ESTRUTURA DE DADOS LINEARES → agrupar dados sequencialmente

a_{i-1} a_i a_{i+1} N = 0 estrutura vazia

acessar valor
alterar valor
excluir struct
copiar struct
determinar tamanho
juntar duas
s1 + s2 = s3
concatenar
intercalar

CRIAR uma estrutura vazia

INSERIR
- qualquer posição insere
- posição específica insere_inicio e insere_fim
- posição adequada insere_ord

REMOVER
- elem espec. EX 1º ocorrência
- todas ocorrencias de 1 elem espec.
- em uma posição específica

remove_inicio remove_fim

CRITÉRIOS de ordenação

NÃO ORDENADA
{-5, 17, 3, 8}

ORDENADAS
{3, 5, 7, 8} ou {8, 7, 5, 3}

critérios de implementação

ALOCAÇÃO

ESTÁTICA
uso de vetores
espaço determinado
nº max de elementos

DINÂMICA
uso de malloc e ponteiros
uso otimizado da mem.
+ difícil

ACESSO

SEQUENCIAL
aritmética de ponteiros
exige movimentação
POSIÇÕES CONSECUTIVAS

ENCADEADO
QUALQUER ÁREA DA MEMORIA
sem movimentação
preservamento da lista
CADEIA de PONTEIROS

IMPLEMENTAÇÃO

lista vazia est/seq
dina/enca

lista cheia est/seq
dina/enca

concatena dinamica

insere ordenada

lista est/seq ordenada
crescente → decrescente

imprime_lista

cabeçalho - dinâ
ord. e ñ ord } código

intercala com cabeçalho

Scanned with CamScanner

	cria_lista	lista_vazia	lista_cheia	insere_elem	remove_elem	insere_ord	remove_ord
ENTRADA	nenhuma (função tipo Lista)	endereço de uma lista Lista lst	Lista lst	Lista lst + int elem	Lista lst + int elem	Lista lst + int elem	Lista lst + int elem
PRÉ CONDIÇÃO	X	X	X	lista != NULL e não cheia	lista != NULL e não estar vazia	lista != NULL e não cheia	lista != NULL e não vazia
PROCESSO	criar lista e deixar vazia	if lst->Fim == 0	if lst->Fim == MAX	insere no [lst->Fim]	percorrer lista remover deslocamento à esq i-1 → i	posição correta sucessor > que elem	percorrer até achar elem
SAÍDA	endereço da lista return lst;	1 se vazia 0 se não	1 cheia 0 se não	1 sucesso 0 Falha	1 sucesso 0 Falha	1 sucesso 0 Falha	1 sucesso 0 Falha
Tipo função	Lista	int	int	int	int	int	int
PÓS CONDIÇÃO	X	X	X	lst->Fim++	lst->Fim--	lst->Fim++	lst->Fim--

Scanned with CamScanner

VETORES

POSIÇÕES CONSECUTIVAS

Acesso direto INDEXADO

EST/SEQ

NÃO ORDENA.

cria_lista lista_vazia lista_cheia insere_elem remove_elem

Inserção i 0+

1	2	5	3	10	

1	2	5	0	3	10

deslocamento à direita

FIM última posição ocupada ou 1º livre

```
int no[MAX];
```

Remoção i –

1	2	5	0	3	10

1	2	5	3	10	

MAX-1

deslocamento à esquerda

```
#DEFINE MAX 10
struct lista{
    int no[MAX];
    int Fim;
};

typedef struct lista *Lista;
```

C — Estrutura

H

imprime_lista

```
void imprime_lista (Lista lst){
    for (int i=0; i<lst->Fim; i++){
        printf("%d", lst->no[i]);
    }
    printf("\n");
}
```

void tamanho

```
void tamanho(Lista lst){
    printf("%d", lst->Fim);
}
```

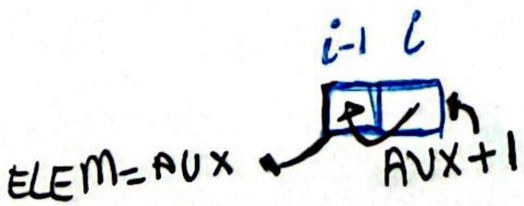
IMPLEMENTAÇÃO ESTÁTICA SEQUENCIAL de listas lineares

remove_elem
- elem entre a lista
- lista vazia
- elem ∉ lista
- último elem

```
int remove_elem(Lista lst, int elem){
    if(lst==NULL || lista_vazia(lst)==1)
        return 0; // folha
    int i, Aux=0;
    // Percorrimento até achar elem ou final da lista
    while(Aux<lst->Fim && lst->no[Aux] != elem)
        Aux++;
    if(Aux==lst->Fim)// ∄ elem
        return 0; // folha
    for(i=Aux+1; i<lst->Fim; i++)
        lst->no[i-1]=lst->no[i];
    lst->Fim --; // -1 elem da lista
    return 1; // sucesso
```

deslocamento a esq. do sucessor até o final

ATUAL SUCESSOR



cria_lista

```
Lista cria_lista(){
    Lista lst;
    lst=(Lista)malloc(sizeof(struct lista));
    if(lst!=NULL) // alocação OK
        lst->Fim=0;
    return lst;
}
```

lista_vazia

```
int lista_vazia(Lista lst){
    if(lst->Fim==0)
        return 1; // Lista vazia
    else
        return 0; // Ñ vazia
}
```

lista_cheia

```
int listacheia(Lista lst){
    if (lst->Fim==MAX)
        return 1; // lista cheia
    else
        return 0; // ñ cheia
}
```

0 max-1 | 1 | 2 | 3 |] max = FIM

insere no final da fila insere_elem

```
int insere_elem (Lista lst, int elem){
    if(lst==NULL || listacheia(lst)==1)
        return 0; // erro de aloca ou listacheia
    lst->no[lst->Fim] = elem; //insere
    lst->Fim++;
    return 1;
```

```
for(int i=lst->fim; i>0; i--){
    lst->no[i]= lst->no[i-
}
lst->no[0] = elem;
lst->Fim++;
```

Scanned with CamScanner

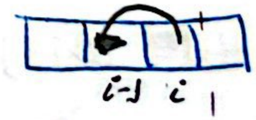
EST/SEQ

ORDENADA

cria lista — lista cheia — insere_ord
lista vazia — remove_ord

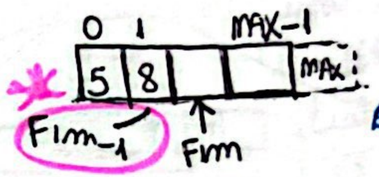
inserção ORDENADA
- elem entre
- lista cheia
- lista vazia
- elem < 1º nó
- elem >= ult

```
int insere_ord (Lista lst, int elem){
    if(lst==NULL || lista_cheia(lst)==1)
        return 0; //falha
    //trata lista vazia ou elem >=ult
    if(lista_vazia(lst)==1 || elem >= lst->no[lst->Fim]){ //Fim-1
        lst->no[lst->Fim]= elem; //insere no final
    }else{
        int i, aux=0;          achar sucessor maior q elem
        while(elem>=lst->no[aux])//percorrimento
            aux++; //achar um elem2 maior que elem
        for(i=lst->Fim; i>aux; i--) //deslocamento   deslocar os elem >elem
            lst->no[i] = lst->no[i-1];
        lst->no[aux] = elem; //insere elem
    }
    lst->Fim++; //avança Fim
    return 1; //sucesso
}
```



IMPLEMENTAÇÃO ESTÁTICA SEQUENCIAL de listas lineares

Lista deve permanecer ord.
Percorrimento da lista



remove_ordenada
- lista vazia
- elem entre
- elem. ult
- elem ∄
- elemento não está na lista
- elem < 1º nó
- elem > ult nó

ENTRE 1º nó < elemento < ult nó

```
int remove_ord (Lista lst, int elem){
    if(lst==NULL || lista_vazia(lst)==1 ||
       elem < lst->no[0] || elem > lst->no[lst->Fim-1])
        return 0; // falha
    int i, Aux=0;
    //Percorrer até achar elem ou nó maior, ou fim
    while(Aux < lst->Fim && lst->no[Aux]<elem)
        Aux++;
    if(Aux==lst->Fim || lst->no[Aux]>elem)   // ∄ elem
        return 0; // falha
    //Deslocamento à esq. do sucessor até o final
    for(i=Aux+1; i<lst->Fim; i++)
        lst->no[i-1] = lst->no[i];
    lst->Fim--;
    return 1;
}
```

Scanned with CamScanner

alocação de elementos individualizados
- informação desejada (valor do elem)
- ponteiro p/ sucessor (ENCADEAMENTO)

garante a formação da lista

nó

fim da lista
prox == NULL

```c
struct no {
    int info;
    struct no *prox;
};
```

C

```h
typedef struct no *Lista;
```

H

Din/Enc NÃO ORDENADA

cria_lista

```c
Lista cria_lista(){
    return NULL;
}
```

L → NULL

lista_vazia

```c
int lista_vazia(Lista lst){
    if (lst == NULL)
        return 1;
    else
    return 0;
}
```

~~lista cheia~~

limitado pelo espaço de memória

IMPLEMENTAÇÃO Dinâmica Encadeada de listas LINEARES

Posições NÃO consecutivos

NEM proporcional ao armazenamento

insere_elem
- lista vazia - lista 1ou +

alteração do end → passagem por referência

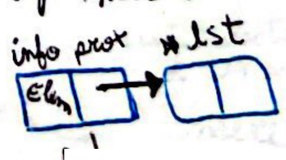
insere_elem

```c
int insere_elem(Lista *lst, int elem){
    // aloca novo nó
    Lista N = (lista) malloc(sizeof(struct no));
    if(N == NULL) {return 0;} //Falha
    N->info = elem; //insere
    N->prox = *lst;
    *lst = N; // aponta p novo no
    return 1;
}
```

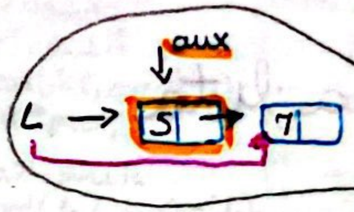
endereço armazenado pela lista

preenchimento do novo nó

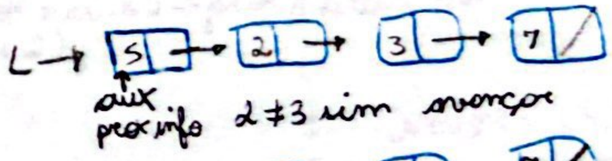
acesso a lista

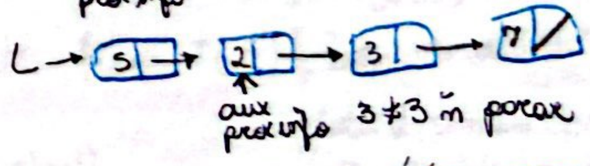


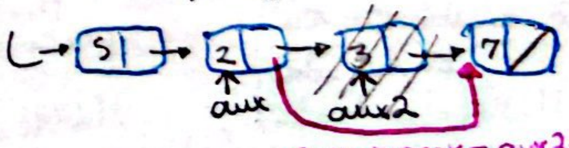
remove_elem
- lista vazia
- elem ∄
- elem = 1º nó
- elem entre 2º e ult




```c
int remove_elem(Lista *lst, int elem){
    if(lista_vazia(*lst) == 1) return 0;
    Lista aux = *lst; //remove = 1º nó
    if (elem == (*lst)->info){
        *lst = aux->prox; // lista aponta p/ 2º nó
        free(aux); //libera memoria alocada
        return 1;
    // Percorrimento até achar o elem ou o final
    while (aux->prox != NULL && aux->prox->info != elem
        aux = aux->prox;        // sucessor de aux
    if(aux->prox == NULL) return 0;
    Lista aux2 = aux->prox; // aponta nó a remover
    aux->prox = aux2->prox; //retira
    free (aux2);
    return 1;
}
```







aux->prox = aux2->pr

```c
int imprime_lista (Lista *lst) {
    Lista aux = *lst;
    if(aux == NULL) return 0;
    while( aux != NULL)
        printf(" %d ", aux->info);
        aux = aux->prox;
    }
    return 1;
```

Scanned with CamScanner

ORDENADA

cria_lista

```
Lista criaLista(){
    return NULL
}
```

lista_vazia

```
int lista_vazia(Lista lst){
    if(lst==NULL)
        return 1;
    else
        return 0;
}
```

insere elemento ordenado

- lista vazia - elem ≤ 1º nó
- elem > ult - elem entre 1º e ult

```
int insere_ord(Lista *lst, int elem){
    Lista N = (lista) malloc(sizeof(struct no));
    if(N==NULL){return 0;}
    N->info = elem;
    if(lista_vazia(*lst)||elem <=(*lst)->info){
    N->prox = *lst; // aponta p/ 1º nó atual
    *lst = N; // lista aponta p/ o novo
    return 1;
    }
    // percorrimento da lista (elem > 1º nó)
    Lista aux = *lst; // aux aponta p/ 1º no
    while (aux->prox != NULL && aux->prox->inf <elem)
        aux = aux->prox; // avança
    N->prox = aux->prox; // se aux é último N->prox = NULL
    aux->prox = N;
    return 1;
}
```

ponteiro auxiliar

insere elem

tamanho

```
int tamanho (Lista *lst){
    Lista aux = *lst;
    int Tam=0;
    while(aux != NULL){
        tam++;
        aux = aux->prox;
    }return tam;
}
```

sucessor de aux

aux->prox->info

concatena

```
Lista concatena(Lista *lst1, Lista *lst2){
    Lista d;
    d=cria_lista();
    Lista aux = lst1;
    Lista aux2 = lst2;
    while(aux != NULL){
        insere_ord(&d, aux->info);
        aux=aux->prox;
    }
    while(aux2 != NULL){
        insere_ord(&d, aux2->info);
        aux2 = aux2->prox;
    }
    return d;
}
```

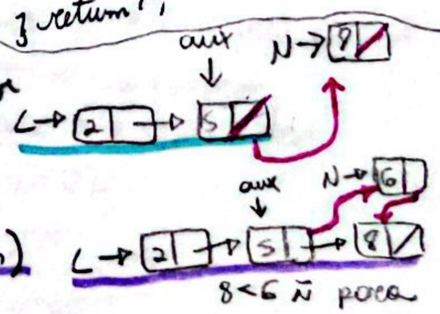
int tam1, int tam2

cabeçalho

```
int cont=0;
while(cont<tam1)
    insere_ord(&d, aux->info)
    aux=aux->prox;
    cont++;
tam=0;
while(cont<tam2)
    insere_ord(&d, aux2->info)
    aux2=aux2->prox;
    cont++;
return d;
```

Implementação DINÂMICA / ENCADEADA de listas lineares

```
int iguais(Lista *lst1, Lista *lst2){
    if(lista_vazia(*lst1)==1 && lista_vazia(*lst2)==1){
        printf("são iguais"); return 0;}
    Lista aux1 = *lst1; Lista aux2 = *lst2;
    while(aux1 != NULL){
        if(aux1->info != aux2->info) printf("≠");
    }
    aux1 = aux1->prox;
    aux2 = aux2->prox;
    printf("=");
}return 1;
```



remove_ord

- lista vazia - elem ult nó
- elem entre - elem ∄ - elem < 1º nó, elem > ult, 1º < elem < ult

```
int remove_ord(Lista *lst, int elem){
    if(lista_vazia(*lst)==1 || elem < (*lst)->info)
        return 0;
    Lista aux = *lst // auxiliar p/ 1º nó
    if(elem==(*lst->info){ // remove 1º nó
        *lst = aux->prox; // lista -> 2º nó
        free(aux);
    }return 1;
```

elem > sucessor

```
// percorrimento até final ou achar elem ou no maior
while(aux->prox != NULL && aux->prox->info < elem)
    aux = aux->prox;
if(aux->prox==NULL)
    return 0;
// remove elem (pos 1º nó)
Lista aux2 = aux->prox; // aponta nó a ser removido
aux->prox = aux2->prox // Retira nó
free(aux2);
return 1;
}
```

```
void imprime_lista (Lista *lst){
    Lista aux;
    aux=*lst;
    if(lista_vazia(lst)==1)
        printf("Lista vazia\n");
    else{
        while(aux != NULL){
            printf("%d", aux->info);
        }} aux = aux->prox;
    }printf("\n");
```

cabeçalho

aux = (*lst) -> prox

intercala

```
Lista intercala(Lista *lst1, Lista *lst2)
Lista d=concatena(lst1, lst2);
Lista intercalada=cria_lista();
Lista aux=d;
while(aux != NULL){
    insere_ord(&intercalada, aux->info);
    aux=aux->prox;
}
return intercalada;
}
```

.H

```
int criaLista();
int listaVazia(Lista lst);
int insere_ord(Lista *lst, int elem);
```

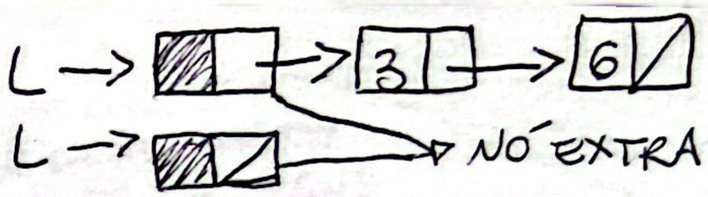
user.C

```
insere_ord(&p, 3);
```

Scanned with CamScanner

NÓ CABEÇALHO

L = {3, 6}
L = { }



↳ alocado dinamicamente

• C
```
struct no {
    int info;
    struct no *prox;
};
```

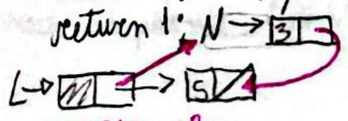
• H
```
typedef struct no *Lista;
```

cria_lista
```
Lista cria_lista( ){
    //aloca nó cabeçalho
    Lista cab;
    cab = (Lista)malloc(sizeof(struct no));
    //coloca lista no estado vazio
    if(cab!=NULL){ //se alocação NÃO falhar
        cab->prox = NULL;
        cab->info = 0;//OPCIONAL guardar qnt
    }
    return cab; // se alocação falhar
                // retorna NULL
}
```

lista_vazia
```
int lista_vazia(Lista lst){
    if(lst->prox ==NULL)
        return 1;
    else
        return 0;
}
```

lista_cheia "lista infinita"
malloc == NULL "sem memória"
DINÂMICA

insere_elem
```
int insere_elem(Lista *lst, int elem){
    Lista N = (Lista) malloc(sizeof(struct no));
    if(N == NULL){return 0;}
    N->info = elem;
    N->prox = (*lst)->prox; //aponta p/ o 1º no atual da lista
    (*lst)->prox = N; //faz o no cab apontar p/ o novo no
    (*lst)->info++; //opcional
    return 1;
}
```



remove_elem
```
int remove_elem(Lista *lst, int elem){
    if(lista_vazia(*lst)==1) return 0;
    Lista aux = *lst;
    //percorrimento até achar elem ou final
    while(aux->prox!=NULL && aux->prox->info!=elem)
        aux = aux->prox;
    if(aux->prox ==NULL) return 0;
    Lista aux2= aux->prox;
    aux->prox = aux2->prox;
    free(aux2);
    ~~return 1;~~
    (*lst)->info--;
    return 1;
}
```

insere_elem_ordenado
insere_ord
```
int insere_ord(Lista *lst, int elem){
    Lista N = (Lista) malloc(sizeof(struct no));
    if(N==NULL) return 0;
    N->info = elem;
    //percorrimento da lista
    Lista aux = *lst; //aponta cabeçalho
    while(aux->prox!=NULL && aux->prox->info<elem)
        aux = aux->prox;
    N->prox = aux->prox;
    aux->prox = N;
    (*lst)->info++;
    return 1;
}
```

remove_ord
```
int remove_ord(Lista *lst, int elem){
    if(lista_vazia(lst)==1) return 0;
    Lista aux = *lst;
    while(aux->prox!=NULL && aux->prox->info<elem)
        aux = aux->prox;
    if(aux->prox = NULL || aux->prox->info>elem)
        return 0;
    Lista aux2 = aux->prox;
    aux->prox = aux2->prox;
    free(aux2);
    (*lst)->info--;
    return 1;
}
```

Scanned with CamScanner

EST/SEQ
Ñ ORDENADA
void menor

```
void menor(Lista lst){
    int menor, pos;
    if (lst->Fim != 0){
        pos=0;
        menor = lst->no[0];
        for (int i=0; i < lst->Fim; i++){
            if (menor > lst->no[i]){
                menor = lst->no[i];
                pos=i;
            }
    }}
    for (int j=pos+1; j< lst->Fim; j++)
        lst->no[j-1]=lst->no[j];

    lst->Fim--;
}
```

ocorrências

```
Lista remove_todos (Lista lst, int elem){
    if(lst == NULL || lista_vazia(lst)==1)
        return NULL;
    Lista aux;
    aux= cria_lista(); // nova lista
    for (int i=0; i<lst->Fim; i++)
        if(lst->no[i] != elem){
            aux->no[aux->Fim]=lst->no[i];
            aux->Fim++;
        }
    }
    free(lst);
    return aux;
}
```

```
Lista concatena (Lista lst1, Lista lst2){
    Lista d; d=cria_lista;
    for (int i=0; i < lst1->Fim; i++){
        d->no[i] = lst1->no[i];
        d->Fim++;
    }
    for (int j=0; j < lst2->Fim; j++){
        d->no[d->Fim]= lst2->no[j];
        d->Fim++;
    }
    return d;
}
```

.H

```
typedef struct lista *Lista
int listavazia(Lista lst);
:
Lista cria_lista();
int insere_elem(Lista lst, int elem);
void imprime_lista(Lista lst);
```

main.c

```
# include "estatica.h"
Lista p;
p = cria_lista();
insere_inicio (p,3);
tamanho(p);
menor(p);

Lista d;
d = cria_lista();
d = concatena(p,q);
```

EST/SEQ
ordenada

```
int menor (Lista lst){
    for(int i=0; i<lst->Fim; i++)
        lst->no[i] = lst->no[i+1];
    lst->Fim--;
    return 1;
}
```

remove_pares

```
Lista remove_pares (Lista lst){
    if(lst == NULL || lista_vazia(lst)==1)
        return NULL;
    Lista aux;
    aux = cria_lista(); // nova lista

    for (int i=0; i < lst->Fim; i++){
        if( lst->no[i] % 2 != 0){
            aux->no[aux->Fim]= lst->no[i];
            aux->Fim++;
        }
    }
    free(lst);
    return aux;
}
```

iguais

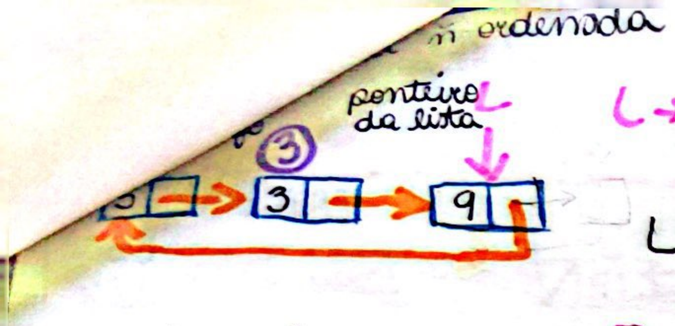
lst1->Fim == lst2->Fim

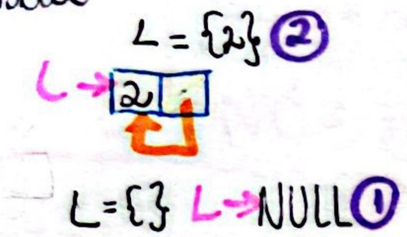
```
int iguais (Lista lst1, Lista lst2){
    if(tamanho(lst1)==tamanho(lst2)){
        for (int i=0; i< lst->Fim; i++){
            if(lst1->no[i] != lst2->no[i])
                return 0;
        }
        return 1;
    }
} return 0;
```

main.c

```
if(iguais(p,q))
    printf("Iguais\n");
else
    printf("Diferentes\n");
```

Scanned with CamScanner






```
struct no{
    int info;
    struct no *prox;
};
typedef struct no * Lista;
```

Insere final

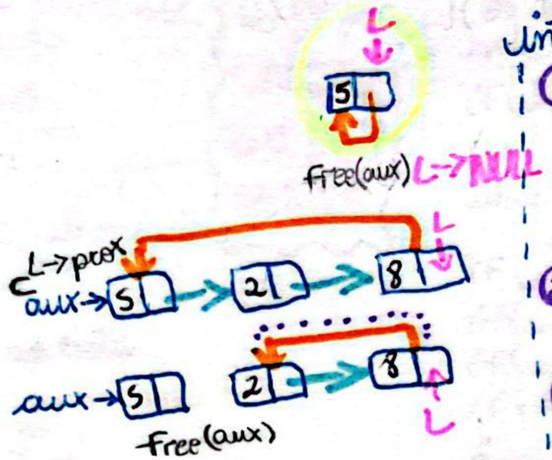
Remove início

"FIFO FILA"

```
Lista cria_lista(){
    return NULL;
}
```

```
int lista_vazia(Lista lst){
    if(lst == NULL)
        return 1; //sucesso
    else
        return 0; //falha
}
```

ENCADEAMENTO! CIRCULAR DINÂMICA ENCADEADA NÃO ORDE.




```
int remove_início(Lista *lst, int *elem){
① // lista vazia
    if(lista_vazia(*lst)==1){return 0;}
    Lista aux = (*lst)->prox; // aux->1º nó
    *elem = aux->info; // retorna valor removido
② if(*lst==(*lst)->prox) //1 nó
        *lst = NULL;
③ else
        (*lst)->prox = aux->prox;
    Free(aux);
    return 1;
}
```

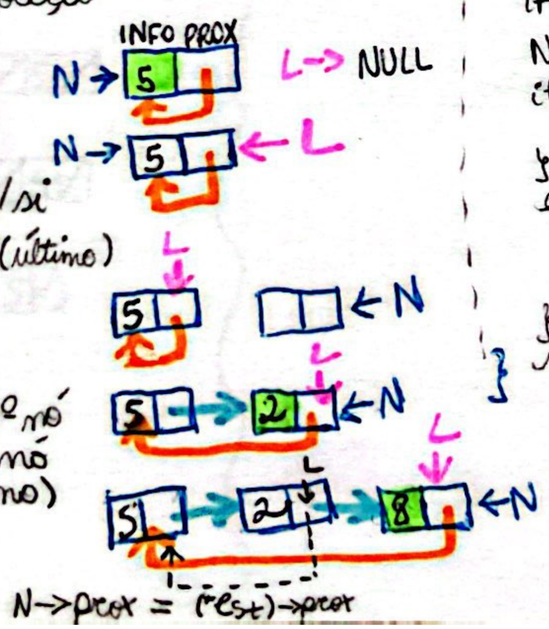
último nó → 1º

ponteiro L → ult nó

L→info info ult no

L→prox→info info 1º nó

```
int insere_final(Lista *lst, int elem){
    //aloca novo nó p/elem
    Lista N=(Lista) malloc(sizeof(struct no));
    if(N==NULL){return 0;}// falha alocação
    N->info = elem; // insere elem
    // Trata lista vazia
    if(lista_vazia(*lst)==1){
① N->prox = N; // novo nó aponta p/si
        *lst = N; // lista aponta novo nó(último)
    }
    // Trata 1 ou + elem
    else{
② N->prox = (*lst)->prox; // novo nó -> 1º nó
        (*lst)->prox = N; // ult. nó -> novo nó
        *lst = N; // lista -> novo nó (último)
③ }
    return 1;
```




```
int insere_ini(Lista *lst, int elem){
    Lista N=(Lista)malloc(sizeof(struct no));
    if(N==NULL) return 0;
    N->info = elem;
    if(lista_vazia(*lst)==1){
        N->prox = N;
        *lst = N;
    }
    else{
        N->prox = (*lst)->prox;
        (*lst)->prox = N;
    }
    return 1;
}
```

Scanned with CamScanner

ENCADEAMENTO DUPLO

Cada nó conhece seu sucessor e antecessor na lista

NÓ = | ANT | INFO | PROX |

DINÂMICA ENCADEADA ORDENADA

L={2}

L={ } → NULL

L → [| 2 |]

L → [| 5 |] → [| 3 |]

```
struct no {
    int info;
    struct no * prox;
    struct no * ant;
};
typedef struct no * Lista;
```

```
Lista cria_lista(){
    return NULL;
}
```

```
int lista_vazia(Lista lst){
    if(lst==NULL)
        return 1;
    else
        return 0;
}
```

Remove elemento

L={ }

elem ∄ na lista

{ L um nó,
elem = 1º nó,
elem = ult nó,
1º nó < elem < ult nó

```
int remove_elem(Lista *lst, int elem){
    if(lista_vazia(*lst) == 1) {return 0;}
    Lista aux = *lst;
    while (aux->prox != NULL && aux->info != elem)
        aux = aux->prox;
    if (aux->info != elem) return 0;
    if(aux->prox != NULL)(aux)->prox->ant = aux->ant;
    if(aux->ant != NULL)(aux)->ant->prox = aux->prox;
    if(aux == *lst) *lst = aux->prox;
    Free(aux);
    return 1;
}
```

```
int insere_elem(Lista *lst, int elem){
    // aloca novo nó
    Lista N=(Lista) malloc(sizeof(struct no));
    if (N==NULL){return 0;}
    N->info = elem; //insere
    N->ant = NULL; // novo nó n tem ant
    N->prox = *lst; //sucessor = end. lista
    if (lista_vazia(*lst)==0)//n vazia
        (*lst)->ant = N;
    *lst = N;
    return 1;
}
```

L vazia ou L com elementos

N → [| 5 |] L → NULL

[| 5 |] ← L

N → [| | |] L → [| 5 |]

N → [| 2 |] → [| 5 |] ← L

L → [| 2 |] → [| 5 |]

Estruturas lineares

Deque

Fila dupla c/ entrada restrita

Scanned with CamScanner

PIA E/S

topo ← TOPO

ult posi ocupada

vazia

T → -1

T+1 prxima livre

```
#define max 20
struct pilha{
    int vetor[max];
    int topo;
};

typedef struct pilha *Pilha;

Pilha cria_pilha(){
    Pilha p;
    p=(Pilha)malloc(sizeof(struct pilha));
    if (p!=NULL)
        p->topo=-1;
    return p;
}

int pilha_vazia(Pilha p){
    if (p->topo==-1)
        return 1;
    else
        return 0;
}

int pilha_cheia(Pilha p){
    if (p->topo==max-1)
        return 1;
    else
        return 0;
}
```

```
push(Pilha p, int elem){
    if(p==NULL || pilha_cheia(p)==1)
        return 0;
    p->topo++;
    p->vetor[p->topo]=elem;
    return 1;
}

pop(Pilha p, int *elem){
    if (p==NULL || pilha_vazia(p)==1)
        return 0;
    *elem=p->vetor[p->topo];
    p->topo--;
    return 1;
}

letopo(Pilha p, int *elem){
    if(p==NULL || pilha_vazia(p)==1)
        return 0;
    *elem=p->vetor[p->topo];
    return 1;
}
```

Din ENC

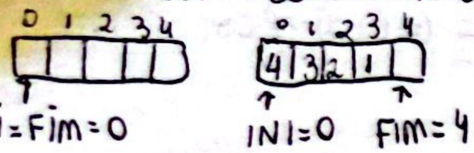
```
struct no{
    int info;
    struct no* prox;
};

typedef struct no *Pilha;

Pilha cria_pilha(){
    return NULL;
}

int pilha_vazia(Pilha p){
    if (p==NULL) return 1;
    else return 0;
}

int push(Pilha *p, int elem){
    Pilha N=(Pilha)malloc(sizeof(struct no));
    if (N==NULL) return 0;
    N->info=elem;
    N->prox=*p
    *p=N; return 1;
}

int pop(Pilha *p, int *elem){
    if(pilha_vazia(*p)==1) return 0;
    Pilha aux=*p;
    *elem=aux->info;
    *p=aux->prox;
    free(aux);
    return 1;
}

int letopo(Pilha *p, int *elem){
    if(pilha_vazia(*p)==1) return 0;
    *elem=(*p)->info;
    return 1;
}
```

Scanned with CamScanner

EST/SEQ

Execução uma posição

max-1 elementos ult=[/]




```
# define max 20
  struct fila {
      int vetor[MAX];
      int ini, fim;
  };
typedef struct fila *Fila;
```

```
Fila criafila(){
  Fila f;
  f=(Fila)malloc(sizeof(struct fila));
  if (f!=NULL){
      f->ini=0;
      f->fim=0;
  }
}return f;
```

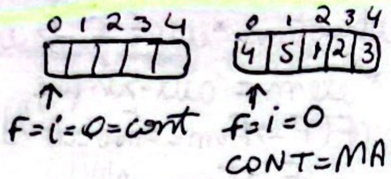
```
int filavazia(Fila f){
  if(f->ini==f->fim)
      return 1;
  } else return 0;
```

```
int filacheia(Fila f){
  if(f->ini==(f->fim+1)%max)
      return 1;
  else
      return 0;
}
```

```
int insere_fim(Fila f, int elem){
  if (filacheia(f)==1)return 0;
  f->vetor[f->fim]=elem;
  f->fim=(f->fim+1)%max;
}return 1;
```

```
int remove_ini(Fila f; int *elem){
  if(filavazia(f)==1)return 0;
  *elem=f->vetor[f->ini];
  f->ini=(f->ini+1)%max;
}return 1;
```

Com contador




```
# define max 20
  struct fila {
      int vetor[max];
      int ini, cont; };
Fila criafila(){
  Fila f=(Fila)malloc(sizeof(struct fila));
  if(f!=NULL){
      f->ini=0;
      f->cont=0;
  }
}return f;
```

```
int filavazia(Fila f){
  if(f->cont==0) return 1;
}else return 0;
```

```
int filacheia(Fila f){
  if(f->cont==MAX)return 1;
}else return 0;
```

```
int insere_fim(Fila f, int elem){
  if(filacheia(f)==1) return 0;
  f->vetor[(f->ini+f->cont)%max]=elem;
  f->cont++;
}return 1;
```

```
int remove_ini(Fila f, int *elem){
  if(filavazia(f)==1)return 0;
  *elem=f->vetor[f->ini];
  f->ini=(f->in+1)%max;
  f->cont--;
}return 1;
```

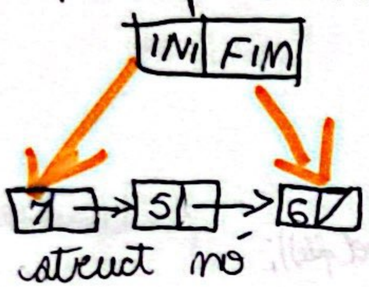
Scanned with CamScanner

Fila Din/En

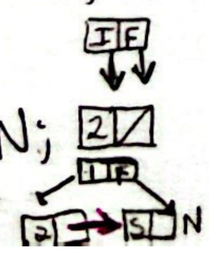
Encadeamento simples

```
struct no{        info - valor do elem
  int info;       prox ->
  struct no *prox;
}
struct fila{
  struct no *ini;
  struct no *fim;
};  typedef struct fila* Fila;
```

struct fila é um nó




```
Fila criafila(){
  Fila f=(Fila)malloc(sizeof(struct fila));
  if (f!=NULL){
    f->ini= NULL;
    f->fim= NULL;
  }
  return f;
}

int filavazia(Fila f){
  if(f->ini==NULL)
    return 1;
  else
    return 0;
}

int insere_fim (Fila f, int elem){
  struct no *N;
  N=(struct no*) malloc(sizeof(struct no));
  if(N==NULL) return 0;
  N->info= elem;
  N->prox= NULL;
  if( filavazia(f)==1) f->ini = N;
  else (f->fim)->prox= N
  f->fim= N; return 1;
}
```




```
int remove_inicio(Fila f, int *elem){
  if(filavazia(f)==1) return 0;
  struct no *aux = f->ini;
  *elem = aux->info;
  if(f->fim==f->ini) //1 nó
    f->fim==NULL;
  f->ini = aux->prox;
  free(aux);
  return 1;
}
```

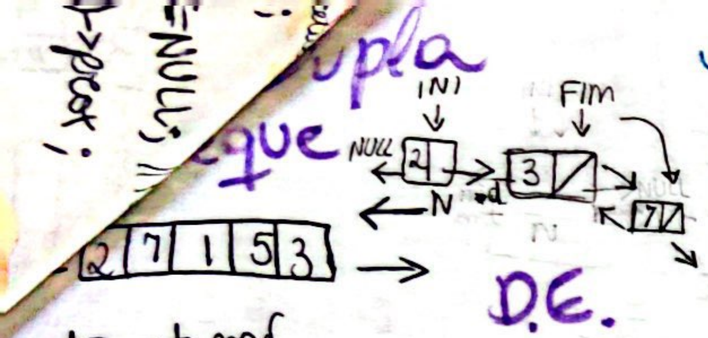
```
int remove_inicio (Fila *f){
  if( filavazia(*f)==1) return
  Fila aux= (*f)->prox;
  *elem = aux->info;
  if(*f==(*f)->prox) *f=NULL; //1 elem
  else
    (*f)->prox = (*aux)->prox;
  free(aux);
  return 1;
```

Encadeamento circular

```
struct no{        typedef struct no*Fila;
  int info;
  struct no*prox;
}
Fila criafila(){return NULL;}

int filavazia(Fila f){
  if(f==NULL) return 1;
  else return 0;
}

int insere_fim (Fila *f, int elem){
  Fila N=(Fila)malloc(sizeof(struct no));
  N->info = elem;  if(N==NULL){return 0};
  if(filavazia(*f)==1){
    N->prox=N;
    *f=N;
  }
  else{
    N->prox=(*f)->prox;
    (*f->prox)=N;
    *f=N;
  }
  return 1;
}
```

Scanned with CamScanner

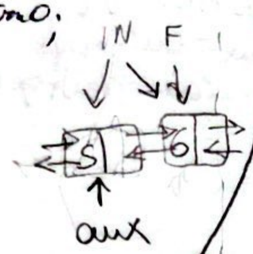
Dupla Deque



D.E.

```
struct no{
    int info;
    struct no *prox;
    struct no * ant;
};
struct posição {
    struct no *ini;
    struct no *fim;
};
typedef struct posição *Deque;
Deque criadeque(){
    Deque iniefim;
iniefim=(Deque)malloc(sizeof(struct posição));
if(iniefim != NULL){
    iniefim->fim = NULL;
    iniefim->ini = NULL;
}
return iniefim;
}
int dequevazia(Deque d){
    if(d->ini==NULL)
    return 1;
    else return 0;
}
```

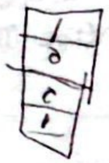
```
int insereinicio(Deque *d, int elem){
    struct no * N=(struct *no) malloc...si..(struct no));
    if(N==NULL) return 0;
    N->info=elem;
    if(deque vazia(*d)==1){
        N->ant=NULL;
        N->prox=NULL;
        (*d)->ini=N;
        (*d)->fim=N;
    }else{
        N->ant=NULL ~~(*d)->ini->ant~~;
        N->prox=(*d)->ini;
        (*d)->ini=N;
    }
    return 1;
}
```




```
int insereFim(Deque *d, int elem){
    mesma coisa
    else{
        N->ant=(*d)->fim
        N->prox=NULL ~~(*d)->fim->prox~~;
        (*d)->fim = N; NULL
    }
    return 1;
}
```

```
void imprime(Deque *d){
    struct no aux=(*d)->ini;
    while(aux != (*d)->fim){
        printf("%d", aux->info);
        aux = aux->prox;
    }
    printf("%d", aux->info);
}
if(Deque vazio(d)==1) ~~return~~ print (vazio)
else
    struct no *aux=(*d)->ini;
    while(aux != NULL)
        print aux->info
        aux = aux->prox
```

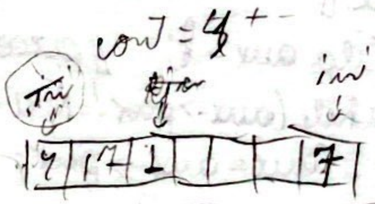
```
int removeinicio(Deque *d, int *elem){
    if(deque vazia(*d)==1) return 0;
    *elem=(*d)->ini->info;  struct no * aux=(*d)->ini;
    if((*d)->ini==(*d)->fim){ //1 elem
        (*d)->ini=NULL; (*d)->fim=NULL;
        free(aux);
    }else{
        (*d)->ini=aux->prox;
        (*d)->ini->ant=NULL;
        free(aux);
    }
    return 1;
}
```



```
int removefim(Deque *d, int *elem){
    mesma coisa
    else{  *elem = aux->info;
aux=(*d)->fim = aux->ant;
    (*d)->fim->prox=NULL;
    free(aux);
}
return 1;
}
```


[ ]

cont = 4



ini = (ini - 1 + MAX) % MAX;

[ordem 7-d] aux <- d = [ordem 4-d] aux <- d

[ordem 7-d] aux <- d = 7 2 0 9

--?

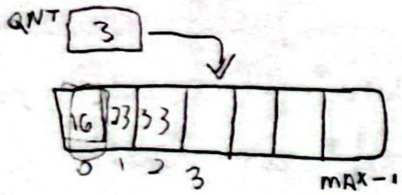
FILA Prioridade

ascendente — ↳ menor priori sai

descendente — ↳ maior priori sai

Ñ ORD - insere no final (fila simples)
remove remove >/< elem

ORD = insere ord
remove no início (fila simples)

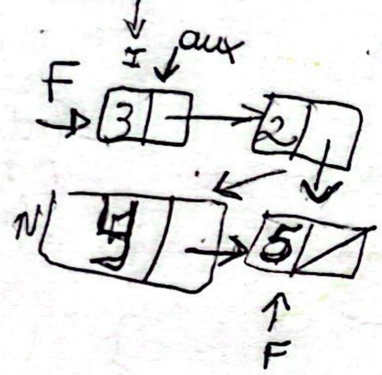



```
struct no{
    int info;
    struct no *prior;
};
struct fila{
    struct no* ini;
    struct no* fim;
};
typedef struct fila* Fila
```

```
Fila criafila(){
Fila f=(Fila) malloc(sizeof(struct fila));
if(f!=NULL){
    f->ini=NULL;
    f->fim=NULL;
    return f;
}
int filavazia(Fila f){
    if(f->ini==NULL)
        return1;
    else
        return0;
}
```

```
void print (Fila f)
Fila aux=f;
while(aux->ini = NULL){
        print (aux->ini)->info
        aux->ini=(aux->ini)->prior;
```

```
int insere_ord(Fila f, int elem){
struct no *N=(struct no*)malloc(sizeof(struct no));
if(N==NULL) return 0;
N->info =elem;
if (filavazia(f)==1){
    f->ini=N;
    N->prior=NULL;
} f->fim=N;
else{
    Fila aux=f;   //percorrimento
    while (aux->prior!=NULL && aux->prior->info <elem){
        aux=aux->prior;
    N->prior=aux->prior;
} aux->prior=N;
return 1;
}
```



asc en

aux->prior->info > elem

desc en

```
int remove_ini (Fila f; int *elem){
    if(filavazia(f)==1) return 0;
    struct no *aux= f->ini;
    *elem = aux->info;
    if(f->ini==f->fim)
        f->fim=NULL;
    f->ini=aux->prior;
    free (aux);
    return 1;
}
```

Fila ES

```
struct fila{
    int no[max];
    int ini, fim;
};
```

↓ cont

desperdício de posição

Fila D/E

Encadeamento circular

```
struct no{
    int info;
    struct no* prior;
};
struct fila{
    struct no* ini;
    struct no* fim;
};
```

Encadeamento simples e prioridade

Deque E/S

```
struct deque{
    int no[max];
    int ini, fim, cont;
};
```

Deque D/E

```
struct no{
    int info;
    struct no*prior;
    struct no* ant;
};
struct deque{
    struct no* ini;
    struct no* fim;
}
```

Pilha E/S

```
struct pilha{
    int vet[max];
    int topo;
};
```

Pilha D/E

```
struct no{
    int info;
    struct no*prior;
};
```

Scanned with CamScanner

E.S.

desperdício de posição

```
struct deque {
    int vet[max];
    int ini, fim, cont;
};
typedef struct deque * Deque;

Deque cria_deque(){
    Deque d;    # define max 10
    d = (Deque) malloc(sizeof(struct deque));
    if(d != NULL){
        d->ini = 0;
        d->fim = 0;
        d->cont = 0;
    }
    return d;
}

int deque_vazia(Deque d){
    if(d->cont == 0) if(d->ini == d->fim)
        return 1;
    else
        return 0;
}

int deque_cheia(Deque d)
    if(d->cont == MAX) if(d->ini == (d->fim+1)%MAX)
        return 1;
    else
        return 0;
}

int insere_inicio(Deque d, int elem){
    if(deque_cheia(d) == 1)
        return 0;
    if(d->fim == 0)
        d->ini = MAX-1;
    if(d->fim > 0)
        d->ini = d->ini-1;
    d->vet[d->ini] = elem;
    d->cont++;
}

int insere_final(Deque, d, int elem){
    if(deque_cheia(d) == 1)
        return 0;
    d->vet[d->fim] = elem;
    d->fim = (d->fim -1) % MAX;
    d->cont ++;
}

int remove_ini(Deque d, int *elem){
    if(deque_vazia(d) == 1)
        return 0;
    *elem = d->vet[d->ini];
    d->ini = (d->ini-1) % MAX;
    d->cont--;
    return 1;
}

int remove_final(Deque d, int *elem){
    if(deque_vazia(d) == 1)
        return 0;
    *elem = d->vet[d->fim];
    d->fim --;
    if(d->fim < 0)
        d->fim = MAX-1;
    d->cont --;
}
```

Scanned with CamScanner