



LISTAS DUPLAMENTE ENCADEADA

SUMÁRIO

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

ANDRÉ BACKES

Estrutura de dados descomplicada em linguagem

C



Diferente da **lista dinâmica encadeada**, esse tipo de lista não possui dois, mas sim **três** campos de informação dentro de cada elemento: os campos **dado**, **prox** e **ant**.

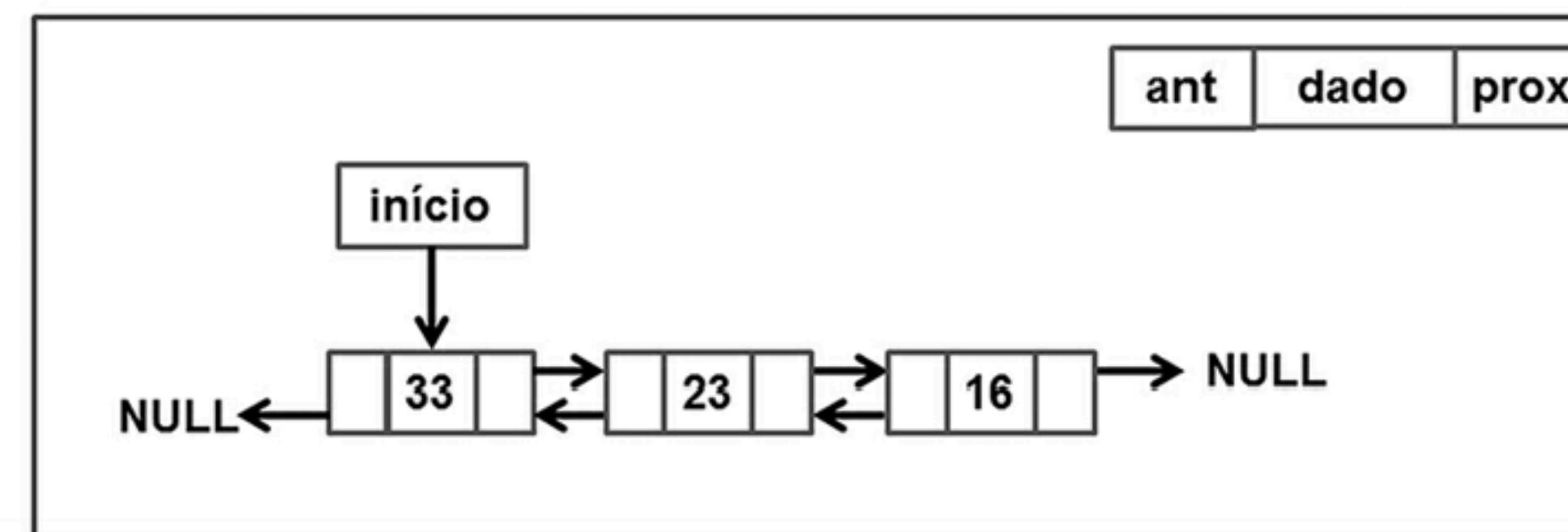


É a existência dos campos **prox** e **ant** que garante que a lista é **duplamente encadeada**.



Após o último elemento, não existe nenhum novo elemento alocado. Sendo assim, o último elemento da lista aponta o campo **prox** para **NULL**. O mesmo vale para o primeiro elemento da lista: não existe ninguém antes dele. Sendo assim, ele aponta o seu campo **ant** para **NULL**.

Lista *li



QUANDO USAR LISTA DUPLAMENTE ENCADEADA

Em geral, usamos esse tipo de lista na seguinte situações:

- Não há necessidade de garantir um espaço mínimo para a execução da aplicação.
- Inserção e remoção em lista ordenada são as operações mais frequentes.
- Tamanho máximo da lista não é definido.
- Necessidade de acessar a informação de um elemento antecessor.

Arquivo ListaDinEncadDupla.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 busca_lista_pos(Lista* li, int pos, struct aluno *al);  
11 int busca_lista_mat(Lista* li, int mat, struct aluno *al);  
12 int insere_lista_final(Lista* li, struct aluno al);  
13 int insere_lista_inicio(Lista* li, struct aluno al);  
14 int insere_lista_ordenada(Lista* li, struct aluno al);  
15 int remove_lista(Lista* li, int mat);  
16 int remove_lista_inicio(Lista* li);  
17 int remove_lista_final(Lista* li);  
18 int tamanho_lista(Lista* li);  
19 int lista_vazia(Lista* li);  
20 int lista_cheia(Lista* li);
```

Arquivo ListaDinEncadDupla.c

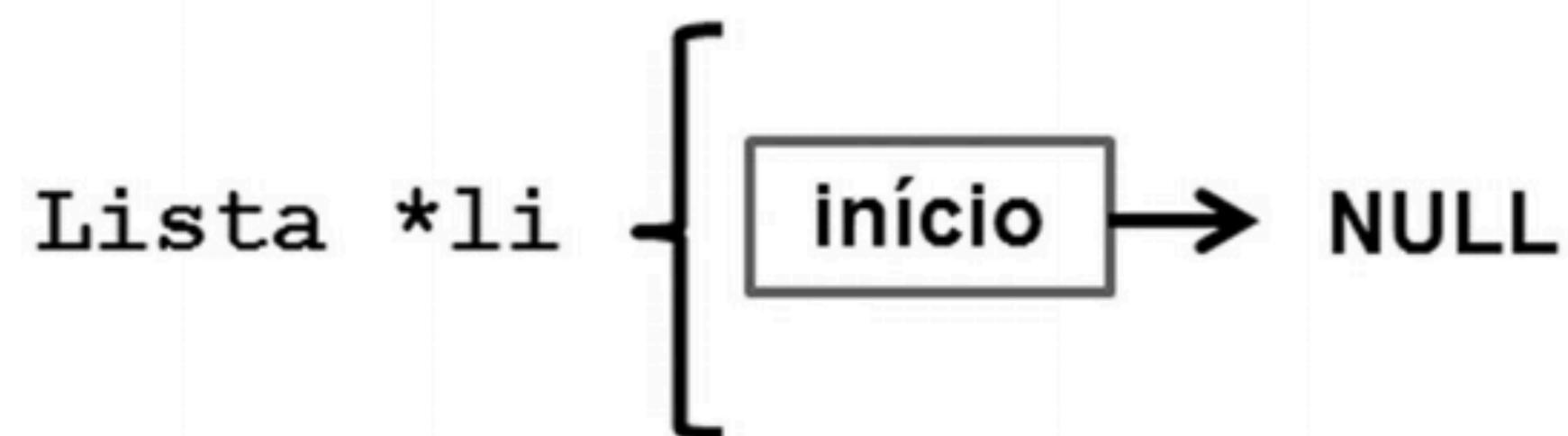
```
01 #include <stdio.h>  
02 #include <stdlib.h>  
03 #include "ListaDinEncadDupla.h" //inclui os protótipos  
04 //Definição do tipo lista  
05 struct elemento{  
06     struct elemento *ant;  
07     struct aluno dados;  
08     struct elemento *prox;  
09 };  
10 typedef struct elemento Elem;
```



Note que não existe diferença entre criar uma **lista dinâmica encadeada** e uma **lista dinâmica duplamente encadeada**.

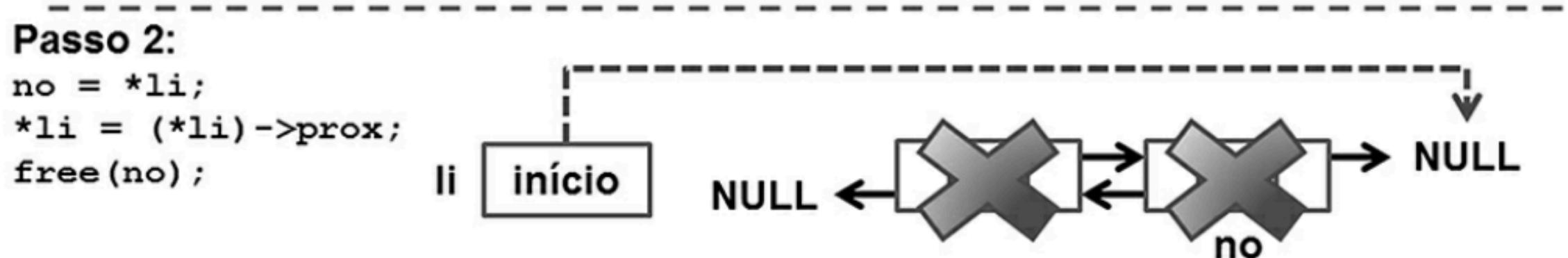
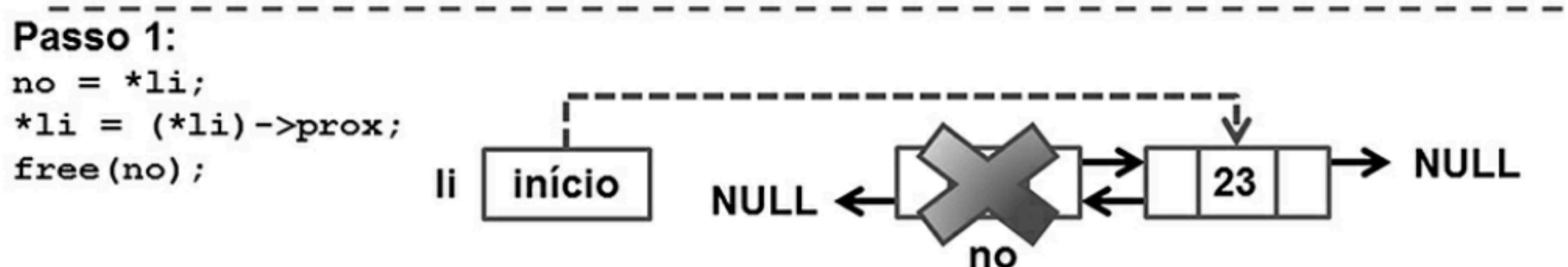
Criando uma lista

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



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 }
```

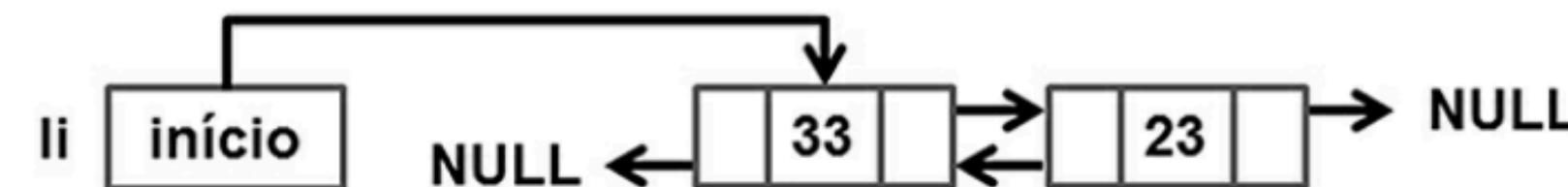


Tamanho da lista

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

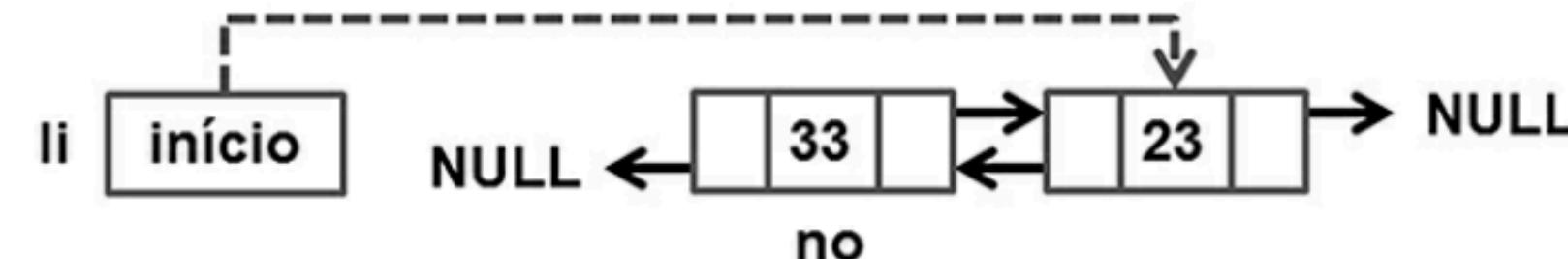
Lista inicial:

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



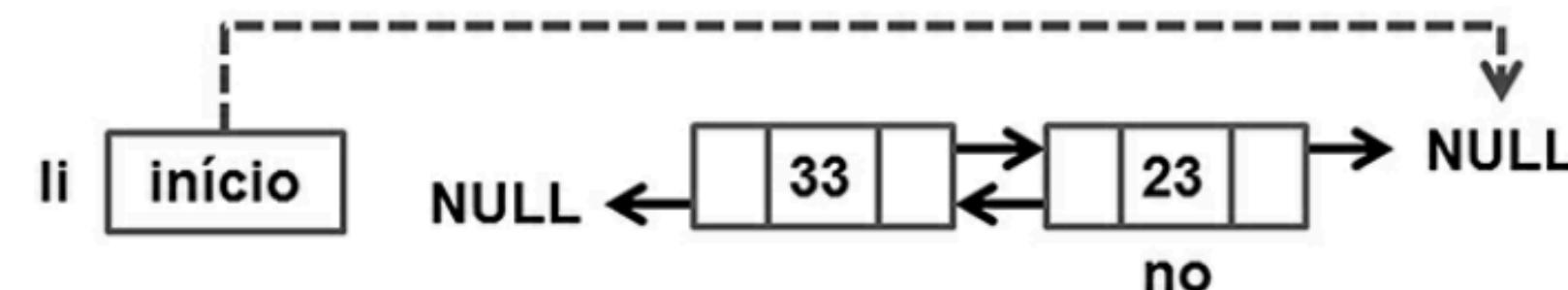
Passo 1:

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



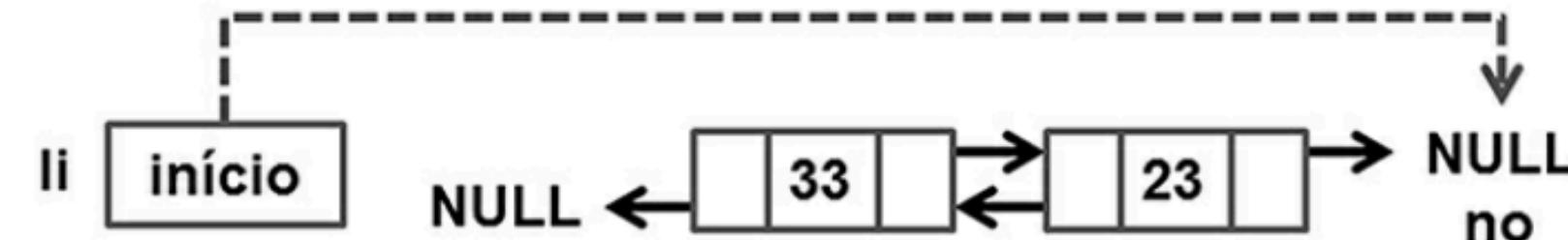
Passo 2:

```
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 }
```

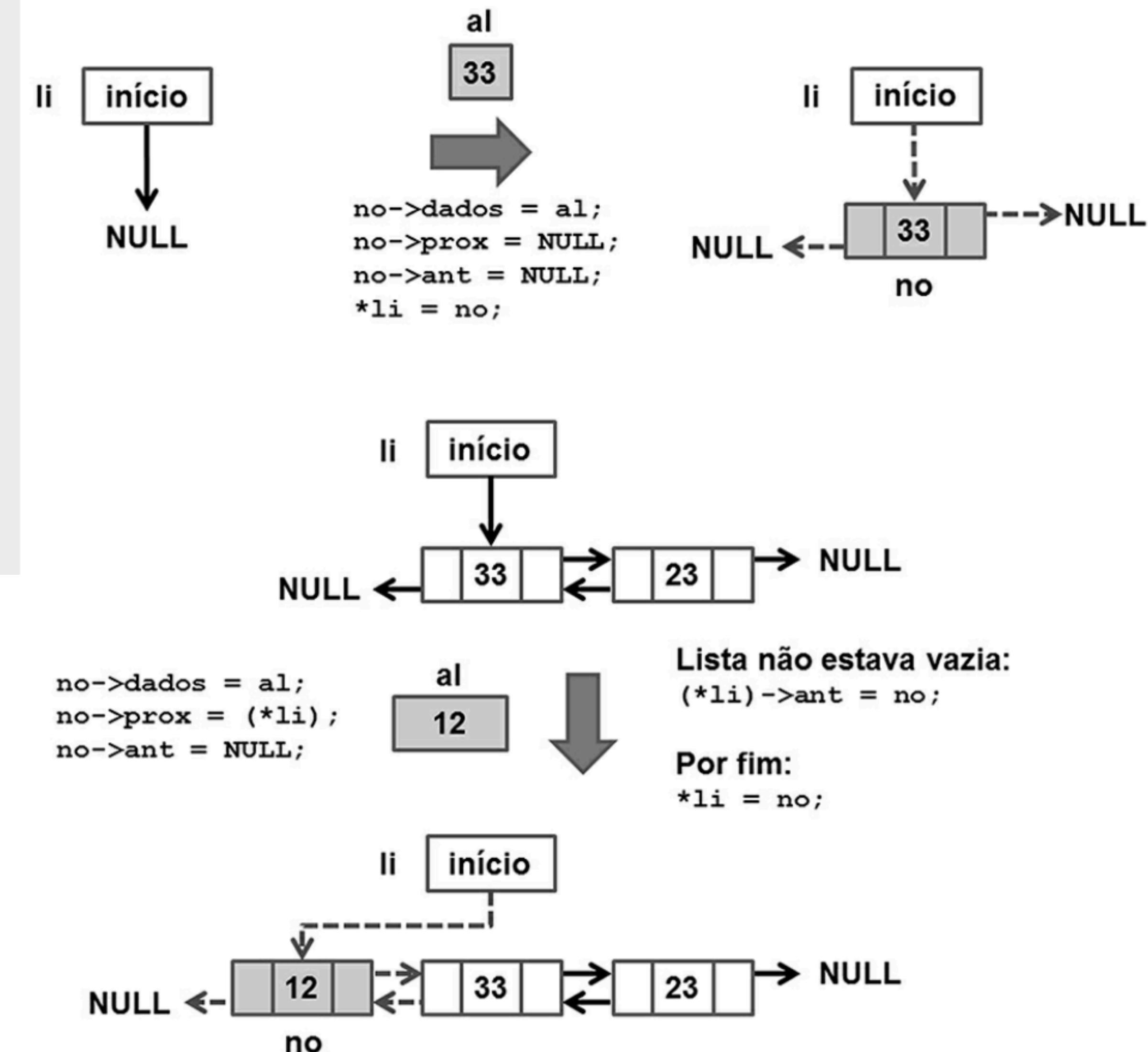


No caso de uma lista com alocação dinâmica, ela somente será considerada cheia quando não tivermos mais memória disponível no computador para alocar novos elementos. Isso ocorrerá apenas quando a chamada da função **malloc()** retornar **NULL**.

Inserindo um elemento no início da lista

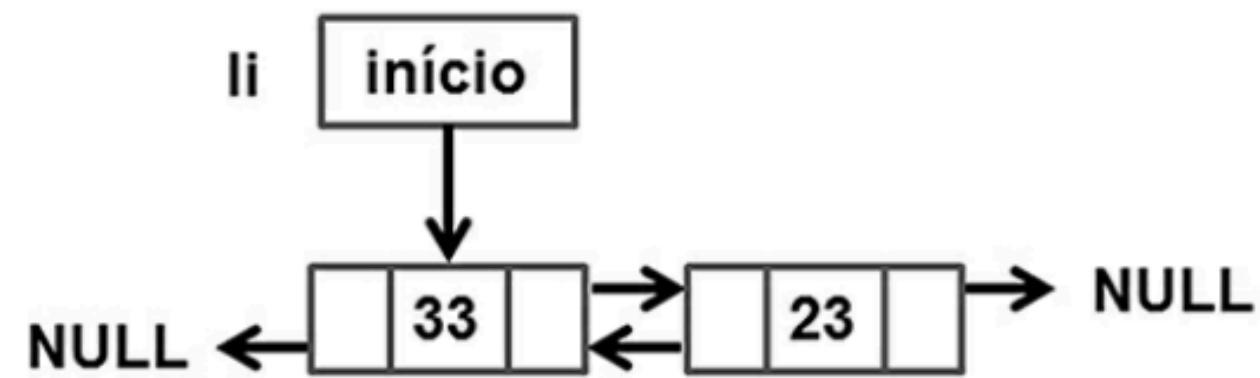
```

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     no->ant = NULL;
11     //lista não vazia: apontar para o anterior!
12     if(*li != NULL)
13         (*li)->ant = no;
14     *li = no;
15     return 1;
16 }
```



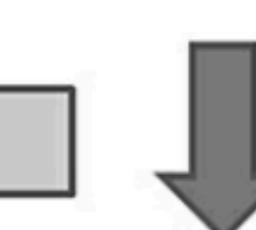
Inserindo um elemento no final da lista

```
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        no->ant = NULL;
12        *li = no;
13    }else{
14        Elemt *aux = *li;
15        while(aux->prox != NULL){
16            aux = aux->prox;
17        }
18        aux->prox = no;
19        no->ant = aux;
20    }
21    return 1;
22 }
```

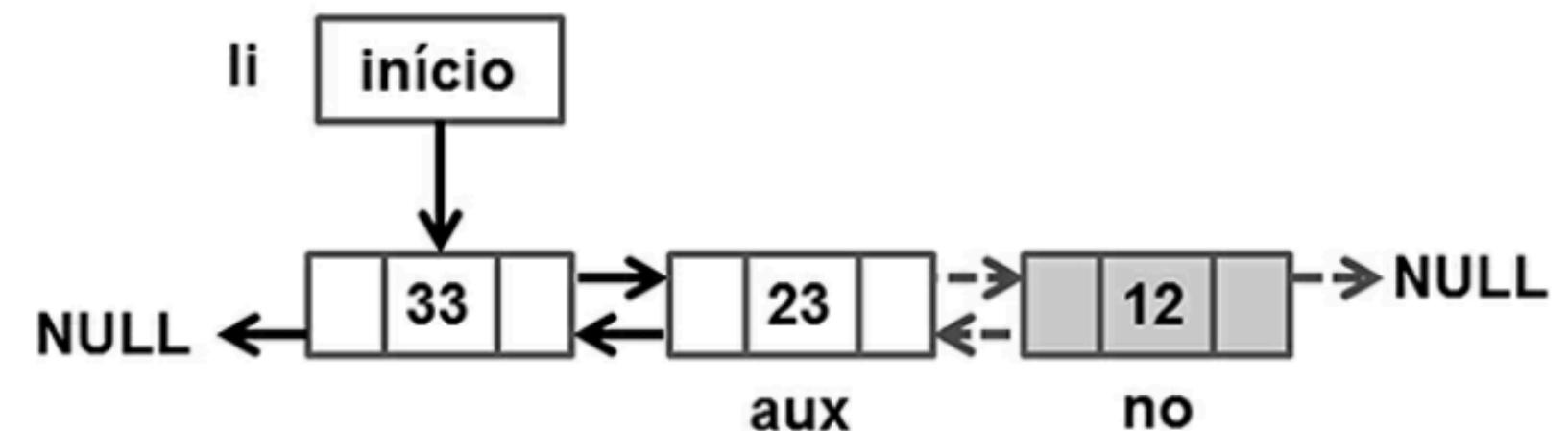


Busca onde Inserir:

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



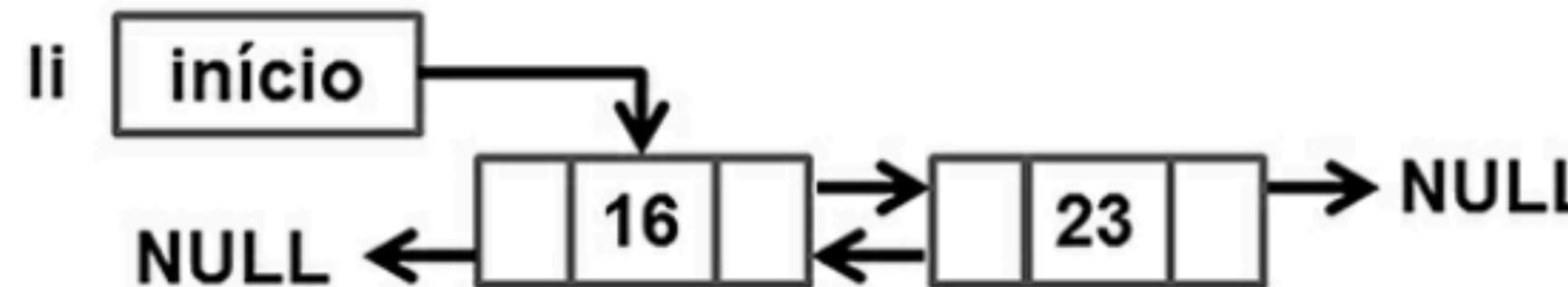
Insere depois de “aux”:
no->dados = al;
no->prox = NULL;
aux->prox = no;
no->ant = aux;



 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. Felizmente, como na **lista dinâmica encadeada**, não é preciso que se mude o lugar dos demais elementos da lista.

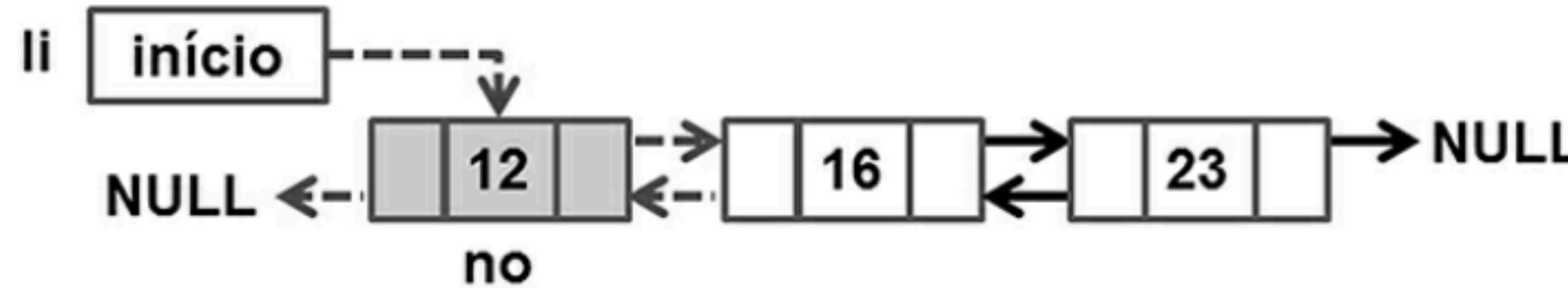
Lista inicial
Busca onde inserir:

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



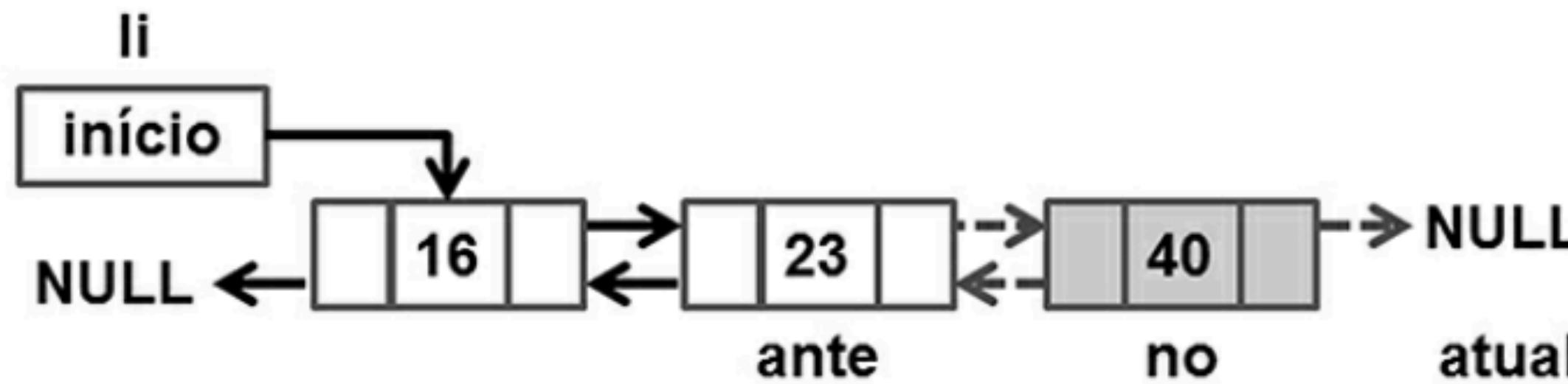
Inserir no início:

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



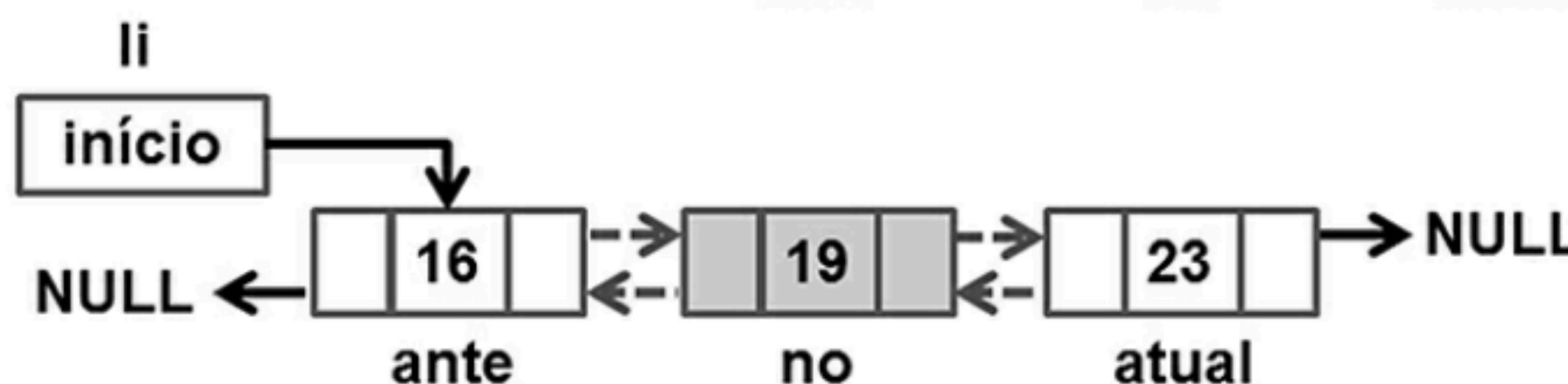
Inserir depois de “ante”:

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



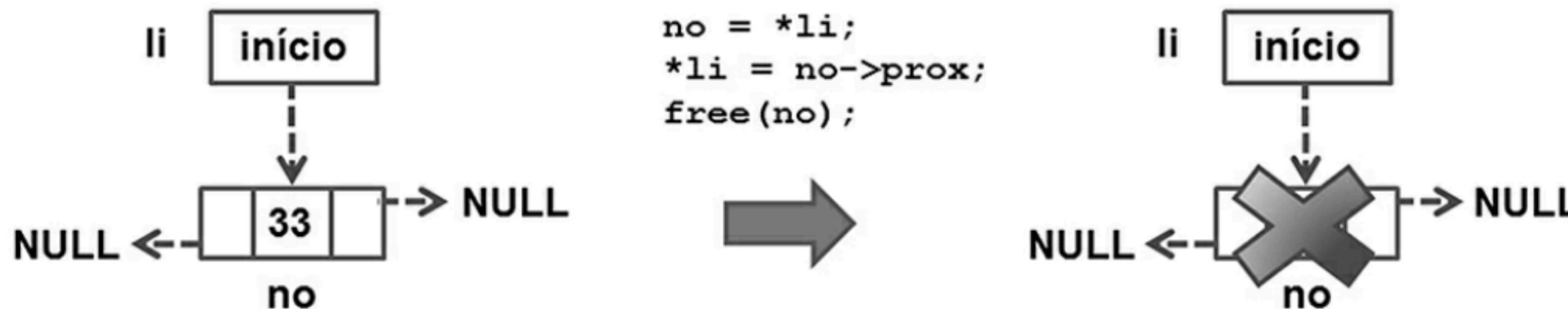
Não é final da lista:

```
atual->ant = no;
```





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

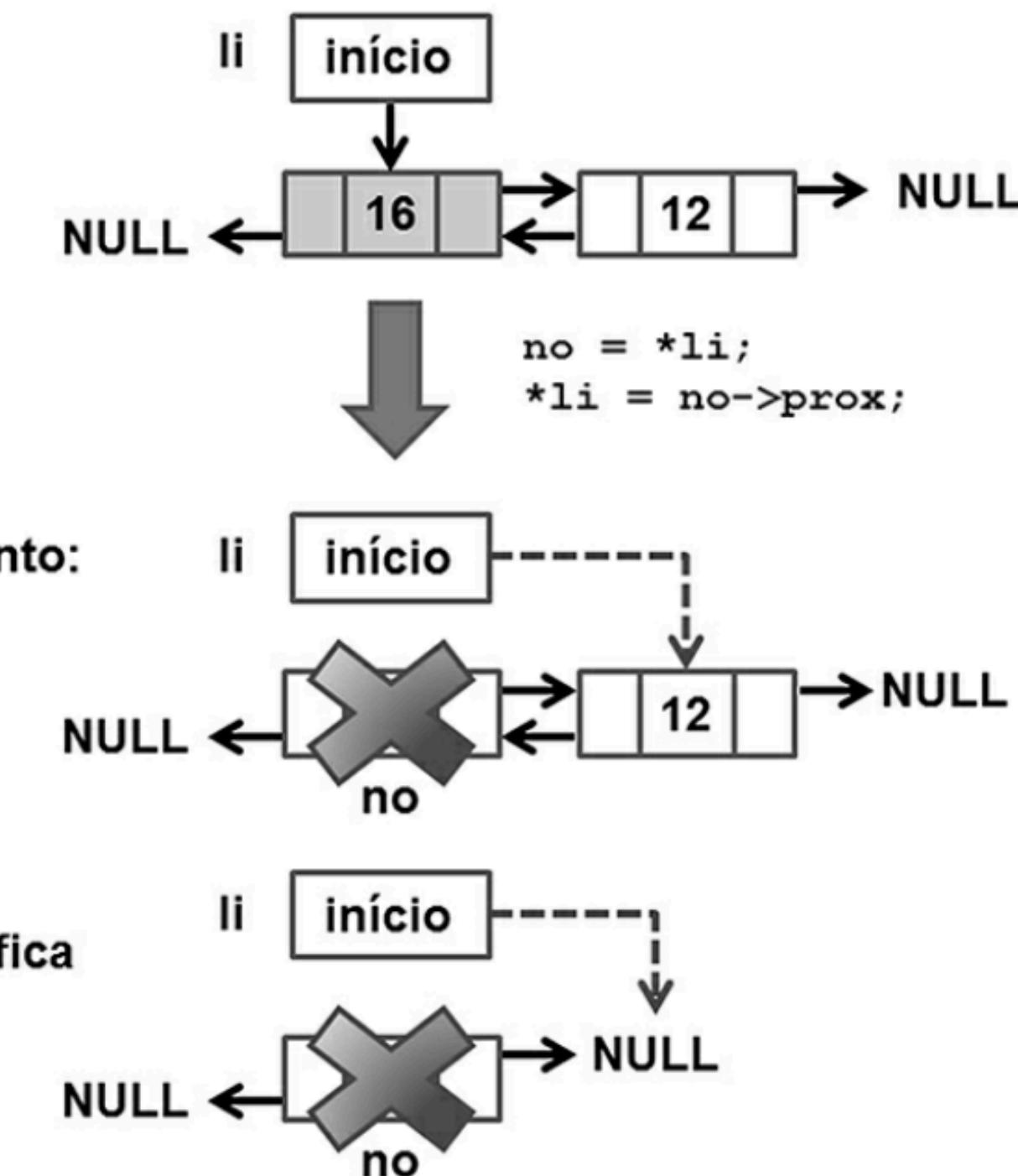


De fato, essa remoção é semelhante à remoção do início de uma **lista dinâmica encadeada**. Porém, temos agora que lidar com o ponteiro que aponta para o elemento anterior da lista.

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     Elemt *no = *li;
08     *li = no->prox;
09     if(no->prox != NULL)
10         no->prox->ant = NULL;
11
12     free(no);
13     return 1;
14 }
```

Lista inicial



Se a lista possui mais de um elemento:
no->prox->ant = NULL;

Por fim:
free(no);

Se "no" é o único elemento, a lista 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     Elemt *no = *li;  
08     while(no->prox != NULL)  
09         no = no->prox;  
10  
11    if(no->ant == NULL)//remover o primeiro e único  
12        *li = no->prox;  
13    else  
14        no->ant->prox = NULL;  
15  
16    free(no);  
17    return 1;  
18 }
```

Procura último elemento da lista:

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

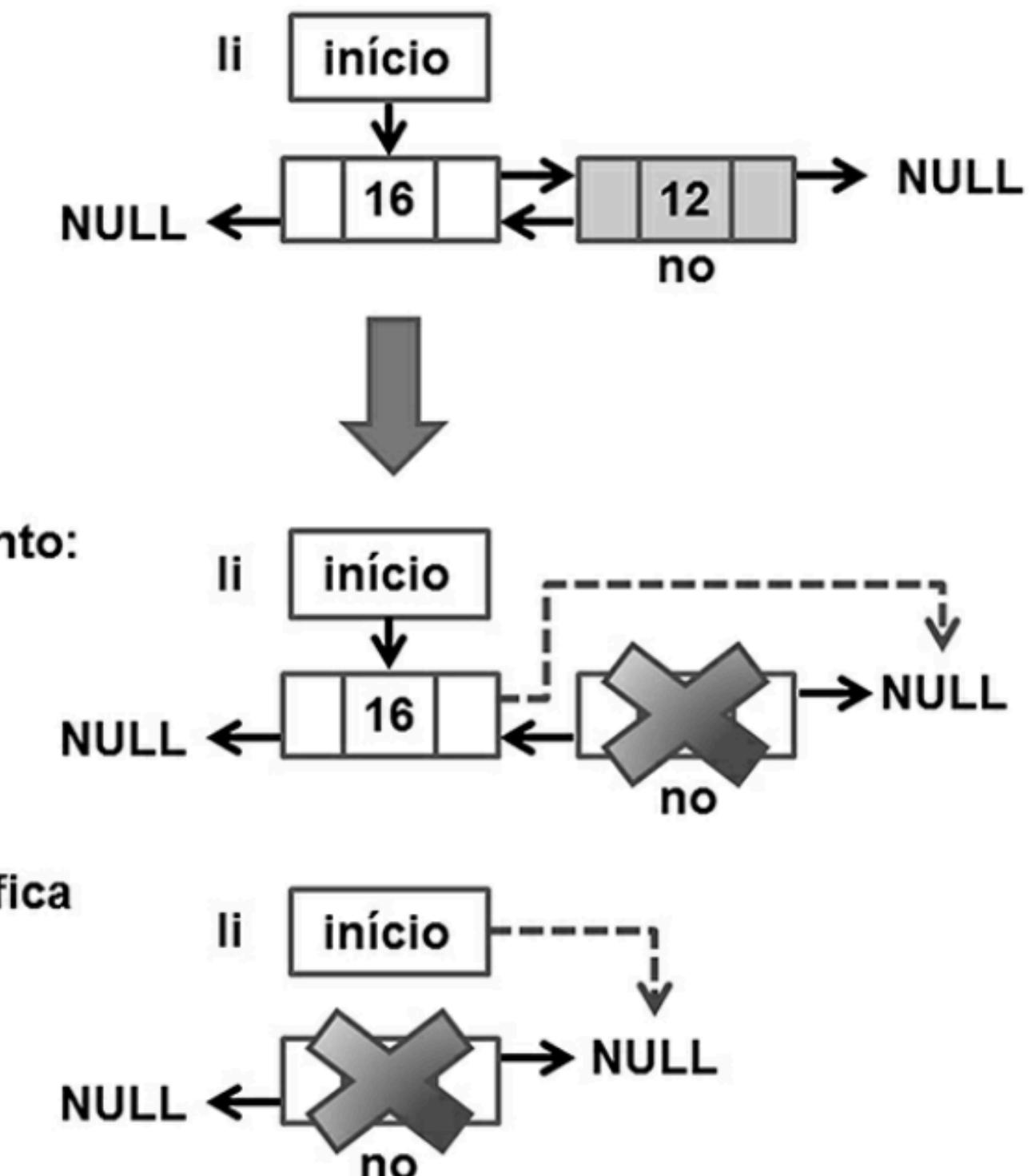
Se a lista possui mais de um elemento:

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

Por fim:

```
free(no);
```

Se “no” é o único elemento, a lista fica vazia.



Removendo um elemento específico da lista

```

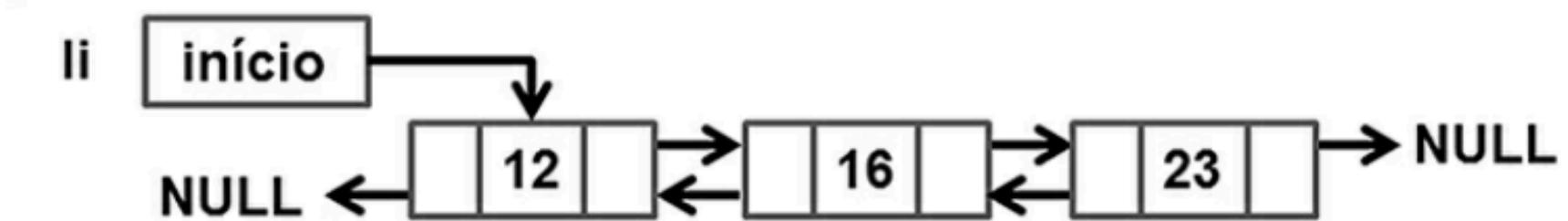
01 int remove_lista(Lista* li, int mat) {
02     if(li == NULL)
03         return 0;
04     if((*li) == NULL)//lista vazia
05         return 0;
06     Elem *no = *li;
07     while(no != NULL && no->dados.matricula != mat) {
08         no = no->prox;
09     }
10     if(no == NULL)//não encontrado
11         return 0;
12
13     if(no->ant == NULL)//remover o primeiro
14         *li = no->prox;
15     else
16         no->ant->prox = no->prox;
17
18     if(no->prox != NULL)//não é o último
19         no->prox->ant = no->ant;
20
21     free(no);
22
23 }
```

Lista inicial

Busca qual remover:

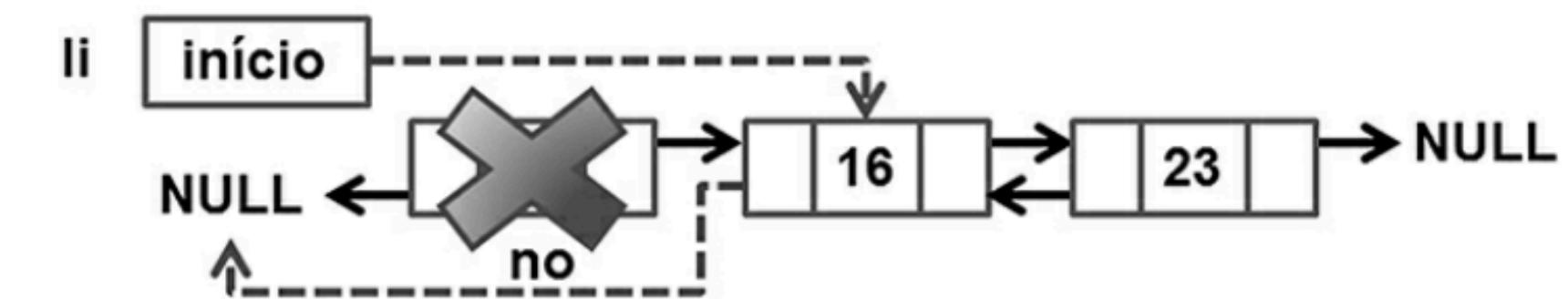
```

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



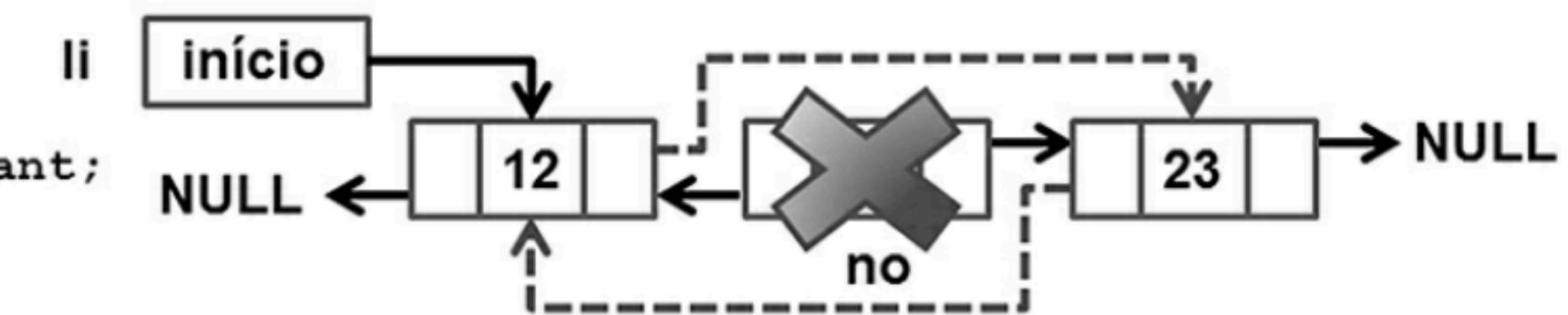
Remover do início:

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



Não está removendo do final:

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

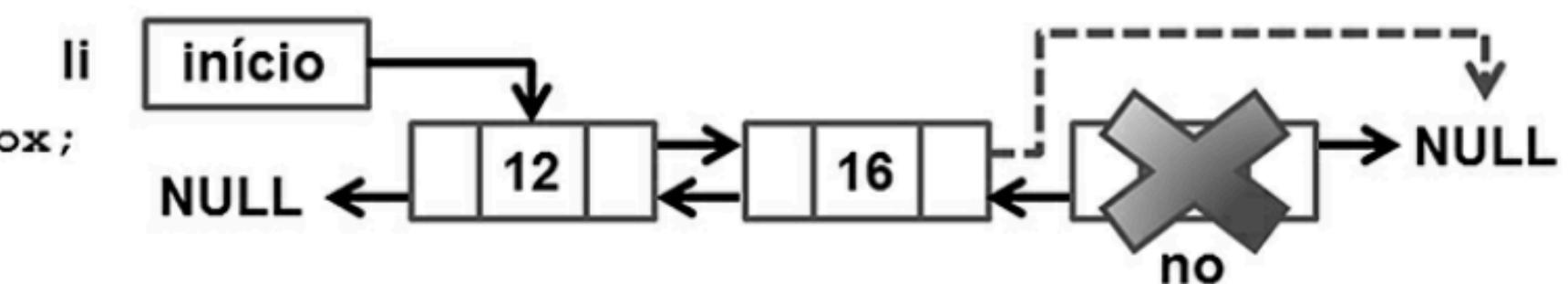


Remover do meio ou do final:

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

Por fim:

```
free(no);
```

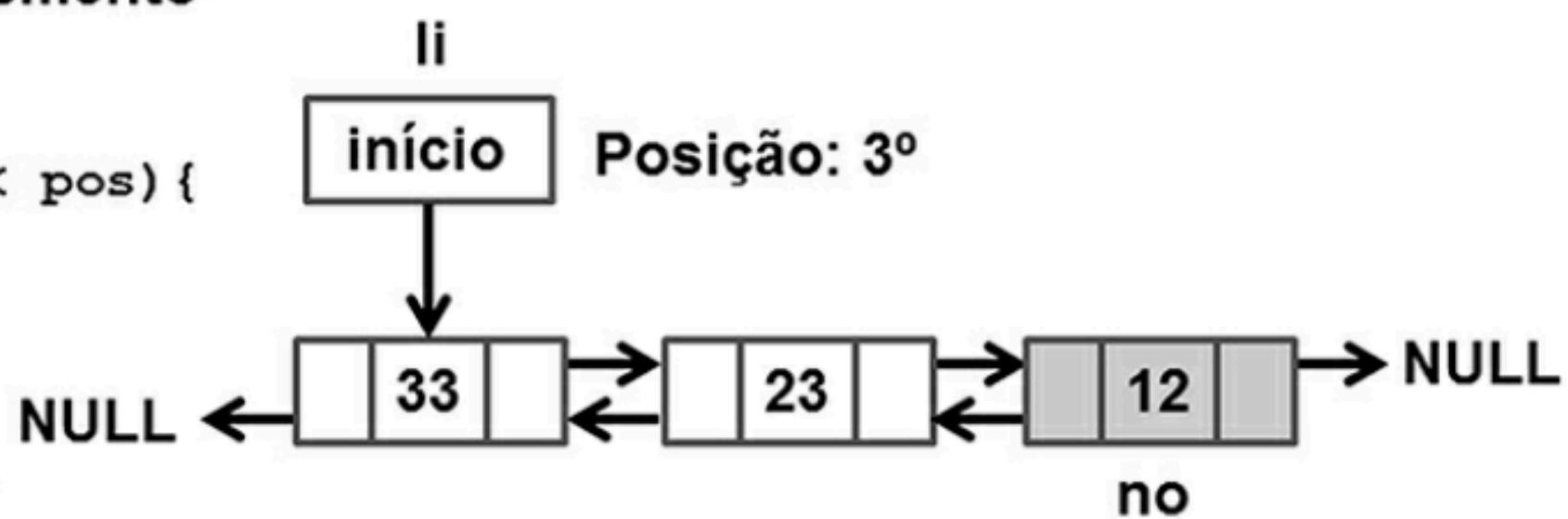


Busca um elemento por posição

```
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 o elemento foi
encontrado e o retorna
if(no == NULL) return 0;
else{
    *al = no->dados;
    return 1;
}
```



Busca um elemento por conteúdo

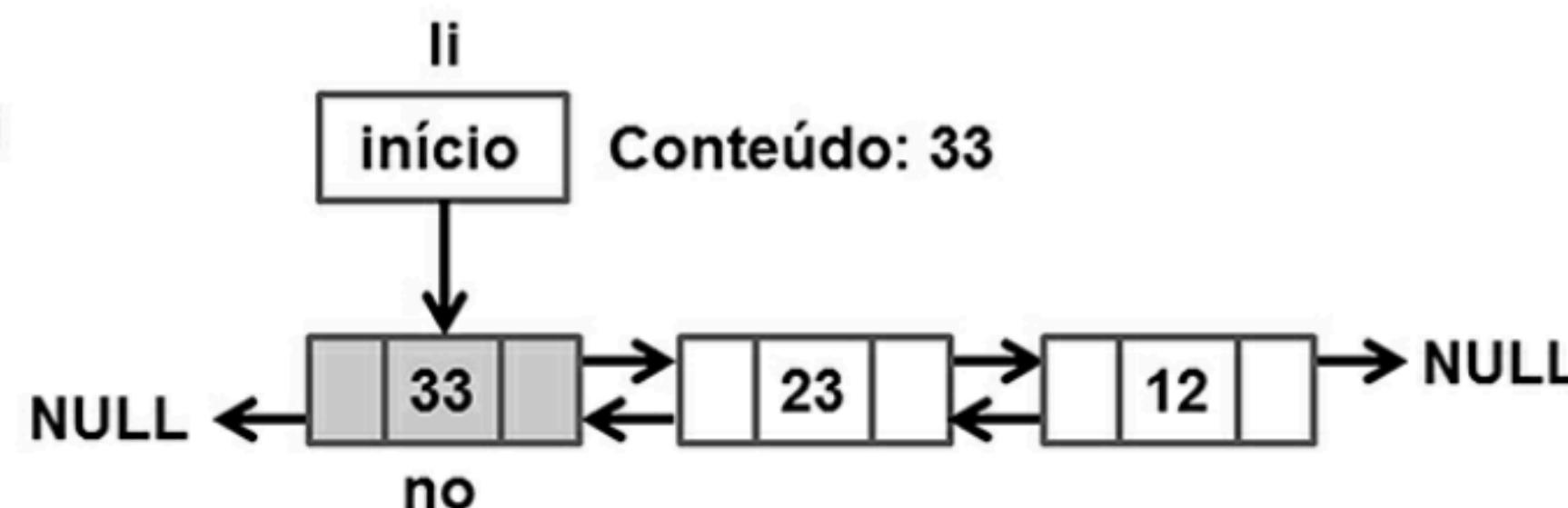
```
01 int busca_lista_mat(Lista* li, int mat, struct aluno *al){  
02     if(li == NULL)  
03         return 0;  
04     Elem *no = *li;  
05     while(no != NULL && no->dados.matricula != mat){  
06         no = no->prox;  
07     }  
08     if(no == NULL)  
09         return 0;  
10     else{  
11         *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) Dada uma lista contendo números inteiros positivos, escreva uma função que calcule:
 - Quantos números pares existem.
 - A média da lista.
 - O maior valor.
 - O menor valor.
 - A posição do maior valor.
 - A posição do menor valor.
 - O número de nós com valor maior do que x .
 - A soma da lista.
 - O número de nós da lista que possuem um número primo.
- 2) Escreva uma função que, dada uma lista L_1 , crie uma cópia dela em L_2 .

TAREFAS

- 3) Escreva uma função que, dada uma lista L1, crie uma cópia dela em L2 eliminando os valores repetidos.
- 4) Escreva uma função que, dada uma lista L1, inverta a lista e a armazene em L2.
- 5) Escreva uma função para verificar se uma lista de inteiros está ordenada ou não. A ordenação pode ser crescente ou decrescente.
- 6) Dadas duas listas ordenadas, L1 e L2, escreva uma função que faça a UNIÃO de ambas em uma terceira lista.
- 7) Dadas duas listas ordenadas, L1 e L2, escreva uma função que faça a INTERSECÇÃO de ambas em uma terceira lista.



OBRIGADO!

RAFAELVC2@GMAIL.COM