

UNIVERSIDADE FEDERAL DE SANTA MARIA
CURSO CIÊNCIA DA COMPUTAÇÃO - BACHARELADO

**TRABALHO DE PESQUISA E IMPLEMENTAÇÃO DA DISCIPLINA DE
ESTRUTURAS DE DADOS “A”**

FILA DE PRIORIDADE (IMPLEMENTADA COMO *HEAP* BINÁRIO)

Gabriel Porto de Freitas
Gustavo Montagner dos Santos
Leandro Brum da Silva Lacorte

Professora Deise de Brum Saccol

Santa Maria/RS

2023

SUMÁRIO

1 INTRODUÇÃO	3
1.1 Definição de filas de prioridade	3
1.2 Breve explicação sobre implementação	3
2 DESENVOLVIMENTO	4
2.1 Fundamentos Teóricos	4
2.1.1 Estrutura de dados: heaps binários	5
2.1.2 Operações básicas (inserção, remoção, consulta)	5
2.2 Heaps Binários e Filas de Prioridade	6
2.2.1 Vantagens da implementação de filas de prioridade	6
2.2.2 Exemplos Ilustrativos com imagens de heaps em diferentes estados (após inserção e remoção)	7
2.3 Algoritmos	8
2.3.1 Inserção em um heap binário	9
2.3.2 Remoção em um heap binário	15
2.3.3 Impressão/Consulta de um heap binário	16
2.4 Aplicações e casos de uso	20
3 CONCLUSÃO	21
REFERÊNCIAS	22

1 INTRODUÇÃO

1.1 Definição de filas de prioridade

Filas de prioridade são estruturas de dados amplamente utilizadas em diversos sistemas, programas e setores gerais na computação. Essas estruturas têm a capacidade de organizar uma sequência de dados, onde cada um possui uma prioridade. Assim, processos, tarefas e execuções podem ser geridos, dependendo de sua importância ou tipo. Um exemplo prático dessa aplicação ocorre em hospitais de emergência, onde pacientes são atendidos com base na gravidade de sua condição.

Existem diversos métodos para implementar uma fila de prioridades, mas todos seguem um padrão. Cada um deve ter operações de inserção e remoção. Essas operações são fundamentais para garantir a funcionalidade da fila, permitindo a adição de elementos com base em suas prioridades e a remoção de elementos de acordo com a ordem de prioridade estabelecida.

Filas de prioridade podem ser estruturas de dados como heap binária, heap de Fibonacci e árvores de busca binária. Essas estruturas são construídas para garantir que as operações de inserção e remoção mantenham a ordem de prioridade correta. Por exemplo, em algoritmos de escalonamento de processos, as filas de prioridades são usadas para determinar qual tarefa deve ser executada a seguir, com base em sua importância relativa ou outros critérios específicos do contexto.

1.2 Breve explicação sobre implementação

A implementação de uma fila de prioridades pode ser feita de diversas maneiras, dependendo da necessidade ou preferência. As filas podem ser compostas por filas ou listas, que podem ter elementos ordenados ou desordenados. A estrutura permite flexibilidade na implementação, mas, mesmo assim, existem fundamentos que devem ser seguidos. Essas filas têm a vantagem de serem rápidas e eficazes.

As estruturas dessas filas diferem entre si. Cada uma delas é escolhida de acordo com as necessidades específicas do problema em questão. Em resumo, as **heaps binárias** são

especialmente úteis para filas de prioridades, as **heaps de Fibonacci** são otimizadas para operações específicas em grafos, e as **árvores de busca binária** são mais versáteis e úteis para armazenar dados de maneira ordenada e realizar operações de busca eficientes.

2 DESENVOLVIMENTO

2.1 Fundamentos Teóricos

As filas de prioridade têm como base princípios teóricos que garantem sua eficiência e utilidade em diversas situações. Esses fundamentos incluem:

- Ordenação:

A essência das filas de prioridade está na classificação dos elementos de acordo com suas prioridades. Cada elemento é associado a uma prioridade que determina sua posição na fila em relação aos demais.

- Uso de Heap:

Muitas implementações de filas fazem uso da estrutura de dados "heap", que é uma árvore binária completa na qual cada nó pai possui um valor menor ou igual aos seus filhos. Essa propriedade de heap viabiliza operações eficientes de inserção e remoção.

- Eficiência de Tempo:

As filas de prioridade são projetadas para apresentar eficiência em relação ao tempo. Geralmente, as operações de inserção, remoção e obtenção do elemento de maior prioridade devem ser eficazes, frequentemente com uma complexidade de $O(\log n)$, onde n é o número de elementos na fila.

Os princípios teóricos das filas de prioridade focam na eficiência da ordenação com base em prioridades e na aplicabilidade dessa estrutura em diversos cenários computacionais. A habilidade de manipular elementos de acordo com sua importância é crucial em muitos algoritmos e sistemas.

2.1.1 Estrutura de dados: heaps binários

Um heap binário é uma árvore binária completa em que cada nó contém uma chave (valor). Dependendo da implementação, esse valor pode ser o maior (max heap) ou menor (min heap) do que as chaves de seus filhos. Em um heap binário máximo, a chave do nó pai é sempre maior que as chaves dos nós filhos, enquanto em um heap binário mínimo, a chave do nó pai é sempre menor.

Essas estruturas de dados utilizam uma representação em forma de árvore binária, e existem propriedades que facilitam as operações de inserção, remoção e ajuste do heap. A propriedade de forma indica que a árvore deve ser completa ou parcialmente completa, o que significa que todos os níveis, exceto possivelmente o último, devem estar preenchidos. Se o último nível não estiver completo, os nós folha nesse nível são preenchidos da esquerda para a direita.

Além disso, a propriedade do heap, como mencionado anteriormente, estabelece que a árvore binária pode ser um max heap ou min heap. Em um max heap, um nó pai deve ter prioridade maior do que seus nós filhos, enquanto em um min heap, um nó pai deve ter prioridade menor do que seus nós filhos. Essa propriedade é fundamental para manter a ordem de prioridade na heap binária, influenciando diretamente nas operações de inserção, remoção e ajuste, garantindo que a propriedade do heap seja preservada ao longo das operações.

2.1.2 Operações básicas (inserção, remoção, consulta)

As operações de inserção, remoção e consulta desempenham um papel fundamental em uma fila de prioridade, proporcionando à estrutura a autonomia e eficácia necessárias.

- Inserção:

Ao realizar a operação de inserção em uma fila de prioridades com heap binário, o objetivo é adicionar elementos à estrutura de dados. Quando um elemento é inserido, ele ocupa a posição no final da árvore. Se necessário, é executado um ajuste ascendente, conhecido como "heapify-up", para assegurar a propriedade de heap. Esse processo movimenta o novo elemento para a posição apropriada na árvore.

- Remoção:

A operação de remoção envolve retirar o elemento de maior (ou menor) prioridade da raiz da árvore. Após a remoção, o último elemento da árvore é movido para a raiz, seguido por uma operação de ajuste descendente, também chamada de "heapify-down", para garantir a preservação da propriedade de heap. Durante essa operação, o elemento é deslocado para baixo na árvore até atingir a posição adequada, mantendo a prioridade em relação aos filhos.

- Consulta:

A operação de consulta permite verificar o elemento de maior (ou menor) prioridade na fila sem removê-lo. Em geral, essa operação consiste em observar a raiz da árvore, que representa o elemento com a prioridade máxima (ou mínima), dependendo se é um max-heap ou min-heap.

Essas operações são cruciais para manipular os dados na fila de prioridades com heap binário, assegurando que a estrutura mantenha sua propriedade fundamental de heap. Isso torna a fila eficiente em diversas aplicações em ciência da computação, como algoritmos de busca, ordenação e otimização de recursos.

2.2 Heaps Binários e Filas de Prioridade

Heaps binários e filas de prioridade estão estreitamente relacionados; os heaps binários contribuem como uma base eficiente para a implementação das filas de prioridade. Essa combinação é fundamental em muitos campos da ciência da computação, representando uma abordagem eficaz para lidar com conjuntos de dados que possuem prioridades distintas. Em resumo, um heap binário é uma estrutura de dados que pode ser empregada para implementar uma fila de prioridade, e é ele que utilizamos em nossos códigos.

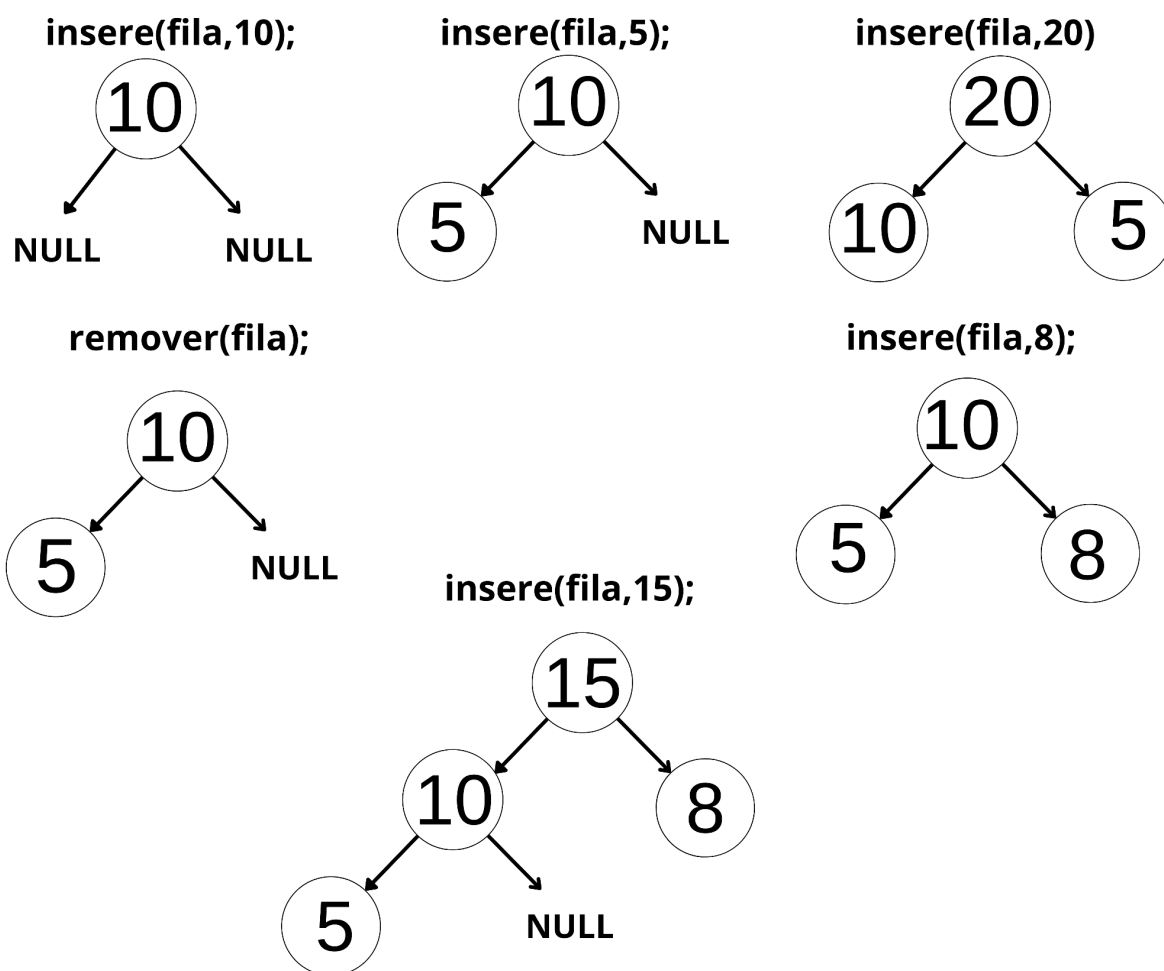
2.2.1 Vantagens da implementação de filas de prioridade

- Acesso eficiente ao maior ou menor elemento:
 - Heaps binários permitem acesso rápido ao elemento de maior ou menor prioridade, já que esses elementos estão sempre nas posições 1 (no caso de uma fila de

prioridade implementada com um heap de máximo) ou na última posição (no caso de um heap de mínimo).

- Inserção e remoção eficientes:
 - Inserir e remover elementos em um heap binário tem complexidade assintótica eficiente, $O(\log n)$, onde n é o número de elementos na fila. Isso permite operações rápidas mesmo em grandes conjuntos de dados.
- Estrutura de dados flexível:
 - Heaps binários são estruturas de dados versáteis que podem ser utilizadas em diferentes contextos além de filas de prioridade. Eles servem como base para algoritmos como o Heapsort e são utilizados em várias aplicações de otimização.
- Implementação relativamente simples:
 - Comparado a outras estruturas de dados complexas, a implementação de um heap binário é relativamente simples. Essa simplicidade facilita a compreensão e a manutenção do código.
- Manutenção da ordem de prioridade:
 - Os heaps binários mantêm automaticamente a ordem de prioridade. Após inserções ou remoções, a estrutura se reorganiza para garantir que o elemento de maior ou menor prioridade esteja sempre acessível rapidamente.

2.2.2 Exemplos Ilustrativos com imagens de heaps em diferentes estados (após inserção e remoção)



Considerando que a cada inserção e remoção o heap seja ajustado, o exemplo acima exemplifica perfeitamente como o processo de inserção e remoção funciona em um heap. O correto ajuste do heap é crucial para manter a propriedade de heap binário e garantir a eficiência das operações.

2.3 Algoritmos

Agora que já temos uma noção mais ampliada de filas de prioridade e heaps binários, vamos implementar um código que consiga inserir um elemento em uma fila de prioridade max heap, remover o maior elemento e consultar na fila. Seguindo os passos e exemplos abaixo, construiremos uma fila de prioridade com heap binário max heap que siga as propriedades necessárias.

Primeiro, vamos criar as estruturas necessárias

```

#include <stdio.h>
#include <stdlib.h>
//cria estruturas
typedef struct {
    int dado;
    int prioridade;
} Item;

typedef struct {
    Item* itens;
    int n;
    int tamanho;
} FilaP;

```

Define as variáveis para armazenar um dado e a prioridade de um elemento na fila.

Define o tamanho e o ponteiro para o array da fila de prioridades.

Com as estruturas definidas, vamos criar a fila.

```

//fila cria
FilaP* criarFilaP(int tam) {
    FilaP* fila = (FilaP*)malloc(sizeof(FilaP));
    fila->itens = (Item*)malloc((tam + 1) * sizeof(Item));
    fila->n = 0;
    fila->tamanho = tam;
    return fila;
}

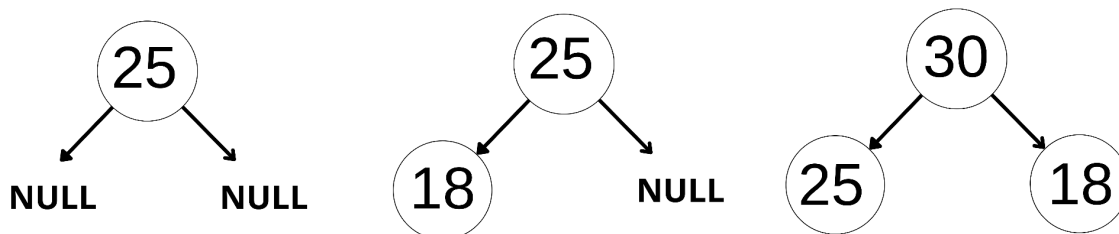
```

A função recebe o tamanho desejado para a fila de prioridade e, assim, cria e retorna uma fila de prioridades.

2.3.1 Inserção em um heap binário

Para inserir um novo elemento em um heap binário, obedecendo às suas propriedades, é necessário adicionar o elemento à primeira posição vaga do heap e garantir que o heap seja

ajustado, caso esse elemento seja maior ou menor que seu pai, dependendo do tipo de implementação (max heap) ou (min heap). No nosso caso, vamos utilizar o (max heap).



Passo 1: Inserir novo elemento no final do heap

Primeiro, criamos uma função que, a cada iteração, insere um elemento na fila. Essa função deve verificar se a fila está vazia, além de chamar outra função que ajusta o heap caso o elemento inserido tenha maior prioridade que seu pai.

A função recebe como parâmetros o ponteiro para a fila, assim como os dados que serão armazenados.

```
void insere(FilaP* fila, int prioridade, int dado) {
    if (fila->n < fila->tamanho) {
        fila->n++;
        fila->itens[fila->n].prioridade = prioridade;
        fila->itens[fila->n].dado = dado;
        ajuste_insere(fila, fila->n);
    } else {
        printf("Fila cheia. \n");
    }
    return;
}
```

```

void insere(FilaP* fila,int prioridade ,int dado) { insere(fila);
    if (fila->n < fila->tamanho) {
        fila->n++;
        fila->itens[fila->n].prioridade = prioridade;
        fila->itens[fila->n].dado = dado;
        ajuste_insere(fila, fila->n);
    } else {
        printf("Fila cheia. \n");
        }
    return;
}

```

Se a fila estiver cheia, não faz nada.
Se houver espaço, inicia a inserção.

Se a fila não estiver cheia, a função incrementa o valor de n para ocupar a próxima posição e armazena os dados nessa posição.

```

void insere(FilaP* fila,int prioridade ,int dado) {
    if (fila->n < fila->tamanho) {
        fila->n++;
        fila->itens[fila->n].prioridade = prioridade;
        fila->itens[fila->n].dado = dado;
        ajuste_insere(fila, fila->n);
    } else {
        printf("Fila cheia\n");
    }
    return;
}

```

complemente fila->n para
pegar a próxima posição da fila.

armazena os dados
do novo elemento.

Agora que inserimos um elemento na fila, devemos chamar uma função para ajustar o heap. Esse procedimento deve ser feito porque não podemos garantir que o novo elemento estará na posição correta. Devemos comparar o elemento com seu pai e, se a prioridade for maior, trocá-los de lugar. Isso deve ser repetido até que esse elemento esteja no topo ou que seu pai tenha maior prioridade. Seguindo os passos abaixo, vamos implementar essa função.

```
//resto do codigo...
```

```
fila->n++;
```

```
fila->itens[fila->n].prioridade = prioridade;
```

```
fila->itens[fila->n].dado = dado;
```

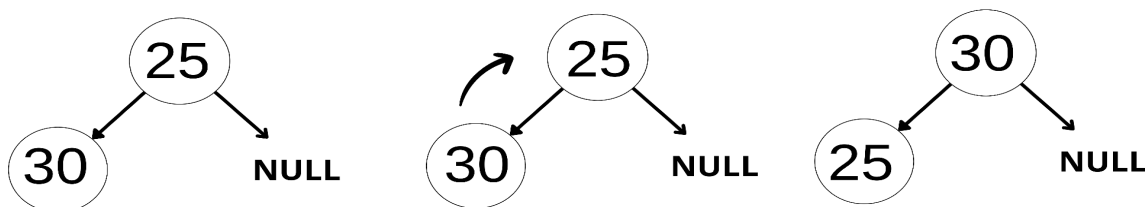
```
ajuste_inserere(fila, fila->n);
```

```
//resto do codigo...
```

Chama a função para
ajustar o heap.

Ajustar o heap

Após a inserção, deve-se ajustar o heap para manter a propriedade do Max Heap. O ajuste envolve mover o novo elemento para cima na árvore, garantindo que o heap mantenha a propriedade de ter o maior elemento na raiz e que os elementos abaixo sigam a estrutura de heap (pai maior que os filhos).



Passo 2: Ajustar o heap

Primeiro, criamos uma função que, a cada inserção de um elemento na fila, verifica se esse elemento tem uma prioridade maior que seu pai e, se tiver, troca-os de lugar. Esse processo continua até que esse elemento esteja no topo ou sua prioridade seja menor que seu pai.

```

void ajuste_inserere(FilaP* fila, int filho) {
    int pai = filho / 2;
    Item temp = fila->itens[filho];

    while (filho > 1 && fila->itens[pai].prioridade <
temp.prioridade) {
        fila->itens[filho] = fila->itens[pai];
        filho = pai;
        pai = filho / 2;
    }
    fila->itens[filho] = temp;
}
  
```

```

void ajuste_insere(FilaP* fila, int filho) {
    int pai = filho / 2;
    Item temp = fila->itens[filho];
    while (filho > 1 && fila->itens[pai].prioridade < temp.prioridade) {
        fila->itens[filho] = fila->itens[pai];
        filho = pai;
        pai = filho / 2;
    }
    fila->itens[filho] = temp;
}

```

Recebe o ponteiro para a fila e o índice do elemento recém inserido.

Com base nas propriedades dos heaps, podemos encontrar o pai de um elemento simplesmente dividindo o seu índice por 2.

```

void ajuste_insere(FilaP* fila, int filho) {
    int pai = filho / 2;
    Item temp = fila->itens[filho];
    while (filho > 1 && fila->itens[pai].prioridade < temp.prioridade) {
        fila->itens[filho] = fila->itens[pai];
        filho = pai;
        pai = filho / 2;
    }
    fila->itens[filho] = temp;
}

```

Obtém o índice do pai e salva o filho.

Depois é só verificar se é o primeiro elemento. Se for, não faz nada; não há o que ajustar. Mas se não for o primeiro, compara-se a sua prioridade com a do seu pai.

```

void ajuste_insere(FilaP* fila, int filho) {
    int pai = filho / 2;
    Item temp = fila->itens[filho];
    while (filho > 1 && fila->itens[pai].prioridade < temp.prioridade) {
        fila->itens[filho] = fila->itens[pai];
        filho = pai;
        pai = filho / 2;
    }
    fila->itens[filho] = temp;
}

```

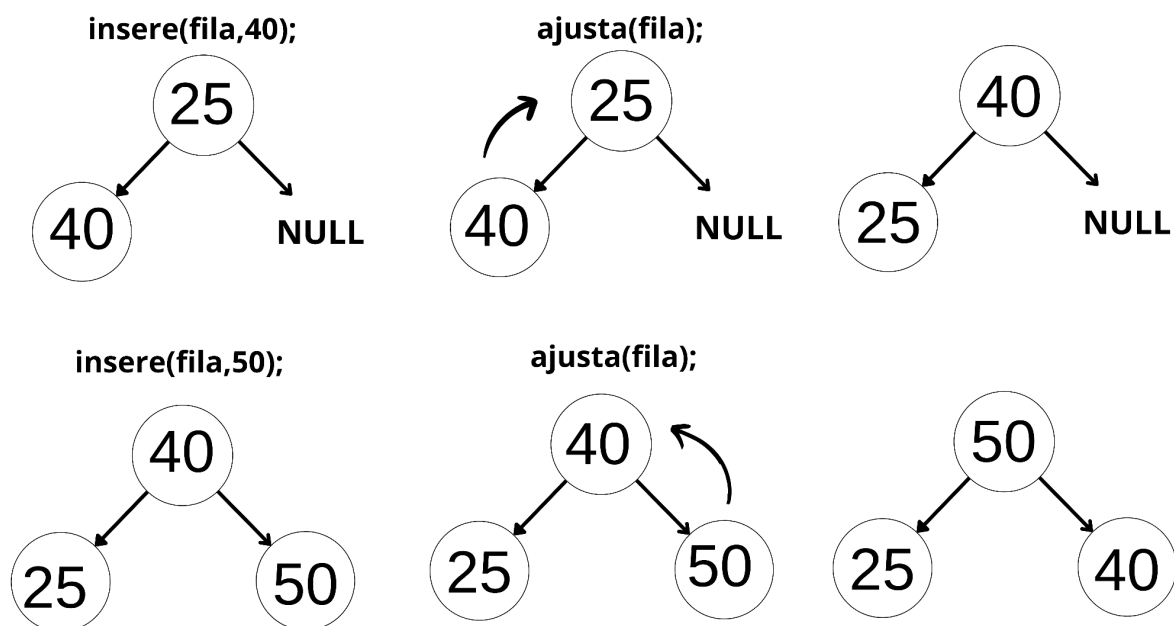
Verifica se é o primeiro elemento e se sua prioridade é maior que a do pai.

Se entrar no while significa que a prioridade do filho é maior do que a do pai; então os dois devem ser trocados de lugar.

```
void ajuste_inserere(FilaP* fila, int filho) {
    int pai = filho / 2;
    Item temp = fila->itens[filho];
    while (filho > 1 && fila->itens[pai].prioridade < temp.prioridade) {
        fila->itens[filho] = fila->itens[pai];
        filho = pai;
        pai = filho / 2;
    }
    fila->itens[filho] = temp;
}
```

Troca os itens de lugar.

Agora o while vai repetir o processo até que o heap se ajuste, respeitando suas propriedades.



2.3.2 Remoção em um heap binário

Verifica se a fila está vazia, remove o item de maior prioridade (assumindo que está na posição 1, e ajusta a estrutura do heap após a remoção.

```
Item remover(FilaP* fila) {  
    if (fila->n == 0) {  
        printf("Fila vazia\n");  
    }  
    int indice_maior_prioridade = 1;  
  
    // Salvar o item de maior prioridade para retornar no final  
    Item item_removido = fila->itens[indice_maior_prioridade];  
  
    // Substituir o item de maior prioridade pelo último item na fila  
    fila->itens[indice_maior_prioridade] = fila->itens[fila->n];  
    fila->n--; // Decrementar o número de itens na fila  
  
    //chama a função para ajustar o heap  
    ajuste_de_remocao(fila, indice_maior_prioridade);  
  
    return item_removido;  
}
```

Esta função é responsável por manter a propriedade do heap após a remoção de um elemento. Ela compara o elemento atual (pai) com seus filhos e faz as trocas necessárias para manter a estrutura do heap válida.

```

void ajuste_de_remocao(FilaP* fila, int pai){
    int filho_esq, filho_dir, maior;

    while (1) {
        maior = pai;
        filho_esq = 2 * pai ;
        filho_dir = 2 * pai + 1;
        // acha os filhos atraves das propriedades do heap
        if (filho_esq < fila->n && fila->itens[filho_esq].prioridade > fila->itens[maior].prioridade){
            maior = filho_esq;
        }
        //verfca se o filho_esq tem uma prioridade maior
        if (filho_dir < fila->n && fila->itens[filho_dir].prioridade > fila->itens[maior].prioridade){
            maior = filho_dir;
        }
        //verfca se o filho_dir tem uma prioridade maior
        if (maior == pai){
            break;
        }
        //se nenhum dos filhos é maior que o pai nao faz nada
        Item aux = fila->itens[pai];
        fila->itens[pai] = fila->itens[maior];
        fila->itens[maior] = aux;
        //se algum for maior troca com o pai
        pai = maior;
    }
}

```

2.3.3 Impressão/Consulta de um heap binário

Para impressão do heap binário, fizemos 3 funções separadas em que cada uma realiza a impressão em um formato diferente.

- `imprimeFilaPDetalhado()`:

Essa função recursiva pode ser chamada para imprimir a fila de prioridades em uma listagem simples ordenada da maior prioridade para a menor.

Passando como parâmetros um ponteiro para a fila e um índice inicial (relativo a em qual nó a impressão deve partir), como na criação da fila definimos o índice inicial como 1, podemos simplesmente utilizar a propriedade do heap binário que diz que: para encontrar o filho da

esquerda basta multiplicar o índice por 2; sendo assim, para encontrar o filho da direita basta incrementar 1 nessa multiplicação.

Primeiro testamos se o número de elementos na fila (n) é nulo, pois sendo assim a fila ainda está vazia e retorna-se uma mensagem.

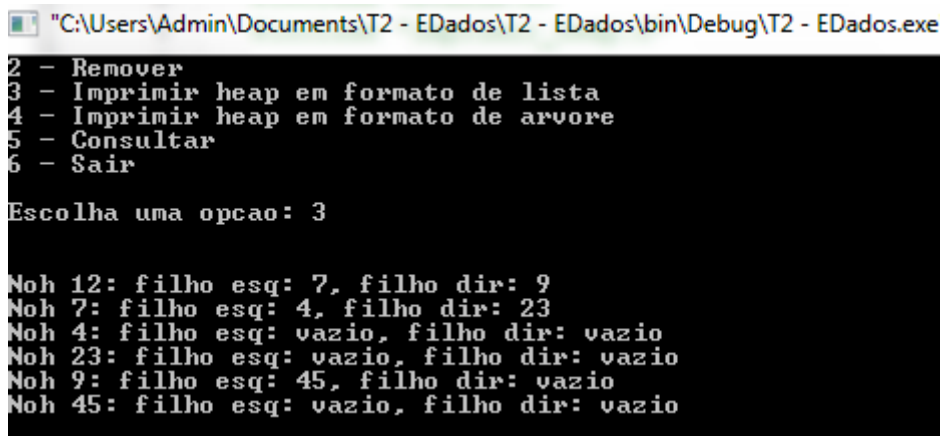
Após, caso o índice esteja dentro do limite atual da fila, imprimimos o nó pai. Depois, testando novamente esse limite, imprimimos os nós filhos da esquerda e da direita, sendo que, caso o índice do filho seja maior que o número de elementos atual, isso significa que o tal filho está vazio.

Por fim, essa função é chamada novamente de forma recursiva, primeiro para o filho da esquerda e depois para o filho da direita, dessa forma percorrendo a fila até que não haja mais elementos a serem imprimidos.

```

5 void imprimeFilaPDetalhado(FilaP* fila, int indice) {
6     if (fila->n == 0) {
7         printf("Fila vazia\n");
8         return;
9     }
10
11     if (indice <= fila->n) {
12         int filho_esq = 2 * indice;
13         int filho_dir = 2 * indice + 1;
14
15         // Imprime informações sobre o nó pai e depois seus filhos
16         printf("Noh %d: ", fila->itens[indice].dado);
17
18         if (filho_esq <= fila->n) {
19             printf("filho esq: %d, ", fila->itens[filho_esq].dado);
20         } else {
21             printf("filho esq: vazio, ");
22         }
23
24         if (filho_dir <= fila->n) {
25             printf("filho dir: %d\n", fila->itens[filho_dir].dado);
26         } else {
27             printf("filho dir: vazio\n");
28         }
29
30         // Chama recursivamente para os filhos
31         imprimeFilaPDetalhado(fila, filho_esq);
32         imprimeFilaPDetalhado(fila, filho_dir);
33     }
34 }

```



```

"C:\Users\Admin\Documents\T2 - EDados\T2 - EDados\bin\Debug\T2 - EDados.exe"
2 - Remover
3 - Imprimir heap em formato de lista
4 - Imprimir heap em formato de arvore
5 - Consultar
6 - Sair

Escolha uma opcao: 3

Noh 12: filho esq: 7, filho dir: 9
Noh 7: filho esq: 4, filho dir: 23
Noh 4: filho esq: vazio, filho dir: vazio
Noh 23: filho esq: vazio, filho dir: vazio
Noh 9: filho esq: 45, filho dir: vazio
Noh 45: filho esq: vazio, filho dir: vazio

```

- `imprimeFilaPARvore()`:

Essa função, também recursiva, funciona de forma semelhante, porém imprime todos os nós de uma vez só, com uma formatação de identações que facilitam visualizar a fila em um formato de árvore convencional com hierarquias.

A função recebe os mesmos parâmetros da anterior, mais uma variável nível inicial que serve para fazer a indentação, e inicialmente deve ser passado 0.

Da mesma forma, verifica se a fila está vazia inicialmente e utiliza a propriedade do heap binário para calcular os filhos.

Em seguida, chama a função recursivamente para o filho da direita. Isso significa que vamos imprimir de forma recursiva começando pelo nó raiz e indo em direção aos filhos da direita.

Após, é feita uma indentação para representar o nível do nó na árvore, com a quantidade de espaços proporcional. Em seguida, imprimimos o valor do nó pai.

Por fim, para continuar a exploração recursiva da árvore, é chamada a função recursivamente para o filho da esquerda.

```

36 void imprimeFilaPARvore(FilaP* fila, int indice, int nivel) {
37     if (fila->n == 0) {
38         printf("Fila vazia\n");
39         return;
40     }
41
42     if (indice <= fila->n) {
43         int filho_esq = 2 * indice;
44         int filho_dir = 2 * indice + 1;
45
46         // Chama recursivamente para o filho da direita
47         imprimeFilaPARvore(fila, filho_dir, nivel + 1);
48
49         // Identação para representar o nível na árvore
50         for (int i = 0; i < nivel; i++) {
51             printf("  ");
52         }
53
54         // Imprime informações sobre o nó pai
55         printf("%d\n", fila->itens[indice].dado);
56
57         // Chama recursivamente para o filho da esquerda
58         imprimeFilaPARvore(fila, filho_esq, nivel + 1);
59     }
60 }

```

```

Menu:
1 - Inserir
2 - Remover
3 - Imprimir heap em formato de lista
4 - Imprimir heap em formato de árvore
5 - Consultar
6 - Sair

Escolha uma opção: 4

      9
     45
12    23
      7
      4

```

- consultaDado():

Uma função mais simples, serve para fazer uma consulta de um determinado dado na fila, recebendo como parâmetro esse dado e um ponteiro para a fila.

Procurando desde o índice 1 até o número de elementos registrados atualmente na fila, o programa percorre um laço for, verificando a correspondência desse dado, e imprimindo caso encontrado.

Utilizando uma variável simples, caso nunca entre no if que indica que foi encontrado, mostra que o dado não existe na fila atual.

```

62 void consultaDado(FilaP* fila, int dado) {
63     int encontrado = 0;
64
65     for (int i = 1; i <= fila->n; i++) {
66         if (fila->itens[i].dado == dado) {
67             printf("Dado %d encontrado na posicao %d da fila. Prioridade: %d\n", dado, i, fila->itens[i].prioridade);
68             encontrado = 1;
69         }
70     }
71
72     if (!encontrado) {
73         printf("Dado %d nao encontrado na fila.\n", dado);
74     }
75 }
76

```

```

Menu:
1 - Inserir
2 - Remover
3 - Imprimir heap em formato de lista
4 - Imprimir heap em formato de arvore
5 - Consultar
6 - Sair

Escolha uma opcao: 5

Digite o dado a ser consultado: 5
Dado 5 nao encontrado na fila.

Menu:
1 - Inserir
2 - Remover
3 - Imprimir heap em formato de lista
4 - Imprimir heap em formato de arvore
5 - Consultar
6 - Sair

```

```

Escolha uma opcao: 5
es
os
ns
Menu:
Digite o dado a ser consultado: 9
Dado 9 encontrado na posicao 3 da fila. Prioridade: 5

```

2.4 Aplicações e casos de uso

- Sistema de Atendimento Médico
 - Em um hospital, a triagem de pacientes pode ser gerenciada por uma fila de prioridade. Pacientes com condições críticas, como parada cardíaca, são atendidos imediatamente, enquanto pacientes com condições menos urgentes podem esperar mais tempo.
- Controle de Tráfego Aéreo

- No controle de tráfego aéreo, os aviões em situações críticas, como emergências médicas a bordo ou baixo combustível, recebem prioridade para pouso ou decolagem em relação a voos regulares.

- Sistemas de Reservas e Bilhetes

- Em um sistema de reservas de companhias aéreas, os passageiros podem ter diferentes níveis de prioridade com base em seu status, como passageiros frequentes ou de classe executiva, permitindo-lhes acesso preferencial a assentos.

3 CONCLUSÃO

Neste trabalho, exploramos o conceito de filas de prioridades, compreendemos como são implementadas com heaps binários e suas utilidades na computação. Com isso, percebemos que filas de prioridades são cruciais para gerenciar dados de forma eficiente, especialmente em situações em que a ordenação e o acesso baseados em prioridade são essenciais.

A implementação com heaps binários destaca-se como uma abordagem eficaz para identificar e manipular rapidamente elementos de alta prioridade, mostrando a versatilidade e aplicabilidade dessas estruturas na computação. Este estudo reforça a associação das filas de prioridade com suas implementações em heaps binários, heaps de Fibonacci e árvores de busca binária. Além disso, destaca a importância dessas filas como ferramentas essenciais em algoritmos e estruturas de dados.

Esses conceitos foram desenvolvidos e projetados para resolver problemas reais, e ao examiná-los de perto, notamos sua presença em muitas áreas do nosso cotidiano. Essas estruturas são poderosas, flexíveis, úteis e eficazes. Por fim, cabe destacar que essas implementações foram construídas exatamente para solucionar desafios na computação.

REFERÊNCIAS

MC-202 Filas de Prioridade e Heap - Instituto de Computação

Fila de prioridades em C - C/C#/C++ - Clube do Hardware

https://wagnergaspar.com/como-implementar-a-estrutura-de-dados-fila-de-prioridade-em-c/#google_vignette