

# Curso de C

## *Memória Dinâmica*

# Memória Dinâmica

## Roteiro:

- Memória dinâmica
- Vetores dinâmicos
- Listas ligadas



# Conceitos: memória **dinâmica** e **estática**

- **Memória estática:**
  - Variáveis declaradas no código
  - Tamanho fixo
  - Número limitado de variáveis

## Programa

```
Memória estática:  
int a;  
int b;
```

# Conceitos: memória **dinâmica** e **estática**

- **Memória estática:**

- Variáveis declaradas no código
- Tamanho fixo
- Número limitado de variáveis

- **Memória dinâmica:**

- Espaços adicionais de memória
- Memória alocada/liberada quando necessário

## Programa

### Memória estática:

```
int a;  
int b;  
int *p;
```

### Memória dinâmica:



# Conceitos

## Passo a passo no programa:

- Solicitar memória dinâmica

### Programa

#### Memória estática:

```
int a;  
int b;  
int *p;
```

#### Memória dinâmica:

???

# Conceitos

## Passo a passo no programa:

- Solicitar memória dinâmica
- Guardar apontador para espaço obtido

### Programa

#### Memória estática:

```
int a;  
int b;  
int *p;
```

#### Memória dinâmica:

???

# Conceitos

## Passo a passo no programa:

- Solicitar memória dinâmica
- Guardar apontador para espaço obtido
- Armazenar dados
  - Qualquer conteúdo

### Programa

#### Memória estática:

```
int a;  
int b;  
int *p;
```

#### Memória dinâmica:

xyz



# Conceitos

## Passo a passo no programa:

- Solicitar memória dinâmica
- Guardar apontador para espaço obtido
- Armazenar dados
  - Qualquer conteúdo
- Liberar memória obtida

### Programa

#### Memória estática:

```
int a;  
int b;  
int *p;
```

#### Memória dinâmica:

~~xyz~~



# Conceitos: obter memória

## Chamar função **malloc**

- Parâmetro: tamanho desejado, em bytes
- Resultado: apontador para espaço na memória

### Programa

#### Memória estática:

```
int a;  
int b;  
int *p;
```

#### Memória dinâmica:

# Conceitos: obter memória

## Chamar função **malloc**

- Parâmetro: tamanho desejado, em bytes
- Resultado: apontador para espaço na memória

```
int *p;  
p = (int*)malloc(4);
```

### Programa

#### Memória estática:

```
int a;  
int b;  
int *p;
```

#### Memória dinâmica:

??

# Conceitos: obter memória

## Chamar função **malloc**

- Parâmetro: tamanho desejado, em bytes
- Resultado: apontador para espaço na memória

```
int *p;  
p = (int*)malloc(4);  
*p = 5;  
printf("%d", *p);
```

### Programa

#### Memória estática:

```
int a;  
int b;  
int *p;
```

#### Memória dinâmica:

5



# Conceitos: liberar memória

## Chamar função **free**:

- Parâmetro: endereço já alocado
- Resultado: libera a área alocada

```
int *p;  
p = (int*)malloc(4);  
*p = 5;
```

### Programa

#### Memória estática:

```
int a;  
int b;  
int *p;
```

#### Memória dinâmica:

5



# Conceitos: liberar memória

## Chamar função **free**:

- Parâmetro: endereço já alocado
- Resultado: libera a área alocada

```
int *p;  
p = (int*)malloc(4);  
*p = 5;  
free(p);
```

### Programa

#### Memória estática:

```
int a;  
int b;  
int *p;
```

#### Memória dinâmica:

~~5~~

# Memória Dinâmica

A blurred background image showing a person's hands using an abacus. The abacus is a traditional calculating tool with a grid of black beads on wooden rods. The person's fingers are visible, moving the beads. The image is semi-transparent, allowing the text to be overlaid.

*Vetor dinâmico*

# Vetor dinâmico

## Função `sizeof`:

### `sizeof(variavel)`

- Retorna quantidade de memória ocupada pela variável

### `sizeof(tipo)`

- Retorna quantidade de memória ocupada por uma variável com este tipo

# Vetor dinâmico

## Função `sizeof`:

### `sizeof`(variavel)

- Retorna quantidade de memória ocupada pela variável

### `sizeof`(tipo)

- Retorna quantidade de memória ocupada por uma variável com este tipo

```
int a;  
printf("%d", sizeof(int));  
printf("%d", sizeof(a));
```

**Saída:**



# Vetor dinâmico

## Função `sizeof`:

### `sizeof`(variavel)

- Retorna quantidade de memória ocupada pela variável

### `sizeof`(tipo)

- Retorna quantidade de memória ocupada por uma variável com este tipo

```
int a;  
printf("%d", sizeof(int));  
printf("%d", sizeof(a));
```

**Saída:**

4

4

# Vetor dinâmico

## Passo a passo:

- Calcular tamanho do vetor com **sizeof**

# Vetor dinâmico

## Passo a passo:

- Calcular tamanho do vetor com **sizeof**
- Reservar memória com **malloc**

# Vetor dinâmico

## Passo a passo:

- Calcular tamanho do vetor com **sizeof**
- Reservar memória com **malloc**
- Acessar elementos com **[ ]** ou **ponteiros**

# Vetor dinâmico

## Passo a passo:

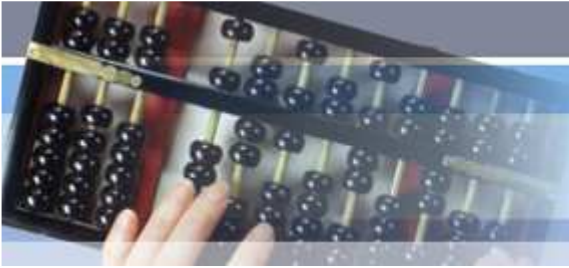
- Calcular tamanho do vetor com **sizeof**
- Reservar memória com **malloc**
- Acessar elementos com **[ ]** ou **ponteiros**
- Liberar memória do vetor com **free**

# Vetor dinâmico

## Exemplo:

```
int *v;  
int tamanho;  
  
scanf("%d", &tamanho);  
v = (int*)malloc(sizeof(int)*tamanho);  
  
v[1] = v[2] + v[3];  
  
*(v+1) = *(v+2) + *(v+3);  
  
free(v);
```

Vetor01



## Conceito: alterar tamanho de memória

### Chamar função **realloc**:

- **Parâmetros:** apontador para espaço antigo, tamanho desejado
- **Resultado:** apontador para novo espaço

#### Programa

##### Memória estática:

```
int a;  
int b;  
int *p;
```

##### Memória dinâmica:

# Conceito: alterar tamanho de memória

## Chamar função **realloc**:

- **Parâmetros:** apontador para espaço antigo, tamanho desejado
- **Resultado:** apontador para novo espaço

```
int *p;  
p = (int*)malloc(4);  
...
```

### Programa

#### Memória estática:

```
int a;  
int b;  
int *p;
```

#### Memória dinâmica:





# Conceito: alterar tamanho de memória

## Chamar função **realloc**:

- **Parâmetros:** apontador para espaço antigo, tamanho desejado
- **Resultado:** apontador para novo espaço

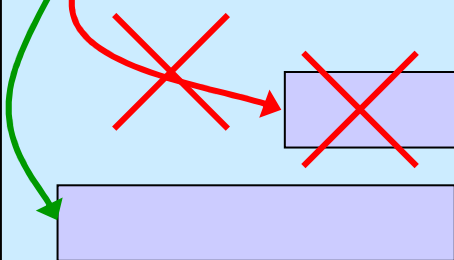
```
int *p;
p = (int*)malloc(4);
...
p = (int*)realloc(p, 8);
```

### Programa

#### Memória estática:

```
int a;
int b;
int *p;
```

#### Memória dinâmica:



# Vetor dinâmico

## Exemplo:

```
int *v;  
int tamanho, n_tamanho;  
  
scanf("%d", &tamanho);  
v=(int*)malloc(sizeof(int)*tamanho);  
  
scanf("%d", &n_tamanho);  
v=(int*)realloc(v,sizeof(int)*n_tamanho);  
  
free(v);
```

Vetor02

# Estrutura dinâmica

- Calcular tamanho da estrutura com **sizeof**

# Estrutura dinâmica

- Calcular tamanho da estrutura com **sizeof**
- Reservar memória com **malloc**

# Estrutura dinâmica

- Calcular tamanho da estrutura com **sizeof**
- Reservar memória com **malloc**
- Acessar membros da estrutura com **->**



# Estrutura dinâmica

- Calcular tamanho da estrutura com **sizeof**
- Reservar memória com **malloc**
- Acessar membros da estrutura com **->**
- Liberar memória da estrutura com **free**

# Estrutura dinâmica

```
struct complexo {  
    float real;  
    float imaginario;  
}  
  
struct complexo *numero;  
  
numero = (struct complexo *)  
    malloc(sizeof(struct complexo));  
  
numero->real = 10.0;  
numero->imaginario = 5.0;  
  
free(numero);
```

# Apontadores

*Listas Ligadas*



# Listas Ligadas

## Idéia principal :

- Vetor:
  - Um único grande bloco de memória
  - Elementos consecutivos adjacentes

# Listas Ligadas

## Idéia principal :

- Vetor:
  - Um único grande bloco de memória
  - Elementos consecutivos adjacentes
- Lista ligada:
  - Cada elemento em posição de **memória separada**
  - Posições **desordenadas** e não consecutivas



The diagram shows three nodes of a linked list arranged horizontally. Each node is represented as a light blue rectangle divided into two parts: a larger left part labeled 'dados' and a smaller right part representing a pointer to the next node.

dados

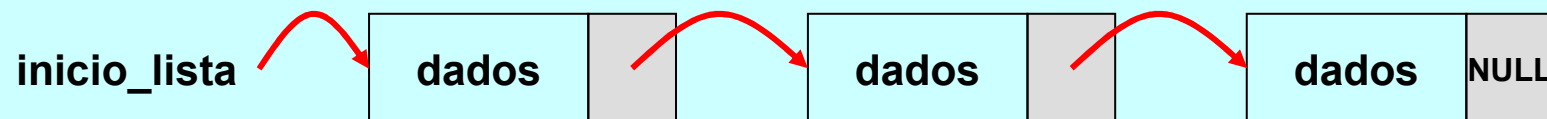
dados

dados


# Listas Ligadas

## Idéia principal :

- Vetor:
  - Um único grande bloco de memória
  - Elementos consecutivos adjacentes
- Lista ligada:
  - Cada elemento em posição de **memória separada**
  - Posições **desordenadas** e não consecutivas
  - **Apontadores** para indicar seqüência correta



# Listas Ligadas: declaração



```
struct elemento {  
    int valor;  
    struct elemento *prox;  
}
```

# Listas Ligadas: declaração

```
struct elemento {  
    int valor;  
    struct elemento *prox;  
}
```

Nome da estrutura

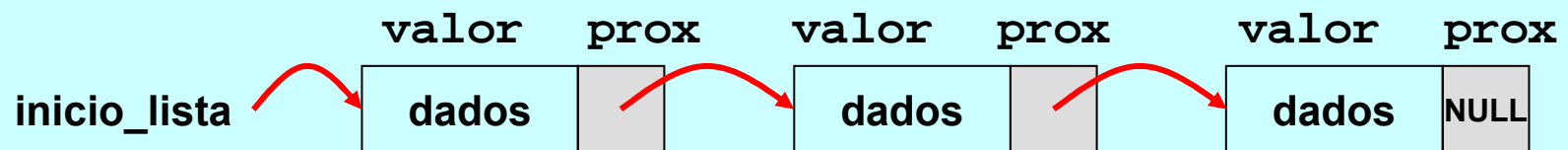
Apontador para próximo elemento na seqüência

# Listas Ligadas: declaração

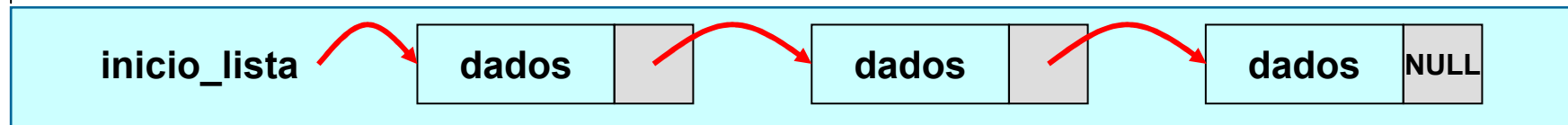
```
struct elemento {  
    int valor;  
    struct elemento *prox;  
}
```

Nome da estrutura

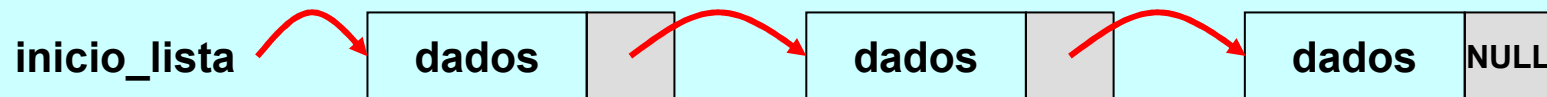
Apontador para próximo elemento na seqüência



# Listas Ligadas: percorrer



# Listas Ligadas: percorrer

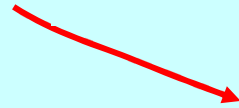


```
struct elemento *atual;  
atual = inicio_lista;  
  
while (atual != null) {  
    printf(" ...", atual->valor);  
  
    atual = atual->proximo;  
  
}
```



# Listas Ligadas: inserir em lista vazia

**inicio\_lista**



**NULL**

# Listas Ligadas: inserir em lista vazia

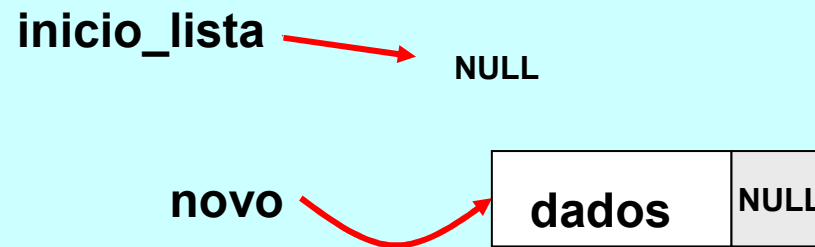
inico\_lista  NULL

novo



```
ново=( . . . *)malloc(sizeof(. . .));
```

# Listas Ligadas: inserir em lista vazia

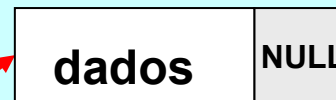


```
novο=(. . . *)malloc(sizeof(. . .));  
novο->valor = . . . ;  
novο->proximo = NULL;
```

# Listas Ligadas: inserir em lista vazia

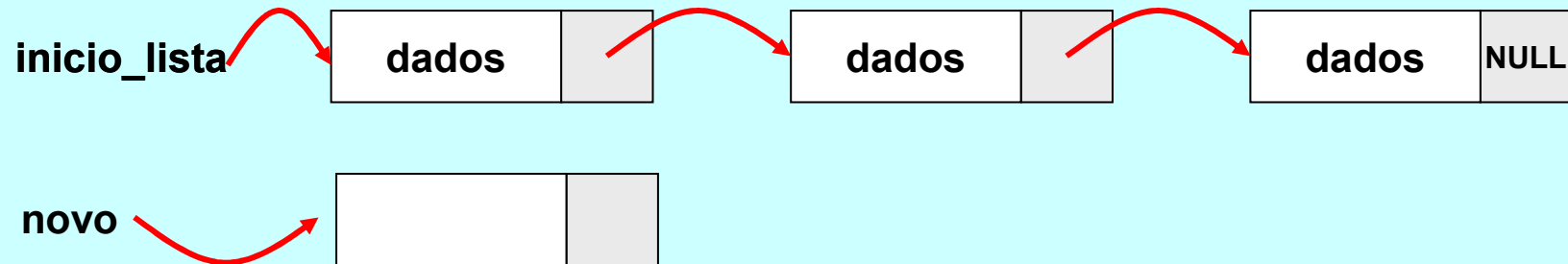
inicio\_lista

novo



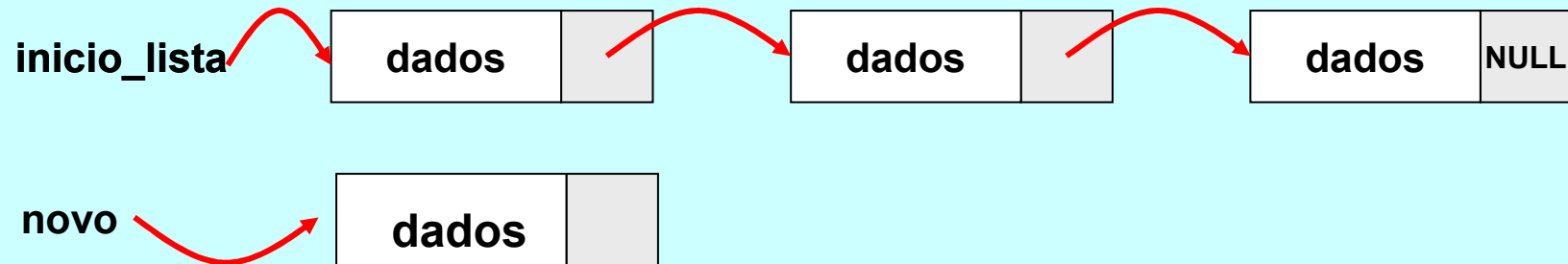
```
novο=(. . . *)malloc(sizeof(. . .));  
novο->valor = . . . ;  
novο->proximo = NULL;  
inicio_lista = novο;
```

# Listas Ligadas: inserir no início, não vazia



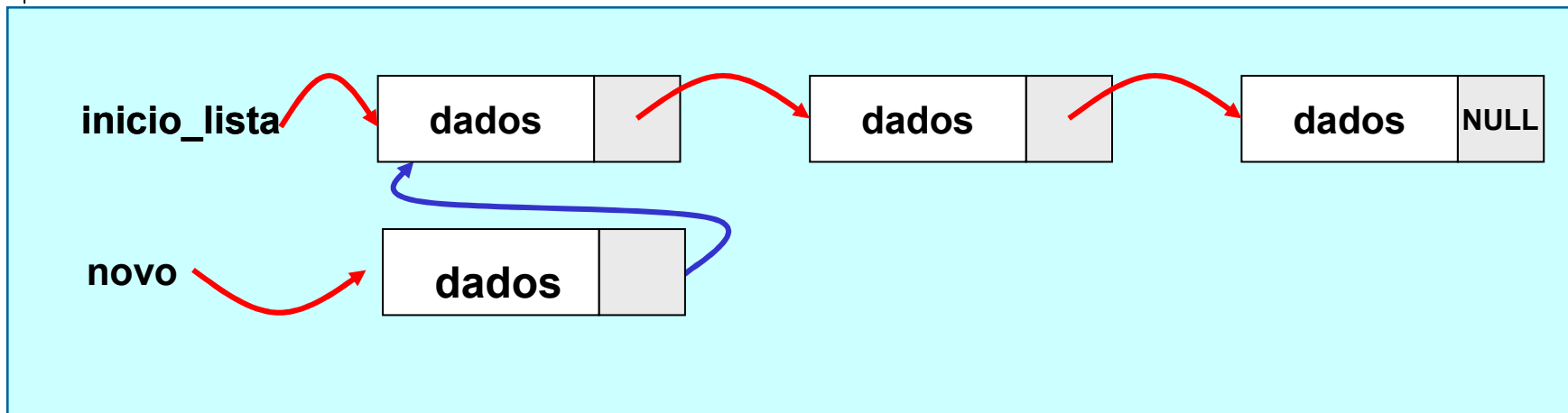
```
novο=(. . . *)malloc(sizeof(. . .));
```

# Listas Ligadas: inserir no início, não vazia



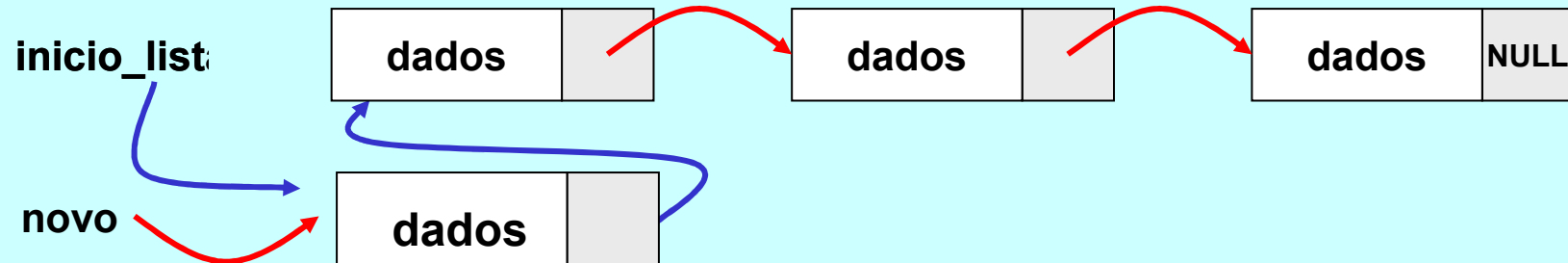
```
novo=(. . . *)malloc(sizeof(. . .));  
novo->valor = . . . ;
```

# Listas Ligadas: inserir no início, não vazia



```
novo=(. . . *)malloc(sizeof(. . .));  
novo->valor = . . . ;  
novo->proximo = inicio_lista;
```

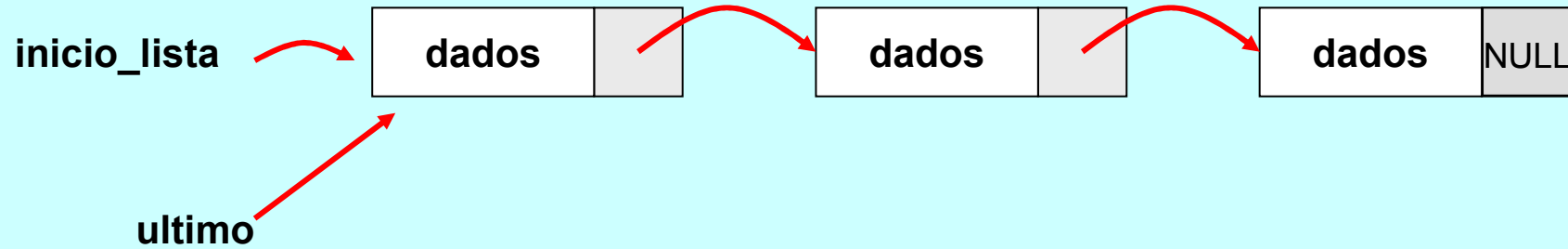
# Listas Ligadas: inserir no início, não vazia



```
novo=(. . . *)malloc(sizeof(. . .));  
novo->valor = . . . ;  
novo->proximo = inicio_lista;  
inicio_lista = novo;
```

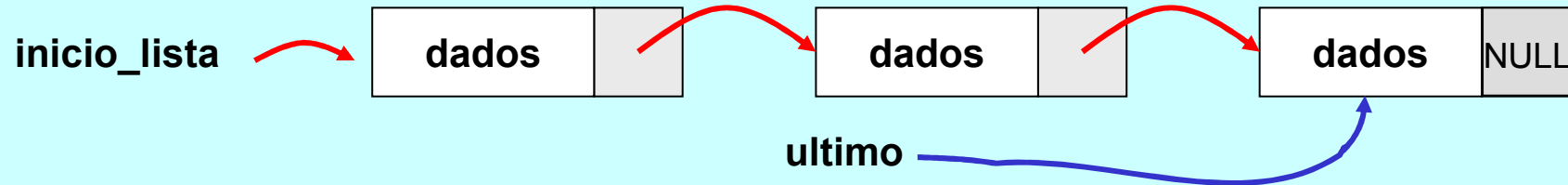


# Listas Ligadas: inserir no fim



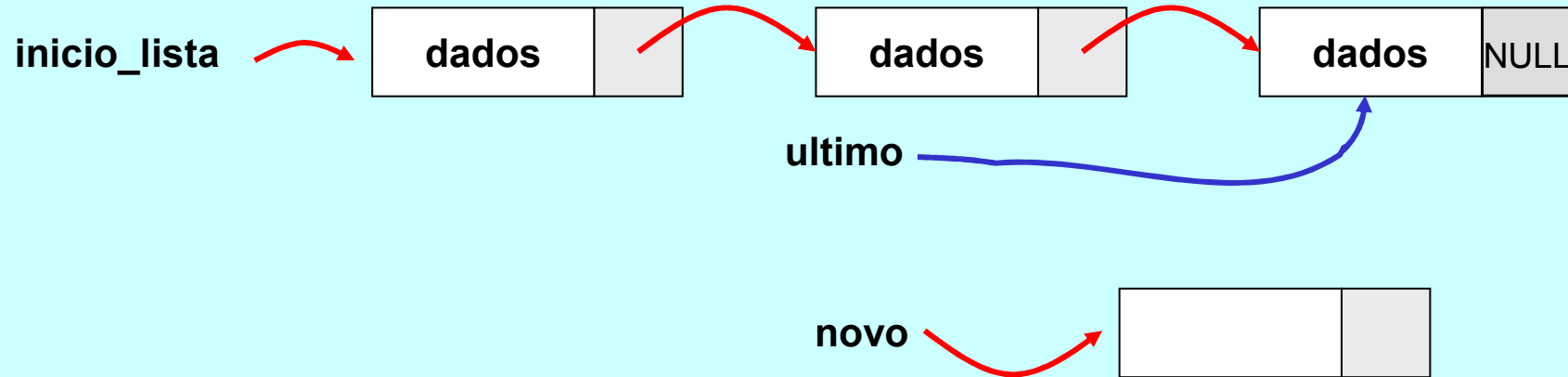
```
struct elemento *ultimo;  
ultimo = inicio_lista;
```

# Listas Ligadas: inserir no fim



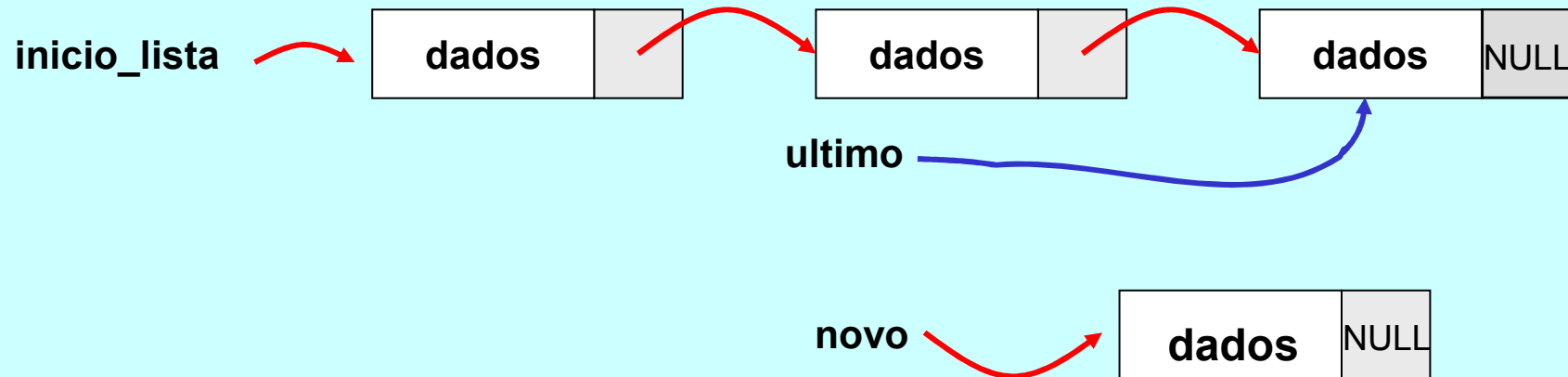
```
struct elemento *ultimo;  
ultimo = inicio_lista;  
while (ultimo->proximo != null) {  
    ultimo = ultimo->proximo;  
}
```

# Listas Ligadas: inserir no fim



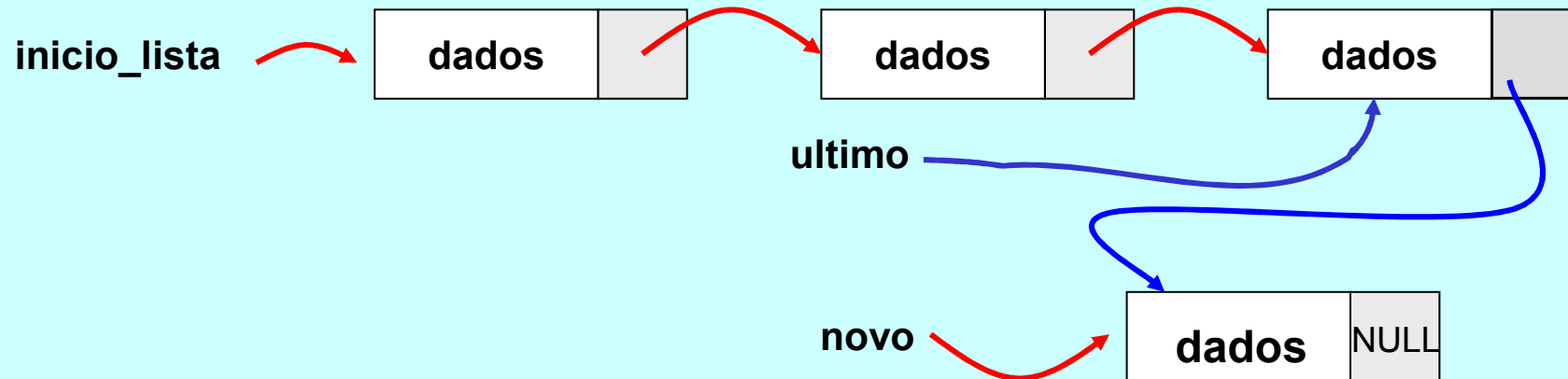
```
struct elemento *ultimo;  
ultimo = inicio_lista;  
while (ultimo->proximo != null) {  
    ultimo = ultimo->proximo;  
}  
  
novo=(. . . *)malloc(sizeof(. . .));
```

# Listas Ligadas: inserir no fim



```
struct elemento *ultimo;  
ultimo = inicio_lista;  
while (ultimo->proximo != null) {  
    ultimo = ultimo->proximo;  
}  
novos=(. . . *)malloc(sizeof(. . .));  
novos->valor = . . . ;  
novos->proximo = NULL;
```

# Listas Ligadas: inserir no fim

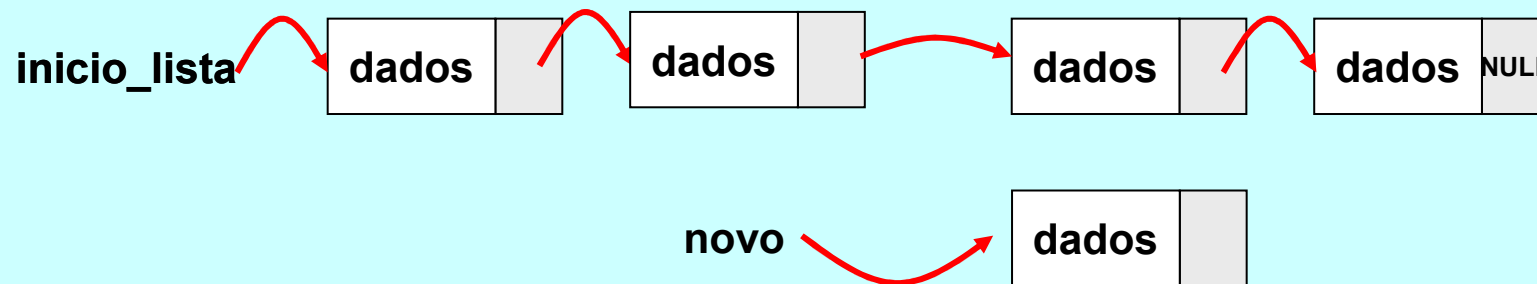


```

struct elemento *ultimo;
ultimo = inicio_lista;
while (ultimo->proximo != null) {
    ultimo = ultimo->proximo; }
novo=(. . . *)malloc(sizeof(. . .));
novo->valor = . . . ;
novo->proximo = NULL;
ultimo->proximo = novo;

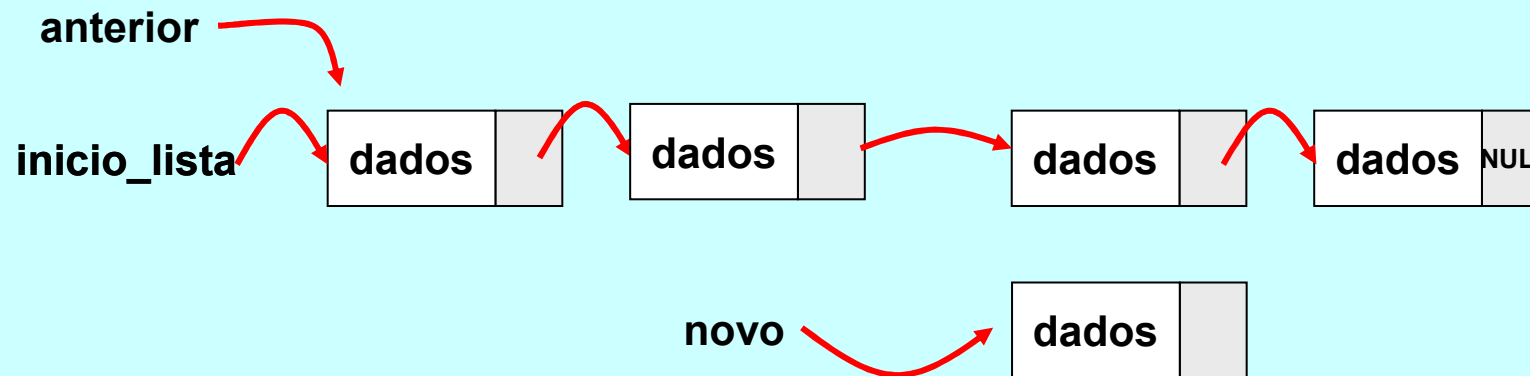
```

# Listas Ligadas: inserir no meio (ordenado)



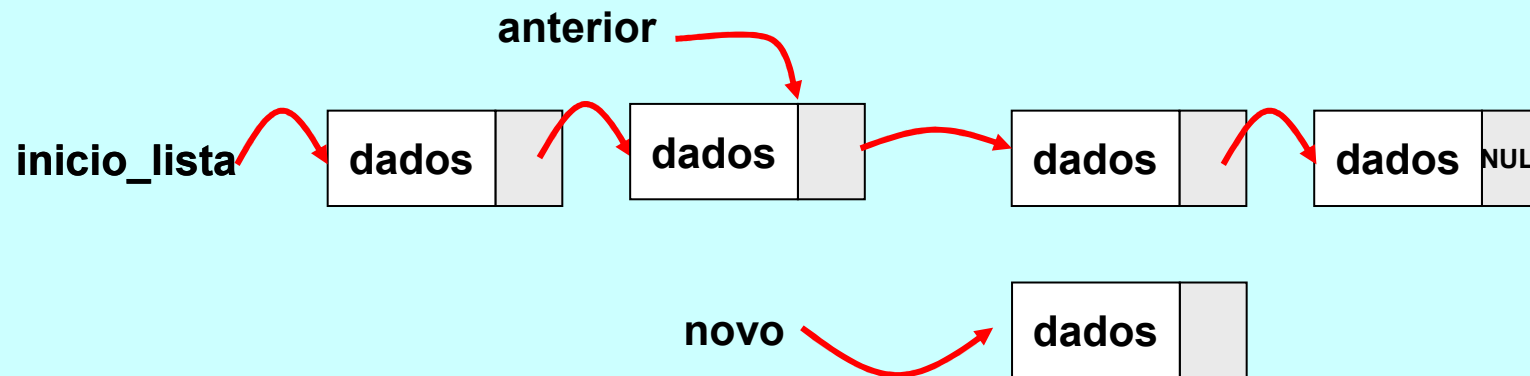
```
novo=(. . . *)malloc(sizeof(. . .));  
novo->valor = . . . ;
```

# Listas Ligadas: inserir no meio (ordenado)



```
novo=(. . . *)malloc(sizeof(. . .));  
novo->valor = . . . ;  
// ordenada por valor  
struct elemento *anterior; anterior = inicio_lista;
```

# Listas Ligadas: inserir no meio (ordenado)



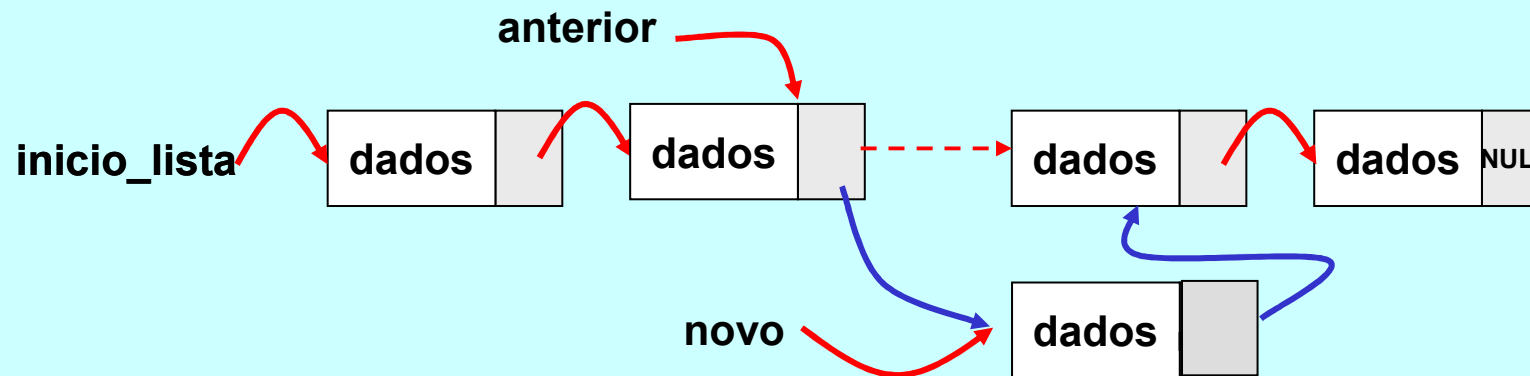
```

novo=(. . . *)malloc(sizeof(. . .));
novo->valor = . . . ;
// ordenada por valor
struct elemento *anterior; anterior = inicio_lista;
while ((anterior-> proximo != NULL)
      && (anterior->proximo->valor < novo->valor)) {
    anterior = anterior->proximo;}

```



# Listas Ligadas: inserir no meio (ordenado)

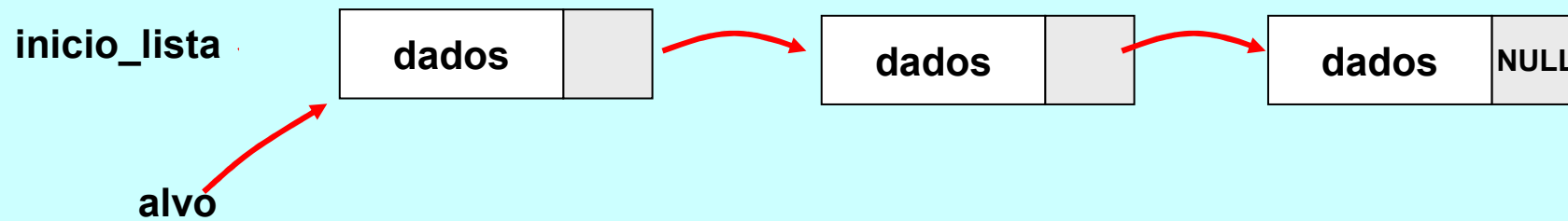


```

novo=(. . . *)malloc(sizeof(. . .));
novo->valor = . . . ;
// ordenada por valor
struct elemento *anterior; anterior = inicio_lista;
while ((anterior-> proximo != NULL)
      && (anterior->proximo->valor < novo->valor)) {
    anterior = anterior->proximo;}
novo->proximo = anterior->proximo;
anterior->proximo = novo;

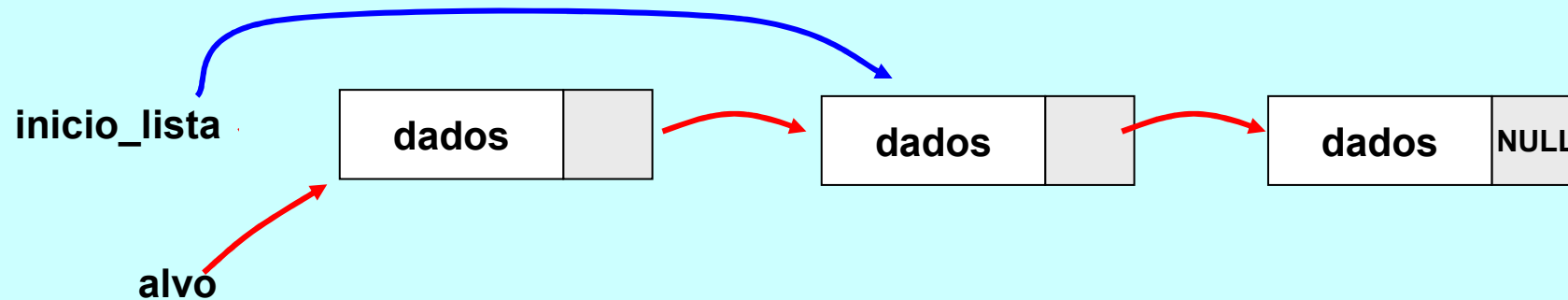
```

# Listas Ligadas: remover do inicio (não vazia)



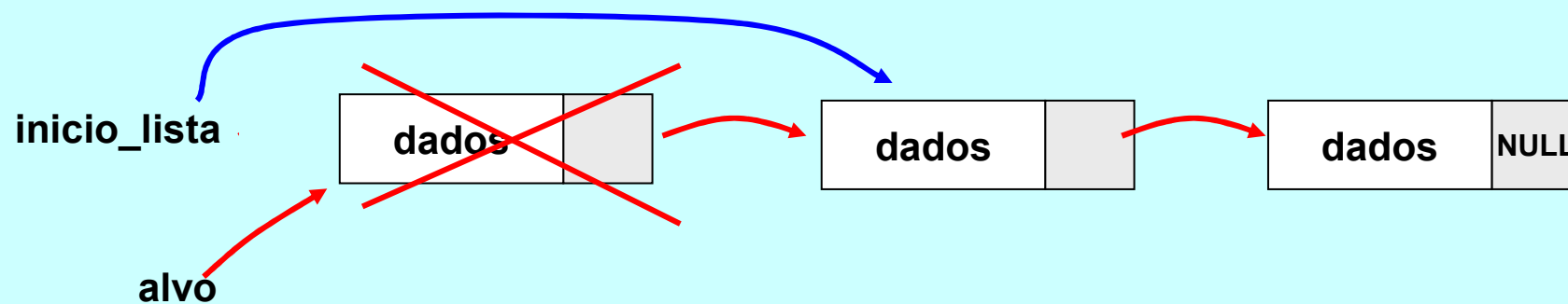
```
alvo = inicio_lista;
```

# Listas Ligadas: remover do inicio (não vazia)



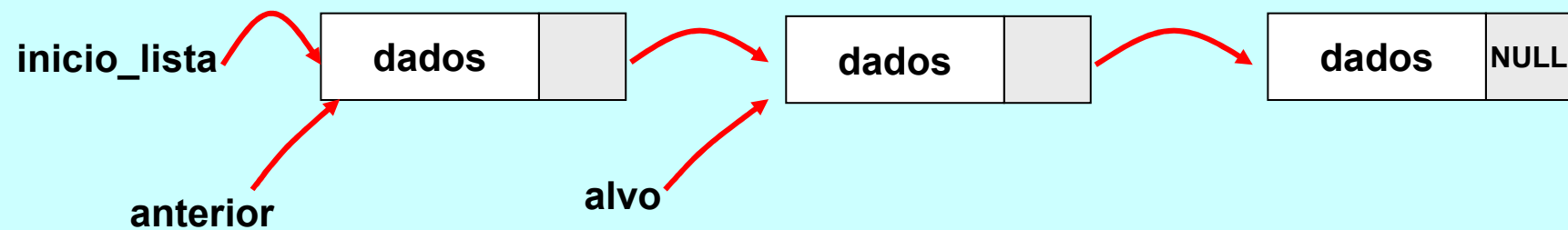
```
alvo = inicio_lista;  
inicio_lista = inicio_lista->proximo;  
// ou: inicio_lista = alvo->proximo;
```

# Listas Ligadas: remover do inicio (não vazia)



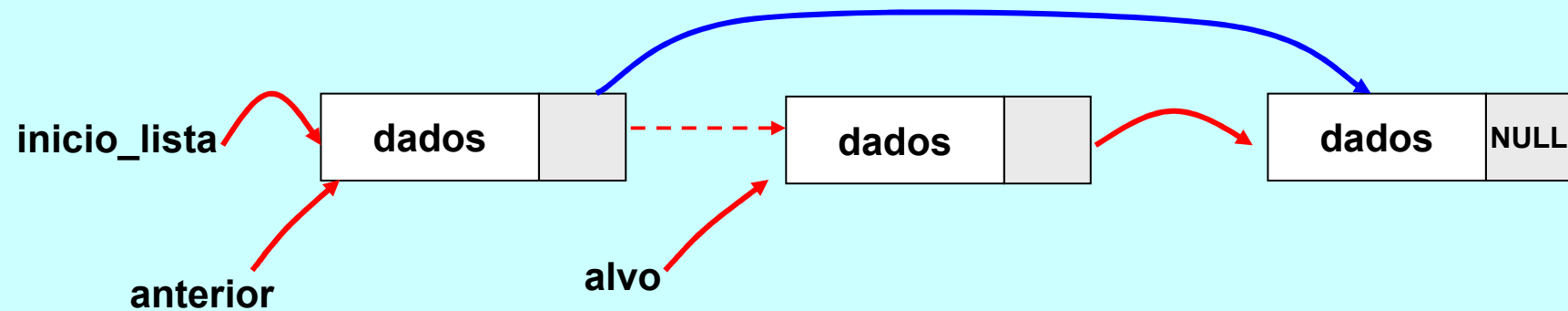
```
alvo = inicio_lista;  
inicio_lista = inicio_lista->proximo;  
// ou: inicio_lista = alvo->proximo;  
free(alvo);
```

# Listas Ligadas: remover no meio ou fim



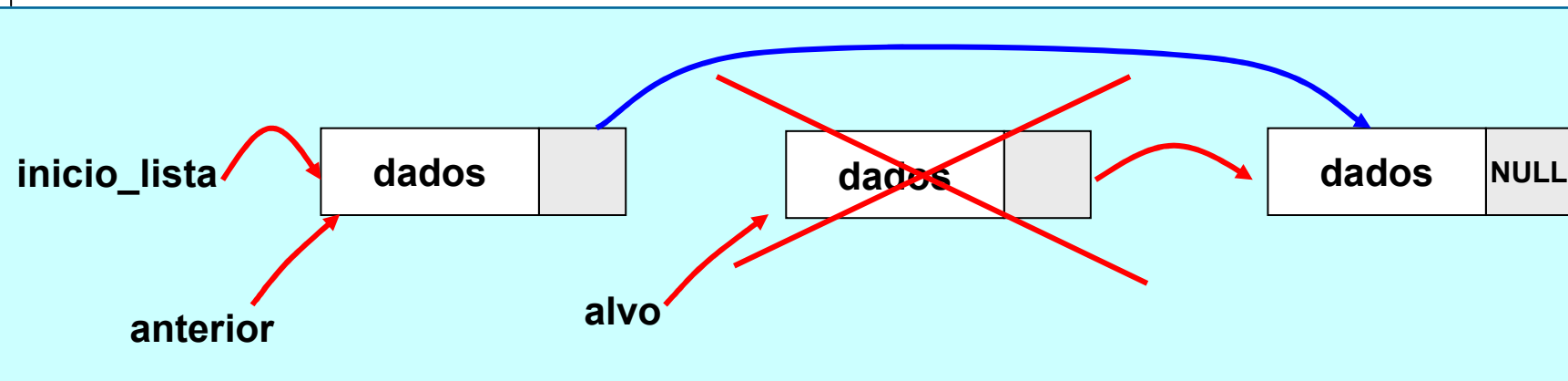
```
// anterior aponta para o elemento  
// imediatamente antes de alvo
```

# Listas Ligadas: remover no meio ou fim



```
// anterior aponta para o elemento  
// imediatamente antes de alvo  
anterior->proximo = alvo->proximo;
```

# Listas Ligadas: remover no meio ou fim

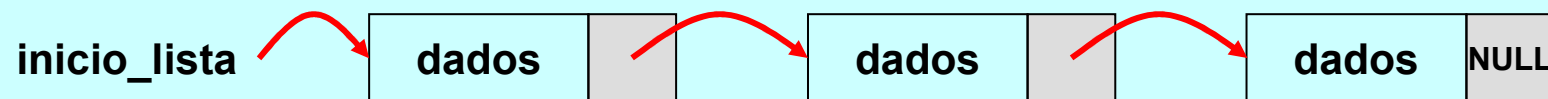


```
// anterior aponta para o elemento
// imediatamente antes de alvo
anterior->proximo = alvo->proximo;
free(alvo);
```

# Listas Ligadas

## Vantagens/Desvantagens:

- Vantagens:
  - Fácil aumento/diminuição no tamanho da lista
  - Reserva memória exatamente sob medida
  - Permite mais de um critério de ordenação

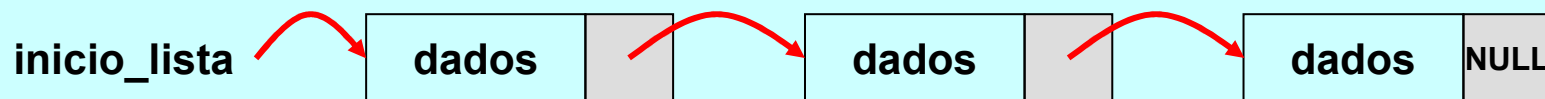




# Listas Ligadas

## Vantagens/Desvantagens:

- Vantagens:
  - Fácil aumento/diminuição no tamanho da lista
  - Reserva memória exatamente sob medida
  - Permite mais de um critério de ordenação
- Desvantagens:
  - Para cada elemento, armazena um apontador
  - Não é possível acessar diretamente um elemento



# Memória Dinâmica



- Vetor2Lista
- Vetor2ListaRec
- ConcatListas
- ConcatListasRec
- InverteLista
- InverteListaRec
- InverteListaNaMemoria
- Vetor2ListaOrd
- Vetor2ListaOrdRec
- OperaListasRec
- OperaComFilas
- Lista2ListaDupla
- Express2Arvore
- DicionarioBinario