

PESQUISA
(Matriz, Fila e Lista)

- **MATRIZ:**

Matriz é uma estrutura de dados do tipo vetor com duas ou mais dimensões. Os itens de uma matriz tem que ser todos do mesmo tipo de dado. Na prática, as matrizes formam tabelas na memória.

Exemplo de declaração de matriz com 2 dimensões usando linguagem C:

```
float Media[5][2];
```

Nesse exemplo:

O valor 5 representa a quantidade de linhas.

O valor 2 representa a quantidade de colunas.

Dizemos que esta matriz é do tipo 5 X 2 e como temos 5 linhas com 2 posições de armazenamento em cada linha, temos capacidade para armazenar até 10 elementos do tipo float. É como se cada uma das 5 posições de um vetor tivesse mais 2 posições dentro delas.

Será necessário utilizar um índice para cada dimensão da matriz, logo uma matriz bidimensional terá 2 índices, um para posicionar a linha, outro para a coluna.

Assim, como no vetor, o índice da primeira posição é zero.

Como atribuir valores a uma matriz?

Suponha a matriz como declarada:

```
float Media[5][2];
```

Para atribuir um valor precisamos identificar a posição usando os índices:

```
Media [0][0] = 5; //Atribui o valor 5 na primeira linha e primeira coluna.
```

```
Media [1][0] = 7; // Atribui o valor 7 na segunda linha, primeira coluna.
```

Como preencher e/ou mostrar uma matriz?

Para fazer o preenchimento de uma matriz, devemos percorrer todos os seus elementos e atribuir-lhes um valor.

Isto pode ser feito tanto gerando valores para cada elemento da matriz, como recebendo os valores pelo teclado.

Um método interessante para percorrer uma matriz é usar duas estruturas de repetição for e duas variáveis inteiras (uma variável "i" e uma "j" como no exemplo a seguir), uma para a linha e a outra para a coluna.

Exemplo:

Suponha uma matriz de 3 linhas por 3 colunas do tipo inteiro. Para percorrer a matriz recebendo seus valores, podemos fazer:

```
int matriz[3][3];
for ( int i=0; i<3; i++ ){
    for (int j=0; j<3; j++ ){
        printf("\nElemento [%d] [%d]: ", i, j);
        scanf ("%d", &matriz[ i ][ j ]);
    }
}
```

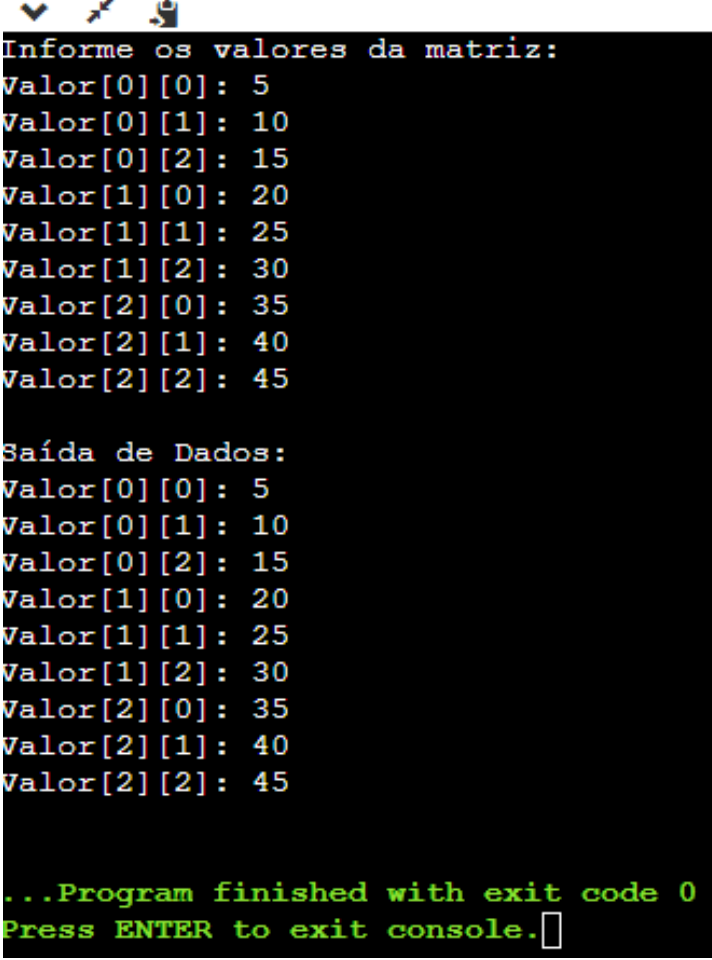
Note que no exemplo acima a variável “i” representa a linha e a “j” a coluna da matriz. Já para mostrar os elementos de uma matriz, o processo é parecido, mas usa-se o “printf()” (comando de saída) ao invés do “scanf()” (comando de entrada).

Exemplo:

```
for ( int i=0; i<3; i++ ){
    for (int j=0; j<3; j++ ){
        printf ("%d", matriz[ i ][ j ]);
    }
}
```

Exemplo de código no Online GDB:

```
main.c
1  #include<stdio.h>
2  //Código Exemplo - Matriz
3  //Pesquisa - Programação Estruturada
4  //Alunos: Sthefanny, Pedro e Luiza
5
6  int main()
7  {
8      int matriz[3][3],i, j;
9      printf ("Informe os valores da matriz:\n");
10
11     for (int i=0; i<3; i++ ){
12         for (int j=0; j<3; j++ ){
13             printf ("Valor[%d][%d]: ", i, j);
14             scanf ("%d",&matriz[i][j]);
15         }
16     }
17     printf("\nSaída de Dados:\n");
18     for(int i=0;i<3;i++){
19         for(int j=0;j<3;j++){
20             printf("Valor[%d][%d]: %d\n",i,j,matriz[i][j]);
21         }
22     }
23     return(0);
24 }
25
```



```
Informe os valores da matriz:
Valor[0][0]: 5
Valor[0][1]: 10
Valor[0][2]: 15
Valor[1][0]: 20
Valor[1][1]: 25
Valor[1][2]: 30
Valor[2][0]: 35
Valor[2][1]: 40
Valor[2][2]: 45

Saída de Dados:
Valor[0][0]: 5
Valor[0][1]: 10
Valor[0][2]: 15
Valor[1][0]: 20
Valor[1][1]: 25
Valor[1][2]: 30
Valor[2][0]: 35
Valor[2][1]: 40
Valor[2][2]: 45

...Program finished with exit code 0
Press ENTER to exit console.
```

Nesse exemplo, utilizamos um par de estruturas de repetição “for” para primeiramente fazer a leitura dos valores da matriz. Fixamos a linha, ou seja, o índice *i* enquanto o índice da coluna *j* varia até preencher todos os elementos da linha.

Depois que uma linha é preenchida, o índice da linha é incrementado e recomeça o preenchimento da coluna por coluna com o loop “for” interno controlado pelo índice *j*. A saída de dados usa o mesmo princípio para percorrer a matriz e exibir os dados.

Outro exemplo:

```

main.c
1  #include <stdio.h>
2  //Alunos: Sthefanny, Luiza, Pedro
3  int main()
4  {   int matriz[3][3];
5      printf("Informe 9 valores para multiplicacao: \n");
6      for(int i=0;i<3;i++){
7          for(int j=0;j<3;j++){
8              scanf("%d",&matriz[i][j]);
9          }
10     }
11     printf("Matriz original: \n");
12     for(int i=0;i<3;i++){
13         for(int j=0;j<3;j++){
14             printf("[%d]",matriz[i][j]);
15         }
16         printf("\n");
17     }
18     for(int i=0;i<3;i++){
19         for(int j=0;j<3;j++){
20             matriz[i][j]=matriz[i][j]*2;
21         }
22     }
23     printf("Matriz multiplicada: \n");
24     for(int i=0;i<3;i++){
25         for(int j=0;j<3;j++){
26             printf("[%d]",matriz[i][j]);
27         }
28         printf("\n");
29     }
30     return 0;
31 }

```

```

v ↗ 🐞
Informe 9 valores para multiplicacao:
1
2
3
4
5
6
7
8
9
Matriz original:
[1] [2] [3]
[4] [5] [6]
[7] [8] [9]
Matriz multiplicada:
[2] [4] [6]
[8] [10] [12]
[14] [16] [18]

...Program finished with exit code 0
Press ENTER to exit console.

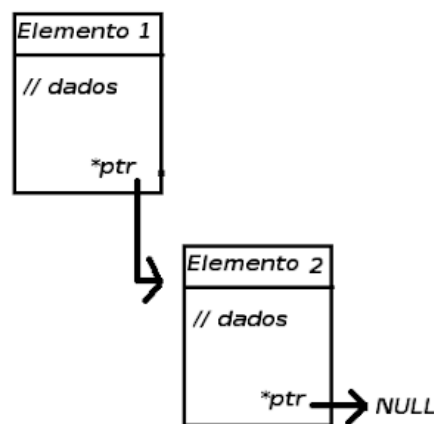
```

Nesse exemplo, é retornada uma matriz com o dobro dos valores inseridos pelo usuário. O princípio e o funcionamento é o mesmo do exemplo anterior.

- **LISTA:**

Em C, uma lista (também conhecida como lista encadeada ou linked list) é uma estrutura de dados dinâmica que permite armazenar e manipular elementos de forma flexível. Diferente de um array, que é uma estrutura de tamanho fixo, uma lista pode crescer ou diminuir dinamicamente à medida que elementos são adicionados ou removidos.

Uma lista encadeada é composta por nós, onde cada nó armazena um valor e um ponteiro para o próximo nó da lista. O último nó da lista tem seu ponteiro apontando para NULL, indicando o fim da lista.



Lembre-se que o ponteiro é para o tipo do nó, então podemos fazer com que o ponteiro do primeiro nó aponte para o segundo nó. E como o nó foi alocado de maneira dinâmica, ele também é representado por um endereço de memória, que será usado pelo ponteiro do elemento 1.

No nó 1, fazemos: **ptr = Elemento2;**

Porém, o ponteiro do segundo nó ainda aponta para outro endereço de memória (lixo). Para podermos identificar que esse elemento é o último da lista, vamos fazer com que o ponteiro do nó 2 aponte para NULL. Não é obrigatório apontar para NULL, mas é recomendado por questões de segurança.

Para contextualizar, vale ressaltar que os ponteiros ou apontadores são variáveis que armazenam o endereço de memória de outras variáveis. Dizemos que um ponteiro “aponta” para uma variável quando contém o endereço da mesma. Os ponteiros podem apontar para qualquer tipo de variável, tendo assim, ponteiros para int, float, double, etc.

Ponteiros são muito úteis quando uma variável tem que ser acessada em diferentes partes de um programa. Neste caso, o código pode ter vários ponteiros espalhados por diversas partes do programa, “apontando” para a variável que contém o dado desejado. Caso este dado seja alterado, não há problema algum, pois todas as partes do programa tem um ponteiro que aponta para o endereço onde reside o dado atualizado.

Em suma, uma lista encadeada é uma sequência de structs, que são os nós da lista, ligados entre si através de ponteiros. Esta sequência pode ser acessada através de um ponteiro para o primeiro nó, que é a cabeça da lista. Cada nó contém um ponteiro que aponta para a

struct que é a sua sucessora na lista. Basicamente, elas são implementadas através de estruturas (associadas aos nós) armazenadas na memória dinâmica e a estrutura que implementa um nó de uma lista ligada deve incluir, além do conteúdo da informação do nó, um ponteiro para o próximo nó.

Ademais, para tirar um nó da lista, por exemplo, vamos usar dois ponteiros que vão percorrer a lista até acharmos o elemento que vamos retirar. Fazemos um deles apontar para esse elemento que vai sair da lista, e o outro ponteiro aponta para o elemento anterior. Há inúmeras possibilidades na utilização das listas, como inserir nós ao final da lista, conectar nós no começo da lista, inseri-los no meio da lista ou excluir algum nó (seja ele o primeiro/cabeça, o último ou estando no meio da lista).

Lista encadeada com cabeça:

Podemos implementar uma lista encadeada de duas maneiras: com cabeça e sem cabeça. Onde a cabeça de uma lista seria um nó, que foi declarado explicitamente no início do programa.

Não usaremos, na verdade, o conteúdo deste nó(também chamado de célula ou cabeça), pois ele não será relevante, já que serve para sinalizar o início da lista para sabermos que ela inicia ali naquele endereço fixo de memória.

Fazemos isso na primeira linha de nossa main():

```
node *LISTA = (node *) malloc(sizeof(node));
```

Esse 'node' é um tipo que definimos (typedef struct), uma estrutura que armazena um número inteiro e um ponteiro para o tipo 'node' (nó, em inglês). A lista, de fato, será colocada no ponteiro deste nó inicial (cabeça), que está em: **LISTA->prox**.

Como acabamos de criar a lista, esse ponteiro deverá apontar para NULL, para sinalizar que a lista está vazia. Vamos fazer isso através do método 'inicia()', que simplesmente faz:

```
LISTA->prox = NULL;
```

Criamos um método para isto, pois este simples ato faz nossa lista ser zerada, sendo possível ser preenchida de novo.

Em seguida, invocamos o método menu(), que irá mostrar ao usuário uma série de opções e irá armazenar a opção escolhida, que será enviada ao método 'opcao()', que irá tratar a opção escolhida através de um teste condicional SWITCH.

Como inserir um nó no início da lista?

Inserir nós no início da lista é bem simples de se fazer, pois temos um head node, ou seja, um nó cabeça, que nos indica onde está o início de nossa lista: **LISTA->prox**.

Primeiro precisamos criar o nosso novo nó, o 'novo'.

Agora devemos fazer dois passos: primeiro fazemos LISTA->prox apontar para nosso novo nó, depois fazemos nosso nó apontar para o próximo elemento da lista. Porém, aqui temos um problema.

Antes o 'LISTA->prox' apontava para um local de memória que tinha o próximo nó.

Mas quando apontamos ele para o nó 'novo', perdemos essa referência. Para resolver isso, basta criar um ponteiro para armazenar o local que está 'LISTA->prox' antes de inserirmos o novo nó.

Vamos chamar de 'oldHead', pois era a cabeça antiga da lista:

```
node *oldHead = LISTA->prox;
```

Então, agora apontamos a cabeça da lista para nosso novo nó:

```
LISTA->prox = novo;
```

E apontamos nosso novo nó para o nó antigo:

```
novo->prox = oldHead;
```

E pronto, inserimos uma estrutura no início da lista.

Como inserir um nó ao final da lista?

Antes de mais nada, devemos alocar memória para este nó e preencher o número que iremos armazenar nele, o que é facilmente feito através da função malloc():

```
node *novo=(node *) malloc(sizeof(node));
```

Lembrando sempre de checar se a memória foi alocada (caso não, devemos dar um exit() no programa, informando o erro).

Agora devemos procurar o final da lista, mas antes de mais nada, devemos checar se ela está vazia através do método vazia(), que simplesmente checa o elemento: `LISTA->prox`.

Se este apontar para NULL, a lista está vazia, afinal, esse elemento é a cabeça da lista.

Caso seja realmente vazia, vamos fazer com que a cabeça aponte para este novo nó que criamos, o 'novo':

```
LISTA->prox = novo;
```

Agora, este nó será o último elemento da lista, então ele DEVE apontar para NULL:

```
novo->prox = NULL;
```

Caso não seja uma lista vazia devemos procurar o final da lista, que é um nó que aponta para NULL. Para isso, vamos declarar um ponteiro de nó:

```
node *tmp;
```

A ideia é fazer esse ponteiro apontar para todos os elementos da lista e checar se ele é o último. Algo como "Hey, você é o último da lista? Não? Próximo! E você, é? Não?

Próximo!..."

Então posicionamos ele apontando para o início da lista, nossa cabeça: `tmp = LISTA->prox`.

Agora vamos percorrer todos os nós, e só vamos parar quando o nó apontar para NULL, ou seja, só vamos parar quando a seguinte comparação retornar valor lógico TRUE:

```
tmp->prox == NULL;
```

Se retornar falso, devemos avançar a lista, e isso é feito da seguinte maneira:

```
tmp = tmp->prox;
```

Isso é feito com um laço while, que sai varrendo todos os nós e só para quando encontra o último:

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

```
    tmp = tmp->prox;
```

Ou seja, nosso ponteiro agora aponta para o próximo elemento da lista. E quando acharmos o ponteiro que aponta para NULL colocamos aquele nó criado anteriormente chamado de 'novo'. Ou seja, em vez do último nó apontar para NULL, vai apontar para novo:

```
tmp->prox = novo;
```

E como ao declarar o 'novo' fizemos ele apontar para NULL, identificamos ele como o último elemento da lista.

Como exibir os elementos de uma lista?

De nada adianta alocar os elementos no início ou fim da lista se não pudermos ver essa lista crescendo diante de nossos olhos. Por isso, vamos criar uma função que será responsável por exibir todos os elementos da lista, do início ao fim.

Início: nó cabeça (LISTA->prox);

Fim: nó que aponta para NULL.

Mais uma vez precisamos criar um ponteiro que vai apontar para cada um dos elementos da lista, e depois imprimir o número armazenado na estrutura.

Vamos chamar esse nó temporário de 'tmp' e ele deve apontar inicialmente para a cabeça da lista:

```
node *tmp = LISTA->prox;
```

Agora vamos imprimir o primeiro elemento (caso exista, senão existir, dizemos que a lista está vazia).

Estamos em um nó, então imprimimos ele e avançamos.

Para imprimir, damos um print no inteiro: `tmp->num`.

E para avançar na lista: `tmp = tmp->prox`.

Devemos parar o avanço quando chegarmos ao fim da lista, ou seja, quando nosso ponteiro temporário apontar para NULL. Logo, isso é feito dentro de um laço while:

```
while( tmp != NULL){  
    printf("%d", tmp->num);  
    tmp = tmp->prox;  
}
```

Dessa forma, os membros da lista serão exibidos.

O que muda para retirar nós de uma lista?

Uma diferença importante é que quando inserimos nós nas funções, elas eram do tipo void, já que tínhamos que colocar um nó ali na lista.

Entretanto, quando retiramos, é importante saber que elementos estão sendo retirados. A explicação disso é o bom senso: quando tiramos um elemento de uma lista, é para trabalhar em cima dele, ver seu conteúdo, passar ele para outra função ou extrair algum dado.

Portanto, nossas funções de retirar elementos irão retornar o nó e serão do tipo:

```
node *retiraInicio(node *LISTA);  
node *retiraFim(node *LISTA);
```

Como retirar um nó do início da lista?

A primeira coisa que devemos fazer para retirar um nó do início é saber se esse nó realmente existe, ou seja, se a lista não está vazia.

Essa simples verificação é feita usando um teste condicional IF:

```
if(LISTA->prox == NULL){  
}
```

E caso dê verdadeira, é porque ela está vazia e não há nada para ser retirado.

Como devemos retornar algo, iremos fazer:

```
return NULL;
```

Isso é necessário para mostrar que nada foi retirado da lista.

Agora iremos fazer a retirada do primeiro elemento da lista através do uso de um ponteiro para o tipo node, o *tmp. Vamos fazer ele apontar para primeiro elemento da lista:

```
tmp = LISTA->prox;
```

Então o primeiro elemento da lista está apontado por 'tmp'. Agora vem a lógica da exclusão de um elemento.

Antes, LISTA->prox aponta para o primeiro elemento, então para excluir esse primeiro elemento, basta fazer com que esse ponteiro aponte para o segundo elemento.

E onde está o segundo elemento? Está em: tmp->prox;

Logo, para excluir o primeiro nó, fazemos:

```
LISTA->prox = tmp->prox;
```

E em seguida retornamos 'tmp', que ainda aponta para aquele nó que foi excluído.

Vale ressaltar que, embora tenhamos tirado ele da lista (fazendo com que ninguém aponte para ele), a memória foi alocada para esse nó, então ele ainda existe e está sendo apontado por 'tmp'.

Como excluir um elemento ao final da lista?

Assim como fizemos para tirar uma struct do início, para tirar do fim primeiro temos que checar se a lista está vazia (mesmo processo do início do exemplo anterior).

Se notar bem, para retirar um elemento do início da lista, usamos dois ponteiros:

LISTA->prox e tmp. Para fazer isso no final da lista, também vamos usar dois ponteiros.

Porém, não vamos poder usar o LISTA->prox pois este está apontado para o início da lista.

Para resolver este problema, iremos usar dois ponteiros neste algoritmo: 'ultimo' e 'penultimo' e como os próprios nomes podem sugerir o 'ultimo' aponta para último elemento da lista e o 'penultimo' para o penúltimo elemento da lista.

Nosso objetivo é fazer o ponteiro 'ultimo' chegar ao fim da lista e o 'penultimo' a penúltima posição. Chegando lá, simplesmente fazemos o ponteiro 'penultimo' apontar para NULL, caracterizando este elemento como o último da lista, e retornamos o ponteiro 'ultimo', que contém o elemento retirado.

Lembre-se que o último elemento da lista é aquele que aponta para NULL. Então vamos fazer o ponteiro 'ultimo' ir percorrendo a lista até chegar num ponto onde 'ultimo->prox' aponta para NULL, e aí chegamos ao fim. Porém, antes do 'ultimo' avançar, devemos fazer com que o ponteiro 'penultimo' receba o valor de 'ultimo'. Colocando isso dentro de um laço while, obtemos o que queríamos da seguinte maneira:

```
while(ultimo->prox != NULL){  
    penultimo = ultimo;  
    ultimo = ultimo->prox;  
}
```

Agora vamos excluir o último da lista, fazendo o penúltimo elemento apontar para NULL, e retornar ele:

```
penultimo->prox = NULL;  
return ultimo;
```

Vale ressaltar o quanto essas ideias são extremamente poderosas! Note que estamos tratando de estruturas, e essas structs podem ser qualquer coisa. Podem guardar dados de funcionários de uma empresa, informações de aluno de uma escola ou cada uma pode ser a bula de um remédio de uma farmácia, no sistema.

Se tivermos 10 funcionários, as listas alocam espaço somente para estes 10.

Se forem 100 alunos, teremos espaço para exatos 100 alunos. E se a farmácia tiver mil tipos de remédios, só vamos usar mil espaços de struct. Nem mais, nem menos.

Precisa guardar mais uma informação? Somente mais uma struct será usada.

- **FILA:**

As filas (queue, em inglês) são um tipo de estrutura dinâmica de dados onde os elementos (ou nós) estão arranjados em lista que obedece as seguintes regras:

- Ao inserir um nó, ele vai para a última posição da estrutura
- Ao retirar um nó, é tirado o primeiro elemento da estrutura

Este tipo de estrutura de dados é dita ser FIFO (First in, first out), ou seja, o primeiro elemento a entrar na estrutura é o primeiro a sair.

O nome fila, por si só, já é auto-explicativo.

Imagine uma fila de banco. A primeira pessoa que chegou na fila, é a que vai ser atendida primeiro. E se chegar mais alguém? Ela vai demorar mais pra ser atendida. E se chegar uma outra? Ela será a última a ser atendida, pois quem está na sua frente é atendido antes. Em suma, sempre que inserimos elementos nessa fila, inserimos ao final. E sempre que retiramos, estamos tirando o primeiro elemento da fila (o mais antigo), pois o que está na frente é o que vai sair antes.

Resumindo: inserimos ao fim, e retiramos do começo. Diferente das pilhas em C (stack), onde colocamos no fim e retiramos do fim.

Agora vamos entender a lógica para programar uma fila em C, do zero.

São exemplos de uso de fila em um sistema:

Controle de documentos para impressão;

Troca de mensagem entre computadores numa rede; etc.

A implementação de filas pode ser realizada através de vetor (alocação do espaço de memória para os elementos é contígua) ou através de listas encadeadas.

Operações com Fila:

Todas as operações em uma fila podem ser imaginadas como as que ocorre numa fila de pessoas num banco, exceto que o elementos não se movem na fila, conforme o primeiro elemento é retirado. Isto seria muito custoso para o computador. O que se faz na realidade é indicar quem é o primeiro.

Criação da fila (informar a capacidade no caso de implementação sequencial - vetor);

Enfileirar (enqueue) - o elemento é o parâmetro nesta operação;

Desenfileirar (dequeue);

Mostrar a fila (todos os elementos);

Verificar se a fila está vazia (isEmpty);

Verificar se a fila está cheia (isFull - implementação sequencial - vetor).

Exemplo de código:

```

1  #include <stdio.h>
2  //Sthefanny, Pedro e Luiza
3
4  struct Fila {
5
6      int capacidade;
7      float *dados;
8      int primeiro;
9      int ultimo;
10     int nItens;
11
12 };
13
14 void criarFila( struct Fila *f, int c ) {
15
16     f->capacidade = c;
17     f->dados = (float*) malloc (f->capacidade * sizeof(float));
18     f->primeiro = 0;
19     f->ultimo = -1;
20     f->nItens = 0;
21
22 }
23
24 void inserir(struct Fila *f, int v) {
25
26     if(f->ultimo == f->capacidade-1)
27         f->ultimo = -1;
28
29     f->ultimo++;
30     f->dados[f->ultimo] = v; // incrementa ultimo e insere
31     f->nItens++; // mais um item inserido
32
33 }
34
35 int remover( struct Fila *f ) { // pega o item do começo da fila
36
37     int temp = f->dados[f->primeiro++]; // pega o valor e incrementa o primeiro
38
39     if(f->primeiro == f->capacidade)
40         f->primeiro = 0;
41
42     f->nItens--; // um item retirado
43     return temp;
44
45 }
46
47 int estaVazia( struct Fila *f ) { // retorna verdadeiro se a fila está vazia
48
49     return (f->nItens==0);
50
51 }
52
53 int estaCheia( struct Fila *f ) { // retorna verdadeiro se a fila está cheia
54
55     return (f->nItens == f->capacidade);
56
57 }
58 void mostrarFila(struct Fila *f){
59
60     int cont, i;
61
62     for ( cont=0, i= f->primeiro; cont < f->nItens; cont++){
63
64         printf("%.2f\t",f->dados[i++]);
65
66         if (i == f->capacidade)
67             i=0;
68
69     }
70     printf("\n\n");
71
72 }

```

```

73
74 void main () {
75
76     int opcao, capa;
77     float valor;
78     struct Fila umaFila;
79
80     // cria a fila
81     printf("\nCapacidade da fila? ");
82     scanf("%d",&capa);
83     criarFila(&umaFila, capa);
84
85     // apresenta menu
86     while( 1 ){
87
88         printf("\n1 - Inserir elemento\n2 - Remover elemento\n3 - Mostrar Fila\n0 - Sair\n");
89         scanf("%d", &opcao);
90
91         switch(opcao){
92
93             case 0: exit(0);
94
95             case 1: // insere elemento
96                 if (estaCheia(&umaFila)){
97                     printf("\nFila Cheia!!!\n\n");
98                 }
99                 else {
100
101                     printf("\nValor do elemento a ser inserido? ");
102                     scanf("%f", &valor);
103                     inserir(&umaFila,valor);
104                 }
105                 break;
106
107             case 2: // remove elemento
108                 if (estaVazia(&umaFila)){
109                     printf("\nFila vazia!!!\n\n");
110                 }
111                 else {
112                     valor = remover(&umaFila);
113                     printf("\n%1f removido com sucesso\n\n", valor) ;
114                 }
115                 break;
116
117             case 3: // mostrar fila
118                 if (estaVazia(&umaFila)){
119                     printf("\nFila vazia!!!\n\n");
120                 }
121                 else {
122                     printf("\nConteudo da fila => ");
123                     mostrarFila(&umaFila);
124                 }
125                 break;
126
127             default:
128                 printf("\nOpcao Invalida\n\n");
129
130
131
132
133
134
135
136
137
138
139
140
141

```