



# Sumário

- 1 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
- 4 Fila
  - Implementação de Fila usando Vetor
  - Implementação de Fila usando Lista Encadeada
- 5 Pilha
  - Implementação de Pilha usando Vetor
  - Implementação de Pilha usando Lista Encadeada
- 6 Complexidade de Tempo das Operações











## TAD Lista

## Implementação

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

















**1ª 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* l, TipoItem x){
    if(l->pos_livre >= TAM_MAX)
        printf("Lista cheia");
    else
        l->item[l->pos_livre++] = x;
}
```



◀ ◻ ▶ ◀ ◻ ▶ ◀ ≡ ▶ ◀ ≡ ▶ ≡ ◻ ↺ 🔍 ↻







```
//Retorna o primeiro item da lista
//Pré-condição: ponteiro não nulo para estrutura lista
//Pós-condição: retorna o primeiro item
TipoItem primeiro(Lista* l) {
    l->pos = -1;
    return proximo(l);
}

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



## Implementação de Lista Estática

## 2ª implementação:

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

# 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



## Variações de Implementação

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

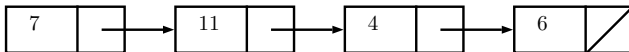
### Estrutura:

```
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

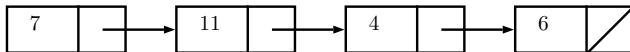
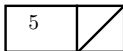
Inserir o elemento 5 na lista abaixo:



## Lista Encadeada Simples

## Inserção na cabeça

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





## Lista Encadeada Simples

## Inserção na cabeça

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

## Lista Encadeada Simples

## Remoção de um elemento

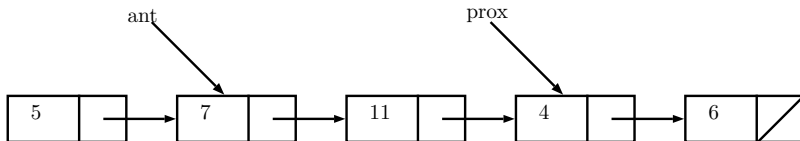
Há 2 casos:

- remoção no meio da lista
- remoção da cabeça

## Lista Encadeada Simples

## Remoção no meio

Remover o elemento 11 na lista abaixo:



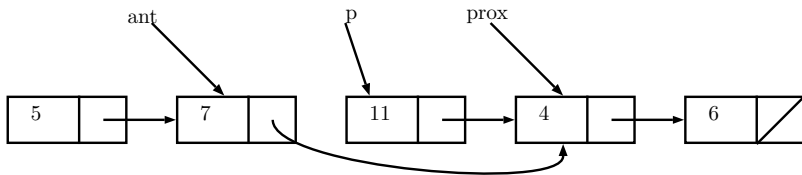


# Lista Encadeada Simples

## Remoção no meio:

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

```
ant->prox = p->prox;
```

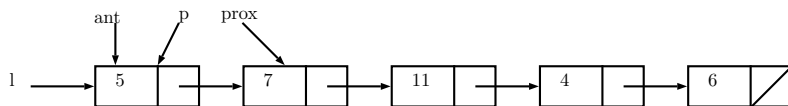


A set of small navigation icons typically found in Beamer presentations, including symbols for back, forward, search, and other slide controls.

## Lista Encadeada Simples

## Remoção na cabeça

Remover o elemento 5 da lista abaixo:





**Remoção:** juntando os 2 casos no código

}



### Estrutura:

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

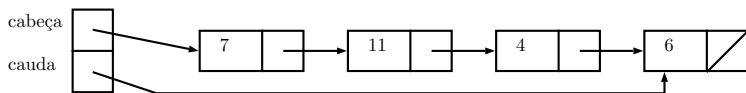
typedef struct no { //estrutura para lista duplamente encadeada
    TipoItem info;
    struct no * ant; // ponteiro para o elemento anterior
    struct no * prox; // ponteiro para próximo elemento
} Lista;
```







# Lista Encadeada com Cabeça e Cauda



### Estrutura:

```
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* nomeArq, 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)

## Lista Encadeada em Arquivo

## 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:

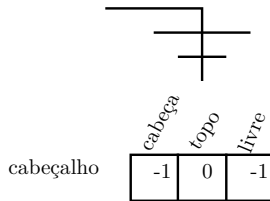
### Função para criar uma lista vazia em um arquivo:

```
//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);
}
```

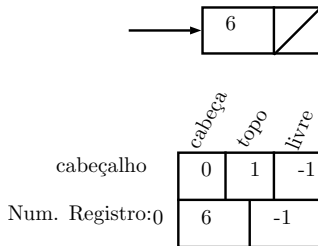
## Lista Encadeada em Arquivo

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



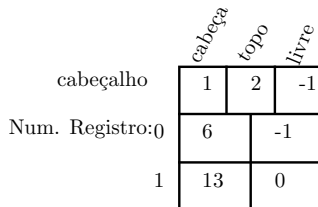
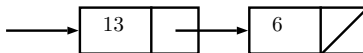
## Lista Encadeada em Arquivo

**Inserção de 6:**



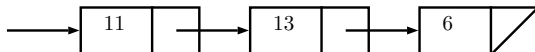
## Lista Encadeada em Arquivo

**Inserção de 13:**



## Lista Encadeada em Arquivo

**Inserção de 11:**



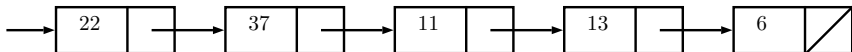
|                 | <i>cabeça</i> | <i>topo</i> | <i>livre</i> |
|-----------------|---------------|-------------|--------------|
| cabeçalho       | 2             | 3           | -1           |
| Num. Registro:0 | 6             | -1          |              |
| 1               | 13            | 0           |              |
| 2               | 11            | 1           |              |





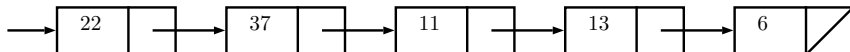
# Lista Encadeada em Arquivo

**Inserção de 22:**



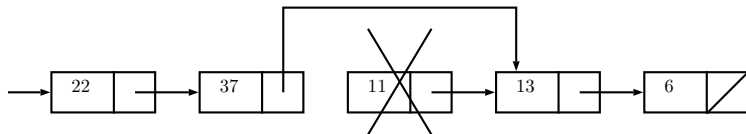
|                 | <i>cabeça</i> | <i>topo</i> | <i>livre</i> |
|-----------------|---------------|-------------|--------------|
| cabeçalho       | 4             | 5           | -1           |
| Num. Registro:0 | 6             |             | -1           |
| 1               | 13            |             | 0            |
| 2               | 11            |             | 1            |
| 3               | 37            |             | 2            |
| 4               | 22            |             | 3            |

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



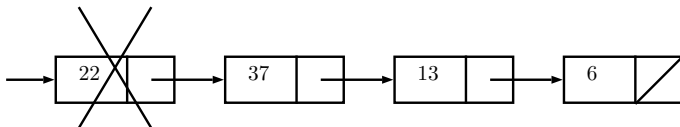
|                 | <i>cabeça</i> | <i>topo</i> | <i>livre</i> |
|-----------------|---------------|-------------|--------------|
| cabeçalho       | 4             | 5           | -1           |
| Num. Registro:0 | 6             |             | -1           |
| 1               | 13            |             | 0            |
| 2               | 11            |             | 1            |
| 3               | 37            |             | 2            |
| 4               | 22            |             | 3            |

### Removendo 11:



|  |                 | <i>cabeça</i> | <i>topo</i> | <i>livre</i> |
|--|-----------------|---------------|-------------|--------------|
|  | cabeçalho       | 4             | 5           | 2            |
|  | Num. Registro:0 | 6             |             | -1           |
|  | 1               | 13            |             | 0            |
|  | 2               | 11            |             | -1           |
|  | 3               | 37            |             | 1            |
|  | 4               | 22            |             | 3            |

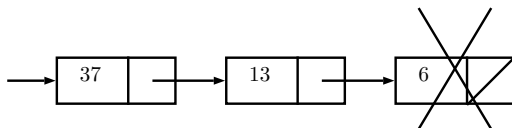
## Removendo 22:



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

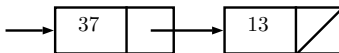
## Lista Encadeada em Arquivo

### Removendo 6:



|                 | <i>cabeça</i> | <i>topo</i> | <i>livre</i> |
|-----------------|---------------|-------------|--------------|
| cabeçalho       | 3             | 5           | 0            |
| Num. Registro:0 | 6             | 4           |              |
| 1               | 13            | -1          |              |
| 2               | 11            | -1          |              |
| 3               | 37            | 1           |              |
| 4               | 22            | 2           |              |

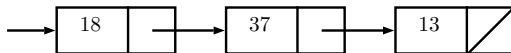
**Inserir 18 na lista abaixo:**



|                 | <i>cabeça</i> | <i>topo</i> | <i>livre</i> |
|-----------------|---------------|-------------|--------------|
| cabeçalho       | 3             | 5           | 0            |
| Num. Registro:0 | 6             |             | 4            |
| 1               | 13            |             | -1           |
| 2               | 11            |             | -1           |
| 3               | 37            |             | 1            |
| 4               | 22            |             | 2            |

## Lista Encadeada em Arquivo

### Inserindo 18:



|                 | <i>cabeça</i> | <i>topo</i> | <i>livre</i> |
|-----------------|---------------|-------------|--------------|
| cabeçalho       | 0             | 5           | 4            |
| Num. Registro:0 | 18            | 3           |              |
| 1               | 13            | -1          |              |
| 2               | 11            | -1          |              |
| 3               | 37            | 1           |              |
| 4               | 22            | 2           |              |





## Lista Encadeada em 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;
}
```

## Lista Encadeada em Arquivo

### 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);
}
```

## Lista Encadeada em Arquivo

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

```
void insere(FILE* arq, 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:**

```
void retira(FILE* arq, TipoItem x){
    cabecalho* cab = le_cabecalho(arq);
    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->info != x){
        pos_ant = pos_aux;
        pos_aux = aux->prox;
        free(aux);
        aux = NULL;
    }
}
```

**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);
}
free(cab);

```

# Filas

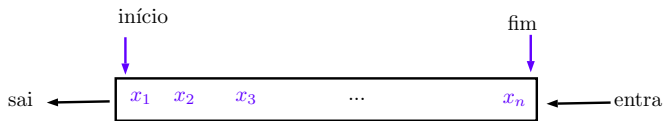
## Fila

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:





## 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; // índices do início e fim da fila
} Fila;
```

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

## Implementação de Fila usando vetor

```
Fila* cria_fila_vazia() {
    Fila* f = (Fila*) malloc(sizeof(Fila));
    f->inicio = 0;
    f->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 última posição 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->vet[f->fim] = x;
        f->fim++;
    }
    else printf("Fila cheia!");
}

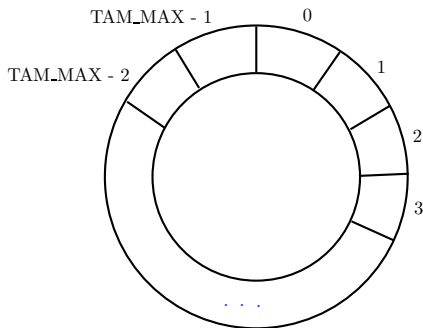
// Desenfileira um elemento
TipoItem* dequeue(Fila * f) {
    if(!vazia(f)) {
        TipoItem* aux = (TipoItem*) malloc(sizeof(TipoItem));
        *aux = f->vet[f->inicio++];
        return aux;
    }
    else {
        printf("Fila vazia!");
        return NULL;
    }
}
```

# 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



- a fila está vazia quando  $\text{inicio} = \text{fim}$
- a fila está cheia quando  $\text{fim} + 1 \bmod \text{TAM\_MAX} = \text{inicio}$
- a posição  $\text{fim}$  deve ficar sempre desocupada para diferenciar 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 índice início

## 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!");
}
```





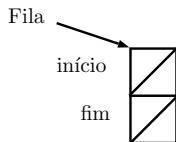
```
typedef int TipoItem;

struct no {
    TipoItem info;
    struct no* prox;
};

typedef struct {
    struct no* inicio;
    struct no* fim;
} Fila;
```

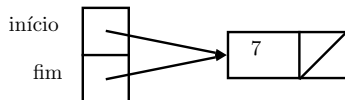
- Uso de lista encadeada com cabeça e cauda para facilitar as inserções e remoções

**Fila vazia:**

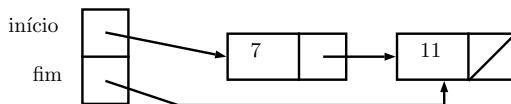


# Fila

**7 inserido:**



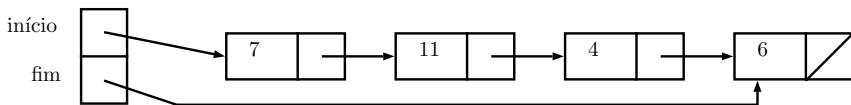
**11 inserido:**







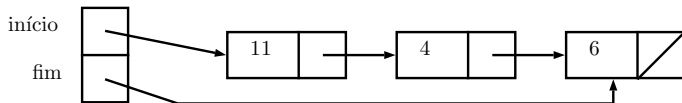
Efetuar 2 operações *dequeue* na fila abaixo:





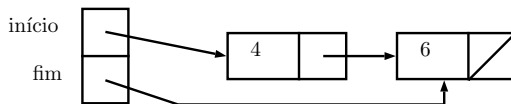
# Fila

Resultado do 1º *dequeue*:



# Fila

Resultado do 2º *dequeue*:



## Implementação de Fila usando lista encadeada

```
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;
}
```

# Pilhas

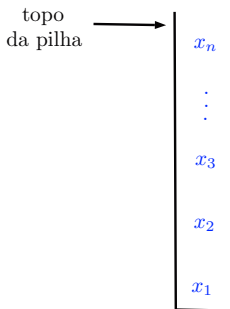
# Pilha

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 de recursão.

# Pilha

## Representação Esquemática:



# Pilhas

## Pilha

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





1. *Journal of Management Studies*, 1997, 34, 1, 1-14.

```
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 se o 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;
    }
}
```

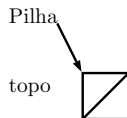
```
typedef int TipoItem;

// 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;
```

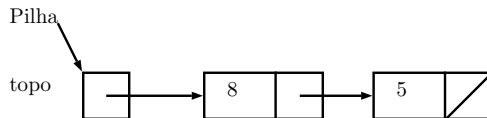
- uso de lista encadeada com cabeça para marcação do topo da pilha.

**Pilha vazia:**



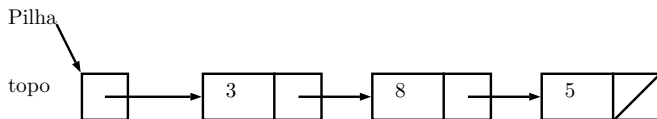


### 8 empilhado:



# Pilha

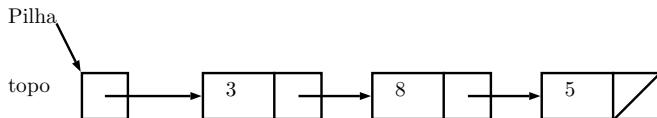
**3 empilhado:**





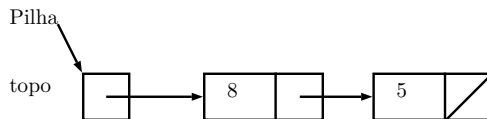
# Pilha

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



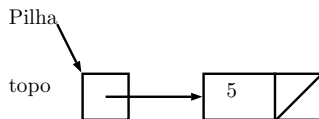
# Pilha

Resultado do 1º pop:



# Pilha

Resultado do 2º *pop*:





- A função `topo` permite consultar o topo da pilha sem desempilhar



## Implementação de Pilha usando lista encadeada

```
//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;
    }
}
```

## Observações quanto à complexidade de tempo

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 com cabeça e cauda | $O(1)$             | $O(1)$            |



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.

## Observações quanto à complexidade de tempo

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

|  |         |         |
|--|---------|---------|
|  | enqueue | dequeue |
| fila circular<br>usando vetor                        | $O(1)$  | $O(1)$  |
| fila usando<br>lista encadeada<br>com cabeça e cauda | $O(1)$  | $O(1)$  |

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

|   | push   | pop    |
|---|--------|--------|
| pilha usando vetor                      | $O(1)$ | $O(1)$ |
| pilha usando lista encadeada com cabeça | $O(1)$ | $O(1)$ |