Listas, Filas e Pilhas

Rômulo César Silva

Unioeste

Abril de 2016





Sumário

- Lista
- 2 Lista Estática
- 3 Lista Dinâmica
 - Lista Encadeada Simples
 - Lista Duplamente Encadeada
 - Lista Duplamente Encadeada Circular
 - Lista Encadeada com Cabeça e Cauda
 - Lista Encadeada em Arquivo
- Fila
 - Implementação de Fila usando Vetor
 - Implementação de Fila usando Lista Encadeada
- Pilha
 - Implementação de Pilha usando Vetor
 - Implementação de Pilha usando Lista Encadeada
- 6 Complexidade de Tempo das Operações





Lista

Uma Lista é uma coleção de itens de dados do mesmo tipo.

Exemplos:

- lista de compras do supermercado
- lista de alunos da turma
- lista de clientes da loja
- lista de peças do estoque



Lista

Características

- Os itens estão dispostos sequencialmente $x_1, x_2, x_3, ..., x_n$, tal que x_1 é o primeiro item e x_n é o último item da lista
- Principais Operações:
 - inserção de um item na lista
 - remoção de um item da lista
 - busca de um item na lista



Lista

Aplicações na Computação

- **Sistemas Operacionais**: gerenciamento de fragmentação da memória principal
- Compiladores: alternativa para implementação da tabela de símbolos
- Computação Gráfica: listas de pontos que formam um polígono
- IA: computação simbólica usada nas linguagens LISP e Prolog
- ... e muitas outras



Principais operações

- inserir(Lista 1, Item x): insere o item x na lista l
- remover(Lista 1, Item x): remove o item x da lista l
- busca(Lista 1, Item x): busca o item x na lista l



Outras operaçõ<u>es</u>

- criar: cria uma lista vazia
- ordenar(Lista 1): colocar os itens em ordem crescente ou decrescente
- inserir(Lista 1, Item x, Posicao p): inserir o ltem x na posição p da lista l
- remover(Lista 1, Posicao p): remove o item que se encontra na posição p da lista l
- concatenar(Lista 11, Lista 12): concatena as listas l1 e l2



Implementação

A estrutura de dados **lista** pode ser implementada de muitas maneiras. A escolha de qual maneira implementar depende da aplicação.



Implementação

Dependendo das restrições que se impõe ao modo de funcionamento das operações de inserção e remoção, a lista recebe um nome específico:

- Fila: a inserção é sempre no final (cauda) da lista e a remoção é sempre do início (cabeça) da lista.
- Pilha: tanto a inserção quanto a remoção é feita sempre do início (cabeça) da lista.



Implementação

As 2 maneiras mais comuns de se implementar lista são:

- lista estática: usando vetores e alocação estática sequencial
- lista dinâmica: usando ponteiros e alocação dinâmica não sequencial



- uso de um vetor alocado estaticamente para armazenar sequencialmente os elementos
- a lista pode ser percorrida em ambas direções
- a inclusão de um novo elemento pode ser feita após o último elemento, sem necessidade de percorrer a lista
- a inclusão de um novo elemento no meio da lista requer o deslocamento de todos os elementos localizados após o ponto de inserção
- a retirada de um elemento requer também o deslocamento dos itens subsequentes para preencher o espaço deixado vazio pelo item removido





```
#define TAM_MAX 1000 // número máximo de itens na lista
typedef int TipoItem;

// estrutura para lista estática
typedef struct {
    TipoItem item[TAM_MAX];
    int pos_livre;
} Lista;
```



- TipoItem representa o tipo dos elementos da lista. Nesta implementação, é um int
- a variável pos_livre marca a última posição livre do vetor
- a lista está vazia se pos_livre é zero
- a lista está cheia se pos_livre é TAM_MAX
- a inserção é feita sempre na última posição livre
- a retirada de um elemento x que esteja na posição i ocasiona o deslocamento dos elementos das posições i+1, i+2, ..., para a posição imediatamente anterior.



1^a implementação - protótipos das funções (.h):

```
//Cria uma lista vazia
Lista* cria_lista_vazia();
//Testa se uma lista é vazia
int vazia(Lista * 1);
//Insere um item na lista
void insere(Lista* 1, TipoItem x);
//Retira o item da lista
void retira(Lista* 1, TipoItem x);
//Imprime os itens da lista
void imprime(Lista* 1);
```



```
1<sup>a</sup> implementação - (.c):
//Cria uma lista vazia
//Pré-condição: nenhuma
//Pós-condição: lista criada
Lista* cria_lista_vazia(){
    Lista * nova_lista = (Lista*) malloc(sizeof(Lista));
    nova_lista->pos_livre = 0;
    return nova_lista;
//Testa se uma lista é vazia
//Pré-condição: ponteiro não nulo para estrutura lista
//Pós-condição: retorna 1 se a lista é vazia ou
//
                         O caso contrário
int vazia(Lista * 1){
   return (1->pos_livre == 0);
```

1^a implementação - (.c):

```
//Insere um item na lista
//Pré-condição: ponteiro não nulo para estrutura lista
//Pós-condição: item é inserido na lista
void insere(Lista* 1, TipoItem x){
   if(1->pos_livre >= TAM_MAX)
     printf("Lista cheia");
   else
     1->item[1->pos_livre++] = x;
```

```
1<sup>a</sup> implementação - (.c):
//Retira o item da lista
//Pré-condição: ponteiro não nulo para estrutura lista
//Pós-condição: item é removido da lista
void retira(Lista* 1, TipoItem x){
   if(vazia(1))
     printf("Lista vazia: sem elementos para retirar");
   else {
     int pos = 0;
     while(pos < 1->pos_livre && x != 1->item[pos])
       pos++;
     if(pos == 1->pos_livre)
        printf("Elemento n\u00e3o encontrado");
     else { int i;
        for(i = pos; i < l->pos_livre; i++)
           1->item[i] = 1->item[i+1];
        1->pos_livre--;
```

```
#define TAM_MAX 1000 // número máximo de itens na lista
typedef int TipoItem;

// estrutura para lista estática
typedef struct {
    TipoItem item[TAM_MAX];
    int primeiro, ultimo;
    int pos; // usado para implementar iterador
} Lista:
```



- TipoItem representa o tipo dos elementos da lista. Nesta implementação, é um int
- a variável primeiro marca a posição do primeiro elemento
- a variável ultimo marca a primeira posição livre após o último elemento
- a variável pos é usada para propósitos de navegação na lista.
- a retirada de um elemento x que esteja na posição i ocasiona o deslocamento dos elementos das posições i+1, i+2, ..., para a posição imediatamente anterior.



2ª implementação:

```
Lista* cria_lista_vazia(){
    Lista * nova_lista = (Lista*) malloc(sizeof(Lista));
    nova_lista->primeiro = 0;
    nova_lista->ultimo = 0;
    nova_lista->pos = -1;
    return nova_lista;
}
int vazia(Lista * 1){
    return (1->primeiro == 1->ultimo);
}
```

```
//Retorna o primeiro item da lista
//Pré-condição: ponteiro não nulo para estrutura lista
//Pós-condição: retorna o primeiro item
TipoItem primeiro(Lista* 1) {
   1->pos = -1;
   return proximo(1);
//Retorna o proximo item da lista
//Pré-condição: ponteiro não nulo para estrutura lista
//Pós-condição: retorna próximo item
TipoItem proximo(Lista* 1) {
1->pos++;
 if(1->pos >= 1->ultimo)
    return -1; // acabou itens
else
    return 1->item[1->pos];
```

```
//Imprime os itens da lista
//Pré-condição: ponteiro não nulo para estrutura lista
//Pós-condição: nenhuma
void imprime(Lista* 1){
  if(vazia(1))
    printf("Lista vazia");
  else {
    int i = 0;
    printf("[");
    for(i = 1->primeiro; i < 1->ultimo; i++)
      printf("%d ", 1->item[i]);
    printf("]\n");
```

```
//imprime usando as funções iteradoras
// para percorrer a lista
void imprime2(Lista* 1){
  if(vazia(1))
    printf("Lista vazia");
  else {
    TipoItem x = primeiro(1);
    printf("[");
    dο
      printf("%d ", x);
    while((x = proximo(1))! = -1);
    printf("]\n");
```



Lista Lista Estática Lista Dinâmica Fila Pilha

Implementação de Lista Dinâmica

- uso de um ponteiro para encadear um elemento com o seguinte
- a alocação de memória é feita à medida que se necessita inserir um novo elemento. Assim, os elementos não ocupam posições contíguas na memória
- a inclusão/retirada de um novo elemento pode ser feita sem necessidade de deslocamento dos demais elementos





Lista Dinâmica

Variações de Implementação

- Lista Dinâmica Simples
- Lista Duplamente Encadeada
- Lista Duplamente Encadeada Circular
- Lista Encadeada com Cabeça e Cauda





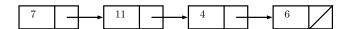
Estrutura:

```
typedef int TipoItem; // tipo dos elementos na lista
typedef struct no { //estrutura para lista encadeada
    TipoItem info;
    struct no * prox; // ponteiro para próximo elemento
} Lista;
```



Inserção na cabeça

Inserir o elemento 5 na lista abaixo:

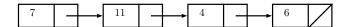




Inserção na cabeça

1º passo: criar o nó para conter o elemento.

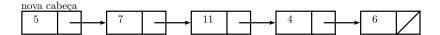






Inserção na cabeça

2º passo: apontar o próximo do nó criado para a cabeça e torná-lo a nova cabeça.





Inserção na cabeça

```
Lista* insere(Lista* 1, TipoItem info) {
  Lista* aux = (Lista*) malloc(sizeof(Lista));
  aux->info = info;
  aux->prox = 1; //encadeia com a cabeça da lista
  return aux;// retorna a nova cabeça
}
```

Remoção de um elemento

Há 2 casos:

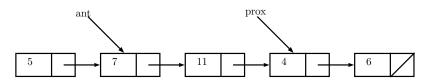
- remoção no meio da lista
- remoção da cabeça





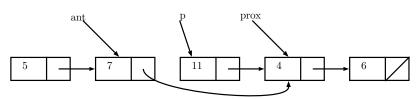
Remoção no meio

Remover o elemento 11 na lista abaixo:



Remoção no meio:

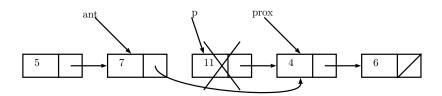
 1^o passo: encadeiar o anterior com o próximo do elemento a ser removido



Remoção no meio

2º passo: liberar o nó do elemento removido

free(p);

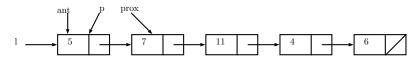






Remoção na cabeça

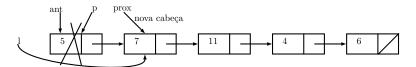
Remover o elemento 5 da lista abaixo:



Remoção na cabeça

1º passo: a cabeça aponta para o próximo elemento

2º passo: o nó da cabeça antiga é liberado



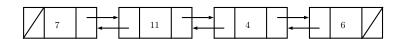


Lista Encadeada Simples

Remoção: juntando os 2 casos no código

```
Lista * retira(Lista* 1, TipoItem info) {
  Lista * ant = 1; // elemento anterior
  Lista * p = 1; // usado para encontrar o elemento
                 // a ser retirado
  while(p!= NULL && p->info != info) { // localiza o elemento
     ant = p;
     p = p->prox;
  if(p != NULL) { // elemento encontrado
    if(p == 1) // remoção na cabeça
       1 = 1->prox; // atualiza a cabeça
    else // remoção no meio
       ant->prox = p->prox;
    free(p); // libera o nó do elemento removido
  else printf("Elemento n\u00e3o encontrado");
  return 1;
```

Lista Duplamente Encadeada





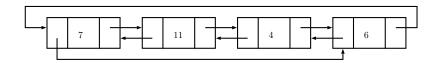
Lista Duplamente Encadeada

Estrutura:

```
typedef int TipoItem; // tipo dos elementos na lista

typedef struct no { //estrutura para lista duplamente encadeada
    TipoItem info;
    struct no * ant; // ponteiro para o elemento anterior
    struct no * prox; // ponteiro para próximo elemento
} Lista;
```

Lista Duplamente Encadeada Circular





Lista Duplamente Encadeada Circular

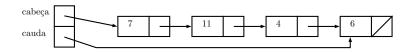
Estrutura:

- é a mesma da Lista Encadeada Circular
- as operações de inserção e remoção devem ser implementadas de modo a garantir a circularidade (o próximo do último é o primeiro, e o anterior do primeiro é o último)





Lista Encadeada com Cabeça e Cauda





Lista Encadeada com Cabeça e Cauda

Estrutura:

```
typedef int TipoItem;

struct no {  // estrutura de nó para lista encadeada
    TipoItem info;
    struct no * prox;
};

typedef struct {  // estrutura para lista encadeada com cabeça e cauda
    struct no* cabeca;
    struct no* cauda;
} Lista;
```

Lista Dinâmica

É importante o aluno dominar a implementação das diferentes variantes!

Por isso:

- implemente primeiro a variante mais simples (Lista Dinâmica Simples) para somente depois implementar as mais complexas.
- ao projetar os algoritmos de inserção e retirada de elementos, procure identificar os casos especiais que existem em cada variante.
- teste cada caso separadamente





- uso de arquivo no lugar da memória principal para armazenar os nós da lista
- se mantêm na memória principal apenas os registros que se está manipulando no momento, devendo ser escritos em seguida no arquivo
- ao invés de usar funções de alocação de memória (malloc), escreve-se um registro no final arquivo ou em uma posição não ocupada.
- o encadeamento, ao invés de usar ponteiros, é feito usando as posições dos registros no arquivo





- pode-se manter uma lista de nós livres que aumenta ou diminui conforme são retirados ou inseridos novos elementos
- é útil ter um cabeçalho no início do arquivo em que se guardam os endereços das cabeças da lista de nós ativos e da lista de nós livres, além da próxima posição após o final do arquivo.
- o mais adequado é usar arquivo-binário e não arquivo-texto para armazenar a lista para dificultar alterações indesejadas diretamente sobre o arquivo.





funções para manipular arquivos em C: abertura

FILE* fopen(char* nomeArg, char* modo)

- nomeArq deve ser um nome válido no sistema operacional
- modo indica quais serão as operações válidas. w+b para leitura e gravação em modo binário
- A função retorna um ponteiro para arquivo (handle)
- no caso de erro, a função retorna um ponteiro nulo (NULL)



funções para manipular arquivos em C: posicionamento

int fseek(FILE* fp, long nro_bytes,int origem)

- a função move a posição corrente nro_bytes a partir de um ponto especificado (origem), que pode ser:
 - SEEK_SET: a partir do início do arquivo
 - SEEK_CUR: a partir da posição corrente
 - SEEK_END: a partir do fim do arquivo





funções para manipular arquivos em C: leitura

unsigned fread(void* buffer, int nro_bytes, int count, FILE* fp)

- buffer: região de memória onde os dados serão armazenados
- nro_bytes: número de bytes lidos por unidade
- count: quantidade de unidades a serem lidas
- a função retorna o número de unidades efetivamente lidas





funções para manipular arquivos em C: escrita

unsigned fwrite(void* buffer, int nro_bytes, int count, FILE* fp)

• parâmetros similares aos de fread, porém para escrita



Estrutura:

```
typedef int TipoItem;
// cabecalho do arquivo
typedef struct {
   int pos_cabeca; //posição do início da lista
   int pos_topo; // la posição não usada no fim do arquivo
   int pos_livre; // posição do início da lista de nós livres
} cabecalho;
// estrutura de no para lista encadeada
typedef struct {
    TipoItem info;
    int prox;
} no;
```

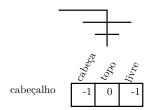


Função para criar uma lista vazia em um arquivo:

```
//Cria uma lista nova em arquivo
//Pré-condição: arquivo aberto para leitura/escrita
//Pós-condição: arquivo é inicializado com uma lista vazia

void cria_lista_vazia(FILE* arq) {
    cabecalho * cab = (cabecalho*) malloc(sizeof(cabecalho));
    cab->pos_cabeca = -1;
    cab->pos_topo = 0;
    cab->pos_livre = -1;
    escreve_cabecalho(arq,cab);
    free(cab);
}
```

Inserir 6, 13, 11, 37 e 22 nesta ordem em uma lista inicialmente vazia:



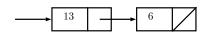


Inserção de 6:





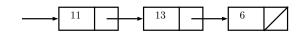
Inserção de 13:



	59985 1	×\$	30	li_{VPe}
cabeçalho	1		2	-1
Num. Registro:0	6			-1
1	13			0



Inserção de 11:

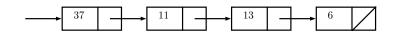


	e569e5	, 	ŞÇ Q	\dot{u}_{VPe}
cabeçalho	2		3	-1
Num. Registro:0	6			-1
1	13			0
2	11			1





Inserção de 37:

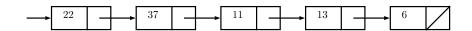


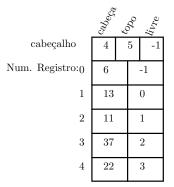
	259925		30	$\dot{u}_{V\!T_{\rm e}}$
cabeçalho	3		4	-1
Num. Registro:0	6			-1
1	13			0
2	11			1
3	37			2





Inserção de 22:







Remover 11, 22 e 6 nesta ordem da lista abaixo:

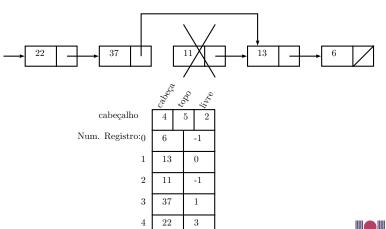


	e569e5	4	80	$\dot{u}_{VP_{\Theta}}$
cabeçalho	4		5	-1
Num. Registro:0	6			-1
1	13		0	
2	11		1	
3	37		2	
4	22			3



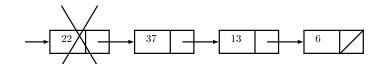
Lista Encadeada em Arquivo

Removendo 11:





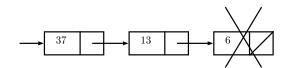
Removendo 22:



cabeçalho	3		5	4
Num. Registro:0	6			-1
1	13	13		0
2	11			-1
3	37			1
4	22			2



Removendo 6:

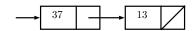


	e599e3	4	90	\dot{li}_{VPe}
cabeçalho	3		5	0
Num. Registro:0	6			4
1	13		-1	
2	11			-1
3	37			1
4	22			2





Inserir 18 na lista abaixo:

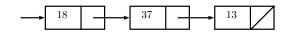


	eJ _o 9e5	, 4	900	$li_{VR_{\Theta}}$
cabeçalho	3		5	0
Num. Registro:0	6			4
1	13			-1
2	11			-1
3	37			1
4	22			2





Inserindo 18:



	3599e3	<i>t</i>	00%	h_{Ye}
cabeçalho	0		5	4
Num. Registro:0	18			3
1	13			-1
2	11			-1
3	37			1
4	22			2





Funções para ler e escrever o cabeçalho do arquivo:

fwrite(cab, sizeof(cabecalho), 1, arq);

```
//Lê o cabeçalho do arquivo contendo as informações da lista
//Pré-condição: arquivo deve estar aberto e ser um arquivo de lista
//Pós-condição: retorna o ponteiro para o cabeçalho lido
cabecalho* le_cabecalho(FILE * arq) {
   cabecalho * cab = (cabecalho*) malloc(sizeof(cabecalho));
   fseek(arq,0,SEEK_SET); // posiciona no início do arquivo
   fread(cab, sizeof(cabecalho), 1, arg);
   return cab;
//Escreve no arquivo o cabeçalho contendo as informações da lista
//Pré-condição: arquivo deve estar aberto e ser um arquivo de lista
//Pós-condição: cabeçalho escrito no arquivo
void escreve_cabecalho(FILE* arg, cabecalho* cab){
   fseek(arq,0,SEEK_SET); //posiciona no início do arquivo
```

Função para ler um nó da lista no arquivo:

```
//lê um nó em uma determinada posição do arquivo
//Pré-condição: arquivo deve estar aberto e ser um arquivo de lista
// pos deve ser uma posição válida da lista
//Pós-condição: ponteiro para nó lido é retornado

no* le_no(FILE* arq, int pos) {
    no* x = malloc(sizeof(no));
    fseek(arq,sizeof(cabecalho)+ pos*sizeof(no),SEEK_SET);
    fread(x,sizeof(no),1,arq);
    return x;
}
```

Função para escrever um nó da lista no arquivo:

```
//Escreve um nó em uma determinada posição do arquivo
//Pré-condição: arquivo deve estar aberto e ser um arquivo de lista
// pos deve ser uma posição válida do arquivo
//Pós-condição: nó escrito no arquivo

void escreve_no(FILE* arq, no* x, int pos){
   fseek(arq,sizeof(cabecalho)+ pos*sizeof(no),SEEK_SET);
   fwrite(x,sizeof(no),1,arq);
}
```

Função para inserir um nó na cabeça da lista no arquivo:

```
void insere(FILE* arg, TipoItem info){
    cabecalho* cab = le_cabecalho(arq);
    no x;
    x.info = info:
    x.prox = cab->pos_cabeca;
    if(cab->pos_livre == -1) { // não há nós livres, então usar o topo
          escreve_no(arq,&x,cab->pos_topo);
          cab->pos_cabeca = cab->pos_topo;
          cab->pos_topo++;
     else { // usar nó da lista de livres
          no* aux = le_no(arq,cab->pos_livre);
          escreve_no(arq,&x,cab->pos_livre);
          cab->pos_cabeca = cab->pos_livre;
          cab->pos_livre = aux->prox;
          free(aux):
    escreve_cabecalho(arq,cab); free(cab);
```

Função para retirar um nó na cabeça da lista no arquivo:

```
//Retira um nó da lista
//Pré-condição: arquivo deve estar aberto e ser um arquivo de lista
//Pós-condição: nó retirado da lista caso pertença a ela
void retira(FILE* arg, TipoItem x){
    cabecalho* cab = le_cabecalho(arg);
    int pos_aux = cab->pos_cabeca;
    int pos_ant = cab->pos_cabeca;
    no* aux = NULL;
    while(pos_aux != -1 && // procura o elemento a ser retirado
          ((aux = le_no(arq,pos_aux))!= NULL) &&
          aux - \sin \theta != x){
      pos_ant = pos_aux;
      pos_aux = aux->prox;
      free(aux);
      aux = NULL;
```

free(cab):

Função para retirar um nó na cabeça da lista no arquivo (cont.):

```
if(pos_aux != -1) { //encontrou o elemento
  if(pos_ant == pos_aux){ // remoção na cabeça
    cab->pos_cabeca = aux->prox;
 else { // remoção no meio
   no * ant = le_no(arq,pos_ant);
    ant->prox = aux->prox;
    escreve_no(arq,ant,pos_ant);
   free(ant);
 aux->prox = cab->pos_livre; // torna o nó removido um nó livre
  cab->pos_livre = pos_aux;
  escreve_no(arq,aux,pos_aux);
  escreve_cabecalho(arq,cab);
 free(aux);
```

Filas

Fila

Uma **fila** é uma lista linear em que as inserções são sempre feitas no final (cauda) e as remoções no início (cabeça).

- O primeiro a entrar é o primeiro a sair (First-In First-Out)
- Exemplo típico: fila de caixa bancário
- Exemplos de aplicações na Computação:
 - **Sistemas Operacionais**: implementação de escalonadores de processos (*scheduler*)
 - Servidores de Banco de Dados: atendimento de requisições de consultas





Fila

Representação Esquemática:





Filas

Fila

- Termo em inglês para fila: queue. Operações mais comuns:
 - Enqueue: enfileirar um elemento
 - Dequeue: desenfileirar um elemento





Implementação de Fila usando vetor

```
#define TAM_MAX 1000 // número máximo de itens na fila
typedef int TipoItem;

// estrutura para fila usando vetor
typedef struct {
    TipoItem vet[TAM_MAX];
    int inicio, fim; // indices do início e fim da fila
} Fila;
```

- elementos são dispostos sequencialmente no vetor
- inicio e fim correspondem à posição do 1º elemento da fila e à primeira posição livre após o último elemento da fila, respectivamente.

Lista Lista Estática Lista Dinâmica Fila Pilha

Implementação de Fila usando vetor

```
Fila* cria_fila_vazia() {
    Fila* f = (Fila*) malloc(sizeof(Fila));
    f \rightarrow inicio = 0;
    f \rightarrow fim = 0;
    return f;
int vazia(Fila* f) {
  return (f->inicio == f->fim);
int cheia(Fila* f) {
  return (f->fim >= TAM_MAX);
```

- uma fila está vazia quando inicio é igual ao fim
- uma fila está cheia se a posição de fim já passou da últiposição válida do vetor (0..TAM_MAX −1)

Implementação de Fila usando vetor

```
// Enfileira um elemento
void enqueue(Fila* f, TipoItem x) {
   if(!cheia(f)){
      f \rightarrow vet[f \rightarrow fim] = x;
      f->fim++:
   else printf("Fila cheia!");
// Desenfileira um elemento
TipoItem* dequeue(Fila * f) {
   if(!vazia(f)) {
     TipoItem* aux = (TipoItem*) malloc(sizeof(TipoItem));
     *aux = f->vet[f->inicio++];
     return aux;
   else {
     printf("Fila vazia!");
     return NULL;
```

Implementação de Fila usando vetor

Problemas com essa implementação:

- como os índices fim e inicio deslocam-se sempre para a direita, a fila pode receber no máximo TAM_MAX elementos, e uma única vez.
- quando inicio e fim valerem TAM_MAX tanto a função vazia quanto a função cheia retornará verdadeiro, sendo incoerente.

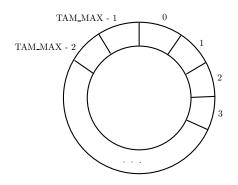
Solução: usar uma estrutura circular





Lista Lista Estática Lista Dinâmica Fila Pilha

Implementação de Fila Circular usando vetor



- a fila está vazia quando inicio é igual a fim
- a fila está cheia quando fim + 1 módulo TAM_MAX é igual a inicio
- a posição fim deve ficar sempre desocupada para diferentia a lista cheia da lista vazia



Implementação de Fila Circular usando vetor

```
// Testa se a fila está vazia
int vazia(Fila* f) {
    return (f->inicio == f->fim);
}

//Testa se a fila está cheia
int cheia(Fila* f) {
    //testa se a posição seguinte ao fim é igual a posição de início
    // usando aritmética modular
    return ((f->fim + 1)%TAM_MAX == f->inicio);
}
```

• a fila está cheia quando o próximo do fim no vetor circular é o indíce inicio



Implementação de Fila Circular usando vetor

```
// Enfileira um elemento
void enqueue(Fila* f, TipoItem x) {
   if(!cheia(f)){
      f->vet[f->fim] = x;
      f->fim++;
      if(f->fim == TAM_MAX) // implementa circularidade
      f->fim = 0;
   else printf("Fila cheia!");
   }
}
```



Implementação de Fila Circular usando vetor

```
// Desenfileira um elemento
TipoItem* dequeue(Fila * f) {
   if(!vazia(f)) {
     TipoItem* aux = (TipoItem*) malloc(sizeof(TipoItem));
     *aux = f->vet[f->inicio];
     f->inicio++:
     if(f->inicio == TAM_MAX) // implementa circularidade
       f \rightarrow inicio = 0:
     return aux;
   else {
         printf("Fila vazia!");
         return NULL;
```



```
typedef int TipoItem;
struct no {
     TipoItem info;
     struct no* prox;
};
typedef struct {
     struct no* inicio;
     struct no* fim;
} Fila;
```

 Uso de lista encadeada com cabeça e cauda para facilitar as inserções e remoções



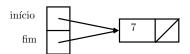
Inserir 7, 11, 4 e 6 em uma fila inicialmente vazia.

Fila vazia:

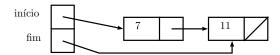




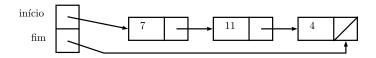




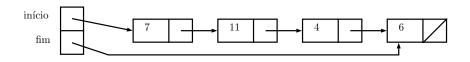




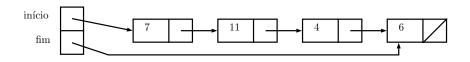




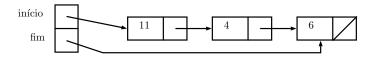




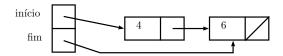
Efetuar 2 operações dequeue na fila abaixo:



Resultado do 1º dequeue:



Resultado do 2º dequeue:



```
int vazia(Fila* f) {
  return (f->inicio == NULL);
}

Fila* cria_fila_vazia() {
  Fila* f = (Fila*) malloc(sizeof(Fila));
  f->inicio = NULL;
  f->fim = NULL;
  return f;
}
```



```
\\Enfileira um elemento
void enqueue(Fila* f, TipoItem x) {
  struct no* aux = (struct no*) malloc(sizeof(struct no));
  aux->info = x;
  aux->prox = NULL;
  if(vazia(f)) {
    f->inicio = aux;
   else {
     f->fim->prox = aux;
  f->fim = aux:
```

```
//Desenfileira um elemento
TipoItem* dequeue(Fila* f) {
  if(!vazia(f)) {
     TipoItem* x = (TipoItem*) malloc(sizeof(TipoItem));
     struct no* aux = f->inicio;
     *x = f->inicio->info:
     if(f->inicio == f->fim) // só tem 1 elemento
        f->fim = NULL;
     f->inicio = f->inicio->prox;
     free(aux):
     return x;
  else return NULL;
```

Lista Lista Estática Lista Dinâmica Pilha

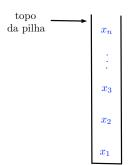
Pilhas

Pilha

Uma pilha é uma lista linear em que tanto as inserções quanto remoções são sempre feitas na cabeça (ou topo da pilha).

- O último a entrar é o primeiro a sair (Last-In First-Out)
- Exemplos típicos: pilha de livros, pilha de pratos.
- Exemplos de aplicações na Computação:
 - Editores de texto: mecanismo desfazer/refazer
 - Compiladores: implementação de analisadores sintáticos de linguagens de programação
 - Compiladores: modelo de ambiente de execução adotado pelas linguagens de programação Pascal, C, C++, Java, entre outras, em que parâmetros e endereço de retorno de cada procedimento/função/subrotina/método invocado é armazenado em uma pilha para controle das chamadas.
 - Algoritmos: pode ser usada como estrutura auxiliar na 🏢 implementação de algoritmos como forma de eliminação unioeste recursão.

Representação Esquemática:





Pilha

- Termo em inglês para pilha: stack. Operações mais comuns:
 - push: empilhar um elemento
 - pop: desempilhar um elemento





Implementação de Pilha usando vetor

```
#define TAM_MAX 1000

typedef int TipoItem;

typedef struct {
    TipoItem vet[TAM_MAX];
    int topo;
} Pilha;
```

• topo marca o topo da pilha, que corresponde à primeira posição livre do vetor.



Implementação de Pilha usando vetor

```
int vazia(Pilha * p) {
return (p->topo == 0);
int cheia(Pilha* p) {
return (p->topo == TAM_MAX);
Pilha* criar_pilha_vazia(){
  Pilha* p = (Pilha*) malloc(sizeof(Pilha));
 p->topo = 0;
  return p;
```

 A pilha está vazia se o topo é igual a zero e está cheia s topo é igual a TAM_MAX



Implementação de Pilha usando vetor

```
//Empilha um elemento
void push(Pilha* p, TipoItem x) {
   if(!cheia(p)){
     p->vet[p->topo++] = x;
   else printf("Pilha cheia!");
//Desempilha um elemento
TipoItem* pop(Pilha* p) {
  if(!vazia(p)){
    TipoItem* aux = (TipoItem*) malloc(sizeof(TipoItem));
    *aux = p->vet[--(p->topo)];
    return aux:
  else {
    printf("Pilha vazia!");
    return NULL;
```

```
typedef int TipoItem;
// estrutura para lista encadeada
struct no {
   TipoItem info;
   struct no* prox;
   };
// estrutura de lista com cabeça
// para implementação de pilha
typedef struct {
  struct no* topo;
} Pilha:
```

 uso de lista encadeada com cabeça para marcação do topo da pilha.

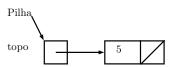


Considere empilhar 5, 8 e 3 em uma pilha inicialmente vazia **Pilha vazia**:



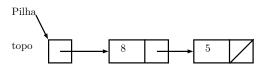


5 empilhado:



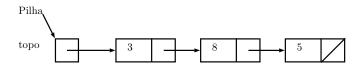


8 empilhado:



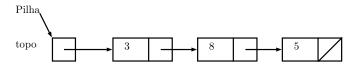


3 empilhado:

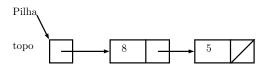




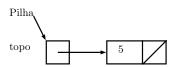
Considere executar 2 operações de desempilhar (pop) na pilha abaixo:



Resultado do 1º pop:



Resultado do 2º pop:





```
int vazia(Pilha* p) {
  return (p->topo == NULL);
}

Pilha* criar_pilha_vazia() {
  Pilha* aux = (Pilha*) malloc(sizeof(Pilha));
  aux->topo = NULL;
  return aux;
}
```

A pilha é vazia se o topo é NULL



```
// Lê o topo da pilha sem desempilhar
TipoItem* topo(Pilha* p) {
   if(!vazia(p)) {
      TipoItem* x = (TipoItem*) malloc(sizeof(TipoItem));
      *x = p->topo->info;
      return x;
   }
   else return NULL;
}
```

 A função topo permite consultar o topo da pilha sem desempilhar



```
// Empilha um item
void push(Pilha* p, TipoItem x) {
   struct no* aux = (struct no*) malloc(sizeof(struct no));
   aux->info = x;
   aux->prox = p->topo;
   p->topo = aux;
}
```

```
//Desempilha um item
TipoItem* pop(Pilha* p) {
   if(!vazia(p)) {
     TipoItem* x = (TipoItem*) malloc(sizeof(TipoItem));
     struct no* aux = p->topo;
     *x = p->topo->info;
     p->topo = p->topo->prox;
     free(aux);
     return x;
   else {
     printf("Pilha vazia!");
     return NULL;
```

Sendo n o número de elementos já inseridos na estrutura:

	inserção na cabeça	inserção na cauda		
lista em vetor	O(n)	O(1)		
lista encadeada simples	O(1)	O(n)		
lista encadeada	O(1)	O(1)		
com cabeça e cauda	0(1)	0(1)		



A operação de busca de um elemento em uma lista, em qualquer uma das variantes é O(n) no pior caso, supondo-se que os elementos estejam desordenados e o elemento procurado é o último.



Sendo n o número de elementos já inseridos na estrutura:

	enqueue	dequeue	
fila circular	O(1)	O(1)	
usando vetor	O(1)	0(1)	
fila usando			
lista encadeada	O(1)	O(1)	
com cabeça e cauda			



Sendo *n* o número de elementos já inseridos na estrutura:

	push	pop
pilha usando vetor	O(1)	O(1)
pilha usando lista	O(1)	O(1)
encadeada com cabeça	0(1)	0(1)

