

Estrutura de Dados para Armazenamento de Informações

Pilhas e Filas em Linguagem C

Recordar é Viver — Listas Encadeadas em C

O que é uma Lista Encadeada?

→ Uma lista encadeada é uma sequência de **nós** que estão **ligados por ponteiros**.

Cada **nó** contém:

- Um **valor** (ex: int valor;)
- Um **ponteiro para o próximo nó** (struct No *prox;)

```
struct No {  
    int valor;           // Dado do nó  
    struct No *prox;    // Ponteiro para o próximo nó  
};
```

✓ Como funciona visualmente?

🔗 Exemplo de uma Lista Encadeada com 3 nós:

```
[10 | * ] → [20 | * ] → [30 | NULL]
```

O ponteiro do último nó aponta para NULL, indicando o **fim da lista**.

Recordar é Viver (Principais Operações que Vimos)

```
// Função para inserir um novo nó no início da lista
struct No* inserirNoInicio(struct No *head, int novoValor) {
    struct No *novoNo = (struct No*)malloc(sizeof(struct No)); // Aloca memória para o novo nó
    novoNo->valor = novoValor; // Atribui o valor desejado ao novo nó
    novoNo->prox = head; // Faz o ponteiro 'prox' do novo nó apontar para o antigo primeiro nó (head)
    return novoNo; // Retorna o novo nó como sendo o novo início (head) da lista
}
```

1. Inserir no Início

- Cria um novo nó.
- Faz o novoNo->prox apontar para o antigo início.
- Atualiza o head (início da lista).

 Útil quando queremos adicionar rapidamente no começo.

struct No* inserirNoInicio(struct No *head, int novoValor)

- Recebe o ponteiro head (início da lista) e o valor a ser inserido.

struct No *novoNo = (struct No*)malloc(sizeof(struct No));

- Aloca memória para um novo nó.

novoNo->valor = novoValor;

- Atribui o valor ao novo nó.

novoNo->prox = head;

- Faz o ponteiro do novo nó apontar para o nó que era o primeiro (head).

return novoNo;

- Retorna o novo nó, que agora passa a ser o novo início da lista.

Recordar é Viver

2. Inserir no Fim

- Cria um novo nó com prox = NULL.
- Percorre a lista até o último nó.
- Faz o último prox apontar para o novo nó.

 Útil para manter a ordem de chegada dos dados.

```
// Função para inserir no fim da lista
struct No* inserirNoFim(struct No *head, int novoValor) {
    // 1. Alocar memória para o novo nó
    struct No *novoNo = (struct No*)malloc(sizeof(struct No));
    // 2. Atribuir o valor ao novo nó
    novoNo->valor = novoValor;
    // 3. O novo nó será o último, portanto 'prox' será NULL
    novoNo->prox = NULL;
    // 4. Se a lista estiver vazia (head == NULL), o novo nó será o primeiro nó
    if (head == NULL) {
        return novoNo;
    }
    // 5. Percorrer a lista até o último nó
    struct No *atual = head;
    while (atual->prox != NULL) {
        atual = atual->prox;
    }
    // 6. Fazer o último nó apontar para o novo nó
    atual->prox = novoNo;
    // 7. Retorna o início da lista (head não muda)
    return head;
}
```

```
struct No* inserirNoFim(struct No *head, int novoValor)
```

- Recebe o início da lista (head) e o valor a ser inserido.

```
struct No *novoNo = (struct No*)malloc(sizeof(struct No));
```

```
novoNo->valor = novoValor;
```

```
novoNo->prox = NULL;
```

- Cria o novo nó com o valor e faz prox = NULL (ele será o último nó).

```
if (head == NULL) {
```

```
    return novoNo;
```

```
}
```

- Se a lista estiver vazia, o novo nó será o primeiro.

```
struct No *atual = head;
```

```
while (atual->prox != NULL) {
```

```
    atual = atual->prox;
```

```
}
```

- Percorre a lista até encontrar o último nó (prox == NULL).

```
atual->prox = novoNo;
```

- Faz o último nó apontar para o novo nó.

```
return head;
```

- Retorna o início da lista (head não mudou).

Recordar é Viver (Operação: Remover um Elemento)

3. Remover um Elemento

- Verifica se a lista está vazia.
 - Percorre até encontrar o valor desejado.
 - Remove o nó ajustando os ponteiros e liberando memória (free()).
-  Tratamos também a remoção do primeiro nó como caso especial.

Remover um nó que **contém um valor específico** da lista.

Exemplo:

Lista antes:

[10 | *] → [20 | *] → [30 | *] → [40 | NULL]

Remover o valor **30**:

[10 | *] → [20 | *] → [40 | NULL]

```

#include <stdio.h>
#include <stdlib.h>
// Estrutura do nó
struct No {
    int valor;
    struct No *prox;
};
// Função para remover um nó com valor específico
struct No* removerElemento(struct No *head, int valorRemover) {
    // 1. Verifica se a lista está vazia
    if (head == NULL) {
        printf("Lista vazia.\n");
        return NULL;
    }
    // 2. Verifica se o valor está no primeiro nó
    if (head->valor == valorRemover) {
        struct No *temp = head;           // Guarda o nó a ser removido
        head = head->prox;              // Atualiza o head para o próximo nó
        free(temp);                     // Libera o nó removido
        return head;                    // Retorna o novo head
    }
    // 3. Caso contrário, percorre a lista procurando o valor
    struct No *anterior = head;
    struct No *atual = head->prox;
    while (atual != NULL && atual->valor != valorRemover) {
        anterior = atual;
        atual = atual->prox;
    }
    // 4. Se não encontrou o valor
    if (atual == NULL) {
        printf("Valor %d não encontrado na lista.\n", valorRemover);
        return head;
    }
    // 5. Encontrou o valor: remove o nó
    anterior->prox = atual->prox; // Pula o nó a ser removido
    free(atual);                 // Libera a memória do nó
    return head;                  // Retorna o head da lista
}

```

if (head->valor == valorRemover) {

- ◆ Verifica se o **primeiro nó** já tem o valor a ser removido.

struct No *temp = head; // Guarda o nó a ser removido

head = head->prox; // Atualiza head para o próximo nó

free(temp); // Libera a memória do nó removido

return head; // Retorna o novo início da lista

- ◆ Remove o primeiro nó:
- Guarda o endereço do nó a ser removido em temp.
- Faz head apontar para o próximo nó.
- Libera a memória do nó removido com free.
- Retorna o novo início da lista.

struct No *anterior = head;

struct No *atual = head->prox;

- ◆ Inicializa dois ponteiros:
- anterior: começa apontando para o primeiro nó.
- atual: começa apontando para o segundo nó.

while (atual != NULL && atual->valor != valorRemover) {

- ◆ Percorre a lista:
- Enquanto não encontrar o valor e não chegar no fim (NULL).
- Avança: anterior recebe atual e atual recebe atual->prox.

}

```

#include <stdio.h>
#include <stdlib.h>
// Estrutura do nó
struct No {
    int valor;
    struct No *prox;
};
// Função para remover um nó com valor específico
struct No* removerElemento(struct No *head, int valorRemover) {
    // 1. Verifica se a lista está vazia
    if (head == NULL) {
        printf("Lista vazia.\n");
        return NULL;
    }
    // 2. Verifica se o valor está no primeiro nó
    if (head->valor == valorRemover) {
        struct No *temp = head;           // Guarda o nó a ser removido
        head = head->prox;              // Atualiza o head para o próximo nó
        free(temp);                     // Libera o nó removido
        return head;                    // Retorna o novo head
    }
    // 3. Caso contrário, percorre a lista procurando o valor
    struct No *anterior = head;
    struct No *atual = head->prox;
    while (atual != NULL && atual->valor != valorRemover) {
        anterior = atual;
        atual = atual->prox;
    }
    // 4. Se não encontrou o valor
    if (atual == NULL) {
        printf("Valor %d não encontrado na lista.\n", valorRemover);
        return head;
    }
    // 5. Encontrou o valor: remove o nó
    anterior->prox = atual->prox; // Pula o nó a ser removido
    free(atual);                 // Libera a memória do nó
    return head;                 // Retorna o head da lista
}

```

```

if (atual == NULL) {
    printf("Valor %d não encontrado na lista.\n", valorRemover);
    return head;
}

◆ Se chegou no fim da lista e não encontrou o valor:
• Imprime mensagem de "não encontrado".
• Retorna a lista sem alterações.

anterior->prox = atual->prox; // Pula o nó a ser removido
free(atual);                // Libera a memória do nó

◆ Se encontrou o valor:
• Faz anterior->prox pular o nó a ser removido (atual).
• Libera a memória do nó removido com free(atual).

return head; // Retorna o início da lista (head não muda)
}

◆ Retorna o início da lista, pois a cabeça da lista não foi
alterada.

```

4. Percorrer a Lista

- Usa um ponteiro auxiliar.
- Vai de nó em nó imprimindo os valores até chegar em NULL.

 É a base de toda leitura ou varredura da lista.

```
// Função para percorrer e imprimir a lista
void imprimirLista(struct No *head) {
    struct No *atual = head; // Cria um ponteiro auxiliar que começa no head
    printf("Lista: ");
    while (atual != NULL) { // Enquanto o ponteiro não for NULL (fim da lista)
        printf("%d -> ", atual->valor); // Imprime o valor do nó atual
        atual = atual->prox; // Avança para o próximo nó
    }
    printf("NULL\n"); // Indica o fim da lista
}
```

```
struct No *atual = head;
```

- ◆ Cria um ponteiro atual que começa apontando para o primeiro nó (head).

```
printf("Lista: ");
```

- ◆ Imprime o título antes de percorrer.

```
while (atual != NULL) {
```

- ◆ Laço que percorre a lista até encontrar o fim (NULL).

```
printf("%d -> ", atual->valor);
```

- ◆ Imprime o valor armazenado no nó atual, seguido de uma seta "→".

```
atual = atual->prox;
```

```
printf("NULL\n");
```

- ◆ Avança para o próximo nó da lista. ◆ Após sair do laço (chegou no fim), imprime NULL para indicar o término.

5. Buscar um Elemento

- Percorre a lista comparando valores.
- Se achar, retorna o ponteiro para o nó.
- Se não achar, retorna NULL.

 Pode ser feito com laço ou de forma recursiva.

```
// Função para buscar um valor na lista
struct No* buscarElemento(struct No *head, int valorBusca) {
    struct No *atual = head; // Ponteiro auxiliar para percorrer
    while (atual != NULL) { // Percorre até o fim da lista
        if (atual->valor == valorBusca) { // Se encontrou o valor
            return atual; // Retorna o endereço do nó encontrado
        }
        atual = atual->prox; // Avança para o próximo nó
    }
    return NULL; // Se não encontrou, retorna NULL
}
```

Agora que dominamos o conceito de **lista encadeada**, podemos **especializar** esse tipo de estrutura em dois comportamentos muito usados em computação:

Pilhas – Conceito Inicial

“Pense no seu navegador. Você vai navegando por páginas... e quando clica em ‘voltar’, ele te leva para a anterior. Como ele sabe qual foi a última página?”

A resposta está na pilha: o navegador **empilha as páginas visitadas**, e quando você clica em ‘voltar’, ele **desempilha** a última.

Portanto: Uma pilha é uma estrutura onde o **último elemento que entra é o primeiro que sai**. Em inglês, dizemos que ela segue a lógica **LIFO – Last In, First Out**.

Conceito de Ponteiro Duplo (Ponteiro de Ponteiro)

1. Começando com Variáveis Simples

```
int x = 10;
```

- x é uma variável do tipo int.
- O valor de x é 10.
- O **endereço de memória** de x pode ser acessado com &x.

2. Ponteiro Simples (*)

```
int *p = &x;
```

- p é um **ponteiro para int** → ou seja, ele guarda o **endereço de x**.
- *p → acessa o **conteúdo** do endereço para o qual p aponta (no caso, o valor de x) - (valor 10).

Nome	Valor guardado	O que aponta
x	10	—
p	&x (endereço de x)	x
*p	10 (valor de x)	—

Conceito de Ponteiro Duplo (Ponteiro de Ponteiro)

✓ 3. Ponteiro para Ponteiro (**)

```
int x = 10;
int *p = &x;
int **pp = &p;
```

pp guarda o endereço de p, ou seja, guarda o valor de p
*pp → acessa p.

**pp → acessa o conteúdo de p, ou seja, o valor de x.

```
#include <stdio.h>

int main() {
    int x = 10;
    int *p = &x;
    int **pp = &p;

    printf("x: %d\n", x);           // 10
    printf("*p: %d\n", *p);         // 10
    printf("**pp: %p\n", *pp);      // endereço de x
    printf("**pp: %d\n", **pp);     // 10

    return 0;
}
```

Expressão	Interpretação	Símbolo	Tipo	O que acessa
p	endereço de x	x	int	Valor direto
*p	valor de x (10)	p	int *	Endereço de x
pp	endereço de p	*p	int	Valor de x
*pp	valor de pp = p	pp	int **	Endereço de p
**pp	valor de *p = x = 10	*pp	int *	Valor de p (endereço de x)
		**pp	int	Valor de x

Por que usar Ponteiro para Ponteiro?

Porque em C, tudo é passado por valor.

! Problema:

Se você passa um ponteiro simples para uma função e tenta mudar onde ele aponta, a alteração não afeta o ponteiro original.

✓ Solução:

Passar o endereço do ponteiro → ou seja, um ponteiro duplo (**).

Exemplo Comparativo

✗ Sem ponteiro duplo (não altera de verdade):

```
void muda(int *p) {
    int a = 99;
    p = &a; // só altera localmente (não muda fora da função)
}
```

✓ Com ponteiro duplo:

```
void muda(int **p) {
    int a = 99;
    *p = &a; // agora muda o ponteiro original!
}
```

Pilhas e Filas

Já Abordamos:

- ✓ Vetores (estrutura linear, estática)
- ✓ Lista Encadeada (estrutura dinâmica, com ponteiros)
- ✓ Ponteiros e Ponteiros de Ponteiros

Agora vem a **especialização de comportamento**:

- ▲ **Pilha (Stack)** – LIFO: Last In, First Out
- ▼ **Fila (Queue)** – FIFO: First In, First Out

Pilha (Stack) – LIFO: Last In, First Out

– Conceito Inicial

Operação	O que faz	Função em C
push	Insere elemento no topo da pilha	void push(struct No **topo, int valor)
pop	Remove e retorna o valor do topo	int pop(struct No **topo)
peek ou top	Retorna o valor do topo sem remover	int peek(struct No *topo)

***topo permite **mudar o ponteiro do topo real** da pilha dentro das funções.*

```

#include <stdio.h>
#include <stdlib.h>
// Definição da estrutura do nó da pilha
struct No {
    int valor;
    struct No *prox;
};
// Função push: insere um novo elemento no topo da pilha
void push(struct No **topo, int valor) {
    struct No *novo = (struct No*)malloc(sizeof(struct No)); // Aloca novo nó
    novo->valor = valor; // Atribui o valor
    novo->prox = *topo; // Faz o novo nó apontar para o antigo topo
    *topo = novo; // Atualiza o topo da pilha
}
// Função pop: remove o topo da pilha e retorna seu valor
int pop(struct No **topo) {
    if (*topo == NULL) {
        printf("Pilha vazia!\n");
        return -1; // Valor que indica erro
    }
    struct No *temp = *topo; // Armazena o nó atual
    int valor = temp->valor; // Pega o valor antes de remover
    *topo = temp->prox; // Atualiza o topo para o próximo nó
    free(temp); // Libera a memória do nó antigo
    return valor; // Retorna o valor removido
}
// Função peek: retorna o valor do topo da pilha sem remover
int peek(struct No *topo) {
    if (topo == NULL) {
        printf("Pilha vazia!\n");
        return -1; // Valor sinalizador
    }
    return topo->valor; // Retorna o valor do topo
}
// Função para imprimir a pilha (visualização)
void imprimirPilha(struct No *topo) {
    printf("Topo da pilha\n");
    while (topo != NULL) {
        printf("%d\n", topo->valor);
        topo = topo->prox;
    }
    printf("NULL\n");
}

```

Implementando

```

int main() {
    struct No *pilha = NULL; // Pilha vazia

    // Inserções
    push(&pilha, 10);
    push(&pilha, 20);
    push(&pilha, 30);

    imprimirPilha(pilha); // Mostra a pilha

    // Consulta o topo sem remover
    printf("Valor no topo (peek): %d\n", peek(pilha));

    // Remove o topo
    int valorRemovido = pop(&pilha);
    printf("Valor removido (pop): %d\n", valorRemovido);

    // Consulta novamente após remoção
    printf("Novo topo (peek): %d\n", peek(pilha));

    imprimirPilha(pilha); // Mostra a pilha final

    return 0;
}

```

Exemplo 2: Navegador Web com Pilha

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
// Estrutura para representar uma página
struct Pagina {
    char url[100];
    struct Pagina *prox;
};
// Função para acessar uma nova página (push)
void acessarPagina(struct Pagina **topo, const char *endereco) {
    struct Pagina *nova = (struct Pagina *)malloc(sizeof(struct Pagina));
    strcpy(nova->url, endereco); // Copia o endereço da nova página
    nova->prox = *topo; // Aponta para a página anterior
    *topo = nova; // Atualiza o topo da pilha
}
// Função para voltar para a página anterior (pop)
void voltarPagina(struct Pagina **topo) {
    if (*topo == NULL) {
        printf("Nenhuma página para voltar.\n");
        return;
    }
    struct Pagina *temp = *topo;
    printf("Voltando da página: %s\n", temp->url);
    *topo = temp->prox; // Atualiza o topo
    free(temp); // Libera o nó removido
}
// Função para mostrar a página atual
void mostrarPaginaAtual(struct Pagina *topo) {
    if (topo == NULL) {
        printf("Nenhuma página aberta.\n");
    } else {
        printf("Página atual: %s\n", topo->url);
    }
}
// Função para mostrar o histórico (todos os nós da pilha)
void mostrarHistorico(struct Pagina *topo) {
    if (topo == NULL) {
        printf("Histórico vazio.\n");
        return;
    }
    printf("Histórico de páginas visitadas:\n");
    while (topo != NULL) {
        printf(" - %s\n", topo->url);
        topo = topo->prox;
    }
}
```

```
// Função principal com menu interativo
int main() {
    struct Pagina *pilha = NULL; // Pilha de páginas
    int opcao;
    char url[100];
    do {
        printf("\n==== MENU DO NAVEGADOR ====\n");
        printf("1. Acessar nova página\n");
        printf("2. Voltar página anterior\n");
        printf("3. Ver página atual\n");
        printf("4. Mostrar histórico\n");
        printf("0. Sair\n");
        printf("Escolha uma opção: ");
        scanf("%d", &opcao);
        getchar(); // Limpa o '\n' deixado pelo scanf
        switch (opcao) {
            case 1:
                printf("Digite a URL da nova página: ");
                fgets(url, sizeof(url), stdin);
                url[strcspn(url, "\n")] = '\0'; // Remove o \n do final
                acessarPagina(&pilha, url);
                break;
            case 2:
                voltarPagina(&pilha);
                break;
            case 3:
                mostrarPaginaAtual(pilha);
                break;
            case 4:
                mostrarHistorico(pilha);
                break;
            case 0:
                printf("Saindo do navegador...\n");
                break;
            default:
                printf("Opção inválida. Tente novamente.\n");
        }
    } while (opcao != 0);
    // Libera memória restante (pilha completa)
    while (pilha != NULL) {
        voltarPagina(&pilha);
    }
    return 0;
}
```

Filas – Conceito Inicial

- Uma **Fila** é uma estrutura de dados linear que segue a lógica **FIFO**
 - **First In, First Out**: o primeiro elemento inserido é o primeiro a ser removido.

Imagine uma **fila na cantina da faculdade**:

- Você chega e entra no **fim da fila** (inserção).
- Quem chegou primeiro, é **atendido primeiro** (remoção no início).
- Ninguém "corta" a fila — a ordem é respeitada!

Fila (Queue) – FIFO: First In, First Out

Operação

Inserção

Remoção

Consulta

Nome em inglês

enqueue

dequeue

peek ou front

O que faz

Adiciona no **fim** da fila

Remove do **início** da fila

Mostra o valor do **início** da fila
sem remover

```

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

// Definição da estrutura do nó da fila
struct No {
    int valor;
    struct No *prox;
};

// Função para enfileirar (enqueue): insere no fim da fila
void enqueue(struct No **inicio, struct No **fim, int valor) {
    struct No *novo = (struct No*)malloc(sizeof(struct No)); // Aloca novo nó
    novo->valor = valor; // Atribui valor ao novo nó
    novo->prox = NULL; // Novo nó será o último, então prox = NULL

    if (*fim == NULL) {
        // Se a fila está vazia, novo nó será o primeiro e o último
        *inicio = novo;
        *fim = novo;
    } else {
        (*fim)->prox = novo; // Conecta o atual último ao novo nó
        *fim = novo; // Atualiza o fim da fila
    }
}

// Função para desenfileirar (dequeue): remove do início da fila
int dequeue(struct No **inicio, struct No **fim) {
    if (*inicio == NULL) {
        printf("Fila vazia!\n");
        return -1;
    }

    struct No *temp = *inicio; // Guarda o nó a ser removido
    int valor = temp->valor; // Armazena o valor para retorno
    *inicio = temp->prox; // Avança o ponteiro do início

    if (*inicio == NULL) {
        // Se a fila ficou vazia, zera também o fim
        *fim = NULL;
    }

    free(temp); // Libera a memória do nó removido
    return valor;
}

```

Implementando

```

// Função para consultar o início da fila (peek)
int peek(struct No *inicio) {
    if (inicio == NULL) {
        printf("Fila vazia!\n");
        return -1;
    }
    return inicio->valor; // Retorna o valor do início sem remover
}

// Função para imprimir a fila do início ao fim
void imprimirFila(struct No *inicio) {
    printf("Início da fila\n");
    while (inicio != NULL) {
        printf(" %d\n", inicio->valor);
        inicio = inicio->prox;
    }
    printf("NULL (fim)\n");
}

// Função principal
int main() {
    struct No *inicio = NULL; // Ponteiro para o início da fila
    struct No *fim = NULL; // Ponteiro para o fim da fila
    // Enfileira três elementos
    enqueue(&inicio, &fim, 10);
    enqueue(&inicio, &fim, 20);
    enqueue(&inicio, &fim, 30);
    // Mostra a fila atual
    imprimirFila(inicio);
    // Consulta o primeiro elemento
    printf("Início da fila (peek): %d\n", peek(inicio));
    // Remove um elemento da fila
    int removido = dequeue(&inicio, &fim);
    printf("Removido da fila (dequeue): %d\n", removido);
    // Mostra a fila após remoção
    imprimirFila(inicio);
    return 0;
}

```

Simulação: Fila de Pacientes – Linguagem C (FIFO)

Ação	O que representa na fila
enqueue	Novo paciente chegando
dequeue	Paciente sendo atendido
peek	Ver o primeiro da fila
Impressão	Mostrar todos os pacientes na ordem

```

#include <stdio.h>
#include <stdlib.h>
// Estrutura do paciente (nó da fila)
struct Paciente {
    char nome[30];           // Nome (sem espaços)
    int idade;                // Idade
    struct Paciente *prox;    // Próximo paciente na fila
};
// Inserir paciente no fim da fila (enqueue)
void enfileirar(struct Paciente **inicio, struct Paciente **fim) {
    struct Paciente *novo = (struct Paciente*)malloc(sizeof(struct Paciente)); // Cria novo paciente
    printf("Digite o nome do paciente (sem espaços): ");
    scanf("%s", novo->nome); // Lê nome do paciente
    printf("Digite a idade do paciente: ");
    scanf("%d", &novo->idade); // Lê idade
    novo->prox = NULL; // Como é o último da fila, não aponta para ninguém
    if (*inicio == NULL) {
        // Se a fila estiver vazia, novo será o primeiro e o último
        *inicio = novo;
        *fim = novo;
    } else {
        // Se já houver pacientes, novo vai para o fim
        (*fim)->prox = novo;
        *fim = novo;
    }
    printf("Paciente inserido na fila com sucesso.\n");
}
// Atender paciente (remover do início da fila)
void atenderPaciente(struct Paciente **inicio, struct Paciente **fim) {
    if (*inicio == NULL) {
        printf("Fila vazia! Nenhum paciente para atender.\n");
        return;
    }
    struct Paciente *temp = *inicio; // Pega o paciente que será atendido
    printf("Atendendo paciente: %s (idade %d)\n", temp->nome, temp->idade);

    *inicio = temp->prox; // Avança o início da fila para o próximo

    if (*inicio == NULL) {
        // Se a fila ficou vazia após atender, fim também vira NULL
        *fim = NULL;
    }
    free(temp); // Libera a memória do paciente atendido
}

```

```

// Mostrar todos os pacientes da fila
void mostrarFila(struct Paciente *inicio) {
    if (inicio == NULL) {
        printf("Fila vazia!\n");
        return;
    }
    printf("\nFila de pacientes:\n");
    while (inicio != NULL) {
        printf("- %s (idade %d)\n", inicio->nome, inicio->idade);
        inicio = inicio->prox; // Avança para o próximo paciente
    }
}

// Função principal com menu
int main() {
    struct Paciente *inicio = NULL; // Ponteiro para o início da fila
    struct Paciente *fim = NULL; // Ponteiro para o fim da fila
    int opcao;
    do {
        // Mostra o menu
        printf("\n===== MENU - FILA DE PACIENTES =====\n");
        printf("1. Inserir paciente na fila\n");
        printf("2. Atender próximo paciente\n");
        printf("3. Mostrar fila\n");
        printf("0. Sair\n");
        printf("Escolha uma opção: ");
        scanf("%d", &opcao); // Lê a opção escolhida
        // Executa a opção escolhida
        if (opcao == 1) {
            enfileirar(&inicio, &fim);
        } else if (opcao == 2) {
            atenderPaciente(&inicio, &fim);
        } else if (opcao == 3) {
            mostrarFila(inicio);
        } else if (opcao == 0) {
            printf("Encerrando o programa...\n");
        } else {
            printf("Opção inválida. Tente novamente.\n");
        }
    } while (opcao != 0); // Repete até o usuário escolher sair
    // Libera a memória da fila (se ainda restar alguém)
    while (inicio != NULL) {
        atenderPaciente(&inicio, &fim);
    }
    return 0;
}

```