

# Estruturas de Dados

## Lista Encadeada

# Lista Encadeada

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

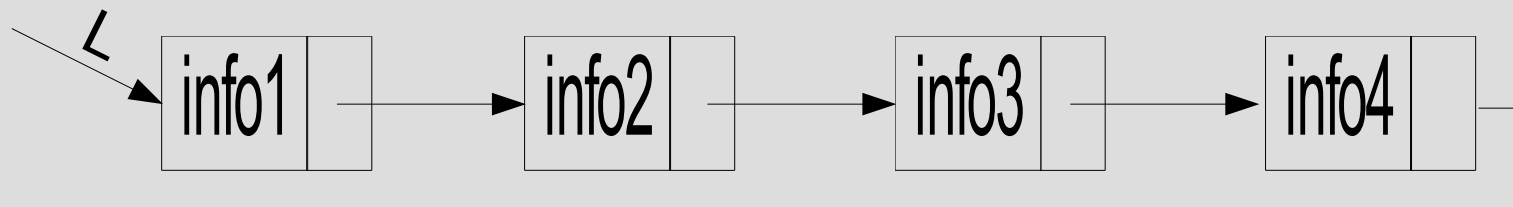


- Em geral, precisamos trabalhar com estruturas de dados dinâmicas que crescem (ou decrescem) à medida que elementos são inseridos (ou removidos)



# Lista Encadeada

- Sequência 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

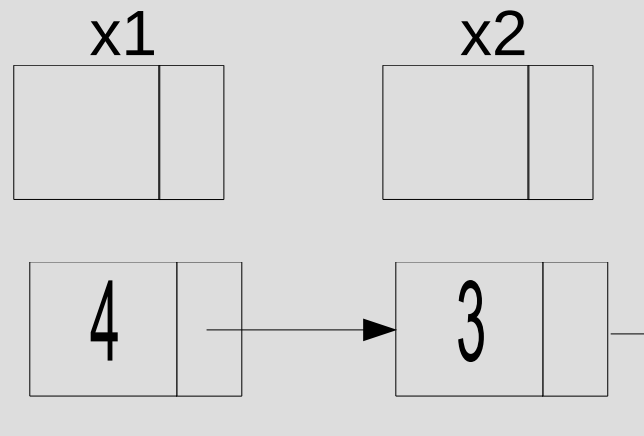


# Definição de Lista

- Considere uma lista encadeada armazenando valores inteiros

```
typedef struct lista Lista;  
struct lista {  
    int info;  
    Lista *prox;  
};
```

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



# Tipo Abstrato de Dado

## Lista Encadeada

Podemos criar um TAD Lista de Inteiros. Para tanto, devemos criar o arquivo lista.h com o nome do tipo e os protótipos.

```
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);
```

# TAD Lista Encadeada

Devemos implementar o TAD Lista Encadeada no arquivo lista.c, definindo o tipo Lista e implementando todas as funções.

```
#include<stdio.h>
#include<stdlib.h>
#include "lista.h"
```

```
struct lista {
    int info;
    Lista *prox;
};
```

# TAD Lista Encadeada

Função que cria uma lista vazia, representada pelo ponteiro NULL

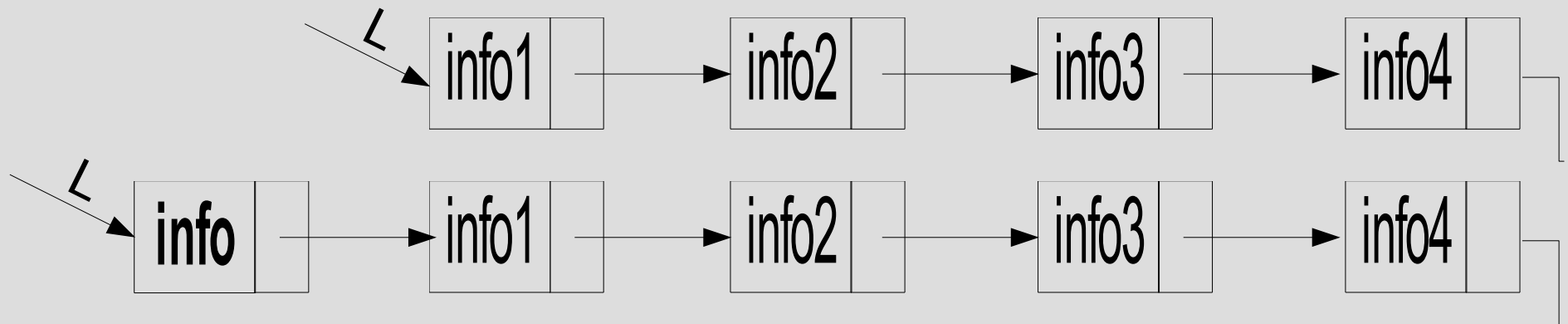
```
/* Cria uma lista vazia.*/  
Lista* lst_cria() {  
    return NULL;  
}
```

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) {  
    return (l==NULL);  
}
```

# TAD Lista Encadeada

Função que insere elemento no início da lista.



```
/* Insere um elemento no início da lista.*/  
Lista* lst_insere(Lista *l, int info){  
    Lista* ln = (Lista*)malloc(sizeof(Lista));  
    ln->info = info;  
    ln->prox = l;  
    return ln;  
}
```



# TAD 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) 2
```

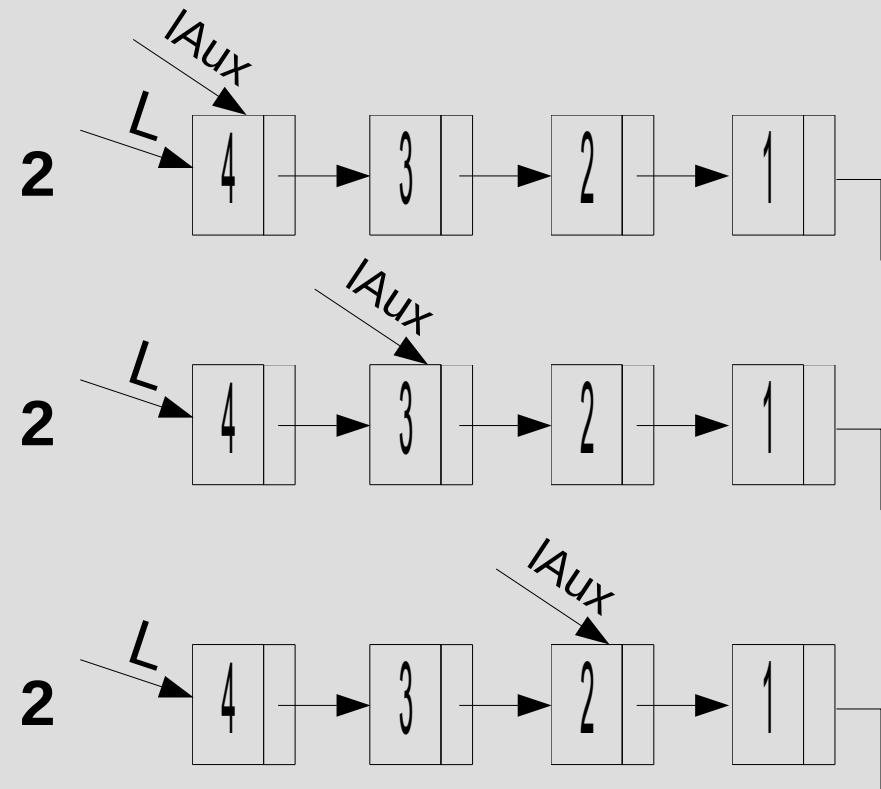
```
            return lAux;
```

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

```
    }
```

```
    return NULL;
```

```
}
```



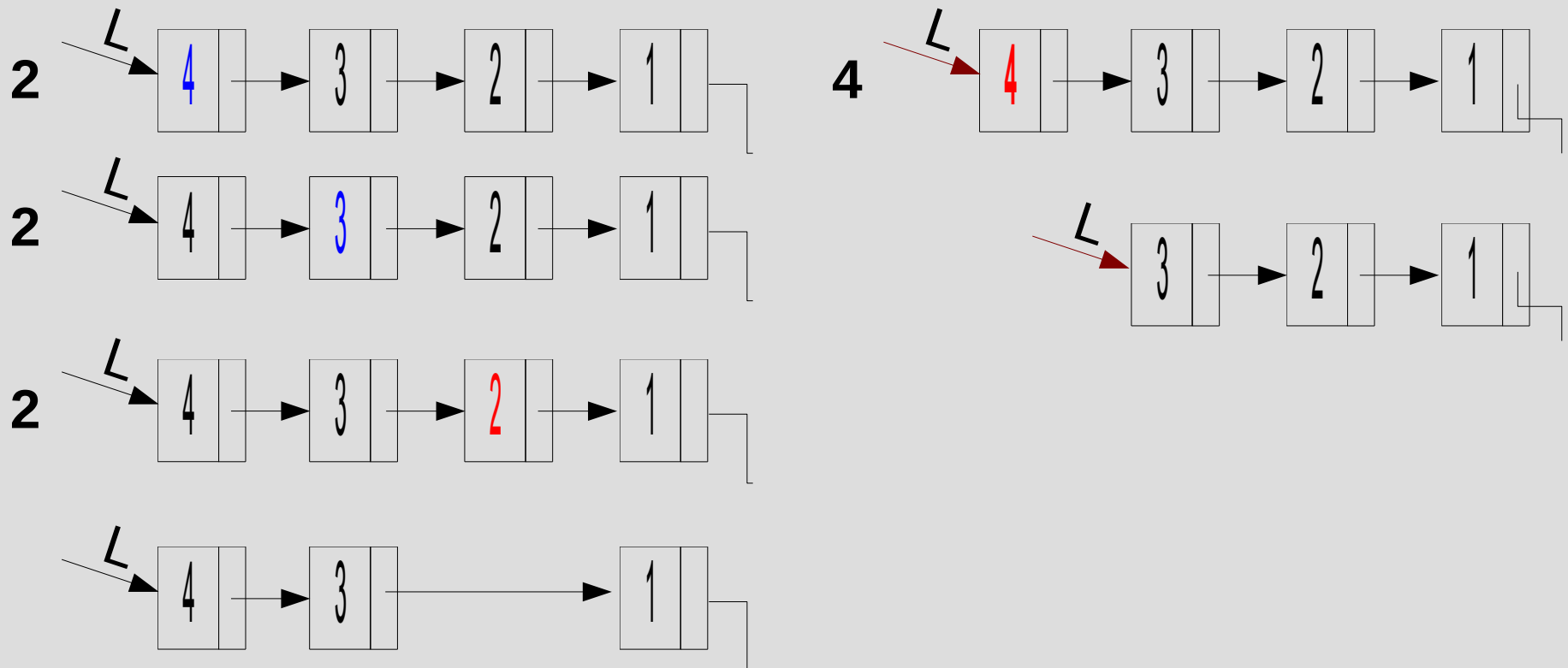
# TAD Lista Encadeada

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;  
    }  
}
```

# TAD Lista Encadeada

Função que remove um elemento de uma lista, retornando a lista alterada

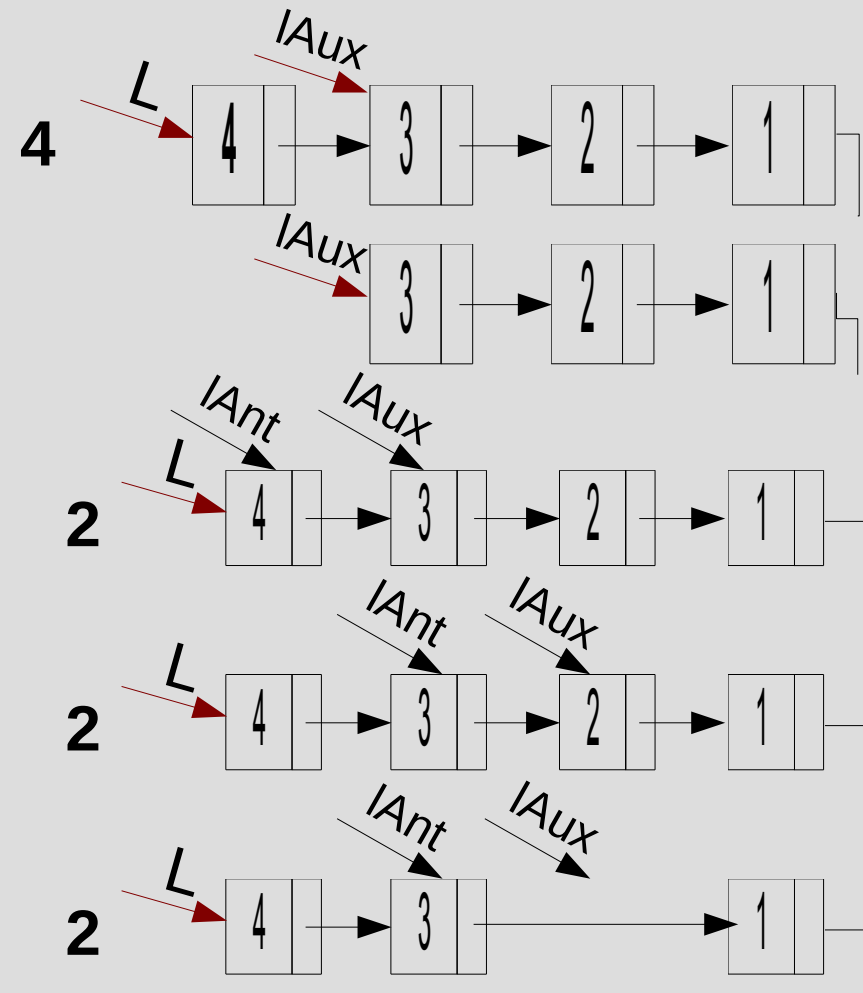


# TAD Lista Encadeada

```

Lista* lst_remove(Lista *l, int info){
    if(l!=NULL){
        Lista* lAux = l->prox;
        if(l->info==info){
            free(l);
            return lAux;
        }
        else{
            Lista* lAnt = l;
            while(lAux!=NULL ){
                if(lAux->info == info){
                    lAnt->prox = lAux->prox;
                    free(lAux);
                    break;
                }else{
                    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) {
```

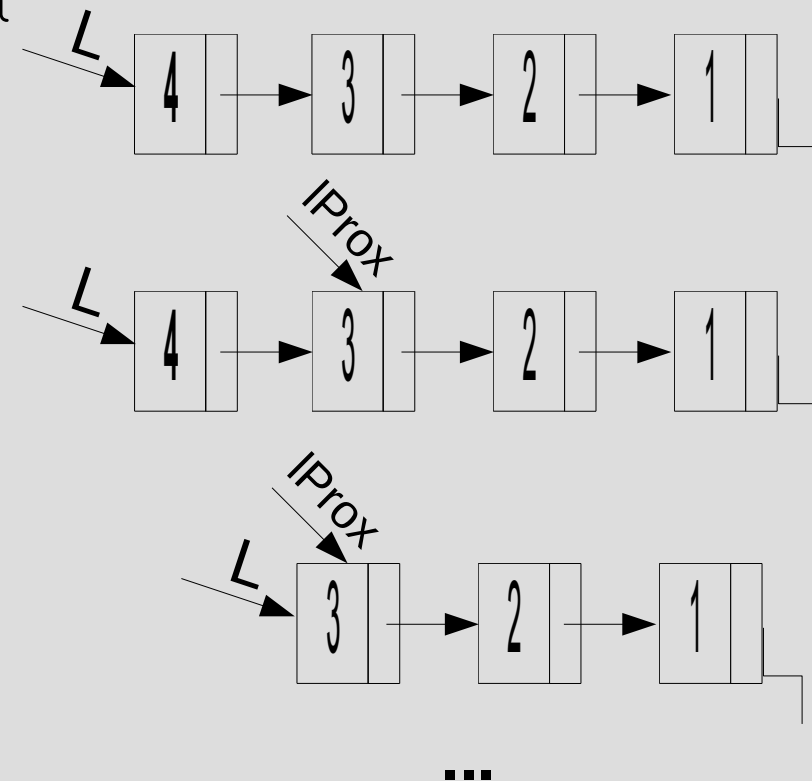
```
        lProx = l->prox;
```

```
        free(l);
```

```
        l = lProx;
```

```
    }
```

```
}
```



# TAD Lista Encadeada

Utilizando o TAD Lista Encadeada.

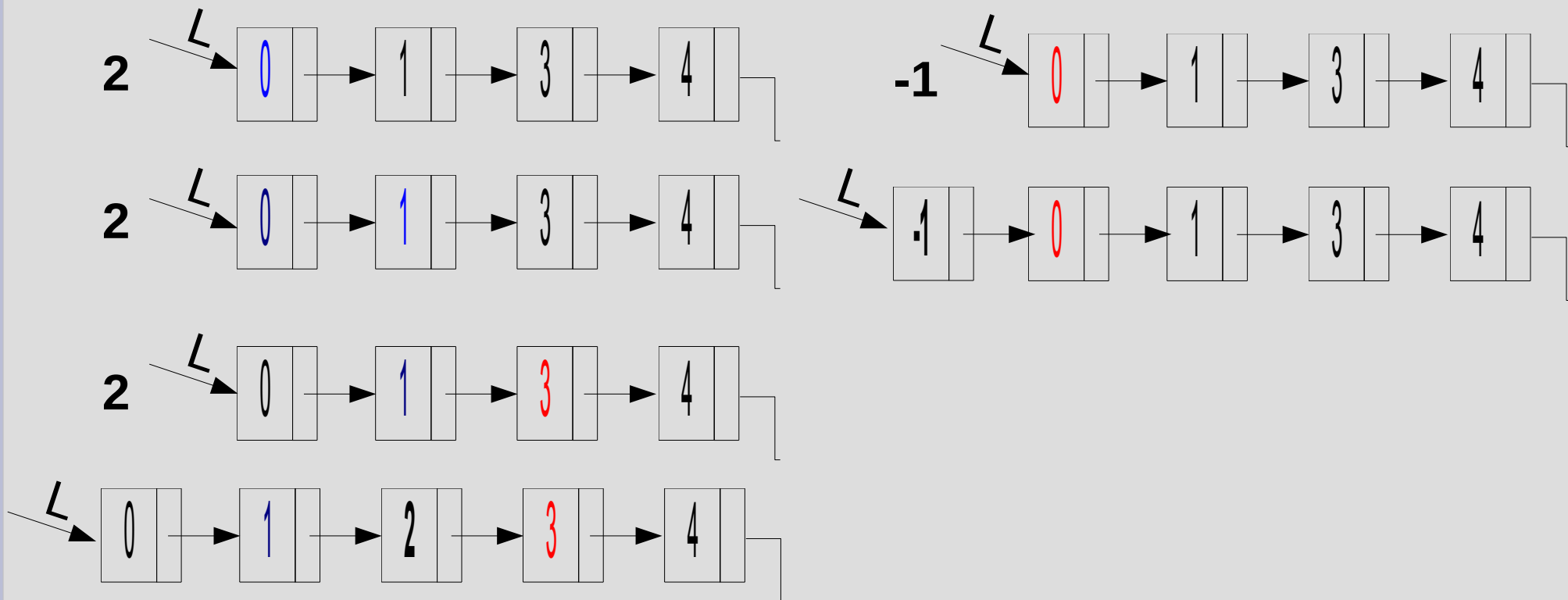
```
#include "lista.h"

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;
}
```

# Lista Encadeada

Para ordenarmos uma lista podemos utilizar a estratégia de ordenação por construção, ou seja, inserimos os elementos em suas corretas posições

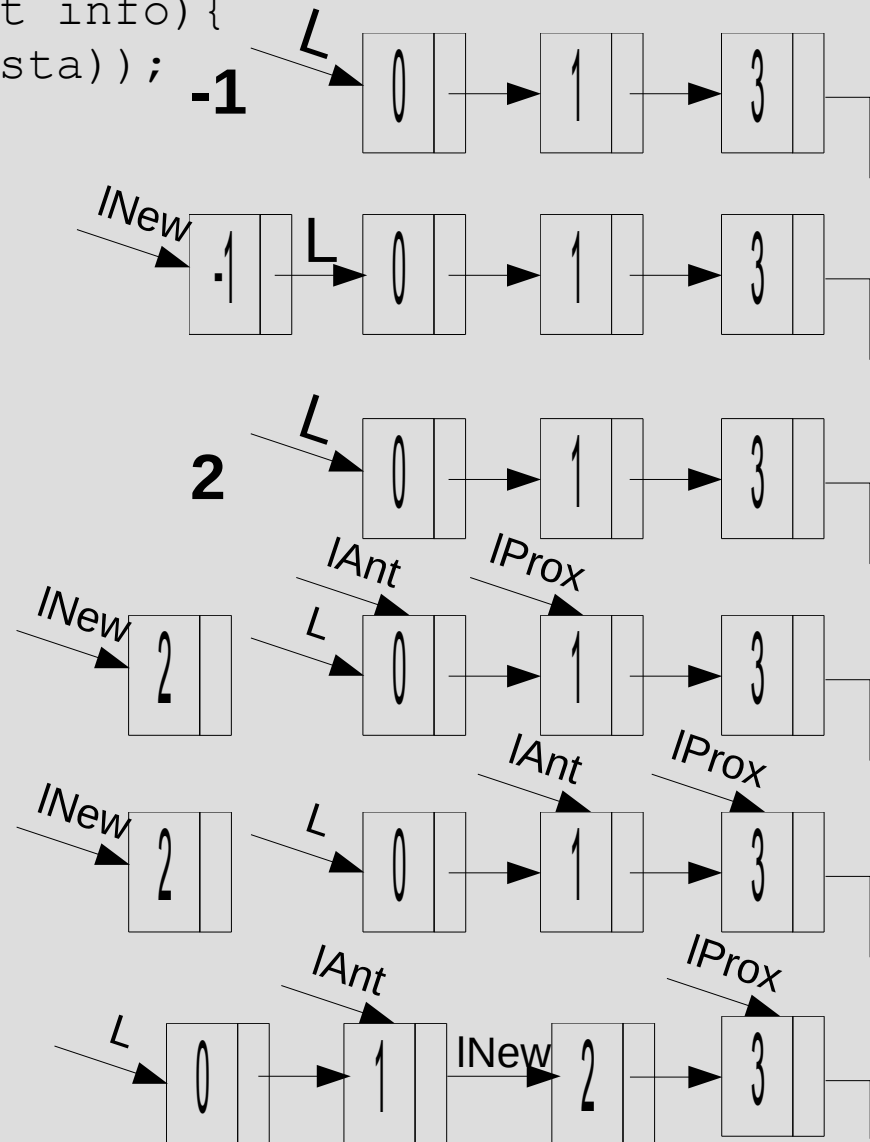


# Ordenação por Construção em Lista Encadeada

```

Lista* lst_inserir_ordenado(Lista *l, int info){
    Lista *lNew = (Lista*)malloc(sizeof(Lista));
    lNew->info = info;
    if(l==NULL){
        lNew->prox = NULL;
        return lNew;
    }else if(l->info>=info){
        lNew->prox = l;
        return lNew;
    }else{
        Lista *lAnt = l;
        Lista *lProx = l->prox;
        while(lProx!=NULL&& lProx->info<info)
        {
            lAnt = lProx;
            lProx = lProx->prox;
        }
        lAnt->prox = lNew;
        lNew->prox = lProx;
        return l;
    }
}

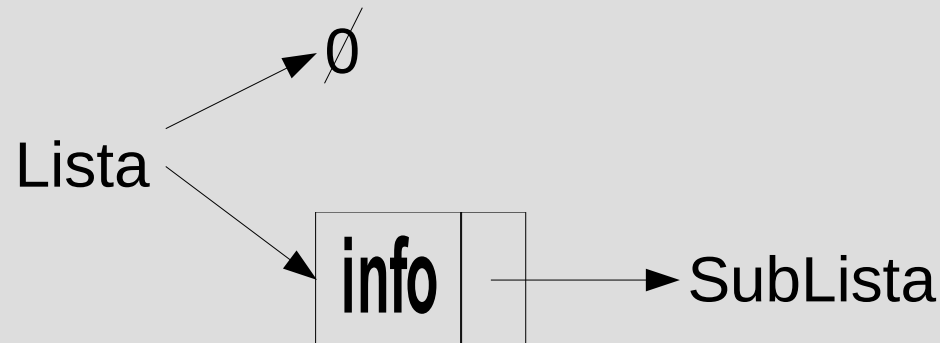
```





# Definição Recursiva de Lista Encadeada

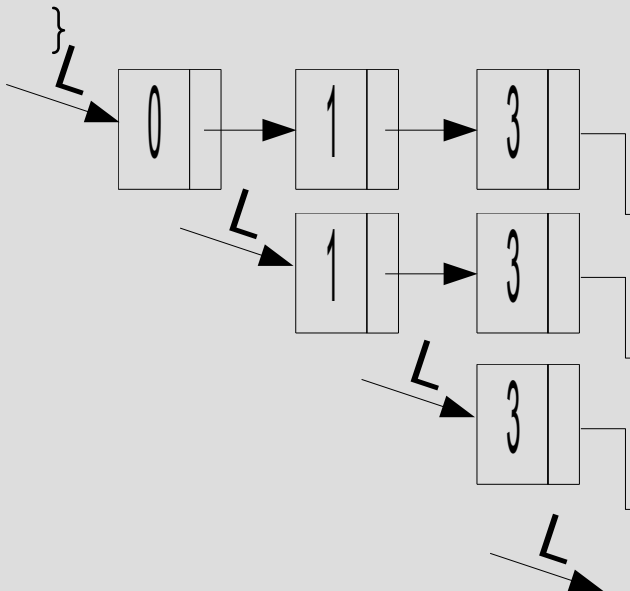
- Podemos dizer que um lista encadeada é representada por:
  - Uma Lista vazia; ou
  - Um Elemento seguido por uma sublista.



# Implementação Recursiva

## Função Imprime Lista

```
void lst_imprime_rec(Lista* l){  
    if(lst_vazia(l))  
        return;  
    else{  
        printf("info: %d\n", l->info);  
        lst_imprime_rec(l->prox);  
    }  
}
```



**info: 0**

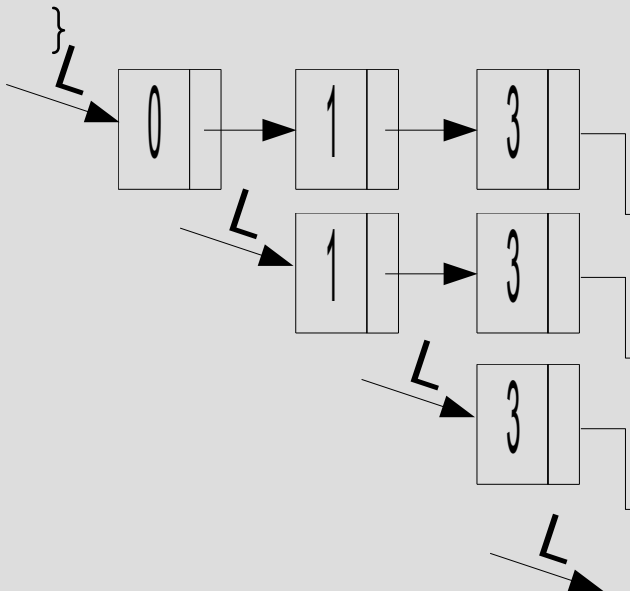
**info: 1**

**info: 3**

# Implementação Recursiva

## Função Imprime Lista Invertida

```
void lst_imprime_invertida_rec(Lista* l) {  
    if(lst_vazia(l))  
        return;  
    else{  
        lst_imprime_invertida_rec(l->prox);  
        printf("info: %d\n", l->info);  
    }  
}
```



**info: 3**

**info: 1**

**info: 0**

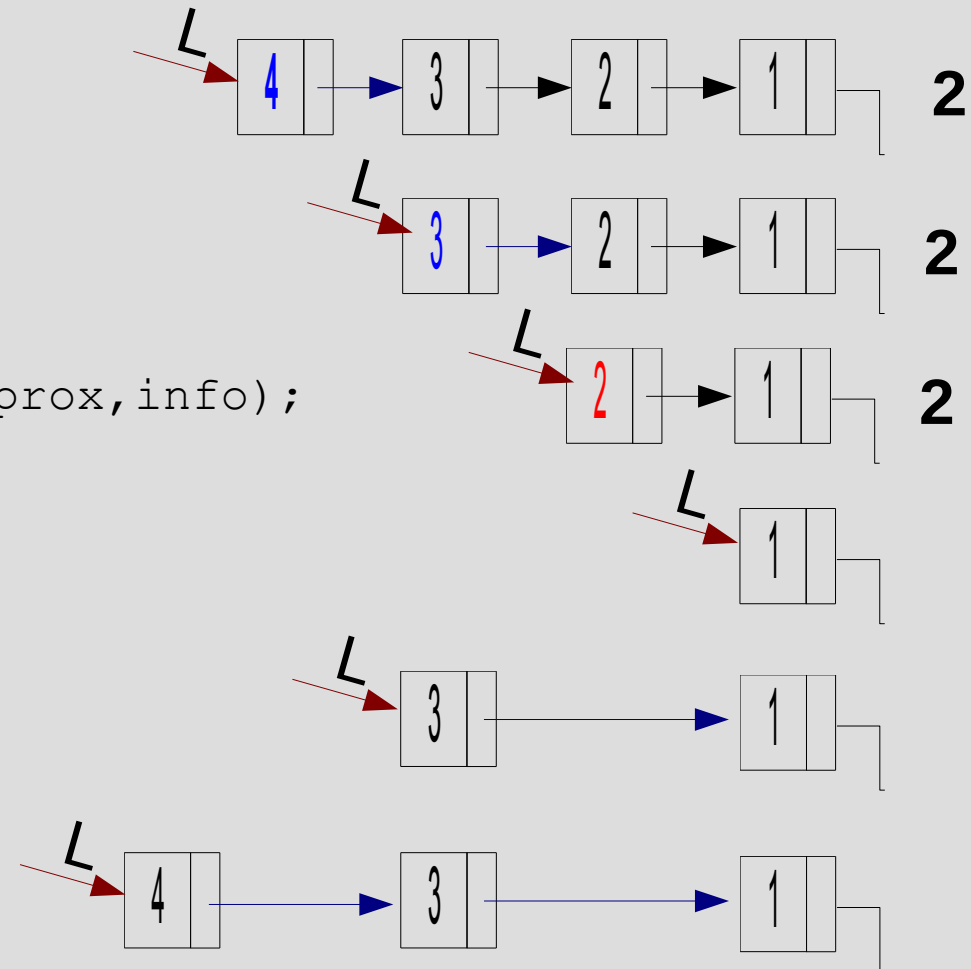
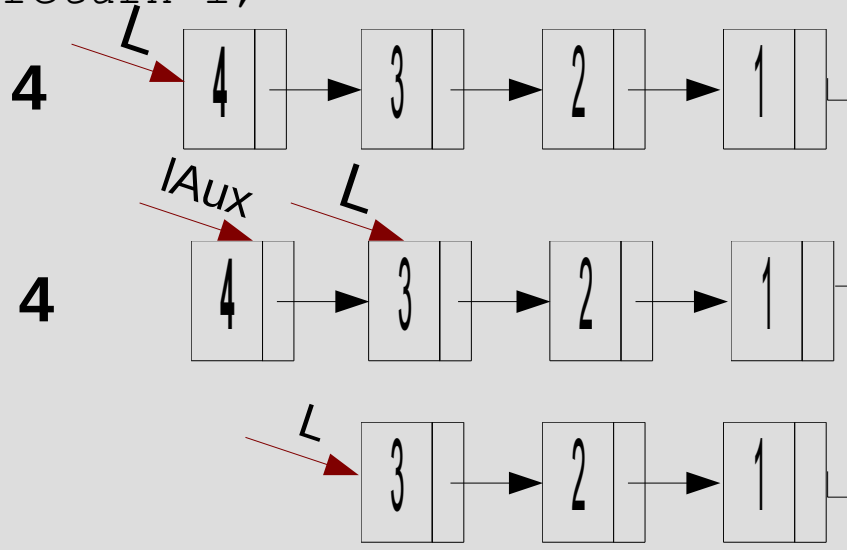
# Implementação Recursiva

## Função Remove Elemento Lista

```

Lista* lst_remove_rec(Lista *l, int info){
    if(!lst_vazia(l))
        if(l->info==info){
            Lista* lAux = l;
            l = l->prox;
            free(lAux);
        }
        else{
            l->prox = lst_remove_rec(l->prox, info);
        }
    return l;
}

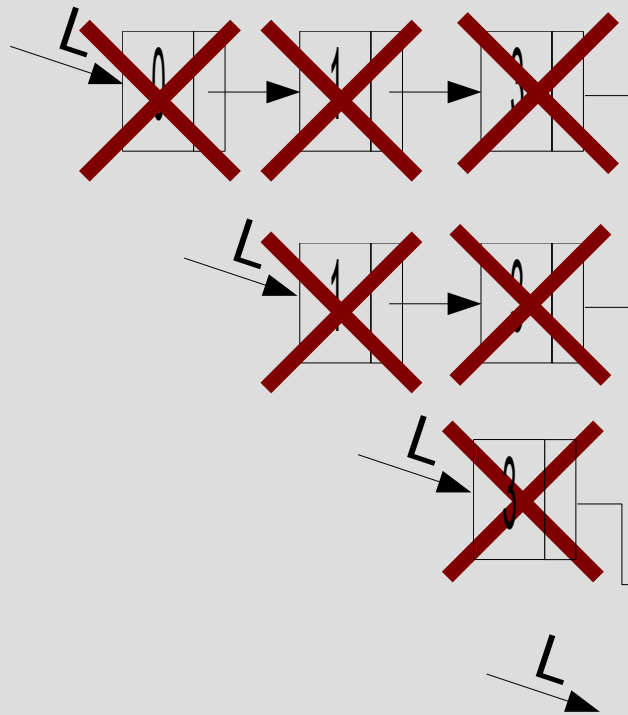
```



# Implementação Recursiva

## Função Libera Lista

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



# Implementação Recursiva

## Função Igualdade Entre Listas

Duas Listas são iguais se elas têm a mesma sequência de elementos

```
int lst_igual_rec(Lista *l1, Lista *l2) {  
    if (lst_vazia(l1) && lst_vazia(l2))  
        return 1;  
    else if (lst_vazia(l1) || lst_vazia(l2))  
        return 0;  
    else  
        return (l1->info==l2->info &&  
                lst_igual_rec(l1->prox, l2->prox));  
}  
}
```

# Referência

- Slides baseados no livro **Introdução a Estruturas de Dados**, Waldemar Celes, Renato Cerqueira e José Lucas Rangel, Editora Campus, 2004.