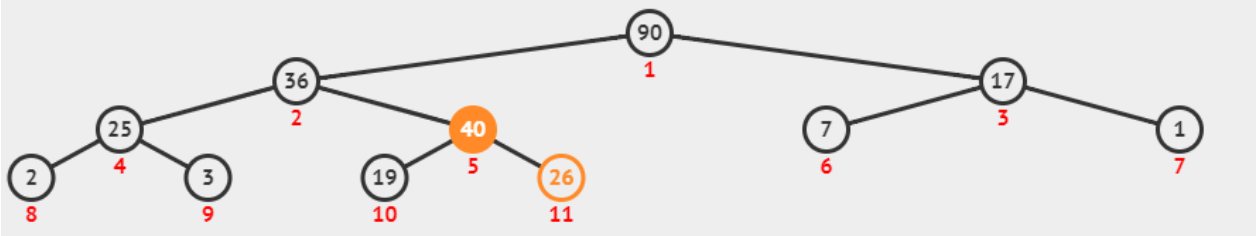
As principais operações realizadas em uma *heap* são a inserção, a remoção do máximo ou mínimo (dependendo da *heap*), a busca, atualização, *heapify*, *heapify-up* ou *sift-up* e *heapify-down* ou *sift-down*. As operações de inserção, remoção, busca e atualização são bastante comuns em outras estruturas de dados. As demais, são bastante específicas da estrutura *heap*: **i**. heapify é utilizado para transformar um vetor qualquer em uma *heap*; **ii**. *heapify-up* e *heapify-down* são operações auxiliares utilizadas para ajustar a estrutura para cima ou para baixo, sendo utilizadas em conjunto com as outras operações.

- **Inserção**: o novo elemento é adicionado no final do vetor para manter a árvore completa. Em seguida, o valor é ajustado para cima (*sift-up*) até restaurar a propriedade da *heap*;
- **Remoção do máximo ou mínimo**: o elemento da raiz é removido, maior ou menor dependendo do tipo de *heap*. O último elemento do vetor é movido para a raiz. Em seguida, aplica-se *heapfyDown* para restaurar a propriedade da *heap*. Normalmente, essa é a operação normal de uma *heap*. Entretanto, se a remoção de qualquer elemento for possível, deve-se buscar esse elemento no vetor, substituí-lo pelo último elemento da *heap*. Dependendo do valor, pode ser necessário reorganizar a *heap*, utilizando *heapifyUp* ou *heapifyDown*;
- ***Heapify* ou construção da *Heap***: a partir de um vetor desordenado, constrói-se uma *heap* de baixo para cima;
- ***Sift*-up (propagação para cima)**: se um elemento tiver prioridade maior que o pai, ele é trocado de posição com o pai. Esse processo se repete até que a propriedade de *heap* seja restaurada. Esta operação também é conhecida como ***heapifyUp***;
- ***Sift*-down**: se um elemento tiver prioridade menor que os seus filhos, ele é trocado com o filho de maior prioridade. O processo se repete até que a propriedade de *heap* seja restaurada. Esta operação também é conhecida como ***heapifyDown***;
- **Atualização de prioridade:** esta não é uma operação tão comum. Nela, o elemento deve ser encontrado no vetor e ter o valor de sua prioridade modificada. De acordo com o novo valor de prioridade, pode existir a necessidade de ajustar a *heap* para restaurar sua propriedade de *heap*, utilizando *heapifyUp* (*sift-up*) ou *heapifyDown* (*sift-down*).

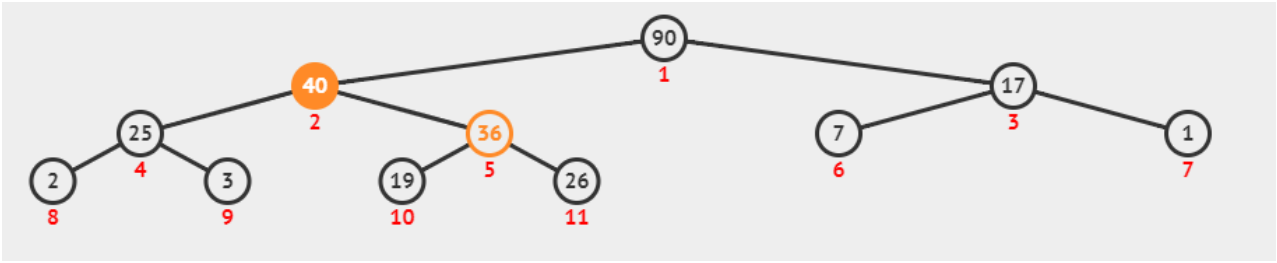
➢ **Exemplo de Inserção: elemento de valor 40**



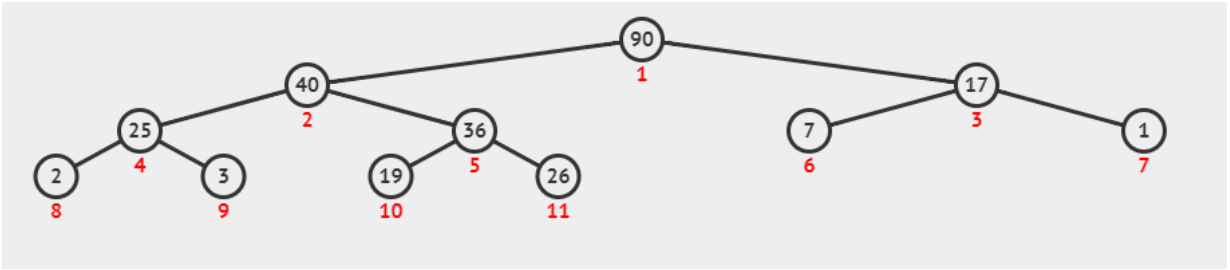
O elemento de valor 40 é adicionado no final do vetor. Após esse procedimento, executa-se o *heapifyUp* para garantir que a *heap* mantenha suas propriedades. Nesse caso, como o pai do elemento de prioridade 40 possui valor de prioridade menor, igual a 26, o elemento de valor 40 troca de posição com o valor do seu pai.



Em seguida, compara-se novamente o valor da prioridade 40 com o valor de prioridade de seu pai na árvore binária completa. Como o valor de seu pai é igual a 36, valor menor que 40, novamente há uma troca de valores.

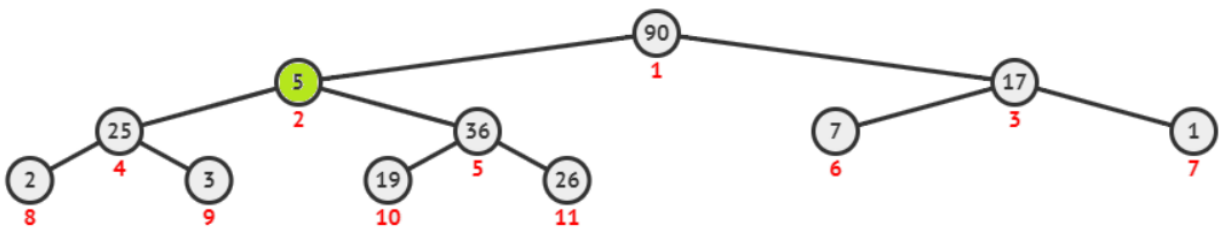


Uma nova comparação é realizada, agora com o valor da raiz. Como o valor da raiz é maior do que o valor do seu filho de valor 40, a *heap* já se encontra ajustada, não sendo necessário continuar com o procedimento de *heapifyUp*.

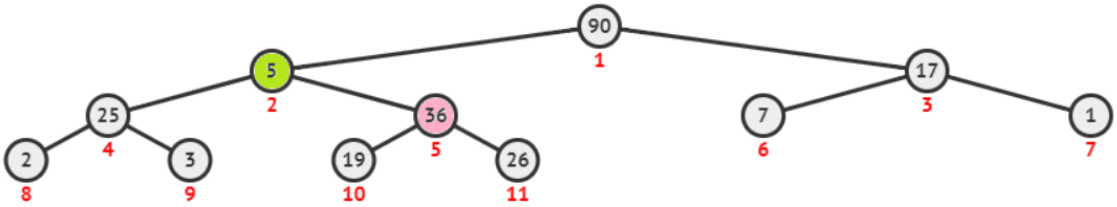


➢ **Atualização: atualização do valor 40 para 5**

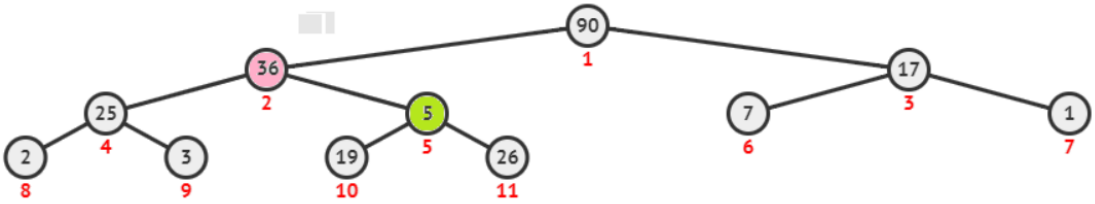
O índice em que se encontra o valor 40 é obtido na *heap* a partir da procura desse valor no vetor. Ao ser encontrado, o valor é atualizado par o valor 5. Com essa mudança, deve-se analisar a necessidade e realizar um *heapifyUp* ou *heapifyDown*.



O índice em que se encontra o valor 40 é obtido na *heap* a partir da procura desse valor no vetor. Ao ser encontrado, o valor é atualizado para o valor 5. Com essa mudança, deve-se analisar a necessidade e realizar um *heapifyUp* ou *heapifyDown*.



O elemento de valor 5 é menor do que o valor do seu pai (valor 90). Dessa maneira, não é necessário realizar a operação de *heapifyUp*. Por outro lado, observa-se que o elemento de valor 5 é menor que o valor de seus filhos. Nesse caso, ele será trocado com o filho de maior valor, que é o 36.



Mais uma vez, há a necessidade de ajustar a *heap* por meio da operação *heapifyUp*, já que o elemento de valor 5 é menor que o valor de seus filhos. Outra vez, deve-se trocar o elemento de valor 5 com o valor do seu filho de maior valor. Logo, o valor 5 é trocado com o valor 26. Após essa troca, as propriedades da *heap* são satisfeitas e os ajustes devido à atualização são encerrados.