**EQUIPE:Leandro Oliveira do Nascimento & Lucas Xavier Paire**

# FILA DE PRIORIDADE

## IMPLEMENTADA COMO HEAP BINÁRIO

# CONCEITOS

## FILAS DE PRIORIDADE

Uma fila de prioridade é uma fila na qual os elementos são inseridos com um nível de prioridade. Quando a função para remover um elemento da fila é chamada, o elemento com a prioridade mais alta é retirado da fila.
As filas de prioridade são amplamente utilizadas em várias situações. Por exemplo, em filas de atendimento em bancos, clientes idosos ou mulheres grávidas têm prioridade mais alta em comparação com outros clientes.
As filas de prioridade podem ser implementadas com listas encadeadas, usando uma variável adicional para representar a prioridade de cada elemento.

```c
struct lista
{
    int dado;
    int prioridade;
    struct lista *prox;
};
typedef struct lista Lista;
```

Nessa struct, temos uma lista, na qual possui uma informacao(dado) que queremos armazenar e uma variavel **prioridade** que será utilizada na hora de realizar a inserção ou remoção(dependendo da implementação.

## OPERAÇÕES

As filas de prioridade prevêem duas operações básicas que são na realidade uma extensão das operações básicas de uma fila comum. São elas:

- Inserir com prioridade
- Remover elemento de mais alta prioridade

Não iremos implementar a fila de prioridade usando listas. Isso porque existe uma implementação que é mais rapida do que listas. Porém, antes precisamos estudar outra estrutura de dados chamada Heap.

## HEAP

A Heap é uma estrutura especializada baseada em árvores que satisfaz a seguinte propriedade:

1. Se B é um nó filho de A, então chave(A) >= chave(B), no caso de **max_heap** ou chave(B) >= chave(A) no caso de **min_heap.**

A propriedade implica em que o nó raiz seja sempre o elemento com maior ou menor chave na estrutura.
Os principais usos da heap são na implementação eficiente de filas de prioridade, em alguns algoritmos como o Algoritmo de Djikstra, e em algoritmos de ordenação de dados tal como o algoritmo heapsort.

**OBS:** A heap é geralmente implementada em um array e não requer ponteiros entre os elementos. Porém na nossa implementação iremos implementar com ponteiros, para sair do convencional(lembrando que com vetores a heap é mais rápida).

# HEAP BINÁRIA

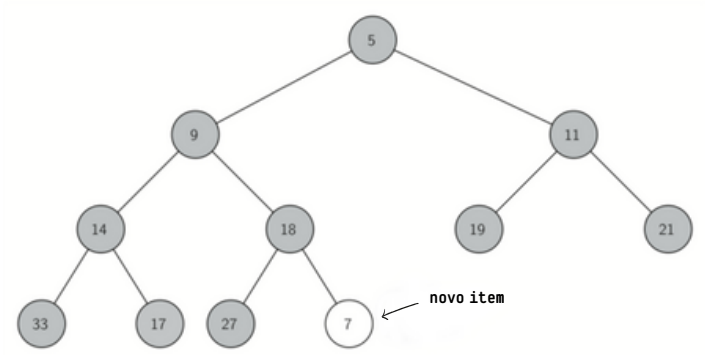
A Heap Binária é um caso a parte da estrutura Heap, poís ela adiciona mais uma propriedade.

**PROPRIEDADES:**

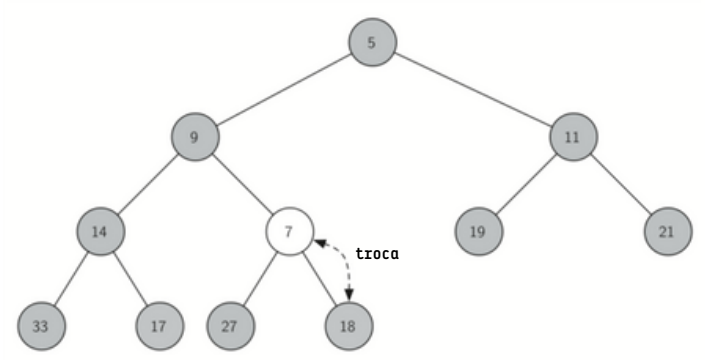
1. Se B é um nó filho de A, então chave(A) >= chave(B), no caso de **max_heap** ou chave(B) >= chave(A) no caso de **min_heap. (PROPRIEDADE DA HEAP)**
2. A árvore binária deve ser completa ou quase completa. Ou seja, todos os níveis exceto possivelmente o último estão completamente preenchidos. Se o último nível não estiver completo, os nós folha deste nível estão completamente preenchidos da esquerda para a direita. **(PROPRIEDADE DA HEAP BINÁRIA)**

O que diferencia uma heap de uma árvore binária qualquer é a forma pelas quais os elementos são **inseridos** e **removidos** da heap. A idéia é inserir os elementos de tal maneira que o elemento de mais alta prioridade esteja sempre localizado na raiz, e a remoção do elemento de mais alta ordem pode ser feita via a remoção da raiz.

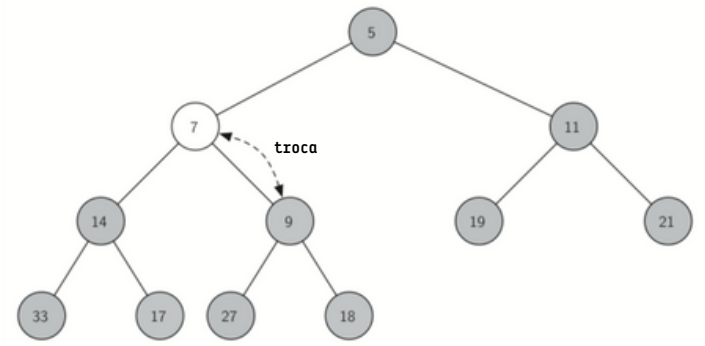
## INSERÇÃO




Podemos inserir um novo elemento sem nos preocuparmos com a propriedade 1 da heap, apenas se preocupando com a 2 propriedade, onde: Se o último nível não estiver completo, os nós folha deste nível estão completamente preenchidos da esquerda para a direita.




Após inserirmos o novo nó, começamos a verificar se o nó inserido tem uma prioridade maior que seu nó pai, se isso foi verdade, então os nós são invertidos, o pai passa ser filho e o filho passa ser pai.




Por meio de recursão, esse processo é repetido até que as propriedades da heap binária sejam respeitadas.

# IMPLEMENTAÇÃO EM C

```c
Heap *fila_prio_inserir(Heap *raiz, int prioridade, int dado)
{
    if (raiz == NULL)
    {
        Heap *novo = insere_na_fila(prioridade, dado);
        return novo;
    }

    if (raiz->esquerda == NULL)
    {
        raiz->esquerda = fila_prio_inserir(raiz->esquerda, prioridade, dado);
    }
    else if (raiz->direita == NULL)
    {
        raiz->direita = fila_prio_inserir(raiz->direita, prioridade, dado);
    }
    else
    {
        if (quant_de_nos(raiz->direita) == quant_de_nos(raiz->esquerda))
        {
            raiz->esquerda = fila_prio_inserir(raiz->esquerda, prioridade, dado);
        }
        else if (quant_de_nos(raiz->esquerda->esquerda) != quant_de_nos(raiz->esquerda->direita))
        {
            raiz->esquerda = fila_prio_inserir(raiz->esquerda, prioridade, dado);
        }
        else
        {
            raiz->direita = fila_prio_inserir(raiz->direita, prioridade, dado);
        }
    }

    raiz = ordena_heap(raiz);
    return raiz;
}
```

A função **fila_prio_inserir** opera da seguinte forma:

1. Primeiro, verifica se a raiz é igual a NULL. Se isso for verdade, retorna o novo elemento inserido na árvore.
2. Se a raiz não for NULL, ele verifica se o nó à esquerda da raiz é NULL. Isso é importante para manter a propriedade 2 de uma heap binária. Se o nó à esquerda for NULL, a função insere o novo elemento na esquerda da raiz.
3. Se o nó à esquerda não for NULL e o nó à direita for NULL, a função insere o novo elemento na direita da raiz.
4. No entanto, se a raiz tiver tanto um nó à esquerda quanto um nó à direita, a função realiza uma comparação do número total de nós nas subárvores esquerda e direita. Isso é feito através da função quant_de_nos, que retorna o número de nós em uma subárvore.
5. A comparação do número de nós nas subárvores não é suficiente para garantir que os nós respeitem a **propriedade 2** de uma heap binária. Portanto, a condição “else if” lida com a última situação. Ela verifica se a quantidade de nós na subárvore à esquerda é diferente da quantidade de nós na subárvore à direita do nó à esquerda. Se forem diferentes, a função implementa o novo elemento na subárvore à esquerda. Caso contrário, ela implementa o novo elemento na subárvore à direita.
6. Por fim, após a inserção, a função **ordena_heap** é chamada. Essa função é responsável por realizar um teste para verificar se a chave associada ao nó filho é tem maior prioridade do que a chave do pai. Se isso for verdade, a função efetua a troca de posição entre o nó pai e o nó filho, garantindo que o nó com a maior prioridade esteja na posição correta.

# FUNÇÕES UTILIZADAS NA INSERÇÃO

```c
Heap *heap_cria_vazia()
{
    return NULL;
}

Heap *insere_na_fila(int prioridade, int dado)
{
    Heap *novo = (Heap *)malloc(sizeof(Heap));
    novo->info.prioridade = prioridade;
    novo->info.dado = dado;
    novo->esquerda = heap_cria_vazia();
    novo->direita = heap_cria_vazia();
    return novo;
}

int quant_de_nos(Heap *raiz)
{
    int somador = 0;
    if (raiz == NULL)
    {
        return 0;
    }

    somador++;
    somador += quant_de_nos(raiz->esquerda);
    somador += quant_de_nos(raiz->direita);

    return somador;
}

Heap *ordena_heap(Heap *raiz)
{
    if (raiz->esquerda != NULL && raiz->esquerda->info.prioridade > raiz->info.prioridade)
    {
        Informacoes temp = raiz->info;
        raiz->info = raiz->esquerda->info;
        raiz->esquerda->info = temp;
    }

    if (raiz->direita != NULL && raiz->direita->info.prioridade > raiz->info.prioridade)
    {
        Informacoes temp = raiz->info;
        raiz->info = raiz->direita->info;
        raiz->direita->info = temp;
    }
    return raiz;
}
```

**Heap *heap_cria_vazia()** - Cria uma árvore vazia;

**Heap *insere_na_fila(int prioridade, int dado)** - Cria um nó na heap, atribui prioridade e dados, e inicializa nós filhos vazios.

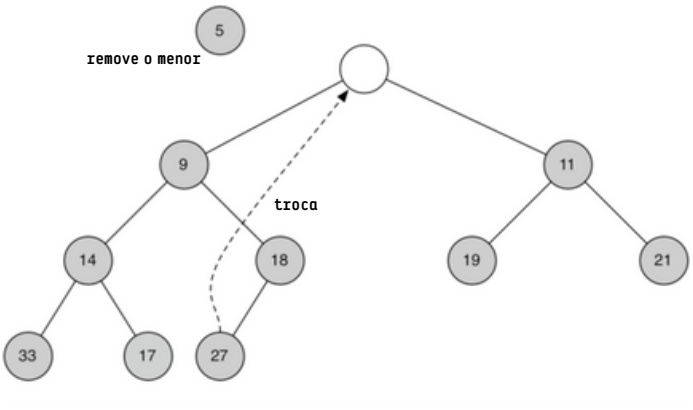
**int quant_de_nos(Heap *raiz)** - Retorna a quantidade de nós possui raiz.

**Heap *ordena_heap** - Faz a troca de informações se o filho tiver uma prioridade mais alta que o pai. A função ordena_heap está de acordo com o **max_heap,** para alterar para **min_heap**, basta trocar '>' por '<'.

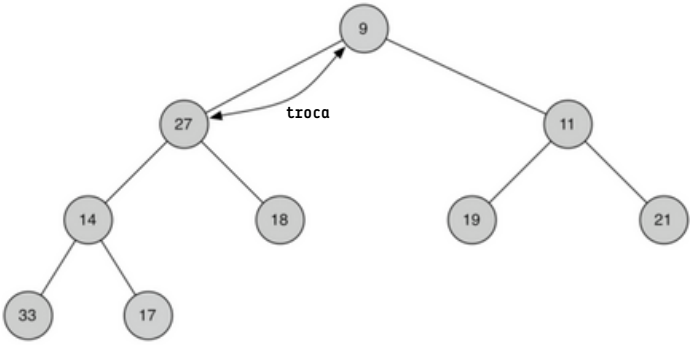
```c
// max_heap
if (raiz->esquerda != NULL && raiz->esquerda->info.prioridade > raiz->info.prioridade)
if (raiz->direita != NULL && raiz->direita->info.prioridade > raiz->info.prioridade)

// min_heap
if (raiz->esquerda != NULL && raiz->esquerda->info.prioridade < raiz->info.prioridade)
if (raiz->direita != NULL && raiz->direita->info.prioridade < raiz->info.prioridade)
```

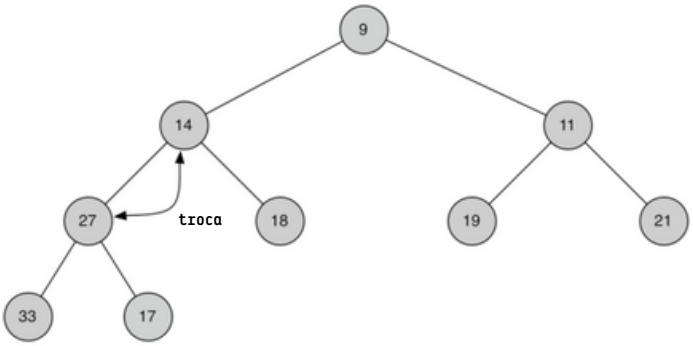
# REMOÇÃO



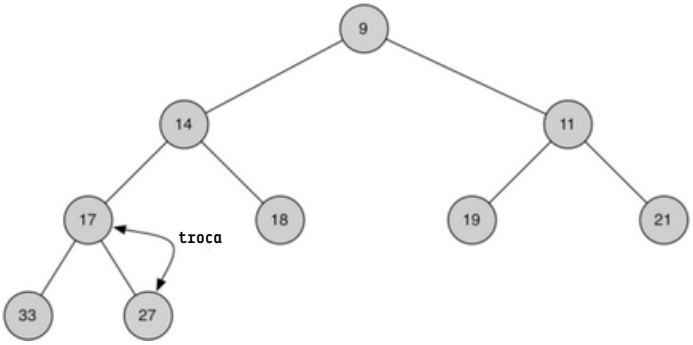

Se chamarmos a função de remover, o elemento que está na raiz da árvore será removido. Para fazer isso, devemos reorganizar novamente a heap. A primeira coisa que fazemos é trocar o conteúdo da raiz com o conteúdo do último elemento da heap. Em seguida, o último nó é liberado com a função free para liberar a memória associada a esse nó.




Após isso, como há um novo nó na raiz, é necessário reorganizar a heap. Para fazer isso, verificamos se a prioridade do nó na raiz é maior do que a prioridade de seus filhos. Se não for, realizamos a troca dos nós para garantir que o nó na raiz tenha a prioridade mais alta.

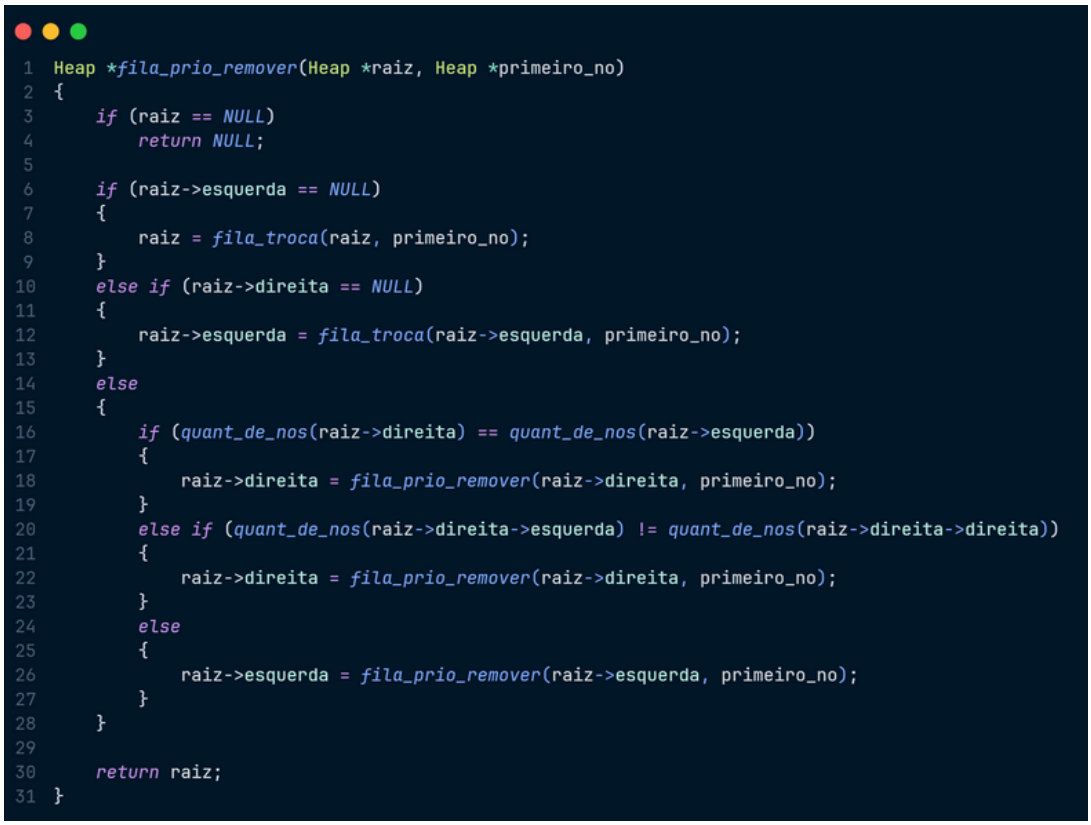



Por meio de recursão, esse processo é repetido até que as propriedades da heap binária sejam respeitadas.




Agora, a heap atende às propriedades da heap binária

# IMPLEMENTAÇÃO EM C



```c
Heap *fila_prio_remover(Heap *raiz, Heap *primeiro_no)
{
    if (raiz == NULL)
        return NULL;

    if (raiz->esquerda == NULL)
    {
        raiz = fila_troca(raiz, primeiro_no);
    }
    else if (raiz->direita == NULL)
    {
        raiz->esquerda = fila_troca(raiz->esquerda, primeiro_no);
    }
    else
    {
        if (quant_de_nos(raiz->direita) == quant_de_nos(raiz->esquerda))
        {
            raiz->direita = fila_prio_remover(raiz->direita, primeiro_no);
        }
        else if (quant_de_nos(raiz->direita->esquerda) != quant_de_nos(raiz->direita->direita))
        {
            raiz->direita = fila_prio_remover(raiz->direita, primeiro_no);
        }
        else
        {
            raiz->esquerda = fila_prio_remover(raiz->esquerda, primeiro_no);
        }
    }

    return raiz;
}
```

A função **fila_prio_remover** opera da seguinte forma:

1. Primeiramente, a função verifica se a raiz é NULL. Se for o caso, significa que a árvore está vazia, e a função retorna NULL.
2. Em seguida, verifica-se se a subárvore esquerda da raiz é NULL. Se for, chama-se a função fila_troca para trocar a raiz com o primeiro nó. Caso contrário, verifica-se se a subárvore à direita da raiz é NULL. Se for, chama a funcao fila_troca para trocar a raiz->esquerda com o primeiro nó.
3. No entanto, se a raiz tiver nós tanto à esquerda quanto à direita, realiza-se uma verificação comparando o número total de nós nas subárvores esquerda e direita. Essa verificação é feita através da função quant_de_nos, que retorna o número de nós em uma subárvore.
4. A comparação do número de nós nas subárvores não é suficiente para garantir que os nós sigam a propriedade 2 de uma heap binária. Portanto, a condição "**else if**" aborda a última situação. Ela examina se a quantidade de nós na subárvore à direita, se a **raiz->direita->esquerda** e **raiz->direita->direita** forem diferentes, a função remove o elemento na subárvore à direita. Caso contrário, remove o elemento na subárvore à esquerda.

# FUNÇÕES UTILIZADAS NA REMOÇÃO

```c

Heap *fila_troca(Heap *raiz, Heap *primeiro_no)
{
    primeiro_no->info = raiz->info;
    return NULL;
}

Heap *ordenar_toda_heap(Heap *raiz)
{
    if (raiz != NULL)
    {
        if (raiz->esquerda != NULL && raiz->esquerda->info.prioridade > raiz->info.prioridade)
        {
            Informacoes temp = raiz->info;
            raiz->info = raiz->esquerda->info;
            raiz->esquerda->info = temp;
            ordenar_toda_heap(raiz->esquerda);
        }

        if (raiz->direita != NULL && raiz->direita->info.prioridade > raiz->info.prioridade)
        {
            Informacoes temp = raiz->info;
            raiz->info = raiz->direita->info;
            raiz->direita->info = temp;
            ordenar_toda_heap(raiz->direita);
        }
    }
    return raiz;
}

Heap *remover(Heap *raiz)
{
    if (raiz == NULL)
        printf("Heap vazia, nao foi possivel remover um elemento!!\n");
    else
    {
        raiz = fila_prio_remover(raiz, raiz);
        ordenar_toda_heap(raiz);

        printf("Elemento removido com sucesso!\n");
    }
    return raiz;
}
```

**Heap *remover(Heap *raiz) -** A função remover verifica se a raiz é nula, indicando uma heap vazia. Se for o caso, exibe uma mensagem informando que não é possível remover um elemento. Se a heap não estiver vazia, chama a função fila_prio_remover para remover o elemento mais prioritário. Em seguida, chama a função ordenar_toda_heap para garantir que a propriedade da heap binária máxima seja mantida após a remoção.

**int quant_de_nos(Heap *raiz)** - Retorna a quantidade de nós possui raiz.

**Heap *ordenar_toda_heap(Heap *raiz)** - percorre a heap e faz trocas para garantir que os elementos mais prioritários estejam na raiz, seguindo a propriedade de uma heap binária máxima. A função ordena_heap está de acordo com o **max_heap,** para alterar para **min_heap**, basta trocar '>' por '<'.

```c
// max_heap
if (raiz->esquerda != NULL && raiz->esquerda->info.prioridade > raiz->info.prioridade)
if (raiz->direita != NULL && raiz->direita->info.prioridade > raiz->info.prioridade)

// min_heap
if (raiz->esquerda != NULL && raiz->esquerda->info.prioridade < raiz->info.prioridade)
if (raiz->direita != NULL && raiz->direita->info.prioridade < raiz->info.prioridade)
```

# CONCLUSÃO

As filas de prioridade são estruturas de dados fundamentais usadas em diversas aplicações para gerenciar elementos com base em suas prioridades. A Heap Binária é um tipo específico de Heap que impõe uma ordem às prioridades dos elementos. O processo de inserção em uma Heap Binária começa com a adição de um novo elemento, seguido por uma reorganização para manter as propriedades da Heap. Na remoção, o elemento de maior (ou menor) prioridade é eliminado primeiro, seguido por um processo de ajuste para restaurar as propriedades da Heap. Dado que nossos algoritmos não foram implementados usando arrays, a recursão desempenhou um papel crucial na manutenção dessas estruturas.

# REFERÊNCIAS BIBLIOGRAFIAS

https://www.facom.ufu.br/~abdala/DAS5102/TEO_HeapFilaDePrioridade.pdf

https://iq.opengenus.org/binary-heap/