

FILAS

Estrutura de Dados

Fila

- Uma fila é uma estrutura de dados dinâmica que admite remoção de elementos e inserção de novos objetos. Mais especificamente, uma fila (= queue) é uma estrutura sujeita à seguinte regra de operação: sempre que houver uma remoção,
- o elemento removido é o que está na estrutura há mais tempo.
- Em outras palavras, o primeiro objeto inserido na fila é também o primeiro a ser removido. Essa política é conhecida pela sigla FIFO (= First-In-First-Out).

Implementação em um vetor

- Suponha que nossa fila mora em um vetor $\text{fila}[0..N-1]$. (A natureza dos elementos do vetor é irrelevante: eles podem ser inteiros, bytes, ponteiros, etc.) Digamos que a parte do vetor ocupada pela fila é
- $\text{fila}[p..u-1]$.

Implementação em um vetor

- O primeiro elemento da fila está na posição p e o último na posição $u-1$. A fila está vazia se $p == u$ e cheia se $u == N$. A figura mostra uma fila que contém os números 111, 222, ..., 666:

0		p						u			N-1
		111	222	333	444	555	666				

Implementação em um vetor

- Para tirar, ou remover (= delete = de-queue), um elemento da fila basta fazer
 - ▣ `x = fila[p++];`
- Isso equivale ao par de instruções `x = fila[p]; p += 1;`, nesta ordem. É claro que você só deve fazer isso se tiver certeza de que a fila não está vazia. Para colocar, ou inserir (= insert = enqueue), um objeto `y` na fila basta fazer
 - ▣ `fila[u++] = y;`

Implementação em um vetor

- Isso equivale ao par de instruções `fila[u] = y; u += 1;`, nesta ordem. Note como esse código funciona corretamente mesmo quando a fila está vazia. É claro que você só deve inserir um objeto na fila se ela não estiver cheia; caso contrário, a fila transborda (ou seja, ocorre um overflow).
- Para ajudar o leitor humano, podemos embalar as operações de remoção e inserção em duas pequenas funções. Se os objetos da fila forem números inteiros, podemos escrever

Implementação em um vetor



```
int tiradafila (void) {  
    return fila[p++];  
}  
  
void colocanafila (int y) {  
    fila[u++] = y;  
}
```

Implementação em um vetor



- Estamos supondo aqui que as variáveis `fila`, `p`, `u` e `N` são globais, isto é, foram definidas fora do código das funções. (Para completar o pacote, precisaríamos de mais três funções: uma que crie uma fila, uma que verifique se a fila está vazia e uma que verifique se a fila está cheia; veja exercício abaixo.)

Implementação em um vetor

□ **Função vazia()**

- ▣ No decorrer de nosso programa sobre estrutura de dados do tipo fila, muitas vezes será necessário checar se a fila está vazia ou não.
- ▣ Isso é feito de uma maneira bem simples: checando o ponteiro "prox" da struct "FILA".
Se apontar pra NULL, a fila está vazia. Do contrário, tem pelo menos um elemento na fila.

Implementação em um vetor

□ **Função insere()**

- ▣ Antes de mais nada, vamos usar a função `aloca()` para reservar um espaço em memória para o novo nó, o "novo". Como este novo nó será o último da fila, seu ponteiro "prox" deve apontar para NULL. Esta foi a primeira parte do processo de se adicionar um elemento em uma fila.

Implementação em um vetor

- A segunda parte é adicionar este novo nó ao final da fila, e para tal, devemos achar o último nó da fila.

Primeiro checamos se a fila está vazia, pois se tiver, basta colocar no novo nó em `FILA->prox`

Caso não esteja, criamos um ponteiro "tmp" que vai apontar para todos os elementos da fila em busca do último. Ele começa no primeiro elemento, que está em "FILA->prox".

Se "tmp->prox" apontar para NULL, o ponteiro aponta para o último da fila.

Senão, devemos seguir adiante com o ponteiro (`tmp = tmp->prox`) até acharmos o último elemento.

Achando, colocamos lá o novo nó, o "novo".

A variável inteira "tam" é para definir o tamanho da fila (número de nós). Usaremos este inteiro na função "exibe()", que vai exibir os elementos da fila.

Implementação em um vetor

□ **Função retira()**

- Vamos agora retirar um elemento da fila.
E segunda a lógica deste tipo de estrutura de dados, vamos retirar o primeiro nó.

Antes de tudo, checamos se a fila não está vazia.

Se estiver, trabalho feito, pois não precisaremos retirar nó algum da estrutura de dados.

Caso a fila não esteja vazia, precisamos identificar o primeiro elemento e o segundo (na verdade, não é obrigado que exista um segundo elemento). O que precisamos fazer é que "FILA->prox" não aponte mais para o primeiro elemento, e sim para o segundo

Implementação em um vetor

- Vamos usar um ponteiro "tmp" para apontar para o primeiro elemento da fila: `tmp = FILA->prox`
Se "tmp" aponta para o primeiro elemento, então "tmp->prox" aponta para o segundo elemento ou NULL, caso a fila só tenha um nó.

Agora vamos fazer a ligação entre o ponteiro base (FILA) e o segundo elemento (ou NULL) da fila:

```
FILA->prox = tmp->prox
```

Pronto. Tiramos o primeiro elemento da jogada, pois se ninguém aponta para ele, ele não faz mais parte da estrutura de dados.

Interessante, não?

Note que declaramos a função "retira()" como sendo do tipo struct Node, pois é uma boa prática retornar o nó que retiramos, pois geralmente retiramos ele da estrutura para fazer algo, trabalhar em cima dele. Depois que retornamos ele pra função "opcao()" liberamos o espaço que havia sido alocado para ele.

Implementação em um vetor

□ **Função `exibe()`**

- ▣ Esta função serve somente para mostrar os números ("num") existentes em cada nó, para você poder adicionar, retirar e ver o que vai acontecendo com a fila.

Ou seja, esta função tem mais propósitos didáticos, pra você ver as coisas realmente funcionando na sua frente, como devem funcionar.

Basicamente pegamos um ponteiro "tmp" e fazemos ele apontar para cada um dos nós, exibindo o número. Para saber o tanto de nós existentes e a ordem, usamos a variável "tam" que é incrementada quando adicionamos nó na fila (função `insere`) e decrementada quando tiramos elementos da estrutura de dados (função `retira`).

Implementação em um vetor

□ Função libera()

- Esta função simplesmente tem por objetivo ir em cada nó e liberar a memória alocada.

Para tal, usamos dois ponteiros.

Um ponteiro aponta para um nó ("atual"), e o outro ponteiro aponta para o nó seguinte ("proxNode").

Liberamos o primeiro ponteiro, ou seja, aquele nó deixa de existir. Se tivéssemos só este ponteiro, nos perderíamos na fila.

Porém, o ponteiro que aponta para o nó seguinte da fila serve pra isso, pois agora temos o ponteiro para o próximo elemento da fila "proxNode").

Agora damos um passo pra frente, fazendo um ponteiro apontar para este próximo elemento ("atual = proxNode"), e o ponteiro próximo, para uma posição a frente ("proxNode = proxNode->prox").

E repetimos o processo até que o nó atual seja NULL (fim da fila).

```
#include <stdio.h>
#include <stdlib.h>

struct Node{
    int num;
    struct Node *prox;
};
typedef struct Node node;

int tam;

int menu(void);
void opcao(node *FILA, int op);
void inicia(node *FILA);
int vazia(node *FILA);
node *aloca();
void insere(node *FILA);
node *retira(node *FILA);
void exhibe(node *FILA);
void libera(node *FILA);
```

```
int main(void)
{
    node *FILA = (node *)
malloc(sizeof(node));

    if(!FILA){
        printf("Sem memoria disponivel!\n");
        exit(1);
    }else{
        inicia(FILA);
        int opt;

        do{
            opt=menu();
            opcao(FILA,opt);
        }while(opt);

        free(FILA);
        return 0;
    }
}
```



```
int menu(void)
{
    int opt;

    printf("Escolha a opcao\n");
    printf("0. Sair\n");
    printf("1. Zerar fila\n");
    printf("2. Exibir fila\n");
    printf("3. Adicionar Elemento na Fila\n");
    printf("4. Retirar Elemento da Fila\n");
    printf("Opcao: "); scanf("%d", &opt);

    return opt;
}
```

```
void opcao(node *FILA, int op)
{
    node *tmp;
    switch(op){
        case 0:    libera(FILA);
                   break;

        case 1:    libera(FILA);
                   inicia(FILA);
                   break;

        case 2:    exibe(FILA);
                   break;


        case 3:    insere(FILA);
                   break;

        case 4:    tmp= retira(FILA);
                   if(tmp != NULL){
                       printf("Retirado: %3d\n\n", tmp->num);
                       libera(tmp);
                   }
                   break;

        default:

            printf("Comando invalido\n\n");


    }
}
```



```
void inicia(node *FILA)
{
    FILA->prox = NULL;
    tam=0;
}

int vazia(node *FILA)
{
    if(FILA->prox == NULL)
        return 1;
    else
        return 0;
}
```

```
node *aloca()
{
    node *novo=(node *)
    malloc(sizeof(node));
    if(!novo){
        printf("Sem memoria disponivel!\n");
        exit(1);
    }else{
        printf("Novo elemento: ");
        scanf("%d", &novo->num);
        return novo;
    }
}
```



```
void insere(node *FILA)
{
    node *novo=aloca();
    novo->prox = NULL;

    if(vazia(FILA))
        FILA->prox=novo;
    else{
        node *tmp = FILA->prox;

        while(tmp->prox != NULL)
            tmp = tmp->prox;

        tmp->prox = novo;
    }
    tam++;
}
```

```
node *retira(node *FILA)
{
    if(FILA->prox == NULL){
        printf("Fila ja esta vazia\n");
        return NULL;
    }else{
        node *tmp = FILA->prox;
        FILA->prox = tmp->prox;
        tam--;
        return tmp;
    }
}
```

```
void exhibe(node *FILA)
{
    if(vazia(FILA)){
        printf("Fila vazia!\n\n");
        return ;
    }
    node *tmp;
    tmp = FILA->prox;
    printf("Fila :");
    while( tmp != NULL){
        printf("%5d", tmp->num);
        tmp = tmp->prox;
    }
    printf("\n    ");
    int count;
    for(count=0 ; count < tam ; count++)
        printf(" ^ ");
    printf("\nOrdem:");
    for(count=0 ; count < tam ; count++)
        printf("%5d", count+1);
    printf("\n\n");
}
```

```
void libera(node *FILA)
{
    if(!vazia(FILA)){
        node *proxNode,
            *atual;

        atual = FILA->prox;
        while(atual != NULL){
            proxNode = atual->prox;
            free(atual);
            atual = proxNode;
        }
    }
}
```