



# Estruturas de Dados I

## Estruturas de Dados - Listas Encadeadas

Prof. Dr. Lidio Mauro Lima de Campos

[limadecampos@gmail.com](mailto:limadecampos@gmail.com)

**Universidade Federal do Pará – UFPA**

**ICEN**

**PPGCC**

# Agenda

- Listas Encadeadas.
- Representações.
- Funções de Acesso.

# Introdução

- Vetores são úteis quando sabemos o número exato (ou aproximado) de elementos que usaremos.
  - Espaço Contíguo de Memória.
  - Acesso Randômico aos elementos.

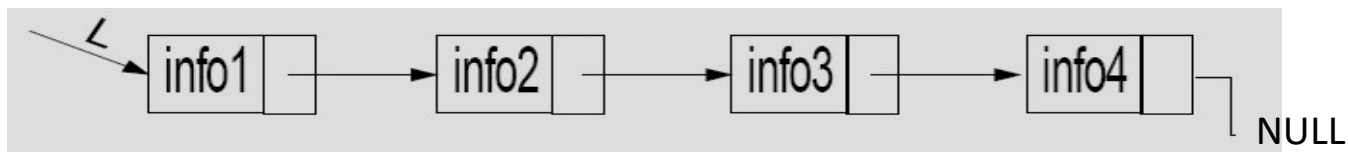


- Em geral, precisamos trabalhar com estruturas de dados dinâmicas que crescem (ou decrescem) à medida que elementos são inseridos (ou removidos).



# Listas Encadeadas

- Seqüência encadeada (via ponteiros) de elementos, chamados de *nós da lista*.
- Cada *nó da lista* é representado por dois campos:
  - a informação armazenada e
  - o ponteiro para o próximo elemento da lista.
- A lista é representada por um ponteiro para o primeiro *nó* o ponteiro do último elemento é NULL.

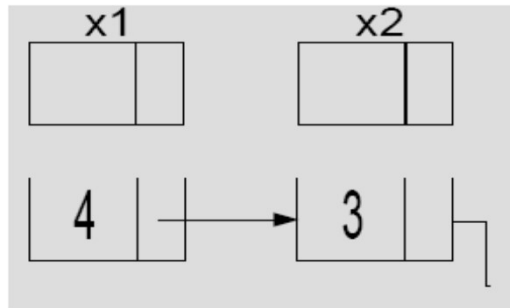


# Definição de uma Lista

- Considere uma lista encadeada armazenando valores inteiros.

```
typedef struct lista //Representação do nó da lista
{
    //Estrutura auto referenciada
    int info;        //informação armazenada na Lista
    Lista *prox;     //ponteiro para a próxima estrutura do mesmo tipo
} Lista;
```

```
Lista x1,x2;
x1.info= 4;
x1.prox= &x2;
x2.info= 3;
x2.prox=NULL;
```



//a estrutura de lista encadeada é representada pelo ponteiro para seu primeiro elemento (tipo Lista\*)

# Tipo Abstrato de Dados Lista Encadeada

- **typedef struct lista Lista;**

/\* Cria uma lista vazia.\*/

- **Lista\* lst\_cria();**

/\* Testa se uma lista é vazia.\*/

- **int lst\_vazia(Lista \*l);**

/\* Insere um elemento no início da lista.\*/

- **Lista\* lst\_insere(Lista \*l, int info);**

/\* Busca um elemento em uma lista.\*/

- **Lista\* lst\_busca(Lista \*l, int info);**

/\* Imprime uma lista.\*/

- **void lst\_imprime(Lista \*l);**

/\* Remove um elemento de uma lista.\*/

- **Lista\* lst\_remove(Lista \*l, int info);**

/\* Libera o espaço alocado por uma lista.\*/

- **void lst\_libera(Lista \*l);**

# Lista Encadeada

- Definindo o tipo Lista e implementando todas as funções.

```
#include<stdio.h>
#include<stdlib.h>
struct lista
{
    int info;
    Lista *prox;
};
```

# Lista Encadeada

- **Função que cria uma lista vazia, representada pelo ponteiro NULL.**

```
/*Cria uma lista vazia.*/  
Lista* lst_cria() //valor de retorno uma lista sem elementos  
{  
    return NULL;  
}
```

- Como a lista é representada pelo ponteiro para o primeiro elemento, uma lista vazia é representada pelo ponteiro NULL.
- **Função que testa se uma lista é vazia, retornando 1 e 0, caso contrário.**

```
/*Testa se uma lista é vazia.*/  
int lst_vazia(Lista *l)  
{  
    // Lista vazia return 1,  
    return (l==NULL);      // Lista não vazia return 0.  
}
```



# Lista Encadeada

- Função que insere elemento no início da lista
- **(CASO 1 – LISTA ESTA VAZIA)**

```
Lista* lst_insere(Lista *l, int info)
{
    Lista* ln = (Lista*)malloc(sizeof(Lista));
    ln->info = info;
    ln->prox = l;
    return ln;
}
```

**l = NULL** após a criação da lista

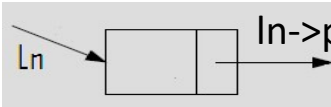


Diagram illustrating the state of the linked list after the first step of insertion (allocating a new node). The pointer `ln` points to a new node. The node's `info` field is empty, and its `prox` pointer points to `ln->prox`.

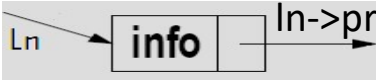


Diagram illustrating the state of the linked list after the second step of insertion (assigning the value to the node). The pointer `ln` points to a node whose `info` field contains the value `info`. The `prox` pointer still points to `ln->prox`.


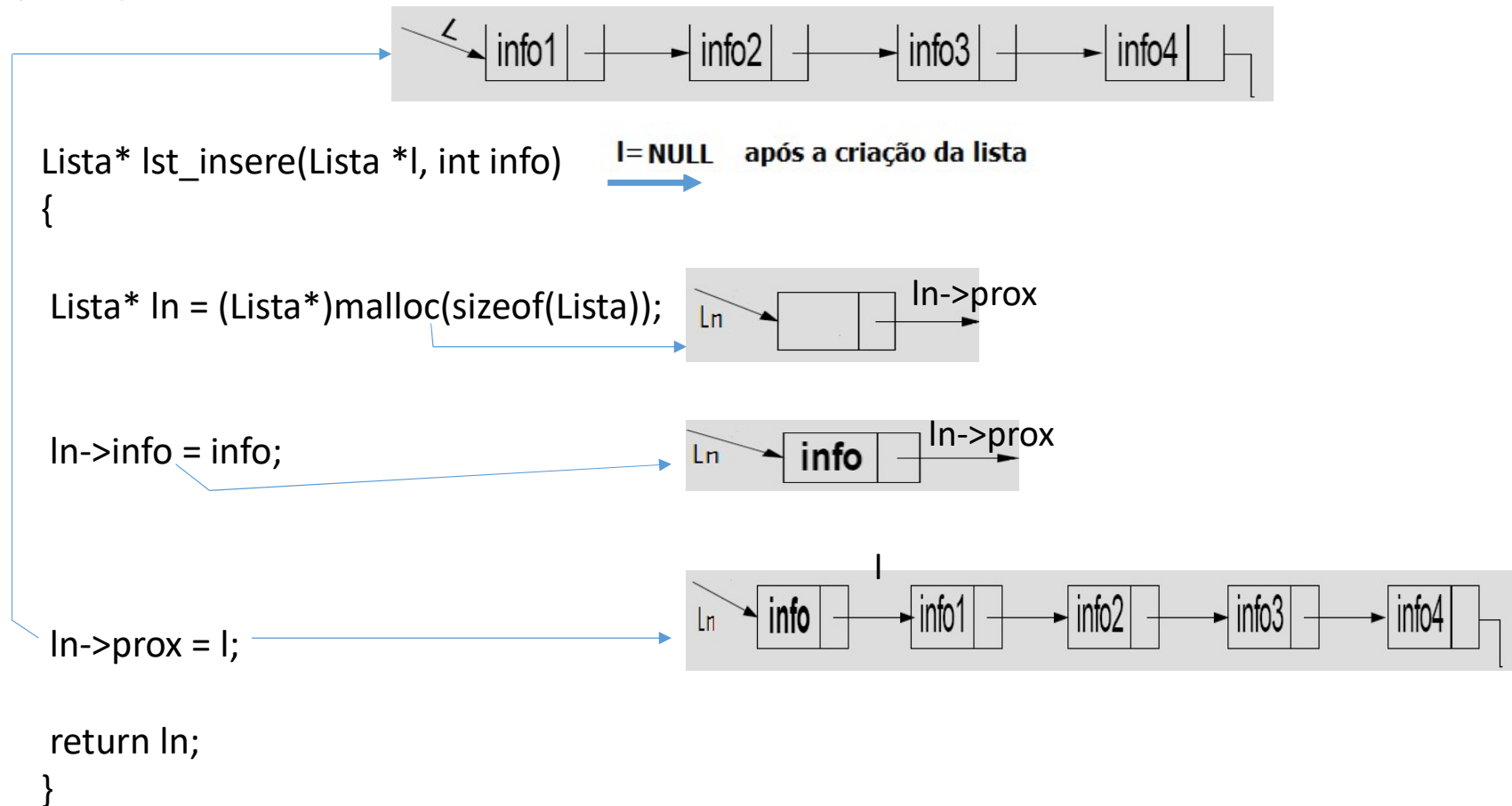


Diagram illustrating the state of the linked list after the third step of insertion (setting the next pointer to NULL). The pointer `ln` points to a node whose `info` field contains the value `info`. The `prox` pointer is now `NULL`.

# Lista Encadeada

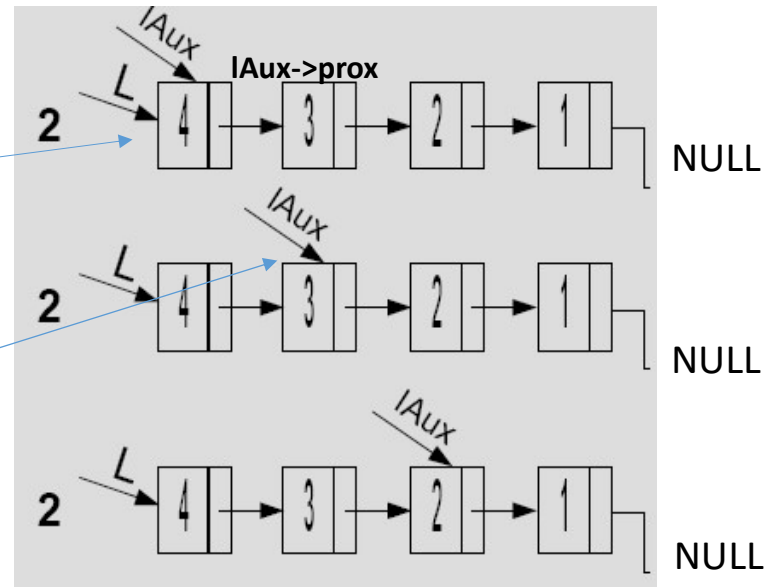
- Função que insere elemento no início da lista (CASO 2 – A LISTA NÃO ESTA VAZIA).



# Lista Encadeada

- Função que imprime uma lista, ou seja, percorre elemento a elemento, imprimindo os elementos.

```
/* Imprime uma lista.*/  
void lst_imprime(Lista *l)  
{  
    Lista* lAux = l;  
    while(lAux!=NULL)  
    {  
        printf("Info = %d\n",lAux->info);  
        lAux = lAux->prox;  
    }  
}
```



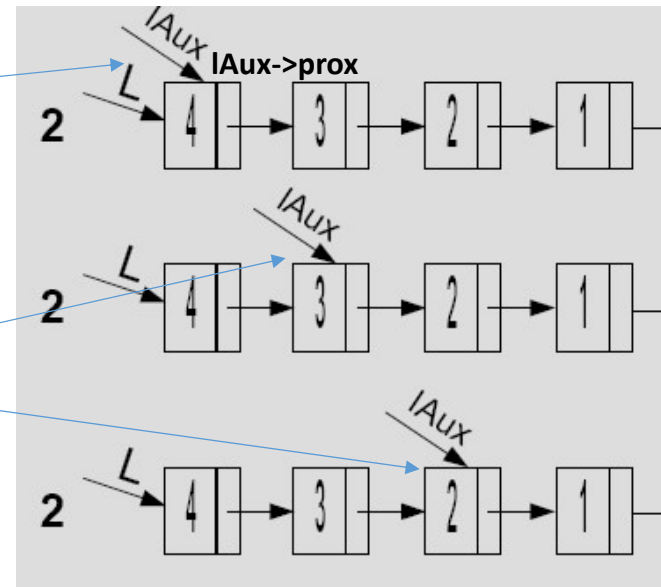
# Lista Encadeada

- Função que busca se um dado elemento pertence a uma lista.

```
/* Busca um elemento em uma lista.*/
```

```
Lista* lst_busca(Lista *l, int info)
```

```
{  
  Lista* lAux = l;  
  while(lAux!=NULL)  
  {  
    if(lAux->info==info)  
      return lAux;  
    lAux = lAux->prox;  
  }  
  return NULL;  
}
```



# Lista Encadeada

Lista\* lst\_remove(Lista \*l, int info) **//CASO 1 - Elemento a ser retirado é o primeiro**

**{ if(l!=NULL) //IAux sempre aponta para o elemento atual da pesquisa sequencial, até encontrar o desejado a ser removido**

**{ Lista\* lAux = l->prox;**

**if(l->info==info){ //primeiro elemento será retirado**

**free(l);**

**return lAux;**

**}**

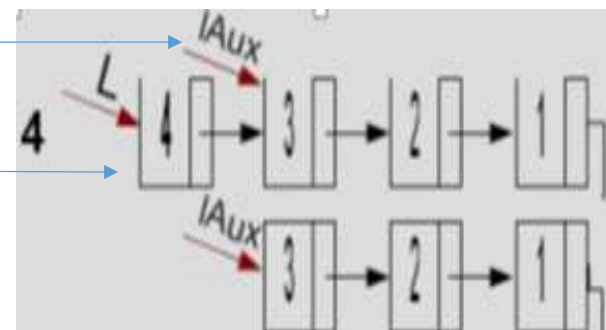
**else**

**.....**

**}**

**return l;**

**}**



# Lista Encadeada

```
Lista* lst_remove(Lista *l, int info)
```

```
{ if(l!=NULL)
```

```
{ Lista* lAux = l->prox;
```

..... # elemento a ser removido não é o primeiro da lista

```
else
```

```
{ Lista* lAnt = l; //não é o primeiro que será retirado, pesquisa sequencialmente elemento
```

```
while(lAux!=NULL)
```

```
{
```

```
if(lAux->info == info){
```

```
lAnt->prox = lAux->prox; free(lAux);
```

```
break;}
```

```
else
```

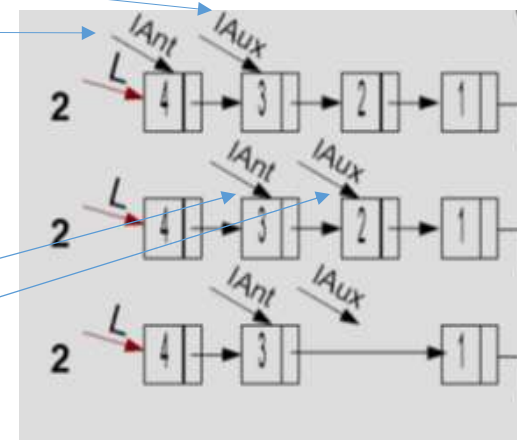
```
{ //enquanto não encontra elemento a remover (lAnt e lAux são movidos frente)
```

```
lAnt = lAux;
```

```
lAux = lAux->prox;}
```

```
}
```

```
} return l; }}
```



# Lista Encadeada

```
Lista* lst_remove(Lista *l, int info)
```

```
{ if(l!=NULL) //lista não é vazia , lAux sempre aponta para o elemento atual da pesquisa sequencial, até encontrar o desejado a ser removido
```

```
{ Lista* lAux = l->prox;
```

```
  if(l->info==info){ //primeiro elemento será retirado
```

```
    free(l); return lAux;
```

```
  }
```

```
  else
```

```
{ Lista* lAnt = l; //não é o primeiro que será retirado, pesquisa sequencialmente elemento
```

```
  while(lAux!=NULL)
```

```
{
```

```
  if(lAux->info == info){
```

```
    lAnt->prox = lAux->prox; free(lAux);
```

```
    break;}
```

```
  else
```

```
{ //enquanto não encontra elemento a remover (lAnt e lAux são movidos frente)
```

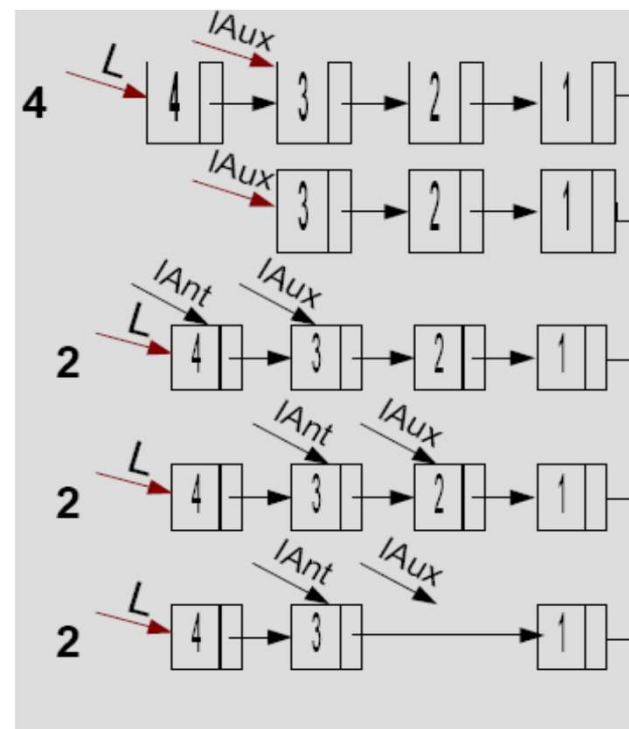
```
  lAnt = lAux;
```

```
  lAux = lAux->prox;}
```

```
}
```

```
}
```

```
  return l; }}
```



# TAD Lista Encadeada

- Função que libera o espaço alocado por uma lista

/\* Libera o espaço alocado por uma lista.\*/

void lst\_libera(Lista \*l)

{

Lista\* lprox;

while(l!=NULL)

{

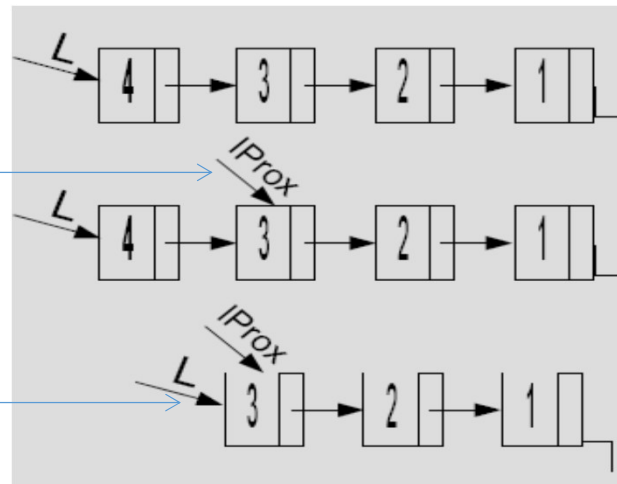
lprox = l->prox;

free(l);

l = lprox;

}

}



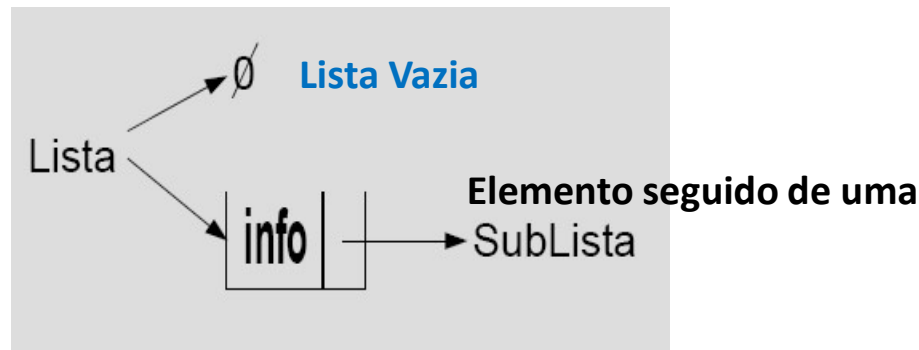


# Lista Encadeada

```
int main (void)
{
    Lista* l = lst_cria();
    l = lst_insere(l,10);
    l = lst_insere(l,20);
    l = lst_insere(l,25);
    l = lst_insere(l,30);
    l = lst_remove(l,10);
    lst_imprime(l);
    return 0;
}
```

## Definição Recursiva de Lista Encadeada

- Podemos dizer que um lista encadeada é representada por:
  - Uma **Lista vazia**; ou
  - Um **Elemento seguido por uma sublista**.



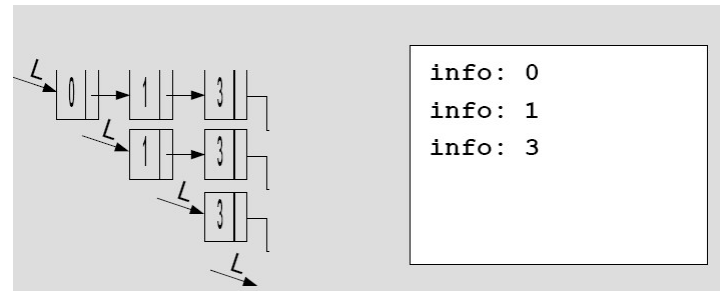
- Nesse último caso, o segundo elemento da lista representa o primeiro da sublista.

# Implementação Recursiva – Função Imprime Lista

- Deve-se usar a definição recursiva, verifica-se se a lista é vazia, caso contrário é composta pelo primeiro nó, dado por l e por uma sublista, dada por l->prox.

- void lst\_imprime\_rec(Lista\* l)

```
{  
  if(lst_vazia(l))  
    return;  
  else  
  {  
    printf("info: %d\n",l->info);/*imprime primeiro elemento*/  
    lst_imprime_rec(l->prox); /*imprime sub-lista*/  
  }  
}
```



# Implementação Recursiva - Função Retira Recursiva

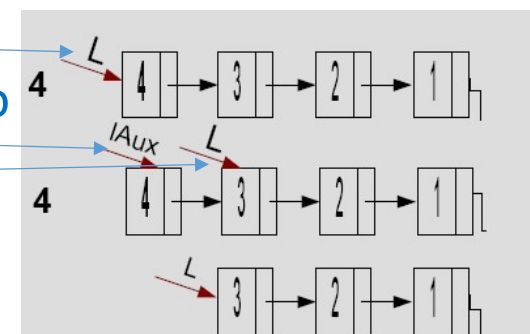
```
Lista* lst_remove_rec(Lista *l, int info){
```

```
if(!lst_vazia(l))
```

**CASO 1 - Elem. a ser removido é o primeiro da Lista**

```
if(l->info==info)
```

```
{Lista* lAux = l; //lAux apontará para elem. a ser removido
```



```
l = l->prox;
```

```
free(lAux);
```

```
}else
```

```
{l->prox=lst_remove_rec(l->prox,info);
```

```
}
```

```
return l;
```

```
}
```

# Implementação Recursiva - Função Retira Recursiva

```
Lista* lst_remove_rec(Lista *l, int info)
```

```
{//a cada chamada recursiva l aponta para l-prox
```

```
if(!lst_vazia(l))
```

```
if(l->info==info)//deseja remover o nó 2
```

```
{ Lista* lAux = l;//elem a ser removido
```

```
l = l->prox;
```

```
free(lAux);
```

```
}else
```

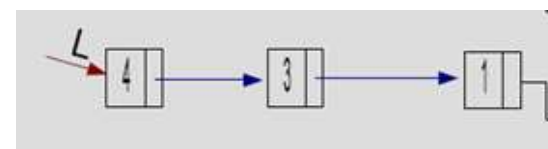
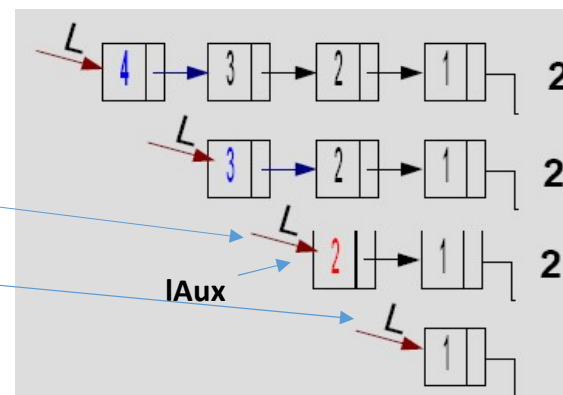
```
{l->prox=lst_remove_rec(l->prox,info);
```

```
}
```

```
return l;
```

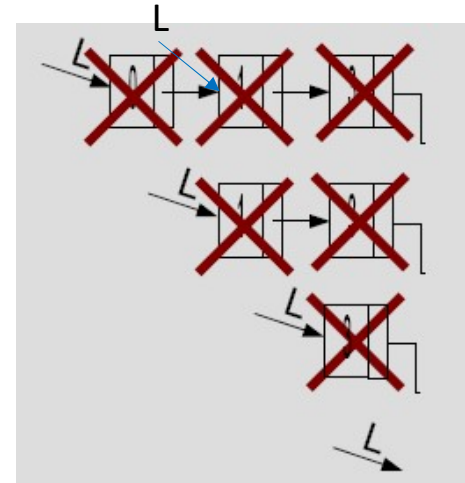
```
}
```

CASO 2 – Elemento está no meio da lista



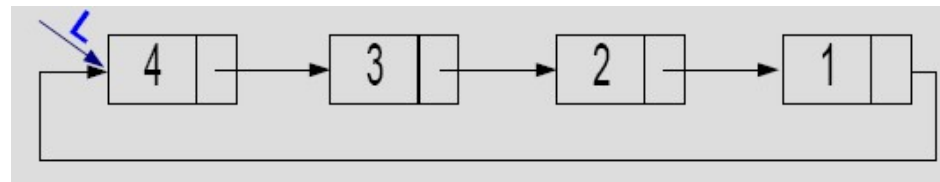
# Implementação Recursiva - Função Libera

```
void lst_libera_rec(Lista *l)
{
    if(!lst_vazia(l))
    {
        lst_libera_rec(l->prox);
        free(l);
    }
}
```

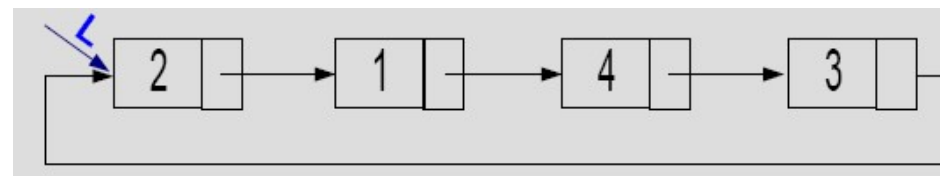


# Listas Circulares

- Algumas aplicações necessitam representar **conjuntos cíclicos**.
- Em uma lista circular **o último elemento tem o primeiro elemento como próximo**.



- A lista pode então ser representada por qualquer elemento da lista.



# Lista Circular

- Funções de acesso: nome do tipo e os protótipos.

```
typedef struct lista_circ ListaCirc;  
/* Cria uma lista circular vazia.*/  
ListaCirc* lst_circ_cria();  
/* Testa se uma lista circular é vazia.*/  
int lst_circ_vazia(ListaCirc *l);  
/* Insere um elemento em uma lista circular.*/  
ListaCirc* lst_circ_insere(ListaCirc *l, int info);  
/* Busca um elemento em uma lista circular.*/  
ListaCirc* lst_circ_busca(ListaCirc *l, int info);  
/* Imprime uma lista circular.*/  
void lst_circ_imprime(ListaCirc *l);  
/* Remove um elemento de uma lista circular.*/  
ListaCirc* lst_circ_remove(ListaCirc *l, int info);  
/* Libera o espaço alocado por uma lista circular.*/  
void lst_circ_libera(ListaCirc *l);
```



# TAD LISTA CIRCULAR – Função Insere

```
ListaCirc* lst_circ_insere(ListaCirc *l, int info)
```

```
{
```

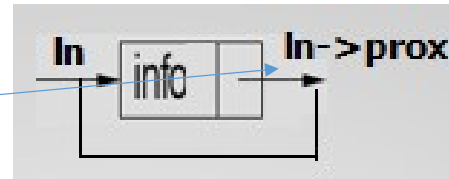
l - NULL

```
    ListaCirc* ln = (ListaCirc*)malloc(sizeof(ListaCirc));
```

```
    ln->info = info;
```

```
    if(l==NULL)//lista vazia
```

**CASO 1- LISTA É VAZIA**



```
    ln->prox = ln;
```

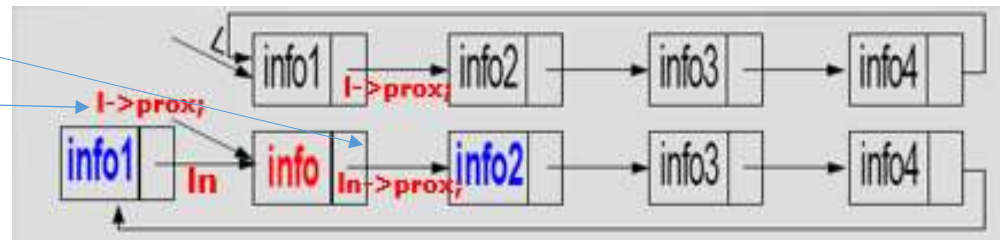
```
    else //lista não é vazia
```

```
{
```

**CASO 2 –LISTA NÃO É VAZIA**

```
    ln->prox = l->prox;
```

```
    l->prox = ln;
```

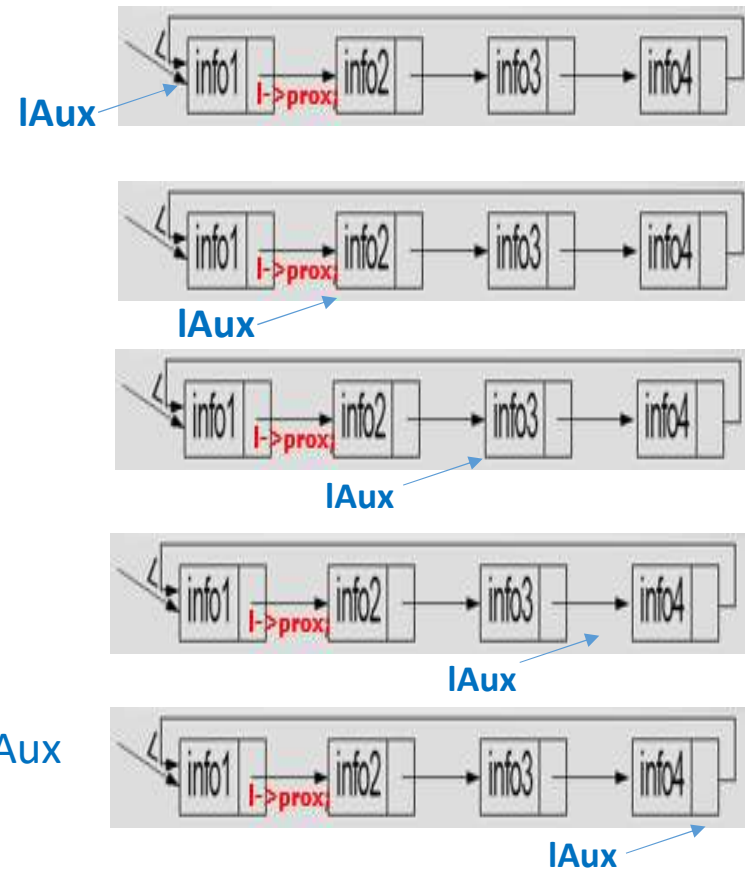


```
return ln;
```

```
}
```

# TAD Lista Circular – Função Imprime

```
void lst_circ_imprime(ListaCirc *l)
{
    if(l!=NULL)
    {
        ListaCirc* lAux = l;
        printf("Lista de Elementos \n");
        do
        {
            printf("Info = %d\n",lAux->info);
            lAux = lAux->prox;
        }
        while(l!=lAux); //chegou ao final, l==lAux
    }
}
```



Ao final da Lista l==lAux

# Lista Circular – Função retira

```
ListaCirc* Ist_retira(ListaCirc* l, int v)
```

```
{
```

```
  ListaCirc* ant=NULL;
```

```
  ListaCirc* p=l;
```

```
  //busca elemento a ser retirado, elemento é o primeiro.
```

```
  while(p!=NULL && p->info!=v) (F)
```

```
  {
```

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

```
  }
```

```
  if(p==NULL)
```

```
    return l;
```

```
  if(ant==NULL) (V)
```

```
    {l=p->prox;} L apontará agora para nodo 2
```

```
  else
```

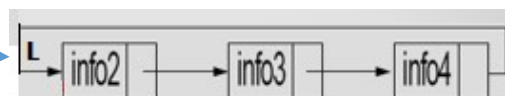
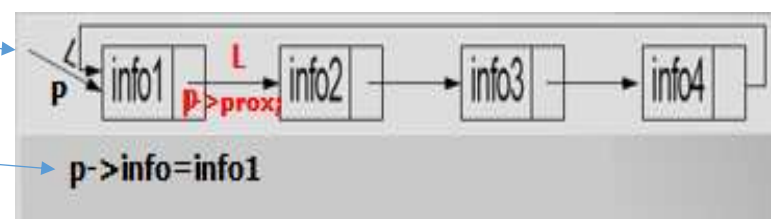
```
    {ant->prox=p->prox; }
```

```
  free(p);
```

```
  return l;
```

```
}
```

## CASO 1- ELEMENTO E O PRIMEIRO



# TAD Lista Circular – Função retira

```
ListaCirc* lst_retira(ListaCirc* l, int v)
```

```
{
```

```
  ListaCirc* ant=NULL;
```

```
  ListaCirc* p=l;
```

```
  while(p!=NULL && p->info!=v)
```

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

```
  if(p==NULL)
```

```
    return l;
```

```
  if(ant==NULL)
```

```
  { l=p->prox;}
```

```
  else
```

```
  { ant->prox = p->prox; } //ant->prox aponta para o elemento seguinte ao nó que contém p.info==3
```

```
  free(p); //remove-se o nó apontado por p , que contém p.info==3
```

```
  return l;
```

```
}
```

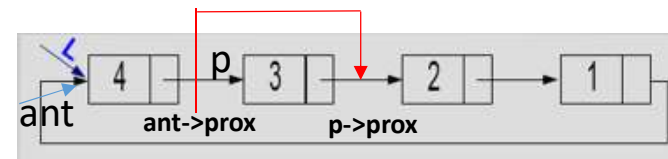
## CASO 2- ELEM.NO MEIO



**3**  
1ª iteração do laço  
p->info!=3 (V)



2ª iteração do laço  
p->info==3 (F)



# Referências

- Thomas H. [Cormen](#), Charles E. [Leiserson](#), Ronald L. [Rivest](#), and Clifford Stein. Algoritmos – Teoria e Prática, Tradução da Segunda Edição. Campus, 2016.
- Ziviani, N. Projeto de Algoritmos Com Implementações em Pascal e C, Pioneira Thomson Learning, 4ed. 2009.
- Waldemar Celes, Renato Cerqueira, José Lucas Rangel, *Introdução a Estruturas de Dados, Editora Campus* (2004)