



# LISTAS SIMPLEMENTE ENCADEADA

# SUMÁRIO

- Estruturas de Dados
- **Listas Simplesmente Encadeada**
- Listas Duplamente Encadeada
- Filas
- Pilhas
- Grafos
- Árvores

ANDRÉ BACKES

# Estrutura de dados descomplicada em linguagem

C

# LISTAS: DEFINIÇÃO

O conceito de lista é algo muito comum para as pessoas.

Trata-se de um relação finita de itens, todos eles contidos dentro de um mesmo tema.

Vários são os exemplos possíveis de listas: itens em estoque em uma empresa, dias da semana, lista de compras do supermercado, etc.

Quanto à inserção/remoção de elementos da lista, temos:

- **Lista convencional:** pode ter elementos inseridos ou removidos de qualquer lugar dela.
- **Fila:** estrutura do tipo FIFO (First In First Out), os elementos só podem ser inseridos no final, e acessados ou removidos do início da lista. Mais informações no Capítulo 6.
- **Pilha:** estrutura do tipo LIFO (Last In First Out), os elementos só podem ser inseridos, acessados ou removidos do final da lista. Mais informações no Capítulo 8.

# OPERAÇÕES BÁSICAS DE UMA LISTA

- Criação da lista.
- Inserção de um elemento na lista.
- Remoção de um elemento da lista.
- Busca por um elemento da lista.
- Destruuição da lista.
- Além de informações com tamanho, se a lista está cheia ou vazia.

# INSERÇÃO



Existem três tipos de inserção: inserção no início, no final ou no meio (isto é, entre dois elementos) da lista.

A operação de inserção no meio da lista é comumente usada quando se deseja inserir um elemento de forma ordenada na lista.



A operação de inserção envolve o teste de estouro da lista, ou seja, precisamos verificar se é possível inserir um novo elemento na lista (ela ainda não está cheia).

# REMOÇÃO



Existem três tipos de remoção: remoção do início, do final ou do meio (isto é, entre dois elementos) da lista.

A operação de remoção do meio da lista é comumente usada quando se deseja remover um elemento específico da lista.



A operação de remoção envolve o teste de lista vazia, ou seja, precisamos verificar se existem elementos dentro da lista antes de tentar removê-los.

# ACESSO AOS ELEMENTOS DA LISTA

- **Acesso sequencial:** os elementos são armazenados de forma consecutiva na memória (como em um array ou vetor). A posição de um elemento pode ser facilmente obtida a partir do início da lista.
- **Acesso encadeado:** cada elemento pode estar em uma área distinta da memória, não necessariamente consecutivas. É necessário que cada elemento da lista armazene, além da sua informação, o endereço de memória onde se encontra o próximo elemento. Para acessar um elemento, é preciso percorrer todos os seus antecessores na lista.

# ALOCAÇÃO DE LISTAS

- **Alocação estática:** o espaço de memória é alocado no momento da compilação do programa. É necessário definir o número máximo de elementos que a lista irá possuir.
- **Alocação dinâmica:** o espaço de memória é alocado em tempo de execução. A lista cresce à medida que novos elementos são armazenados, e diminui à medida que elementos são removidos.

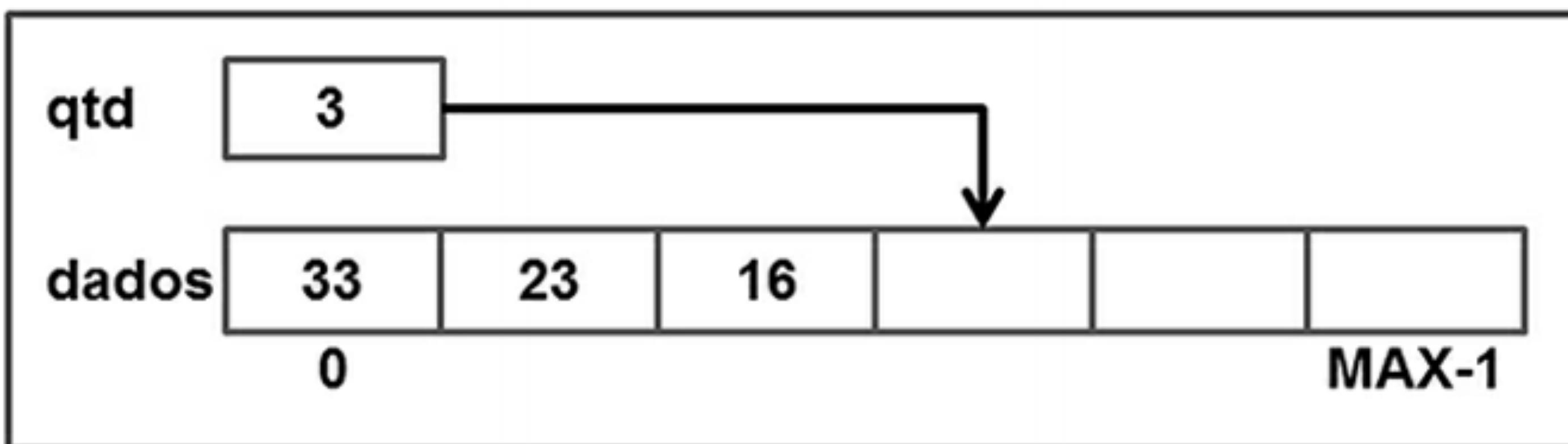
# I) LISTA SEQUENCIAL ESTÁTICA

Ela é definida utilizando um array, de modo que o sucessor de um elemento ocupa a posição física seguinte deste.



Além do array, essa lista utiliza um campo adicional (**qtd**) que serve para indicar o quanto do array já está ocupado pelos elementos (dados) inseridos na lista.

```
Lista *li;
```



# VANTAGENS X DESVANTAGENS

Várias são as vantagens em se definir uma lista utilizando um array

- Acesso rápido e direto aos elementos (índice do array).
- Tempo constante para acessar um elemento.
- Facilidade para modificar as suas informações.

Infelizmente, o uso de arrays também tem suas desvantagens:

- Definição prévia do tamanho do array e, consequentemente, da lista.
- Dificuldade para inserir e remover um elemento entre outros dois: é necessário deslocar os elementos para abrir espaço dentro do array.

# QUANDO USAR UMA LISTA LINEAR

- Listas pequenas.
- Inserção e remoção apenas no final da lista.
- Tamanho máximo da lista bem definido.
- A busca é a operação mais frequente.

## Arquivo ListaSequencial.h

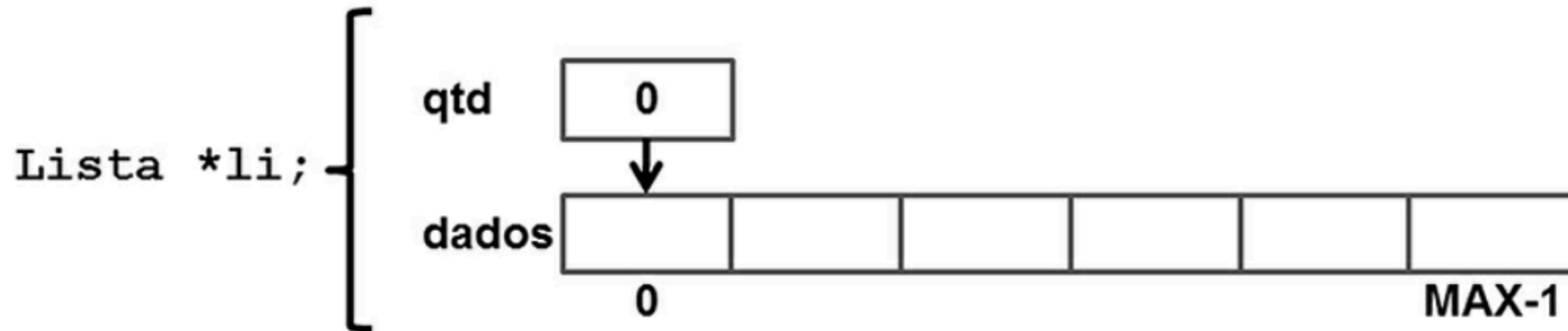
```
01 #define MAX 100
02 struct aluno{
03     int matricula;
04     char nome[30];
05     float n1,n2,n3;
06 };
07 typedef struct lista Lista;
08
09 Lista* cria_lista();
10 void libera_lista(Lista* li);
11 int busca_lista_pos(Lista* li, int pos, struct aluno *al);
12 int busca_lista_mat(Lista* li, int mat, struct aluno *al);
13 int insere_lista_final(Lista* li, struct aluno al);
14 int insere_lista_inicio(Lista* li, struct aluno al);
15 int insere_lista_ordenada(Lista* li, struct aluno al);
16 int remove_lista(Lista* li, int mat);
17 int remove_lista_inicio(Lista* li);
18 int remove_lista_final(Lista* li);
19 int tamanho_lista(Lista* li);
20 int lista_cheia(Lista* li);
21 int lista_vazia(Lista* li);
```

## Arquivo ListaSequencial.c

```
01 #include <stdio.h>
02 #include <stdlib.h>
03 #include "ListaSequencial.h" //inclui os protótipos
04 //Definição do tipo lista
05 struct lista{
06     int qtd;
07     struct aluno dados[MAX];
08 };
```

## Criando uma lista

```
01 Lista* cria_lista() {  
02     Lista *li;  
03     li = (Lista*) malloc(sizeof(struct lista));  
04     if(li != NULL)  
05         li->qtd = 0;  
06     return li;  
07 }
```



## Destruindo uma lista

```
01 void libera_lista(Lista* li) {  
02     free(li);  
03 }
```

## Tamanho da lista

```
01 int tamanho_lista(Lista* li) {  
02     if(li == NULL)  
03         return -1;  
04     else  
05         return li->qtd;  
06 }
```



Basicamente, retornar se uma **lista sequencial estática** está cheia consiste em verificar se o valor do seu campo **qtd** é igual a **MAX**.

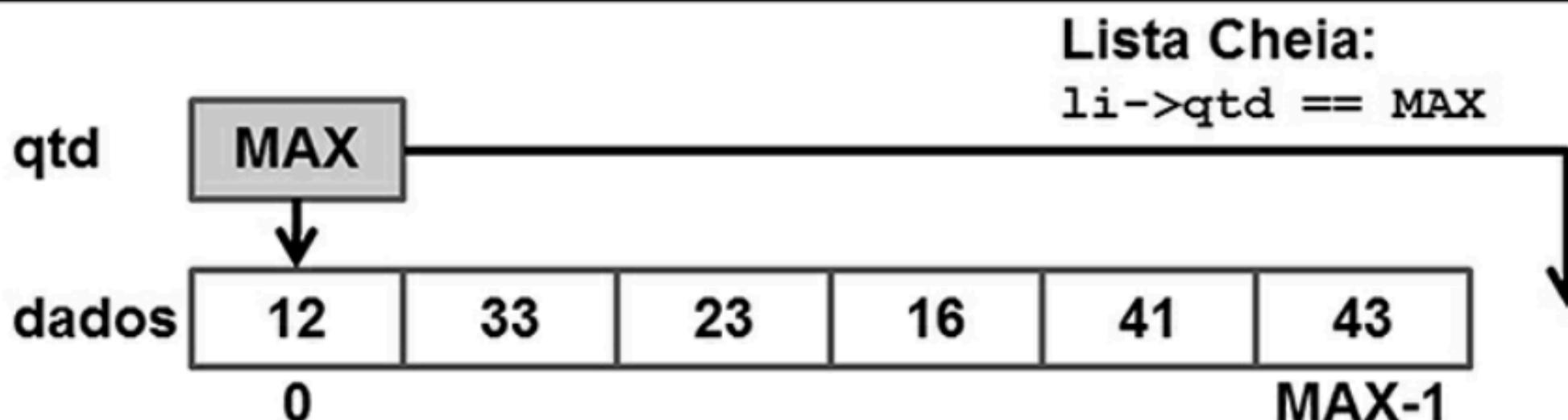


FIGURA 5.9

## Retornando se a lista está cheia

```
01 int lista_cheia(Lista* li) {  
02     if(li == NULL)  
03         return -1;  
04     return (li->qtd == MAX);  
05 }
```



Basicamente, retornar se uma **lista sequencial estática** está vazia consiste em verificar se o valor do seu campo **qtd** é igual a **ZERO**.

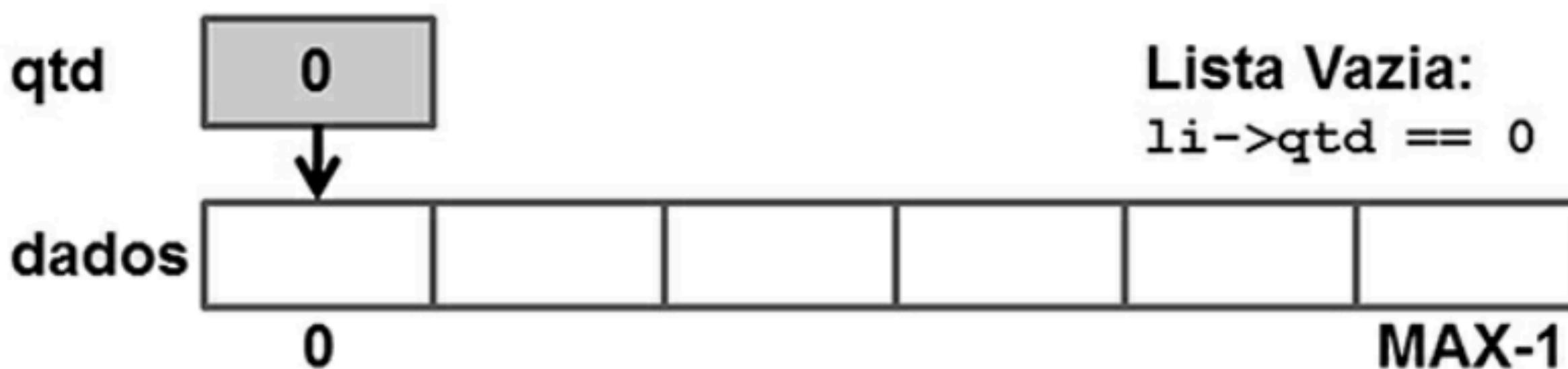


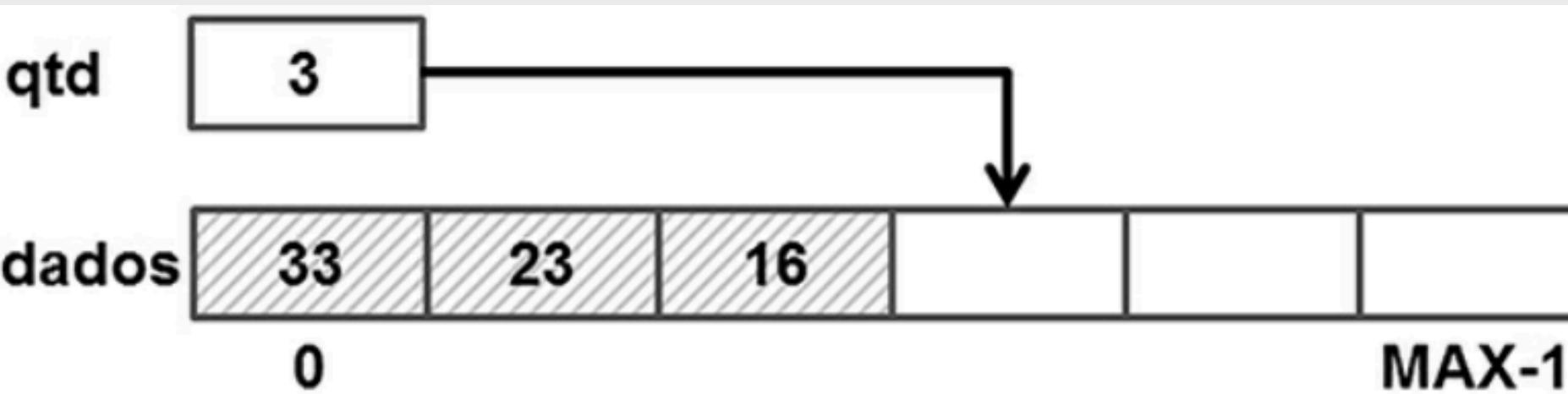
FIGURA 5.11

### Retornando se a lista está vazia

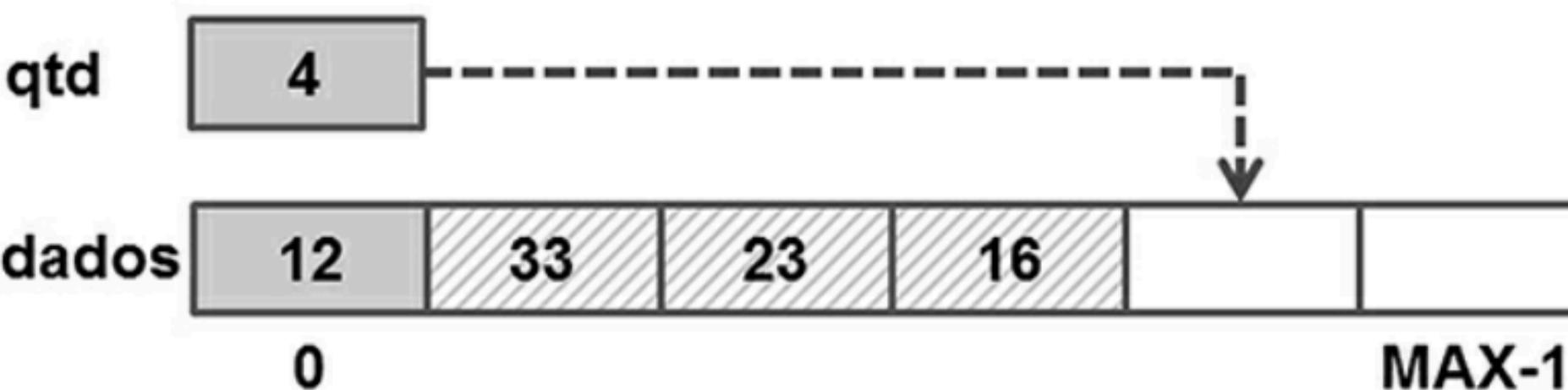
```
01 int lista_vazia(Lista* li) {  
02     if(li == NULL)  
03         return -1;  
04     return (li->qtd == 0);  
05 }
```

## Inserindo um elemento no início da lista

```
01 int insere_lista_inicio(Lista* li, struct aluno al){  
02     if(li == NULL)  
03         return 0;  
04     if(li->qtd == MAX) //lista cheia  
05         return 0;  
06     int i;  
07     for(i=li->qtd-1; i>=0; i--)  
08         li->dados[i+1] = li->dados[i];  
09     li->dados[0] = al;  
10     li->qtd++;  
11     return 1;  
12 }
```



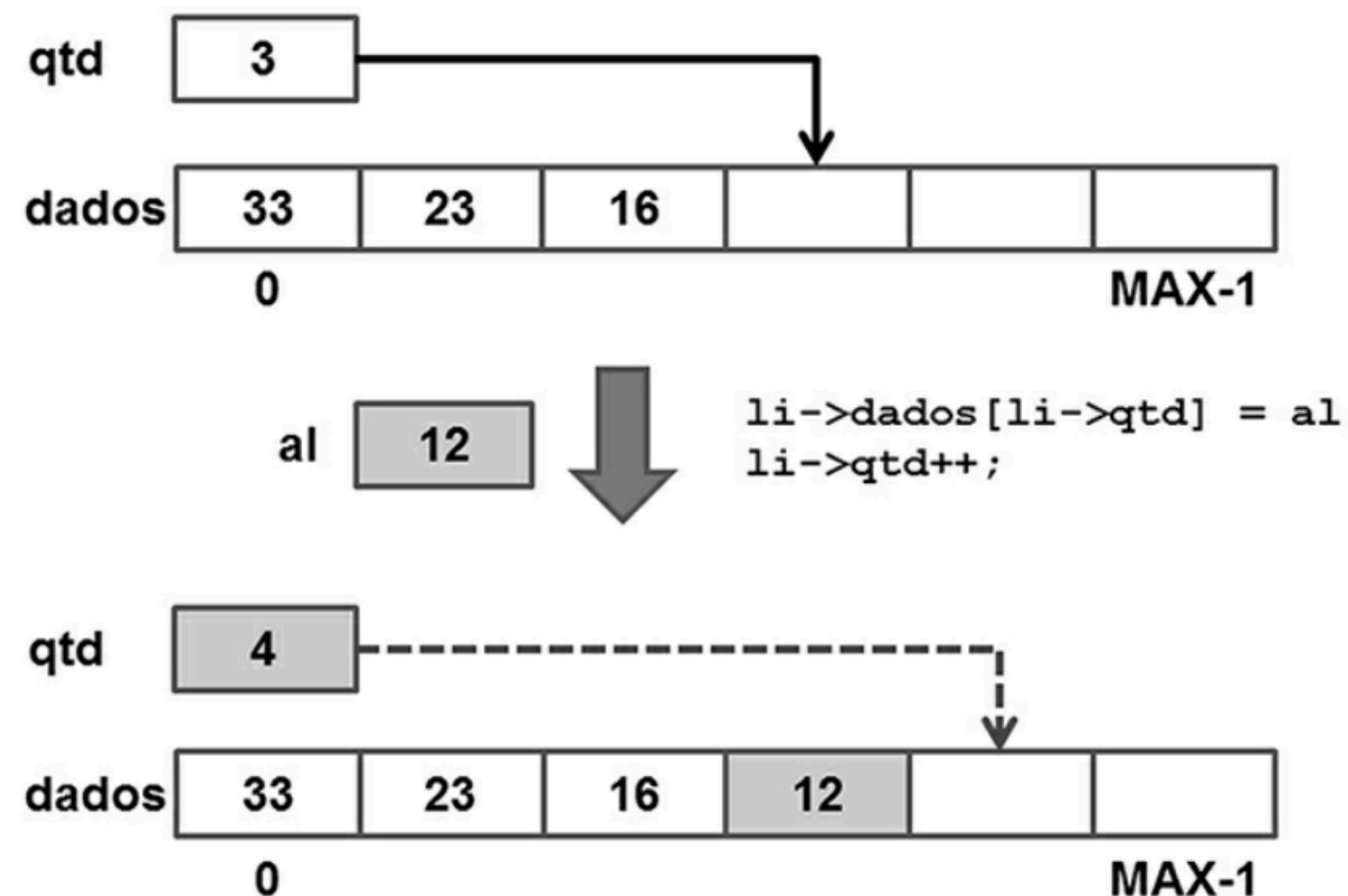
**al** **Desloca os elementos uma posição para frente e insere:**  
**for(i=li->qtd-1; i>=0; i--)**  
 **li->dados[i+1] = li->dados[i];**  
**li->dados[0] = al;**  
**li->qtd++;**



Ao inserir no início da lista, deve-se deslocar os elementos à direita.

## Inserindo um elemento no final da lista

```
01 int insere_lista_final(Lista* li, struct aluno al){  
02     if(li == NULL)  
03         return 0;  
04     if(li->qtd == MAX)//lista cheia  
05         return 0;  
06     li->dados[li->qtd] = al;  
07     li->qtd++;  
08     return 1;  
09 }
```



Inserir um elemento de forma ordenada em uma **lista sequencial estática** é uma tarefa simples, mas trabalhosa.



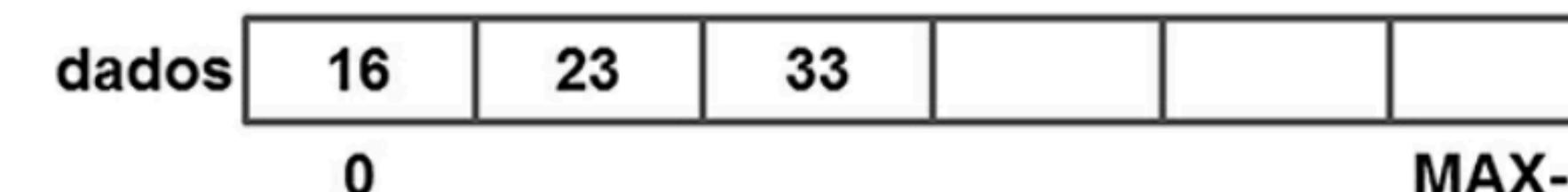
Isso ocorre porque precisamos procurar o ponto de inserção do elemento na lista, o qual pode ser no início, no meio ou no final da lista. Nos dois primeiros casos (início ou meio), é preciso mudar o lugar dos demais elementos da lista.

#### Inserindo um elemento de forma ordenada na lista

```
01 int insere_lista_ordenada(Lista* li, struct aluno al) {  
02     if(li == NULL)  
03         return 0;  
04     if(li->qtd == MAX)//lista cheia  
05         return 0;  
06     int k,i = 0;  
07     while(i<li->qtd && li->dados[i].matricula < al.matricula)  
08         i++;  
09  
10    for(k=li->qtd-1; k >= i; k--)  
11        li->dados[k+1] = li->dados[k];  
12  
13    li->dados[i] = al;  
14    li->qtd++;  
15    return 1;  
16 }
```

# INSERÇÃO ORDENADA DE ELEMENTOS

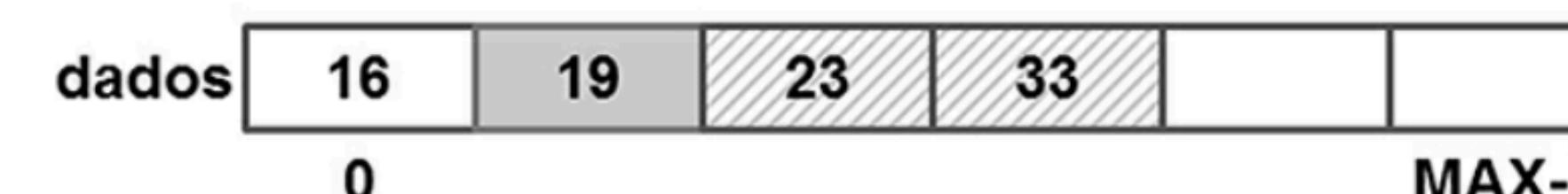
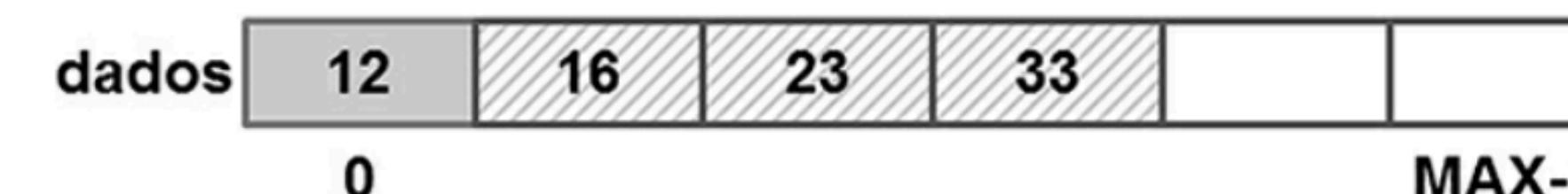
**Lista inicial  
Busca onde Inserir:**



```
int k,i = 0;  
while(i < li->qtd && li->dados[i].matricula < al.matricula)  
    i++;
```

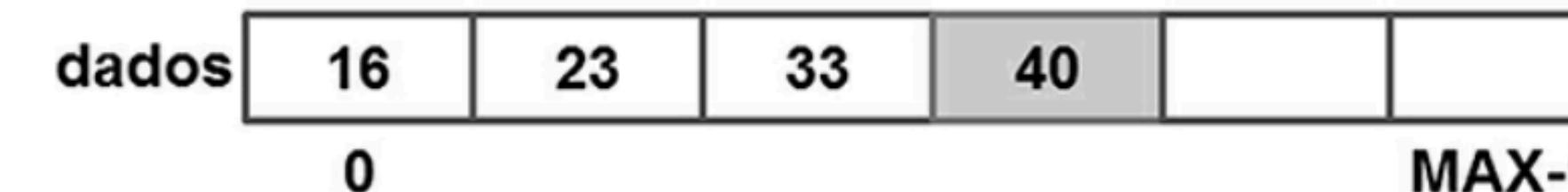
-----  
**Inserção no início ou no meio: desloca elementos**

```
for(k=li->qtd-1; k >= i; k--)  
    li->dados[k+1] = li->dados[k];
```



-----  
**Inserir elemento**

```
li->dados[i] = al;  
li->qtd++;
```



# REMOÇÃO DO INÍCIO



No caso de uma lista implementada usando um array, ela somente será considerada vazia quando a quantidade de elementos (campo **qtd**) for igual ao valor **ZERO**, indicando que nenhuma posição do array está ocupada por elementos.



Como na inserção no início, a remoção de um elemento do início de uma **lista sequencial estática** necessita que se mude o lugar dos demais elementos da lista.

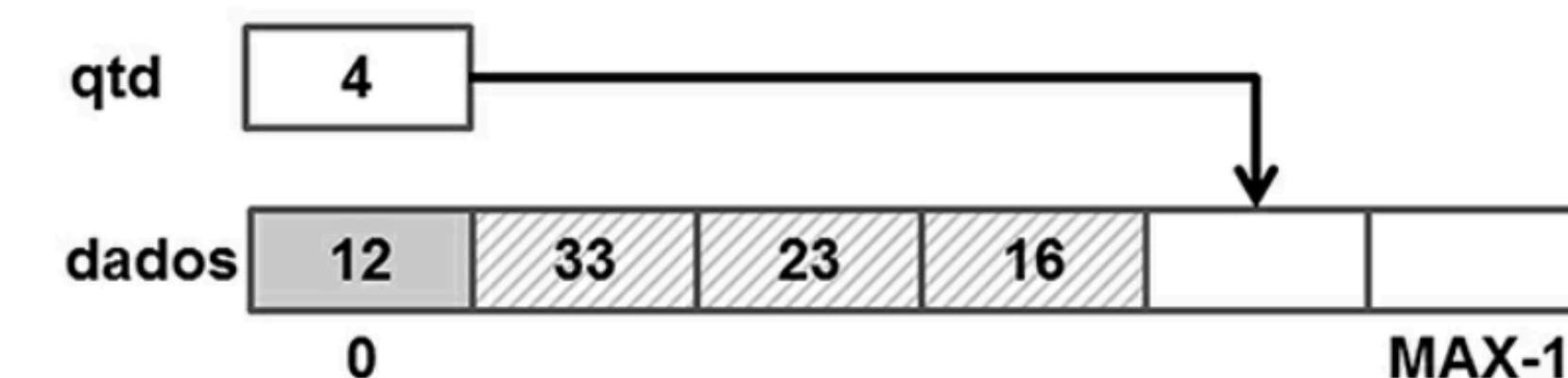


Note que o último elemento da lista fica duplicado. Isso não é um problema, já que aquela posição duplicada é considerada não ocupada por elementos da lista.

## Removendo um elemento do início da lista

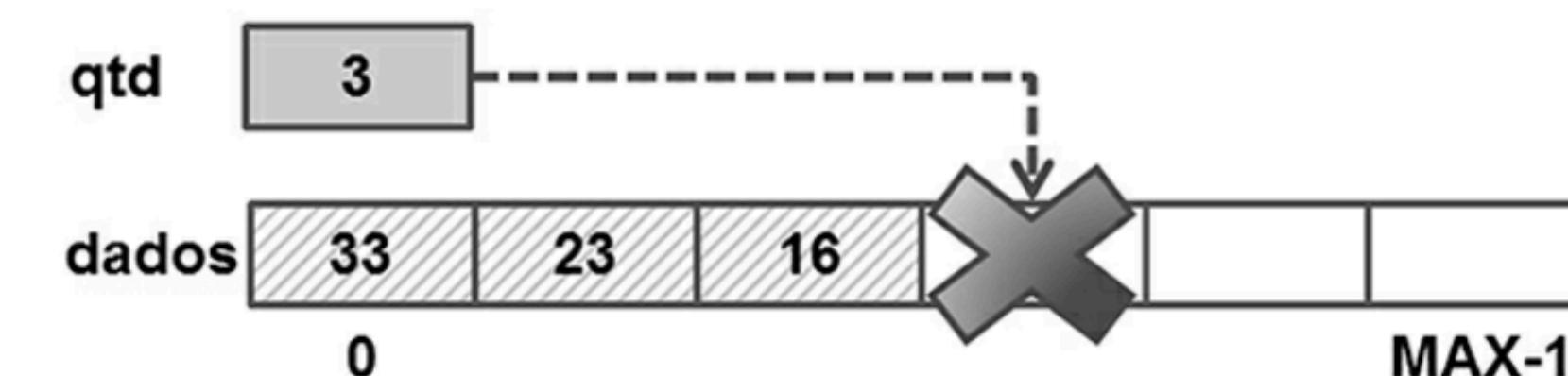
```
01 int remove_lista_inicio(Lista* li) {  
02     if(li == NULL)  
03         return 0;  
04     if(li->qtd == 0)//lista vazia  
05         return 0;  
06     int k = 0;  
07     for(k=0; k< li->qtd-1; k++)  
08         li->dados[k] = li->dados[k+1];  
09     li->qtd--;  
10 }  
11 }
```

FIGURA 5.19



Desloca os elementos uma posição para trás:

```
for(k=0; k< li->qtd-1; k++)  
    li->dados[k] = li->dados[k+1];  
li->qtd--;
```



# REMOÇÃO DO FIM



Diferente da remoção do início, a remoção do final de uma **lista sequencial estática** não necessita que se mude o lugar dos demais elementos da lista. Temos apenas que alterar a quantidade de elementos na lista.



Note que o elemento removido continua no final da lista. Isso não é um problema, já que aquela posição é considerada não ocupada por elementos da lista.

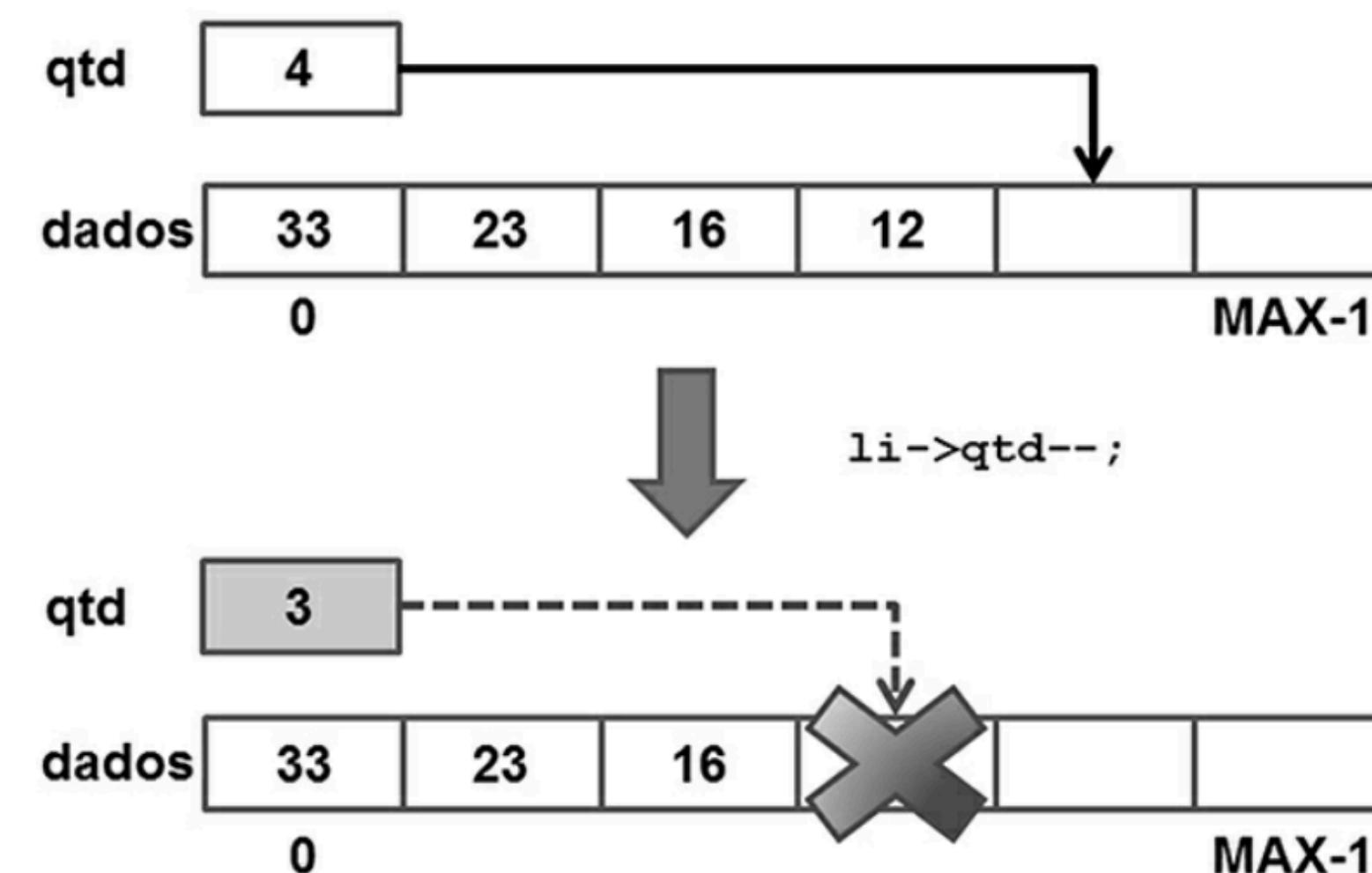


Isso ocorre porque precisamos procurar o elemento a ser removido na lista, o qual pode estar no início, no meio ou no final da lista. Nos dois primeiros casos (início ou meio), é preciso mudar o lugar dos demais elementos da lista após a remoção.

## Removendo um elemento do final da lista

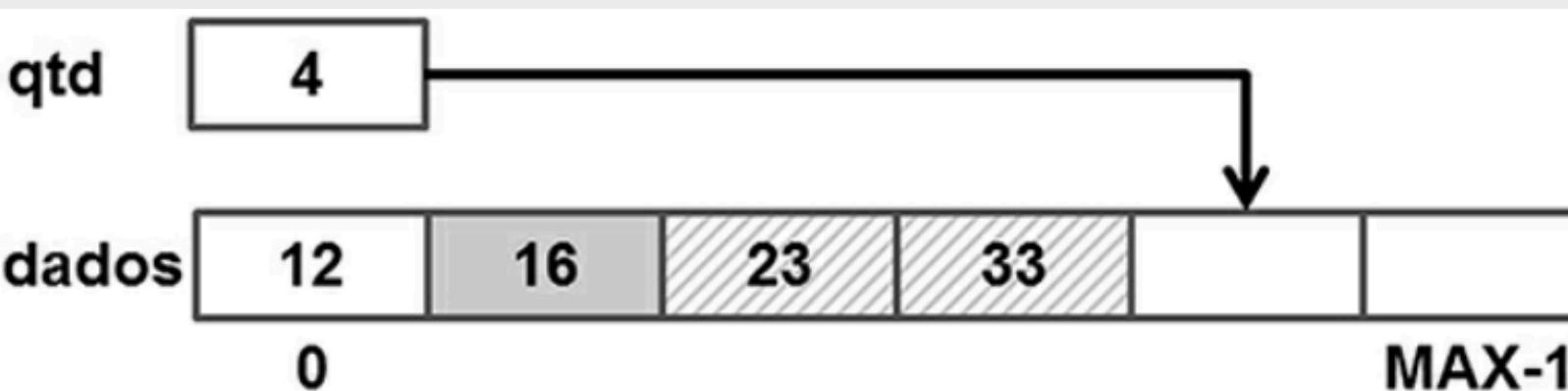
```
01 int remove_lista_final(Lista* li) {  
02     if(li == NULL)  
03         return 0;  
04     if(li->qtd == 0) //lista vazia  
05         return 0;  
06     li->qtd--;  
07     return 1;  
08 }
```

FIGURA 5.21



## Removendo um elemento específico da lista

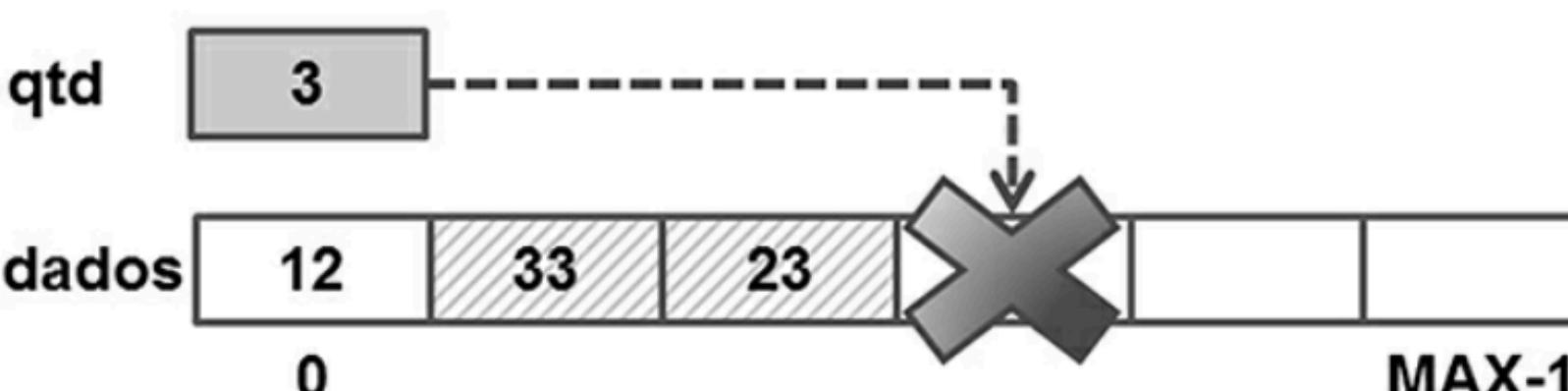
```
01 int remove_lista(Lista* li, int mat) {
02     if(li == NULL)
03         return 0;
04     if(li->qtd == 0)//lista vazia
05         return 0;
06     int k, i = 0;
07     while(i<li->qtd && li->dados[i].matricula != mat)
08         i++;
09     if(i == li->qtd)//elemento não encontrado
10         return 0;
11
12     for(k=i; k< li->qtd-1; k++)
13         li->dados[k] = li->dados[k+1];
14     li->qtd--;
15     return 1;
16 }
```



Procura elemento a ser removido:

```
while(i<li->qtd && li->dados[i].matricula != mat)
    i++;
```

Desloca os elementos uma posição para trás:  
for(k=i; k< li->qtd-1; k++)
 li->dados[k] = li->dados[k+1];
 li->qtd--;

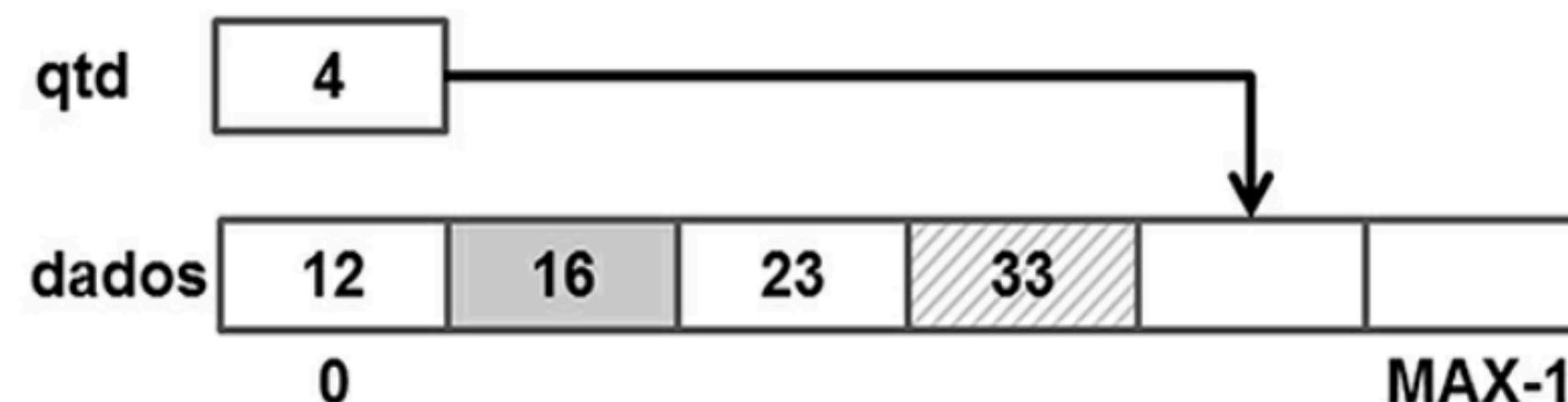


# OTIMIZANDO A REMOÇÃO

## Removendo um elemento específico da lista

```
01 int remove_lista_otimizado(Lista* li, int mat) {  
02     if(li == NULL)  
03         return 0;  
04     if(li->qtd == 0)  
05         return 0;  
06     int i = 0;  
07     while(i<li->qtd && li->dados[i].matricula != mat)  
08         i++;  
09     if(i == li->qtd) //elemento não encontrado  
10         return 0;  
11     li->qtd--;  
12     li->dados[i] = li->dados[li->qtd];  
13     return 1;  
14 }  
15 }
```

 Não se esqueça: a otimização da operação de remoção somente deve ser utilizada quando alterar a ordem dos elementos da lista não comprometer o desempenho da aplicação. Se a ordem dos elementos é um atributo importante da lista, então essa otimização não deve ser utilizada.

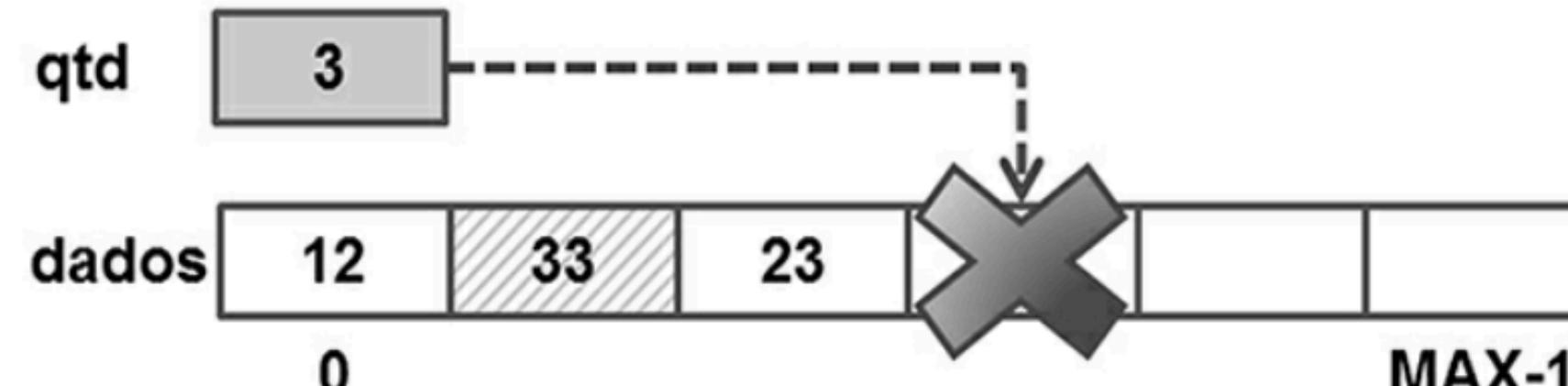


Procura elemento a ser removido:

```
while(i<li->qtd && li->dados[i].matricula != mat)  
    i++;
```

Copia o último elemento da lista para o lugar do elemento removido:

```
li->qtd--;  
li->dados[i] = li->dados[li->qtd];
```

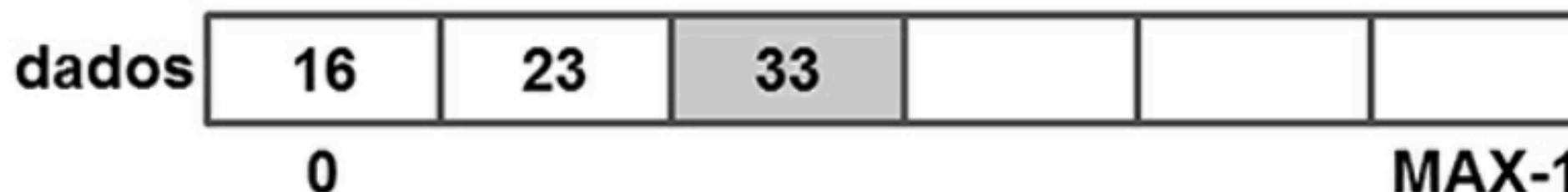


### Busca um elemento por posição

```
01 int busca_lista_pos(Lista* li,int pos,struct aluno *al){  
02     if(li == NULL || pos <= 0 || pos > li->qtd)  
03         return 0;  
04     *al = li->dados[pos-1];  
05     return 1;  
06 }
```

FIGURA 5.27

Posição: 3º



\*al = li->dados[pos-1];



Na busca por conteúdo, precisamos percorrer a lista à procura do elemento desejado.

## Busca um elemento por conteúdo

```
01 int busca_lista_mat(Lista* li, int mat, struct aluno *al){  
02     if(li == NULL)  
03         return 0;  
04     int i = 0;  
05     while(i<li->qtd && li->dados[i].matricula != mat)  
06         i++;  
07     if(i == li->qtd) //elemento não encontrado  
08         return 0;  
09  
10    *al = li->dados[i];  
11    return 1;  
12 }
```



Perceba que a primeira condição do comando **while** será sempre falsa se a lista estiver vazia (campo **qtd** igual a **ZERO**). Assim, não é preciso tratar isoladamente esse caso.

### Conteúdo: 23

dados	16	23	33			
	0					MAX-1

### Busca pelo elemento:

```
while(i<li->qtd && li->dados[i].matricula != mat)  
    i++;
```

### Achou o elemento:

```
*al = li->dados[i];
```

## 2) LISTA DINÂMICA ENCADEADA

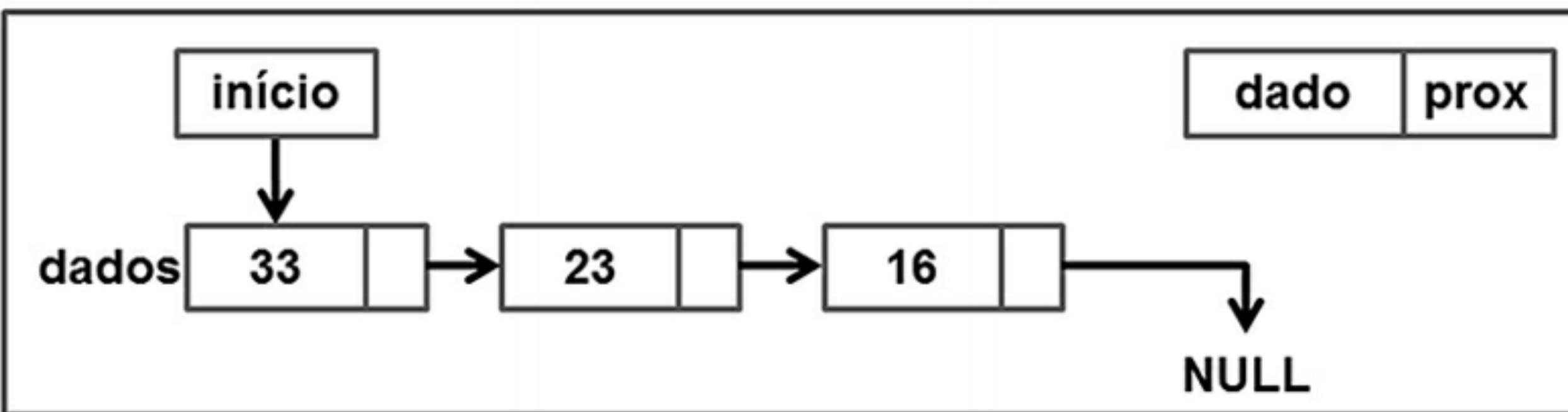
Cada elemento da lista é alocado dinamicamente, à medida que os dados são inseridos dentro da lista, e tem sua memória liberada, à medida que é removido.

Esse elemento nada mais é do que um ponteiro para uma estrutura contendo dois campos de informação: um campo de dado, utilizado para armazenar a informação inserida na lista, e um campo prox, que nada mais é do que um ponteiro que indica o próximo elemento na lista.



Após o último elemento, não existe nenhum novo elemento alocado. Sendo assim, o último elemento da lista aponta para **NULL**.

`Lista *li`



# DIFERENÇA NA DECLARAÇÃO



Desse modo, criamos um indicador que nunca muda sua posição na memória e que aponta para o início da **lista dinâmica encadeada**.

- Lista \*li; //Declaração de uma **lista sequencial estática** (ponteiro).
- Lista \*li; //Declaração de uma **lista dinâmica encadeada** (ponteiro para ponteiro).

## Arquivo ListaDinEncad.h

```
01 struct aluno{  
02     int matricula;  
03     char nome[30];  
04     float n1,n2,n3;  
05 };  
06 typedef struct elemento* Lista;  
07  
08 Lista* cria_lista();  
09 void libera_lista(Lista* li);  
10 int insere_lista_final(Lista* li, struct aluno al);  
11 int insere_lista_inicio(Lista* li, struct aluno al);  
12 int insere_lista_ordenada(Lista* li, struct aluno al);  
13 int remove_lista(Lista* li, int mat);  
14 int remove_lista_inicio(Lista* li);  
15 int remove_lista_final(Lista* li);  
16 int tamanho_lista(Lista* li);  
17 int lista_vazia(Lista* li);  
18 int lista_cheia(Lista* li);  
19 int busca_lista_mat(Lista* li, int mat, struct aluno *al);  
20 int busca_lista_pos(Lista* li, int pos, struct aluno *al);
```

## Arquivo ListaDinEncad.c

```
01 #include <stdio.h>  
02 #include <stdlib.h>  
03 #include "ListaDinEncad.h" //inclui os protótipos  
04 //Definição do tipo lista  
05 struct elemento{  
06     struct aluno dados;  
07     struct elemento *prox;  
08 };  
09 typedef struct elemento Elem;
```

## Criando uma lista

```
01 Lista* cria_lista() {  
02     Lista* li = (Lista*) malloc(sizeof(Lista));  
03     if(li != NULL)  
04         *li = NULL;  
05     return li;  
06 }
```



Para liberar uma lista que utilize alocação dinâmica, e seja encadeada, é preciso percorrer toda a lista liberando a memória alocada para cada elemento inserido nela.

Lista \*li



início → NULL

## Destruindo uma lista

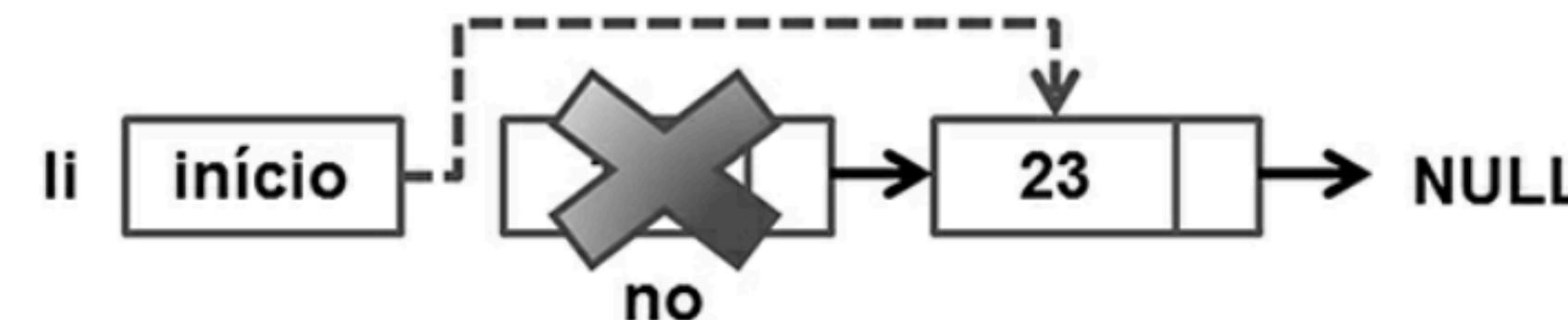
```
01 void libera_lista(Lista* li) {  
02     if(li != NULL){  
03         Elemt no;  
04         while((*li) != NULL){  
05             no = *li;  
06             *li = (*li)->prox;  
07             free(no);  
08         }  
09         free(li);  
10     }  
11 }
```

**Lista inicial**



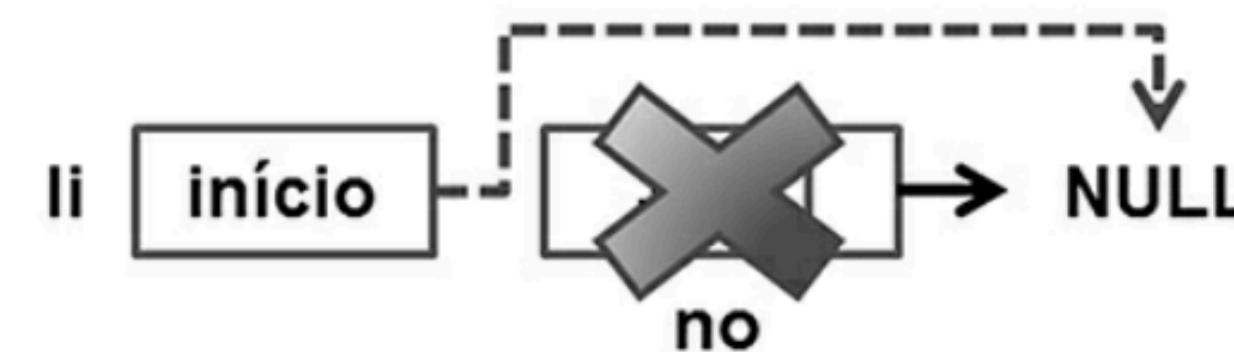
**Passo 1:**

```
no = *li;  
*li = (*li)->prox;  
free(no);
```



**Passo 2:**

```
no = *li;  
*li = (*li)->prox;  
free(no);
```



**Fim:**

```
no == NULL
```



## Tamanho da lista

```
01 int tamanho_lista(Lista* li) {  
02     if(li == NULL)  
03         return 0;  
04     int cont = 0;  
05     Elem* no = *li;  
06     while(no != NULL) {  
07         cont++;  
08         no = no->prox;  
09     }  
10     return cont;  
11 }
```



Para saber o tamanho de uma lista que utilize alocação dinâmica, e seja encadeada, é preciso percorrer toda a lista contando os elementos inseridos nela, até encontrar o seu final.

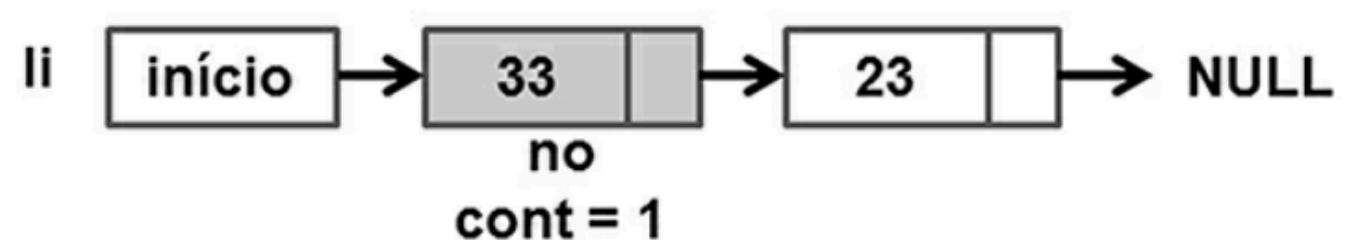
**Lista inicial:**

```
cont = 0;  
no = *li;
```



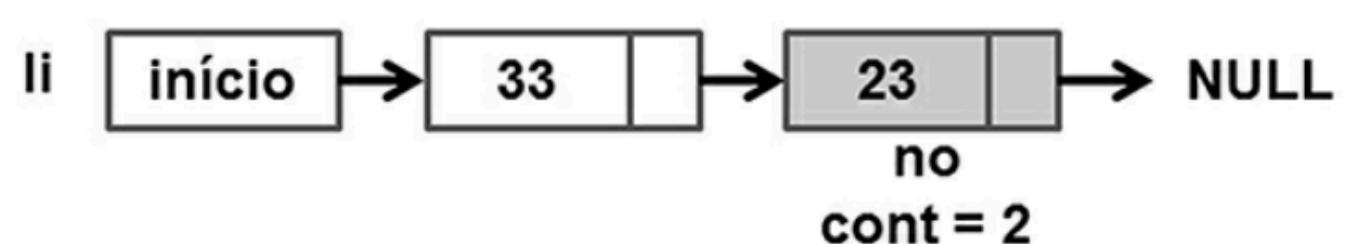
**Passo 1:**

```
cont++;  
no = no->prox;
```



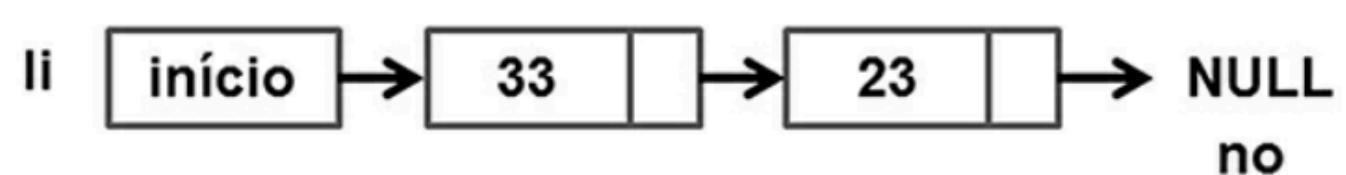
**Passo 2:**

```
cont++;  
no = no->prox;
```



**Fim:**

```
no == NULL
```



## Retornando se a lista está vazia

```
01 int lista_vazia(Lista* li) {  
02     if(li == NULL)  
03         return 1;  
04     if(*li == NULL)  
05         return 1;  
06     return 0;  
07 }
```



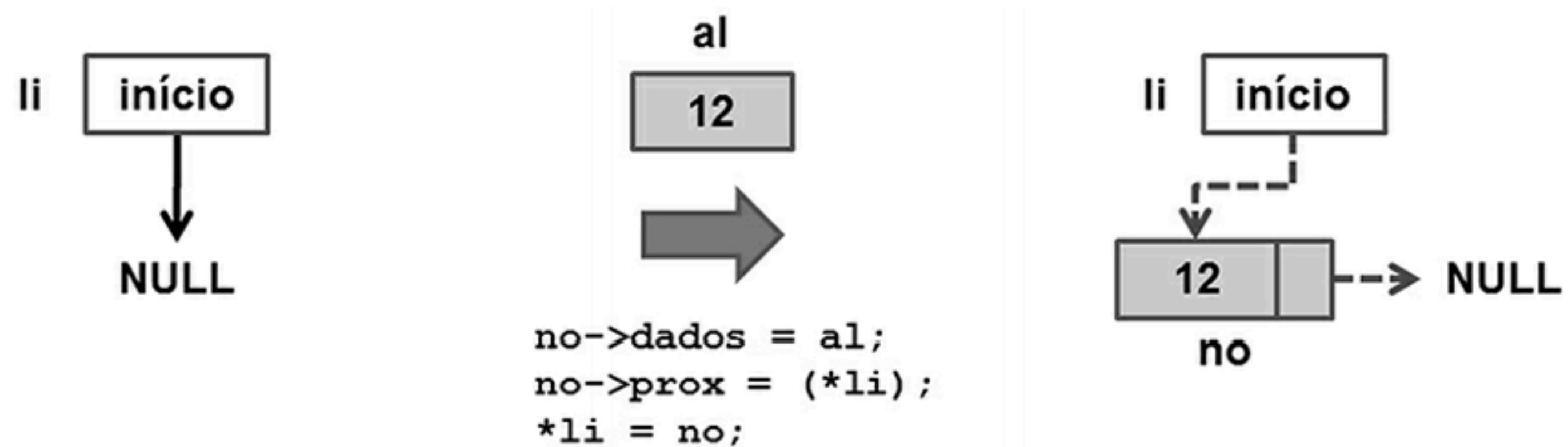
No caso de uma lista com alocação dinâmica, ela somente será considerada cheia quando não tivermos mais memória disponível no computador para alocar novos elementos. Isso ocorrerá apenas quando a chamada da função **malloc()** retornar **NULL**.

## Exemplo

```
01 int lista_cheia(Lista* li) {  
02     return 0;  
03 }
```



Uma **lista dinâmica encadeada** será considerada vazia sempre que o conteúdo do seu “início” apontar para a constante **NULL**.

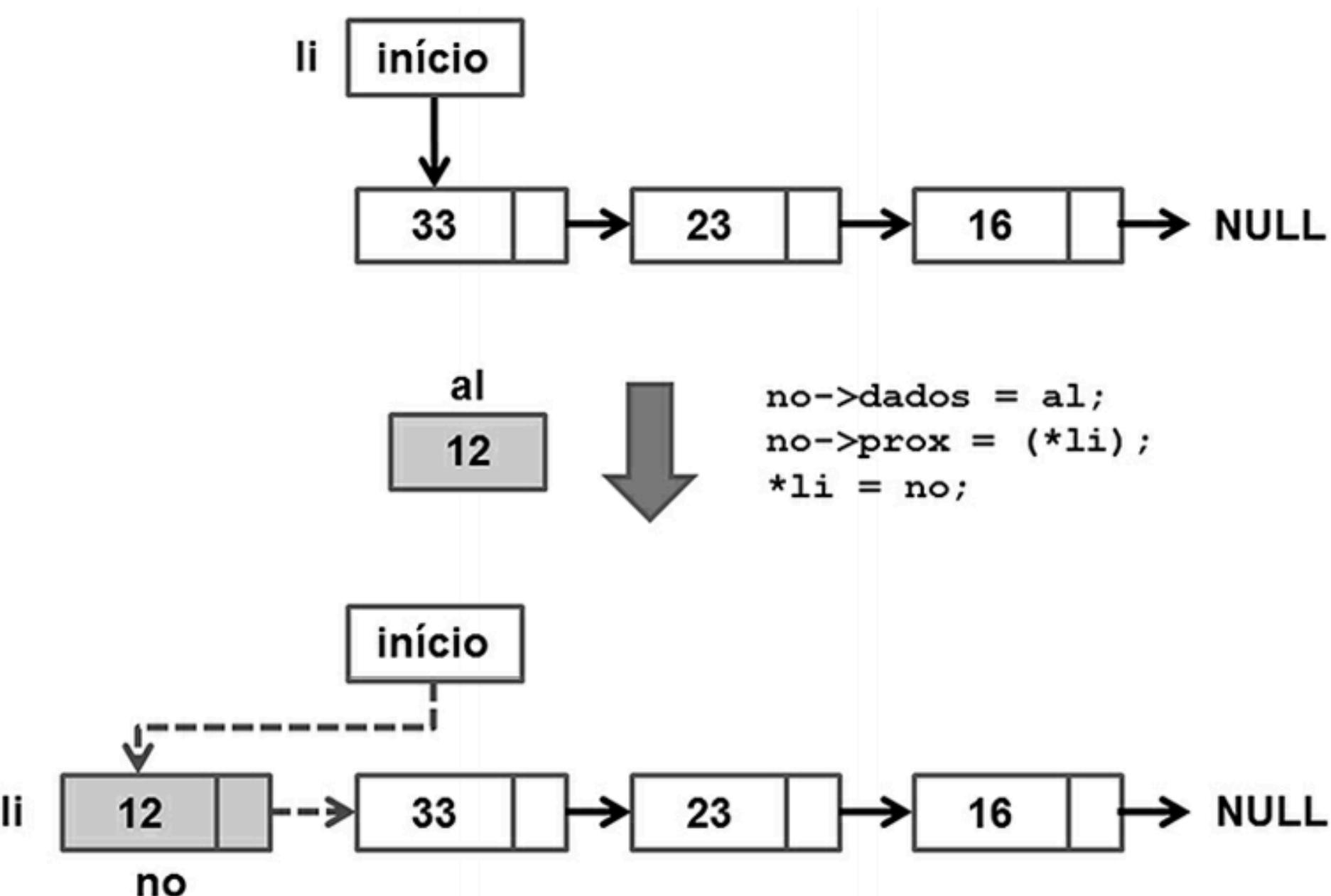


 Diferente da **lista sequencial estática**, a inserção no início de uma **lista dinâmica encadeada** não necessita que se mude o lugar dos demais elementos da lista.

### Inserindo um elemento no início da lista

```

01 int insere_lista_inicio(Lista* li, struct aluno al) {
02     if(li == NULL)
03         return 0;
04     Elem* no;
05     no = (ELEM*) malloc(sizeof(ELEM));
06     if(no == NULL)
07         return 0;
08     no->dados = al;
09     no->prox = (*li);
10     *li = no;
11     return 1;
12 }
  
```

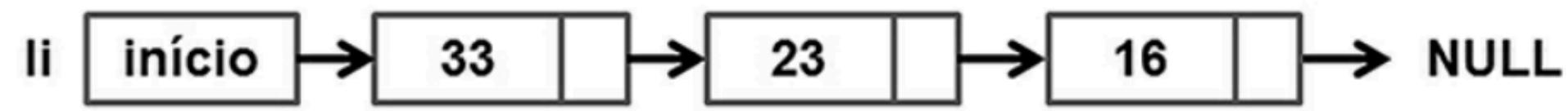




Como na inserção no início, a inserção no final de uma **lista dinâmica encadeada** não necessita que se mude o lugar dos demais elementos da lista. Porém, é preciso percorrer a lista toda para descobrir o último elemento e assim fazer a inserção após ele.

### Inserindo um elemento no final da lista

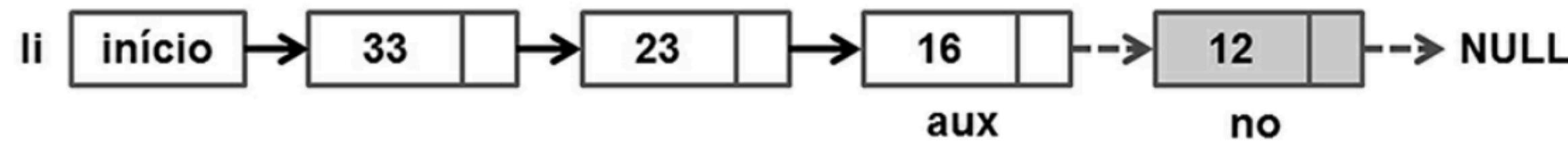
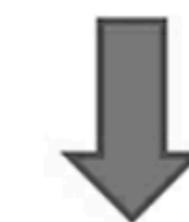
```
01 int insere_lista_final(Lista* li, struct aluno al){  
02     if(li == NULL)  
03         return 0;  
04     Elemt *no;  
05     no = (Elemt*) malloc(sizeof(Elemt));  
06     if(no == NULL)  
07         return 0;  
08     no->dados = al;  
09     no->prox = NULL;  
10     if((*li) == NULL) //lista vazia: insere inicio  
11         *li = no;  
12     else{  
13         Elemt *aux;  
14         aux = *li;  
15         while(aux->prox != NULL){  
16             aux = aux->prox;  
17         }  
18         aux->prox = no;  
19     }  
20     return 1;  
21 }
```



#### Busca onde Inserir:

```
aux = *li;  
while(aux->prox != NULL){  
    aux = aux->prox;  
}
```

al [12]



**Insere depois de “aux”:**  
no->dados = al;  
no->prox = NULL;  
aux->prox = no;

Isso ocorre porque precisamos procurar o ponto de inserção do elemento na lista, o qual pode ser no início, no meio ou no final da lista. Porém, diferente da **lista sequencial estática**, a inserção ordenada em uma **lista dinâmica encadeada** não necessita que se mude o lugar dos demais elementos da lista.

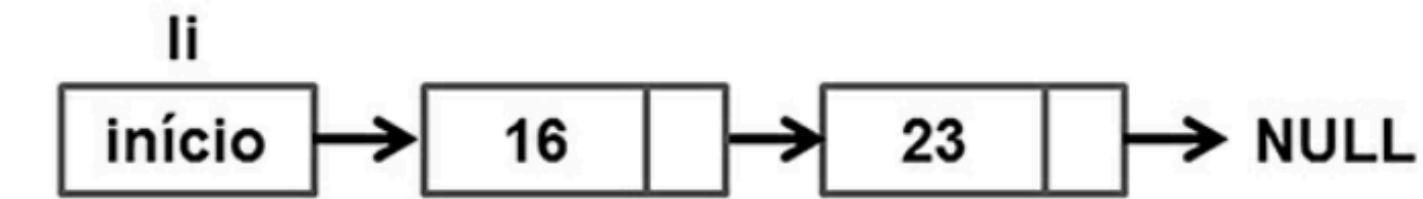
## Inserindo um elemento de forma ordenada na lista

```
01 int insere_lista_ordenada(Lista* li, struct aluno al){  
02     if(li == NULL)  
03         return 0;  
04     Elemt *no;  
05     no = (Elemt*) malloc(sizeof(Elemt));  
06     if(no == NULL)  
07         return 0;  
08     no->dados = al;  
09     if((*li) == NULL){//lista vazia: insere inicio  
10         no->prox = NULL;  
11         *li = no;  
12         return 1;  
13     }  
14     else{  
15         Elemt *ant, *atual = *li;  
16         while(atual != NULL &&  
17                 atual->dados.matricula < al.matricula){  
18             ant = atual;  
19             atual = atual->prox;  
20         }  
21         if(atual == *li){//insere inicio  
22             no->prox = (*li);  
23             *li = no;  
24         }else{  
25             no->prox = atual;  
26             ant->prox = no;  
27         }  
28         return 1;  
29     }
```

**Lista inicial**

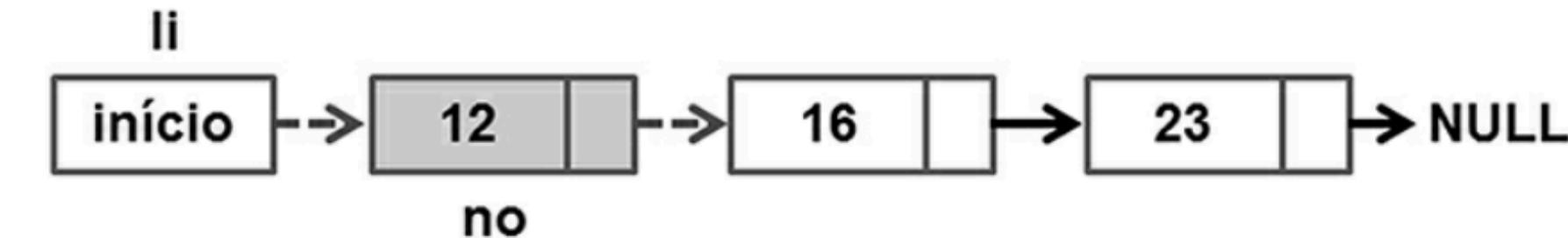
**Busca onde Inserir:**

```
atual = *li;  
while(atual != NULL && atual->dados.matricula < al.matricula){  
    ant = atual;  
    atual = atual->prox;  
}
```



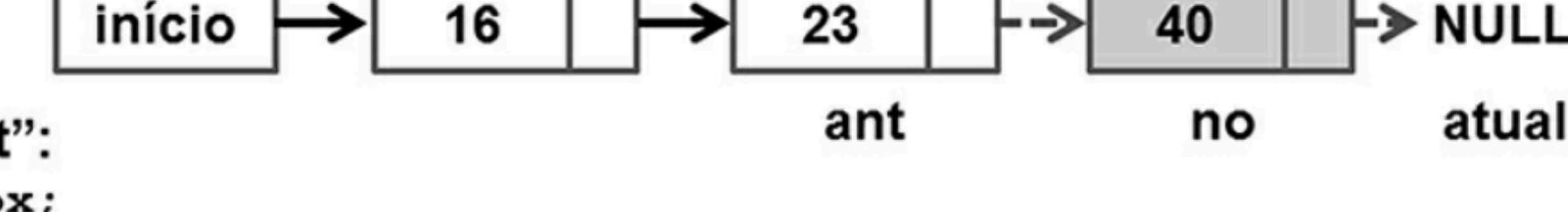
**Inserir no inicio:**

```
no->prox = (*li);  
*li = no;
```



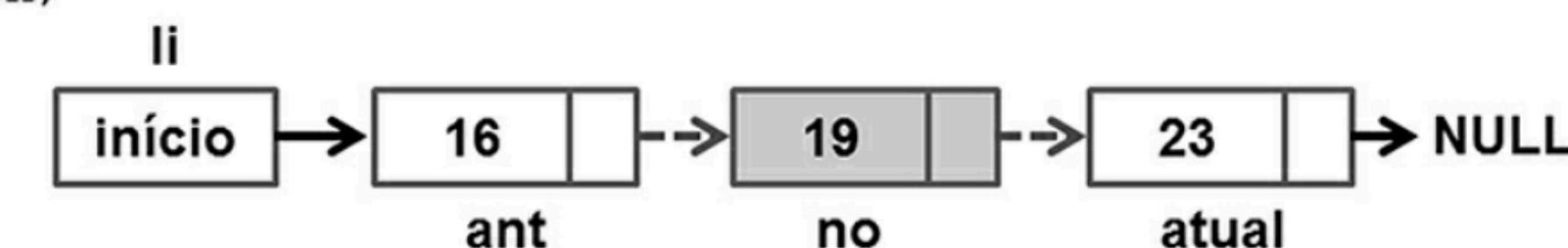
**Inserir depois de “ant”:**

```
no->prox = ant->prox;  
ant->prox = no;
```



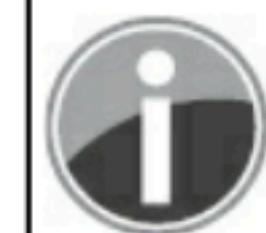
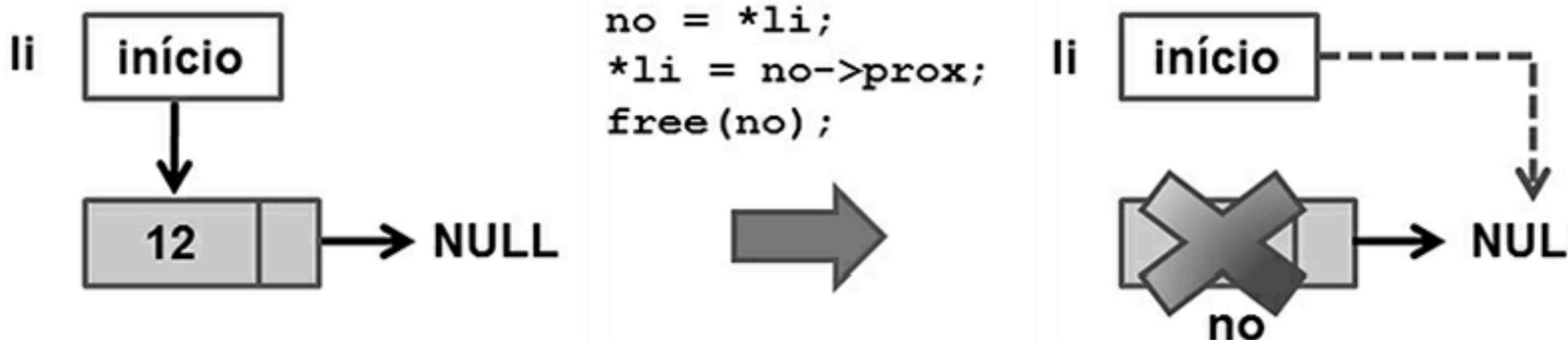
**Inserir depois de “ant”:**

```
no->prox = ant->prox;  
ant->prox = no;
```





No caso de uma lista com alocação dinâmica, ela somente será considerada vazia quando o seu início apontar para a constante **NULL**.



Diferente da **lista sequencial estática**, a remoção do início de uma **lista dinâmica encadeada** não necessita que se mude o lugar dos demais elementos da lista.

## Removendo um elemento do início da lista

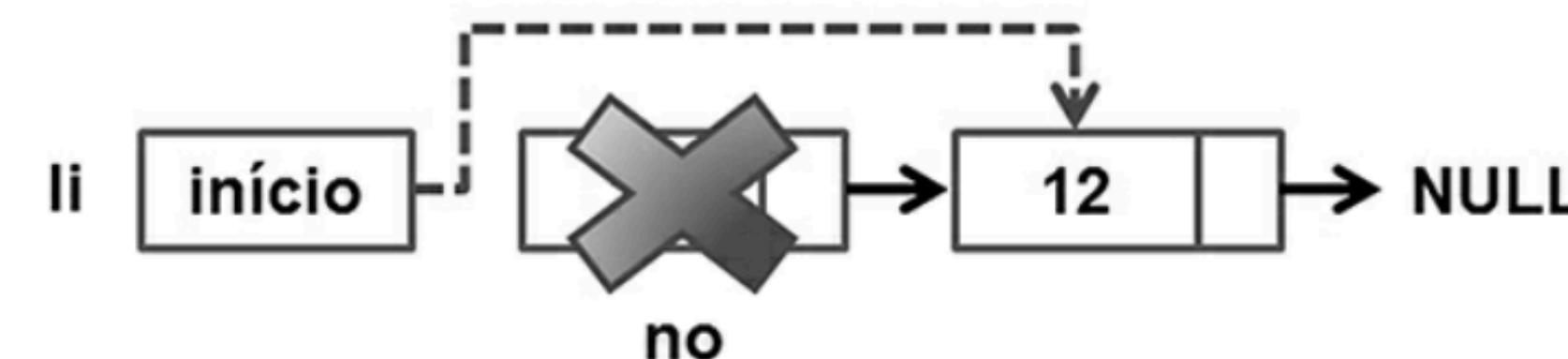
```
01 int remove_lista_inicio(Lista* li) {  
02     if(li == NULL)  
03         return 0;  
04     if((*li) == NULL)//lista vazia  
05         return 0;  
06  
07     Elem *no = *li;  
08     *li = no->prox;  
09     free(no);  
10     return 1;  
11 }
```

**Lista inicial**

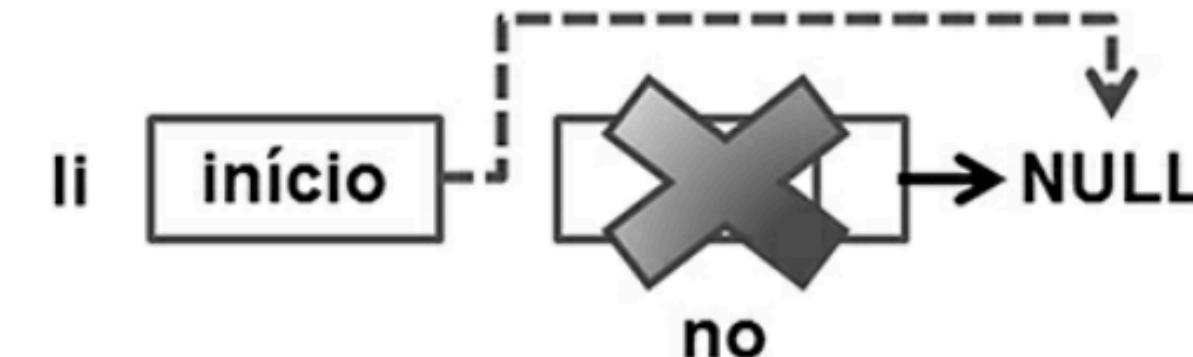


↓  
`*li = no->prox;  
free(no);`

**Se a lista possuir mais de um elemento, o início aponta para o segundo**



**Se a lista possuir um único elemento, ela fica vazia**



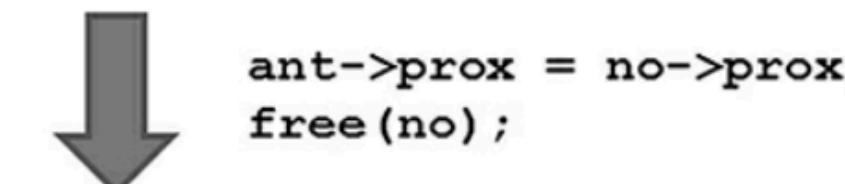
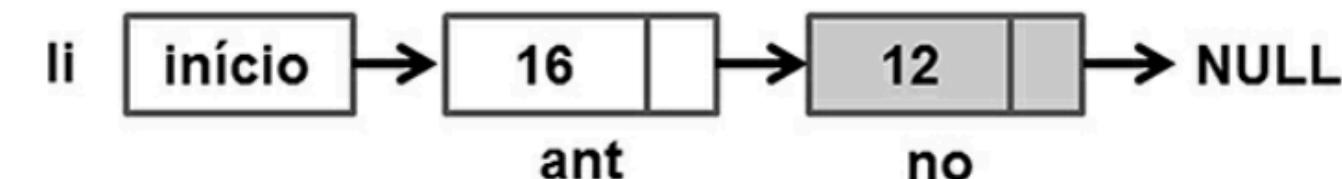
## Removendo um elemento do final da lista

```
01 int remove_lista_final(Lista* li){  
02     if(li == NULL)  
03         return 0;  
04     if((*li) == NULL)//lista vazia  
05         return 0;  
06  
07     Elem *ant, *no = *li;  
08     while(no->prox != NULL){  
09         ant = no;  
10         no = no->prox;  
11     }  
12  
13     if(no == (*li))//remover o primeiro?  
14         *li = no->prox;  
15     else  
16         ant->prox = no->prox;  
17     free(no);  
18     return 1;  
19 }
```

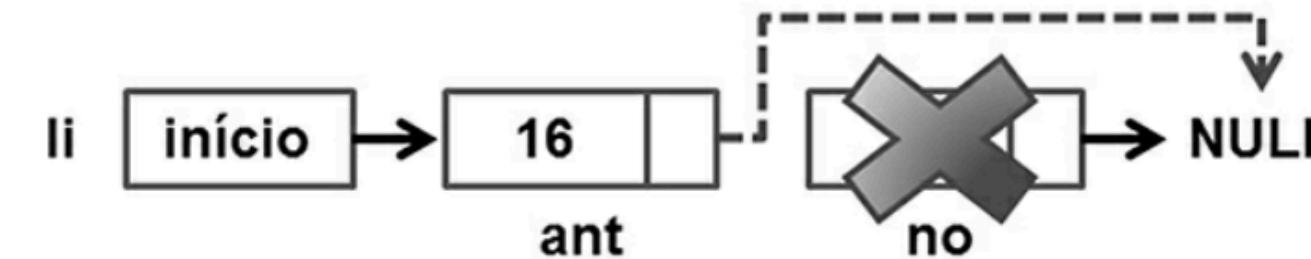
### Lista inicial

#### Busca o último elemento:

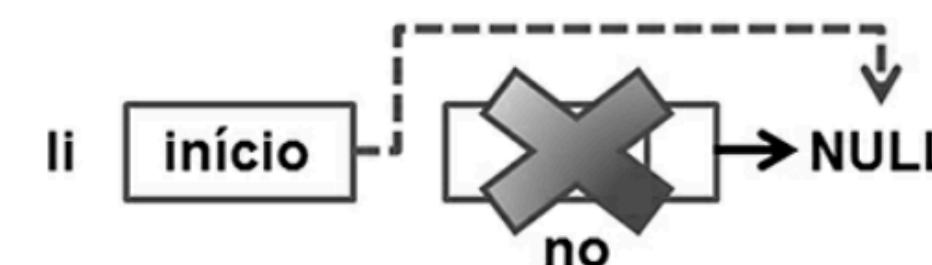
```
no = *li;  
while(no->prox != NULL){  
    ant = no;  
    no = no->prox;  
}
```



Se a lista possui mais de um elemento, ant aponta para NULL

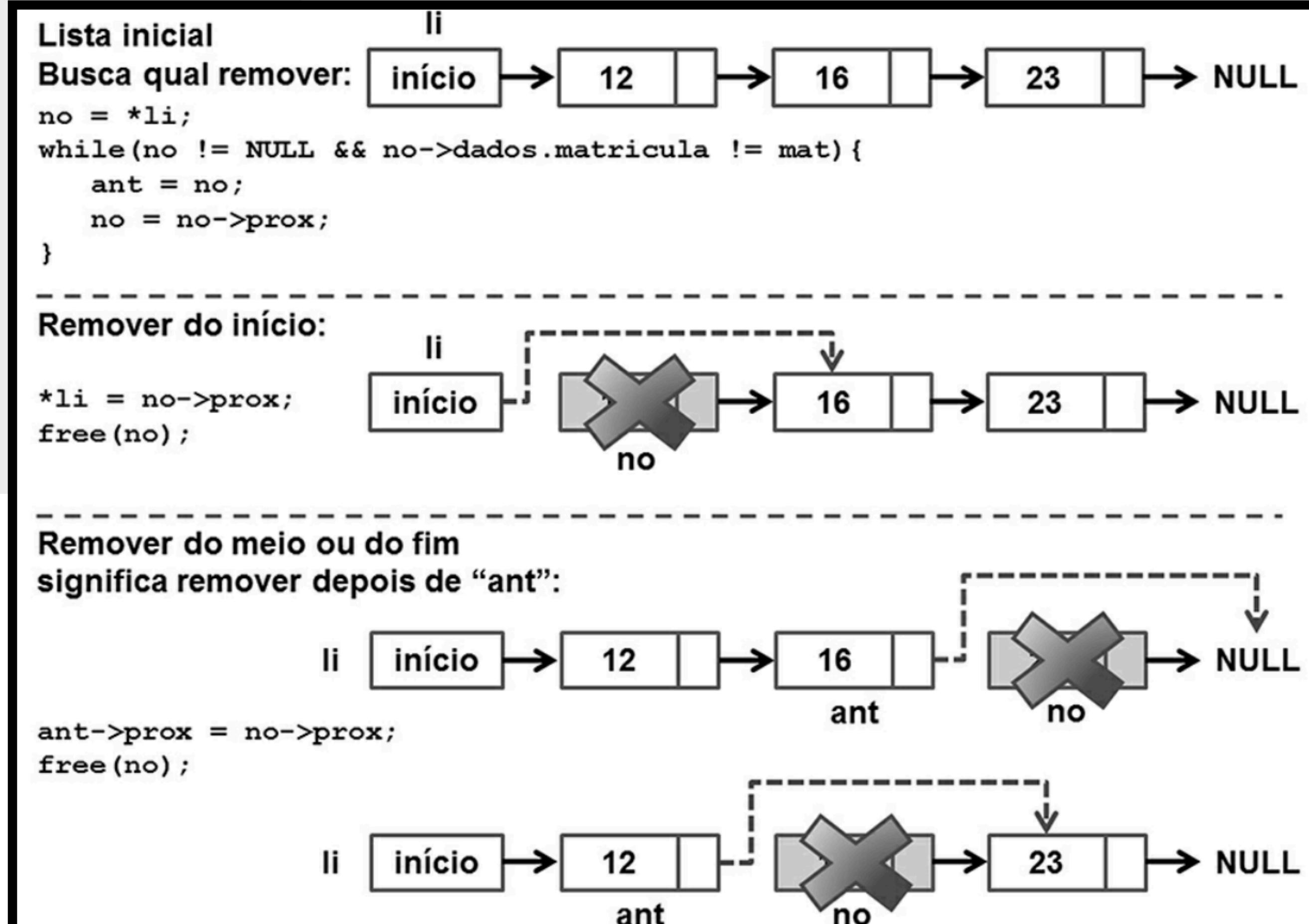


Se "no" é o único elemento da lista, a lista fica vazia.



## Removendo um elemento específico da lista

```
01 int remove_lista(Lista* li, int mat) {  
02     if(li == NULL)  
03         return 0;  
04     if((*li) == NULL)//lista vazia  
05         return 0;  
06     Elemt *ant, *no = *li;  
07     while(no != NULL && no->dados.matricula != mat) {  
08         ant = no;  
09         no = no->prox;  
10    }  
11    if(no == NULL)//não encontrado  
12        return 0;  
13  
14    if(no == *li)//remover o primeiro?  
15        *li = no->prox;  
16    else  
17        ant->prox = no->prox;  
18    free(no);  
19    return 1;  
20 }
```





Em uma lista que utilize alocação dinâmica, e seja encadeada, a busca sempre envolve a necessidade de percorrer a lista.

### Busca um elemento por posição

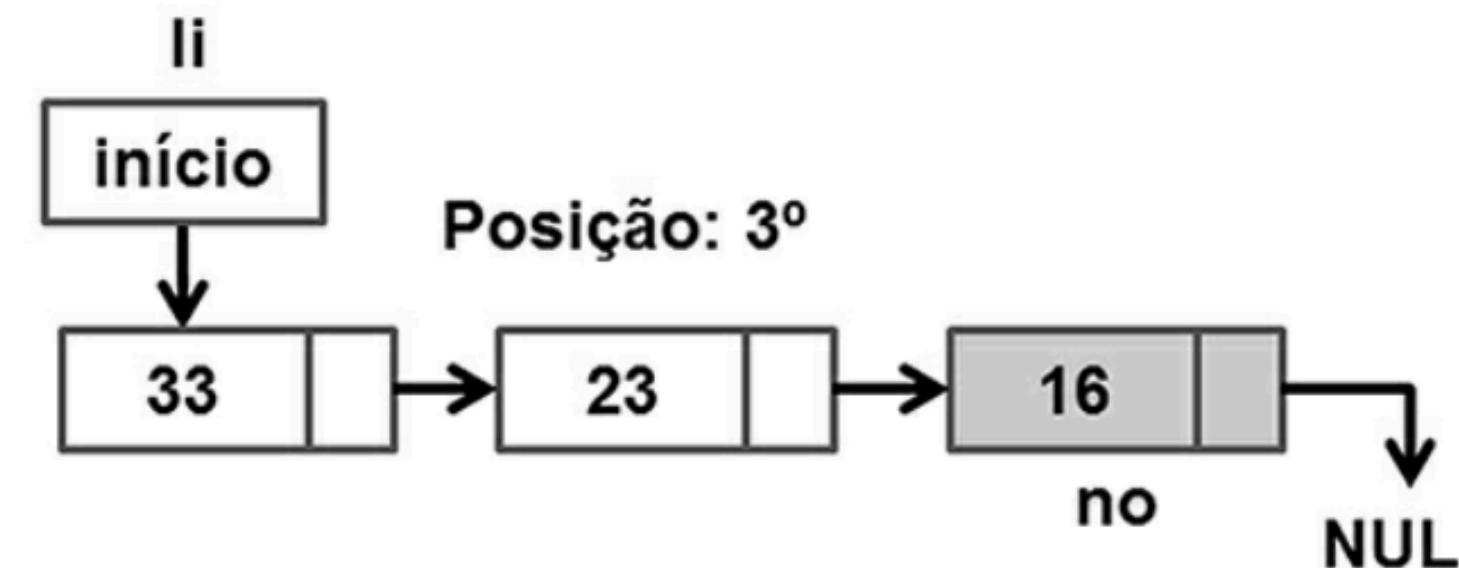
```
01 int busca_lista_pos(Lista* li, int pos, struct aluno *al) {
02     if(li == NULL || pos <= 0)
03         return 0;
04     Elemt *no = *li;
05     int i = 1;
06     while(no != NULL && i < pos) {
07         no = no->prox;
08         i++;
09     }
10     if(no == NULL)
11         return 0;
12     else{
13         *al = no->dados;
14         return 1;
15     }
16 }
```

### Busca pela posição do elemento

```
no = *li;
int i = 1;
while(no != NULL && i < pos) {
    no = no->prox;
    i++;
}
```

**Verifica se a posição foi encontrada e a retorna**

```
if(no == NULL) return 0;
else{
    *al = no->dados;
    return 1;
}
```



## Busca um elemento por conteúdo

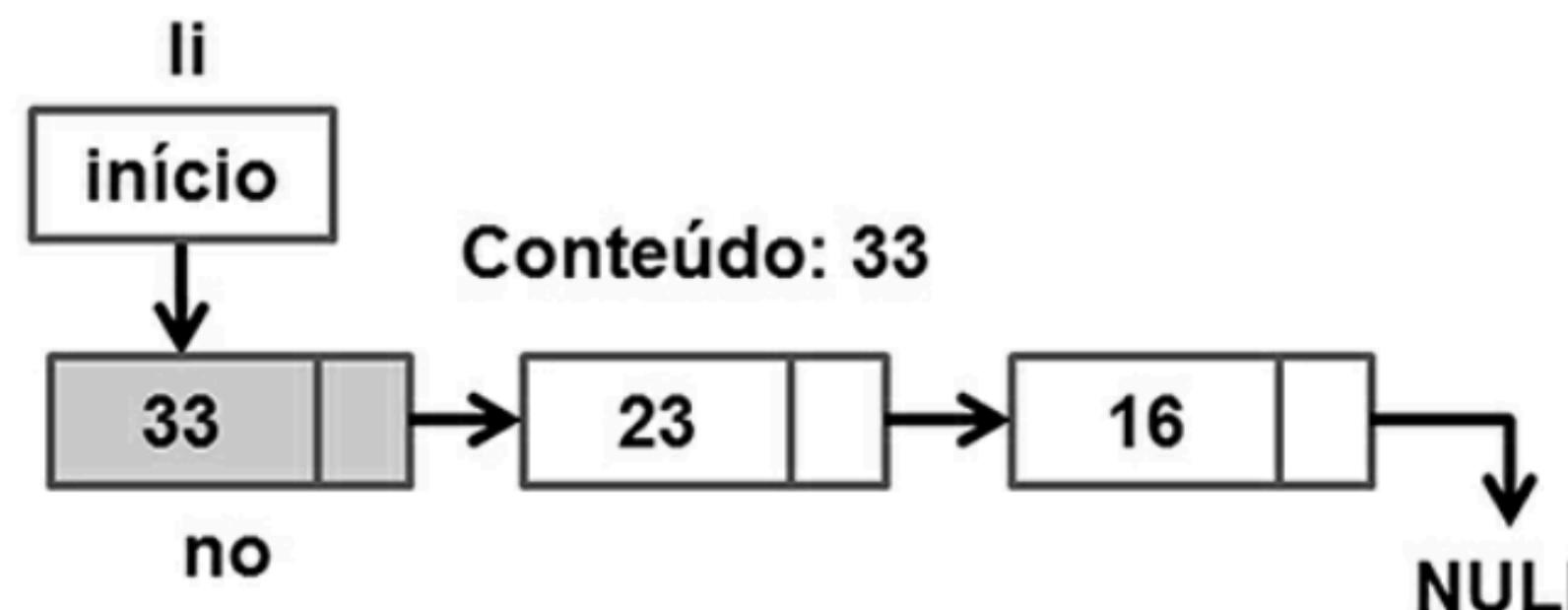
```
01 int busca_lista_mat(Lista* li, int mat, struct aluno *al){  
02     if(li == NULL)  
03         return 0;  
04     Elem *no = *li;  
05     while(no != NULL && no->dados.matricula != mat){  
06         no = no->prox;  
07     }  
08     if(no == NULL)  
09         return 0;  
10     else{  
11         *al = no->dados;  
12         return 1;  
13     }  
14 }
```

### Busca pelo conteúdo do elemento

```
no = *li;  
while(no != NULL && no->dados.matricula != mat)  
    no = no->prox;
```

Verifica se o elemento foi  
encontrado e o retorna

```
if(no == NULL) return 0;  
else{  
    *al = no->dados;  
    return 1;  
}
```



# TAREFAS

- 1) Escreva uma função que receba duas listas e retorne uma terceira contendo as duas primeiras concatenadas. Faça a função para todos os tipos de listas: estática e dinâmica (encadeada, circular e duplamente encadeada).
- 2) Faça uma função para remover os n primeiros elementos de uma lista. A função deve retornar se a operação foi possível ou não. Faça a função para todos os tipos de listas: estática e dinâmica (encadeada, circular e duplamente encadeada).
- 3) Faça uma função para remover os n últimos elementos de uma lista. A função deve retornar se a operação foi possível ou não. Faça a função para todos os tipos de listas: estática e dinâmica (encadeada, circular e duplamente encadeada).

# TAREFAS

4) Dada uma lista que armazena a struct produto, escreva a função que busca o produto de menor preço. Faça a função para a lista sequencial estática e dinâmica encadeada.

```
struct produto{  
    int codido;  
    char nome[30];  
    float preco;  
    int qtd;  
};
```

5) Escreva uma função que receba a posição de dois elementos da lista e os troque de lugar. A função deve retornar se a operação foi possível ou não. Faça a função para todos os tipos de listas: estática e dinâmica (encadeada).



# OBRIGADO!

RAFAELVC2@GMAIL.COM