

LISTAS

Prof. André Backes | @progdescomplicada

Definição

- Conceito muito comum para as pessoas
 - É uma relação finita de itens, todos de um mesmo tema
 - itens em estoque em uma empresa
 - dias da semana
 - lista de compras do supermercado
 - convidados de uma festa
 - etc.

Listas



33

23

16

15

43

58

Definição

- Na computação, é uma estrutura de dados linear utilizada para armazenar e organizar dados
 - É uma sequência de elementos do mesmo tipo
 - É um controle de fluxo muito comum
 - Pode possuir elementos repetidos, ser ordenada ou não, dependendo da aplicação

Listas



Definição

- Quanto a 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 são inseridos no final, e acessados ou removidos do início da lista
 - pilha: estrutura do tipo **LIFO** (*Last In First Out*)
 - os elementos só podem ser inseridos, acessados ou removidos do final da lista

Definição

- Existem duas implementações principais para uma lista
- Alocação estática
 - Os elementos são armazenados de forma consecutiva na memória (array)
 - É necessário definir o número máximo de elementos da lista
- Alocação dinâmica
 - O espaço de memória é alocado em tempo de execução
 - A lista cresce e diminui com o tempo
 - Cada elemento da lista armazena o endereço de memória do próximo

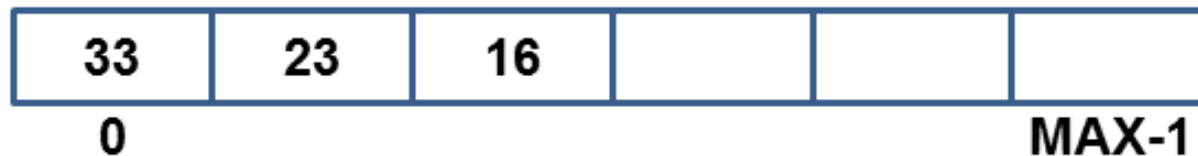
Aplicações

- Filtragem e Classificação de Dados
 - Armazenamento de dados que precisam ser filtrados, ordenados ou agrupados
- Armazenamento Sequencial de Dados
 - Armazenamento de dados em ordem específica, facilitando o acesso, etc
- Representação de Grafos e Tabelas Hash
 - Criação de listas de adjacência e resolver colisões por meio de encadeamento
- Buffers de Entrada e Saída
 - São usadas para armazenar temporariamente dados em sistemas de leitura/gravação de arquivos ou comunicação em rede
- Manipulação de Texto
 - Armazenamento de *strings*, linhas ou palavras, em editores de texto

LISTA ESTÁTICA

Lista Estática

- Lista definida utilizando alocação estática e acesso sequencial dos elementos
 - Trata-se do tipo mais simples de lista possível
 - Essa lista é definida utilizando um array, o sucessor de um elemento ocupa a posição física seguinte



Lista Estática

- Vantagens
 - Acesso direto aos elementos ($O(1)$ para acesso)
 - Simples de implementar
- Desvantagens
 - Tamanho fixo (memória alocada previamente)
 - Ineficiente para inserções/remoções em posições intermediárias ($O(n)$)
- Aplicações
 - Vetores de dados fixos
 - Tabelas de hash simples
 - Listas de alunos em turmas pequenas

Lista Estática | TAD

- Utiliza um array para armazenar os elementos
 - Vantagem: fácil de criar e destruir, acesso rápido e direto aos elementos
 - Desvantagem: necessidade de definir previamente o tamanho da lista, dificuldade para inserir e remover um elemento entre outros dois

```
struct aluno{
    int matricula;
    char nome[30];
    float n1,n2,n3;
};

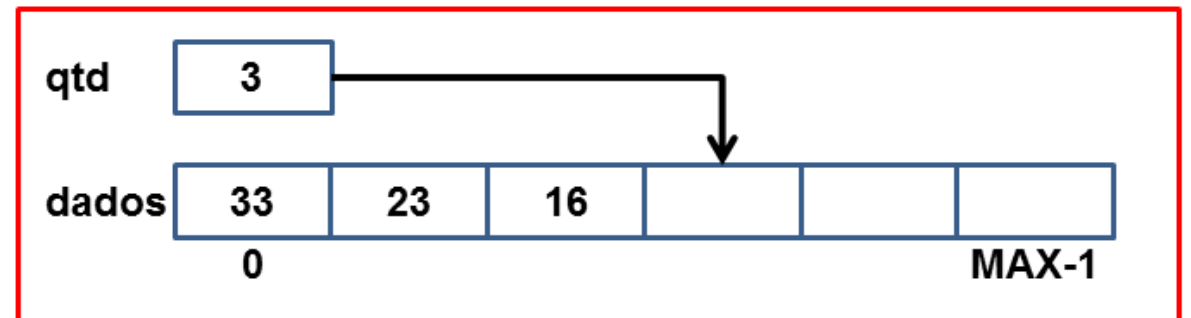
//Definição do tipo lista
#define MAX 100
struct lista{
    int qtd;
    struct aluno dados[MAX];
};

typedef struct lista Lista;
```

Lista Estática | TAD

- A fila é mantida usando apenas um array e a quantidade
 - Indica o quanto do array já está ocupado
- Utiliza alocação estática e acesso sequencial
 - Trata-se do tipo mais simples de lista possível

Lista *li;



Lista Estática | Criação e liberação

- Criação

- Aloca uma área de memória para a lista
- Corresponde a memória necessária para armazenar a estrutura da lista

```
Lista* cria_lista() {  
    Lista *li;  
    li = (Lista*) malloc(sizeof(struct lista));  
    if(li != NULL)  
        li->qtd = 0;  
    return li;  
}
```

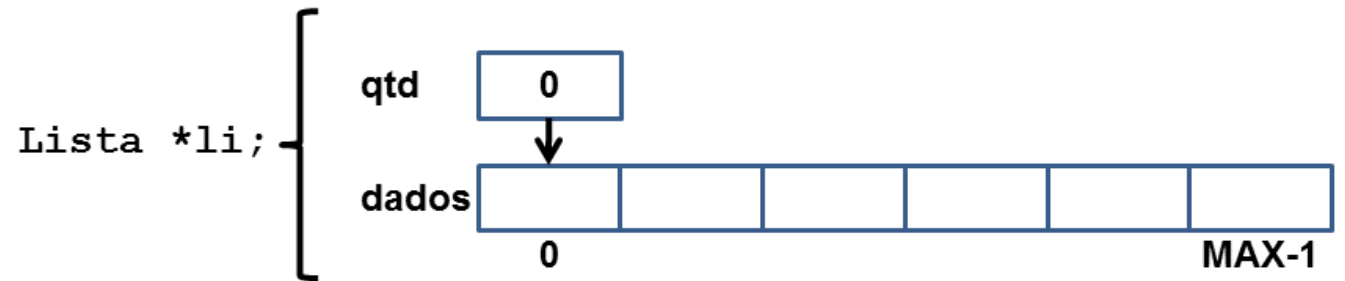
- Liberação

- Basta liberar a memória alocada para a estrutura lista

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

Lista Estática | Criação e liberação

- Ao criar a lista, ela está vazia
 - qtd é igual a ZERO

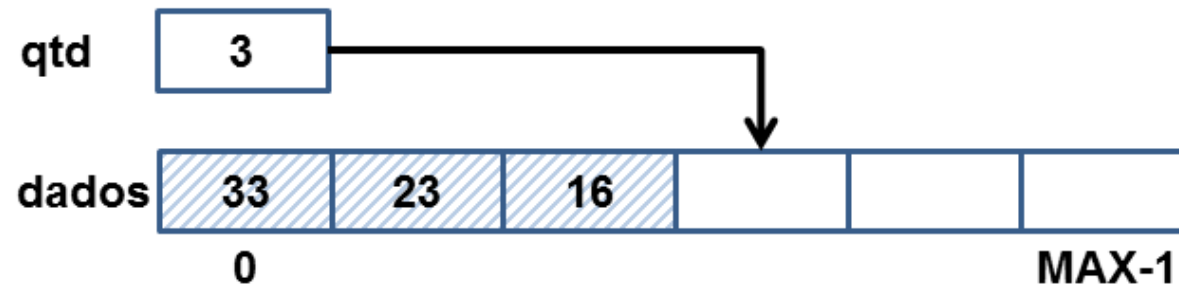


Lista Estática | Inserção no início

- Envolve movimentar todos os elementos da lista uma posição para frente no array
- Precisamos verificar
 - se a lista existe
 - se a lista está cheia
- E só depois
 - mover os elementos
 - copiar os dados
 - incrementar a quantidade

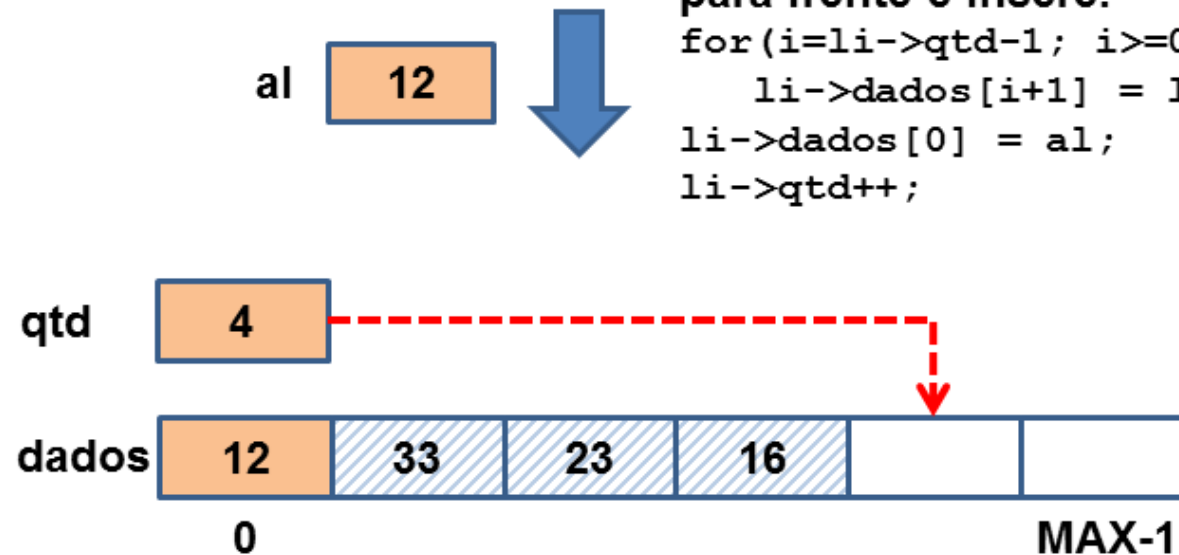
```
int insere_lista_inicio(Lista* li, struct aluno al){  
    if(li == NULL)  
        return 0;  
    if(li->qtd == MAX)//lista cheia  
        return 0;  
    int i;  
    for(i=li->qtd-1; i>=0; i--)  
        li->dados[i+1] = li->dados[i];  
    li->dados[0] = al;  
    li->qtd++;  
    return 1;  
}
```

Lista Estática | Inserção no início



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++;
```

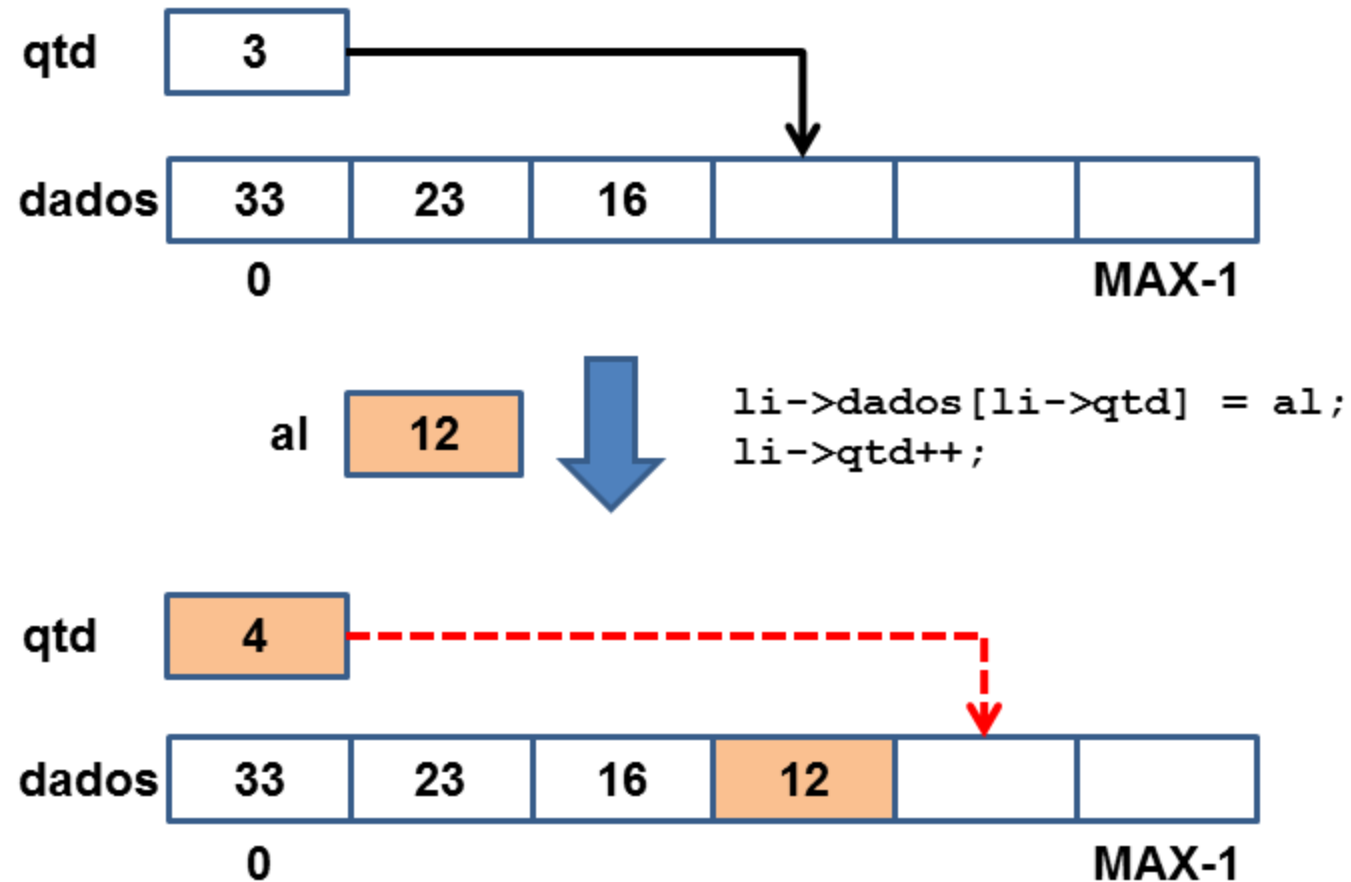


Lista Estática | Inserção no final

- Operação simples e direta
 - qtd indica a posição vaga
- Precisamos verificar
 - se a lista existe
 - se a lista está cheia
- E só depois
 - copiar os dados
 - incrementar a quantidade

```
int insere_lista_final(Lista* li, struct aluno al){  
    if(li == NULL)  
        return 0;  
    if(li->qtd == MAX) //lista cheia  
        return 0;  
    li->dados[li->qtd] = al;  
    li->qtd++;  
    return 1;  
}
```


Lista Estática | Inserção no final



Lista Estática | Inserção ordenada

- Similar a inserção no início
 - Envolve movimentar alguns elementos uma posição para frente no array
 - Precisamos procurar o ponto de inserção
 - Lembra um pouco o *insertionsort*

```
int insere_lista_ordenada(Lista* li, struct aluno al){  
    if(li == NULL || li->qtd == MAX)  
        return 0;  
  
    int k, i = 0;  
    while(i < li->qtd &&  
        li->dados[i].matricula < al.matricula)  
        i++;  
  
    for(k = li->qtd - 1; k >= i; k--)  
        li->dados[k+1] = li->dados[k];  
  
    li->dados[i] = al;  
    li->qtd++;  
    return 1;  
}
```

Lista Estática | Inserção ordenada

Lista inicial
Busca onde Inserir:

dados	16	23	33			
	0					MAX-1

```
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];
```

dados	12	16	23	33		
	0					MAX-1

dados	16	19	23	33		
	0					MAX-1

Inserir elemento

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

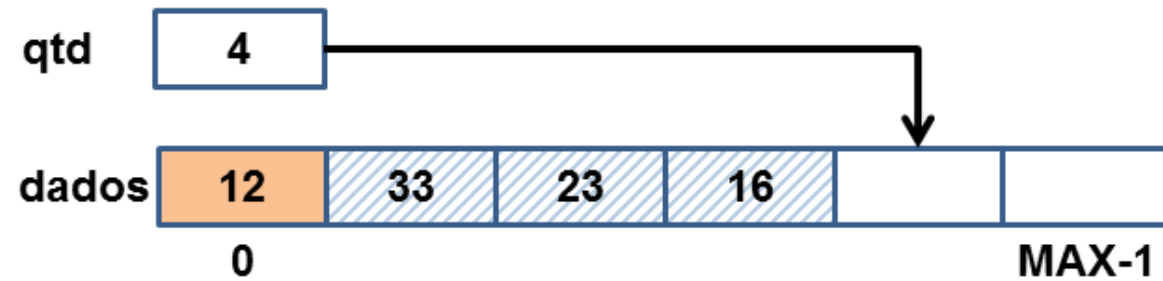
dados	16	23	33	40		
	0					MAX-1

Lista Estática | Remoção do início

- Como na inserção, envolve movimentar todos os elementos da lista uma posição para trás no array
- Precisamos verificar
 - se a lista existe
 - se a lista está vazia
- E só depois
 - mover os elementos
 - diminuir a quantidade

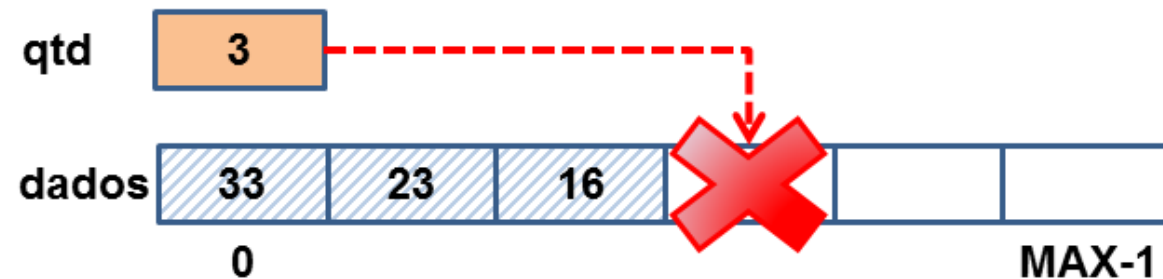
```
int remove_lista_inicio(Lista* li){  
    if(li == NULL)  
        return 0;  
    if(li->qtd == 0) //lista vazia  
        return 0;  
    int k = 0;  
    for(k=0; k< li->qtd-1; k++)  
        li->dados[k] = li->dados[k+1];  
    li->qtd--;  
    return 1;  
}
```

Lista Estática | Remoção do início



Desloca os elementos uma posição para trás:

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

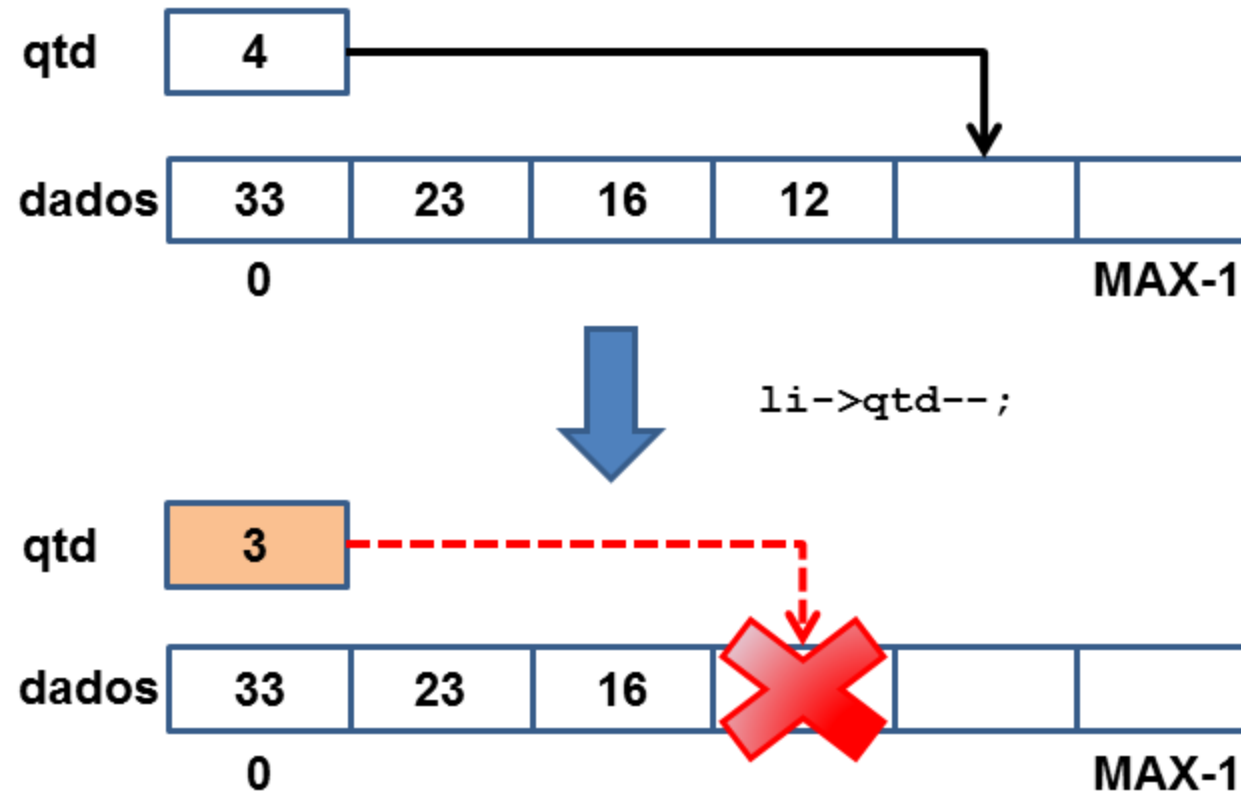


Lista Estática | Remoção do final

- Operação simples e direta
 - qtd indica a posição vaga
 - Basta diminuir a quantidade de elementos na lista
- Precisamos verificar
 - se a lista existe
 - se a lista está cheia

```
int remove_lista_final(Lista* li){  
    if(li == NULL)  
        return 0;  
    if(li->qtd == 0) //lista vazia  
        return 0;  
    li->qtd--;  
    return 1;  
}
```

Lista Estática | Remoção do final



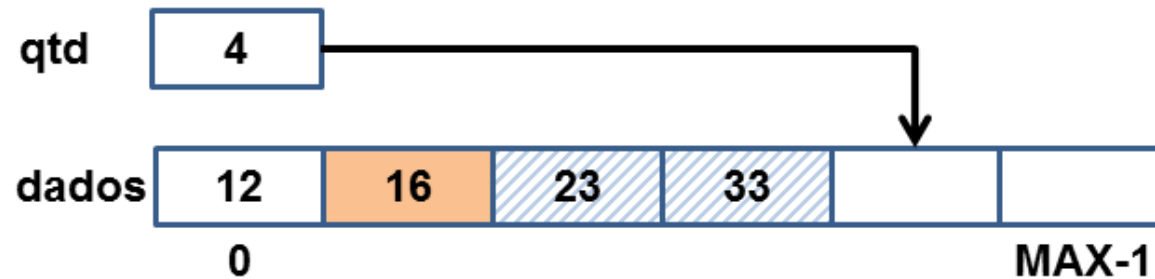
Lista Estática | Remoção ordenada

- Similar a remoção do início
 - Envolve movimentar alguns elementos uma posição para trás no array
 - Precisamos procurar o elemento a ser removido
 - Lembra um pouco o *insertionsort*

```
int remove_lista(Lista* li, int mat){
    if(li == NULL)
        return 0;
    if(li->qtd == 0) //lista vazia
        return 0;
    int k,i = 0;
    while(i < li->qtd && li->dados[i].matricula != mat)
        i++;
    if(i == li->qtd) //elemento nao encontrado
        return 0;

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


Lista Estática | Remoção ordenada



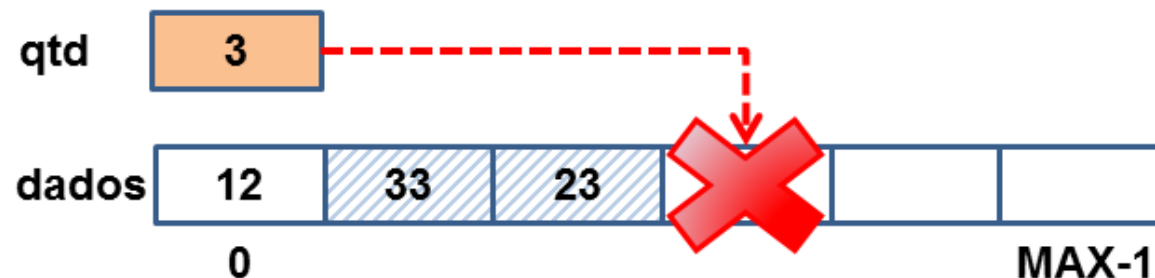
Procura elemento a ser removido:

```
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--;
```



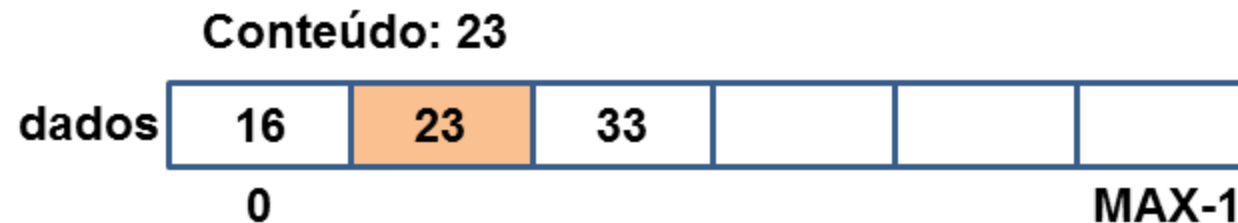
Lista Estática | Acesso

- Podemos acessar qualquer elemento
- A busca pode ser por
 - posição
 - conteúdo

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

```
int busca_lista_mat(Lista* li, int mat, struct aluno *al){  
    if(li == NULL) return 0;  
    int i = 0;  
    while(i < li->qtd && li->dados[i].matricula != mat)  
        i++;  
    if(i == li->qtd) //elemento nao encontrado  
        return 0;  
  
    *al = li->dados[i];  
    return 1;  
}
```

Lista Estática | Acesso



Busca pelo elemento:

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

Achou o elemento:

```
*a1 = li->dados[i];
```

Lista Estática | Complexidade

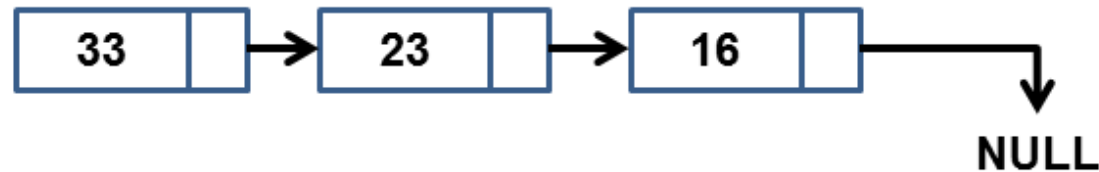
- Ao lado são mostradas as complexidades computacionais das principais operações na Lista Sequencial Estática

Operação	Início	Final	Ordenada
Inserção	$O(N)$	$O(1)$	$O(N)$
Remoção	$O(N)$	$O(1)$	$O(N)$
Busca	$O(N)$		

LISTA DINÂMICA

Lista Dinâmica

- Lista definida utilizando alocação dinâmica e acesso encadeado dos elementos
 - Seus elementos são ponteiros alocados dinamicamente
 - Um elemento é alocado a medida que os dados são inseridos na lista, e tem sua memória liberada a medida que é removido



Lista Dinâmica

- Vantagens
 - Tamanho flexível (memória alocada conforme necessário)
 - Melhor utilização dos recursos de memória
 - Não precisa movimentar os elementos nas operações de inserção e remoção
- Desvantagens
 - Acesso sequencial ($O(N)$ para busca de elementos)
 - Maior uso de memória devido aos ponteiros
- Aplicações
 - Implementação de pilhas, filas, grafos (lista de adjacência)

Lista Dinâmica | TAD

- Cada elemento (nó) armazena os dados e um ponteiro para o próximo
 - Cada nó equivale a uma posição do array que define a Lista Estática.
- Problema
 - Como manter o início da lista?

```
//Definição do tipo lista
struct elemento{
    struct aluno dados;
    struct elemento *prox;
};

typedef struct elemento Elem;
typedef struct elemento* Lista;
```


Lista Dinâmica | TAD

- Explicando melhor o problema
 - O nó inicial da lista pode mudar, mas não é aconselhável que a lista mude de lugar na memória
- Lista Estática
 - Inserção/remoção altera apenas o conteúdo da estrutura da lista e não onde ela está na memória
- Lista Dinâmica
 - Não temos uma estrutura para a lista, apenas para seus elementos
 - Inserção e remoção no início alteram o endereço onde a lista se inicia

```
//Definição do tipo lista
struct elemento{
    struct aluno dados;
    struct elemento *prox;
};

typedef struct elemento Elem;
typedef struct elemento* Lista;
```

Lista Dinâmica | TAD

- Solução

- Criar uma estrutura apenas para manter o início da lista (nó descritor)
 - Semelhante a Lista Estática
- Usar um ponteiro para ponteiro
 - Permite guardar o endereço do ponteiro que representa o primeiro nó da lista

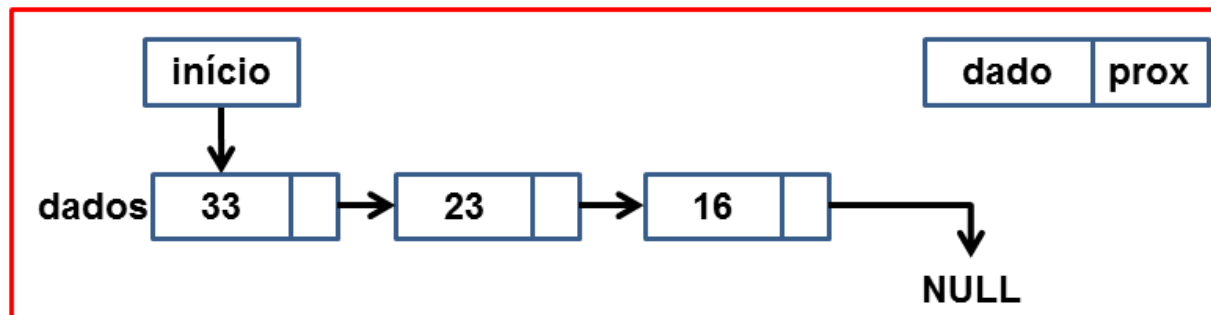
```
//Definição do tipo lista
struct elemento{
    struct aluno dados;
    struct elemento *prox;
};

typedef struct elemento Elem;
typedef struct elemento* Lista;
```

Lista Dinâmica | TAD

- A posição final da lista aponta para NULL
- O início da lista é mantido usando um ponteiro para ponteiro
 - `Lista *li;`
 - `struct elemento **li;`

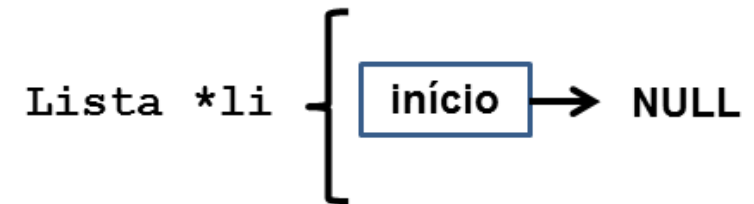
Lista *li



Lista Dinâmica | Criação

- Faz a alocação de uma área de memória para armazenar o endereço do início da lista
 - Equivale a criar uma lista vazia

```
Lista* cria_lista(){  
    Lista* li = (Lista*) malloc(sizeof(Lista));  
    if(li != NULL)  
        *li = NULL;  
    return li;  
}
```



Lista Dinâmica | Liberação

- Para liberar uma lista dinâmica é preciso percorrer toda a lista liberando a memória alocada para cada elemento inserido
 - Equivale a remoção do início
 - Processo termina ao encontrar NULL
- Ao final, liberamos a memória da lista em si

```
void libera_lista(Lista* li) {  
    if(li != NULL) {  
        Elem* no;  
        while((*li) != NULL) {  
            no = *li;  
            *li = (*li)->prox;  
            free(no);  
        }  
        free(li);  
    }  
}
```

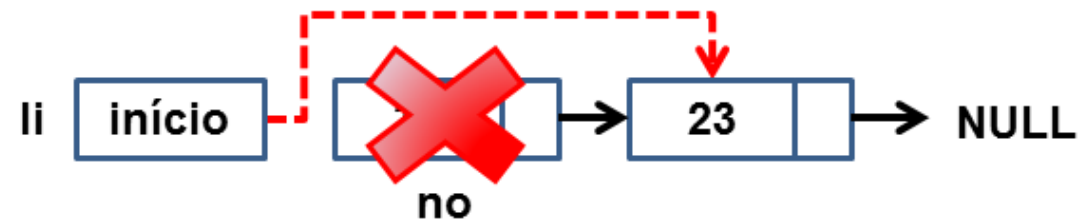
Lista Dinâmica | Liberação

Lista inicial



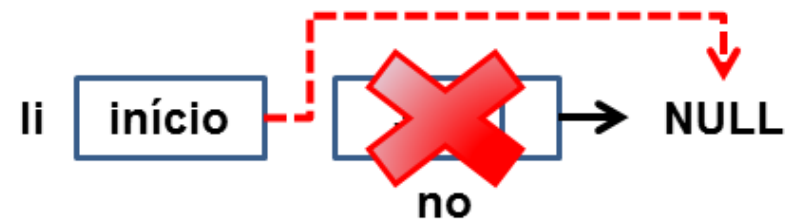
Passo 1:

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



Passo 2:

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



Fim:

```
no == NULL
```

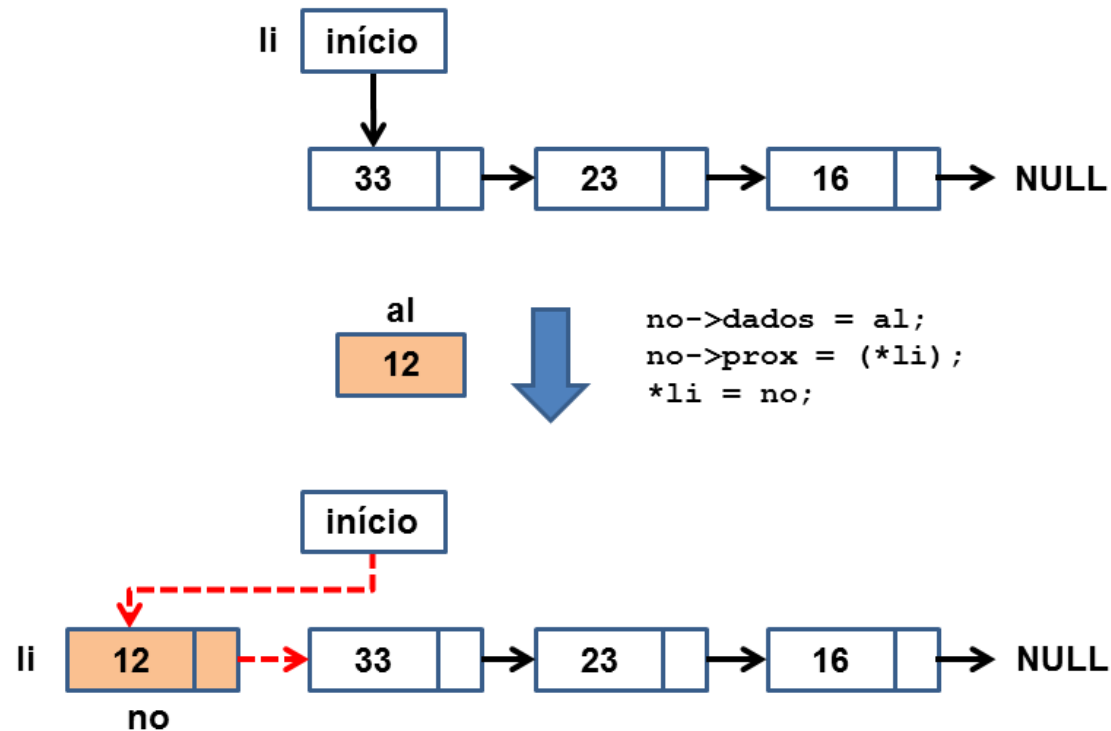
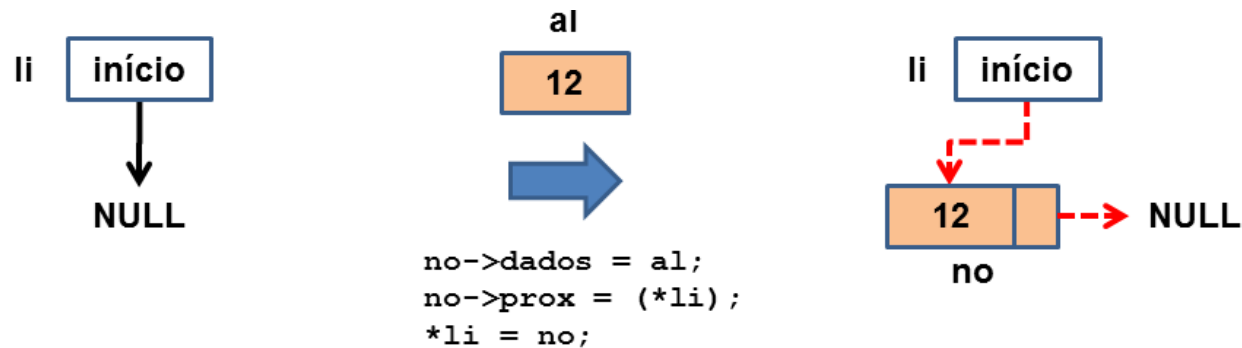


Lista Dinâmica | Inserção no início

- Tarefa simples
 - Envolve alocar espaço para o novo elemento e ajustar o início
- Basicamente, se a lista existe
 - alocar memória para o novo nó
 - copiar os dados
 - mudar o início

```
int insere_lista_inicio(Lista* li, struct aluno al){  
    if(li == NULL)  
        return 0;  
    Elem* no;  
    no = (Elem*) malloc(sizeof(Elem));  
    if(no == NULL)  
        return 0;  
    no->dados = al;  
    no->prox = (*li);  
    *li = no;  
    return 1;  
}
```

Lista Dinâmica | Inserção no início



Lista Dinâmica | Inserção no final

- Tarefa simples, mas trabalhosa
 - Envolve alocar espaço para o novo elemento e ajustar o início
- Basicamente, se a lista existe
 - alocar memória para o novo nó
 - copiar os dados
 - procurar o último elemento
 - apontar o último para o novo

```
int insere_lista_final(Lista* li, struct aluno al)
{
    if(li == NULL)
        return 0;
    Elem *no = (Elem*) malloc(sizeof(Elem));
    if(no == NULL)
        return 0;
    no->dados = al;
    no->prox = NULL;
    if((*li) == NULL){//lista vazia: insere início
        *li = no;
    }else{
        Elem *aux;
        aux = *li;
        while(aux->prox != NULL){
            aux = aux->prox;
        }
        aux->prox = no;
    }
    return 1;
}
```

Lista Dinâmica | Inserção no final



Busca onde Inserir:

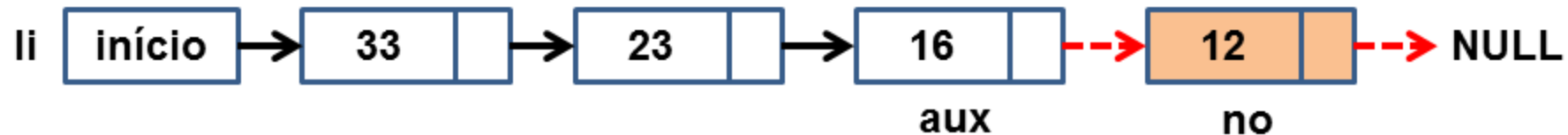
```
aux = *li;  
while(aux->prox != NULL) {  
    aux = aux->prox;  
}
```

al

12

Insere depois de “aux”:

```
no->dados = al;  
no->prox = NULL;  
aux->prox = no;
```



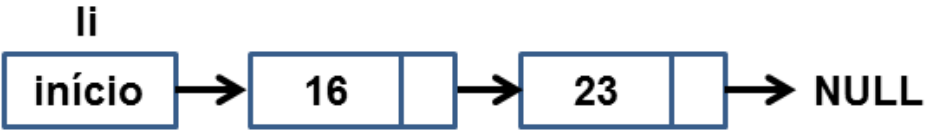
Lista Dinâmica | Inserção ordenada

- Envolve procurar o local de inserção
 - pode ser no início, meio ou final da lista
 - também pode ser uma lista vazia
- Basicamente, se a lista existe
 - alocar memória para o novo nó
 - copiar os dados
 - Achou último elemento com matricula menor?
 - aponta-lo para o novo
 - Caso contrário
 - Sou maior que todos? Insere no final
 - Sou menor que todos? Insere no início

```
int insere_lista_ordenada(Lista* li, struct aluno al)
{
    if(li == NULL) return 0;
    Elem *no = (Elem*) malloc(sizeof(Elem));
    if(no == NULL)
        return 0;
    no->dados = al;
    if((*li) == NULL){//lista vazia: insere início
        no->prox = NULL;
        *li = no;
        return 1;
    }else{
        Elem *ant, *atual = *li;
        while(atual != NULL &&
            atual->dados.matricula < al.matricula){
            ant = atual;
            atual = atual->prox;
        }
        if(atual == *li){//insere início
            no->prox = (*li);
            *li = no;
        }else{
            no->prox = atual;
            ant->prox = no;
        }
        return 1;
    }
}
```

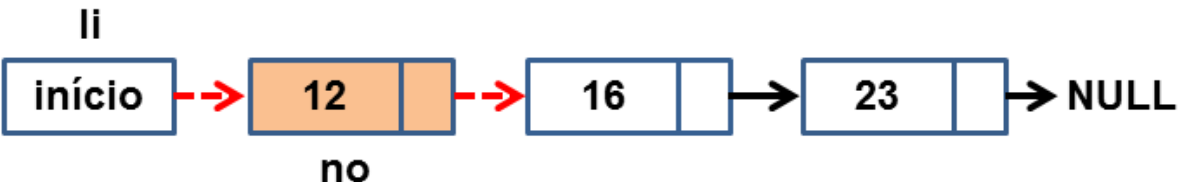
Lista Dinâmica | Inserção ordenada

Lista inicial
Busca onde Inserir:



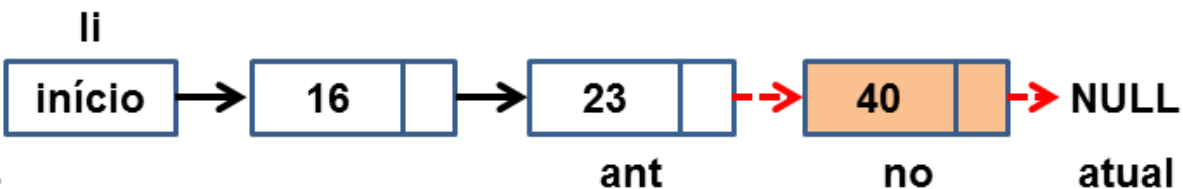
```
atual = *li;  
while(atual != NULL && atual->dados.matricula < al.matricula){  
    ant = atual;  
    atual = atual->prox;  
}
```

Inserir no início:

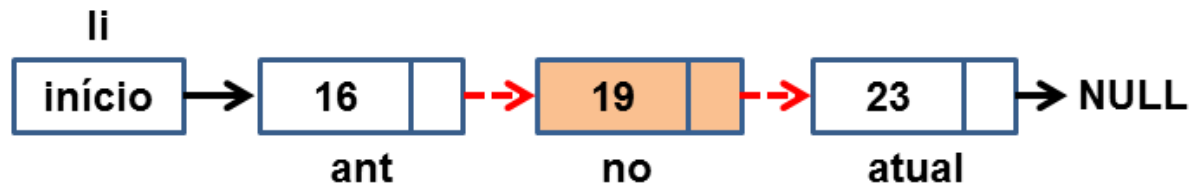


```
no->prox = (*li);  
*li = no;
```

Inserir depois de "ant":



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

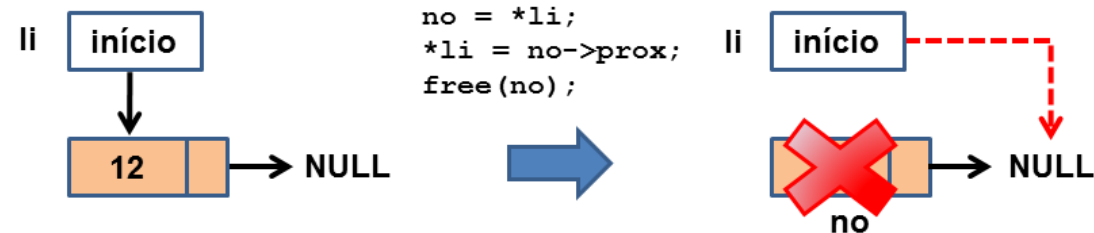


Lista Dinâmica | Remoção do início

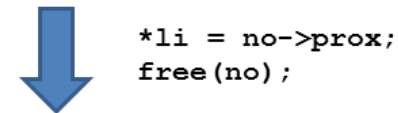
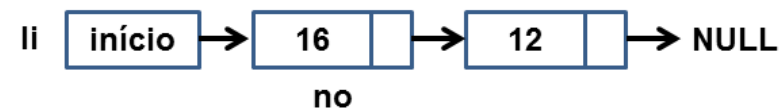
- Envolve liberar a memória do elemento removido e ajustar alguns ponteiros
- Primeiro, verificamos se
 - a lista existe
 - a lista possui elementos
- Em seguida
 - mudar o início da lista
 - liberar a memória do nó

```
int remove_lista_inicio(Lista* li){  
    if(li == NULL)  
        return 0;  
    if((*li) == NULL)//lista vazia  
        return 0;  
  
    Elem *no = *li;  
    *li = no->prox;  
    free(no);  
    return 1;  
}
```

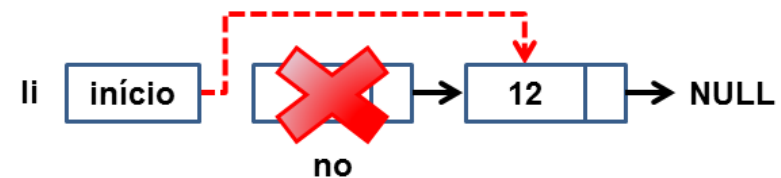
Lista Dinâmica | Remoção do início



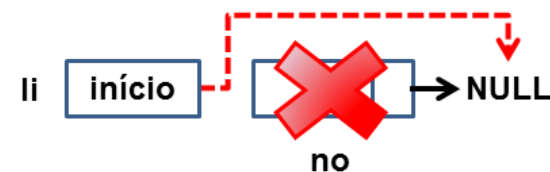
Lista inicial



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



Se a lista possui um único elemento, ela fica vazia



Lista Dinâmica | Remoção do final

- Tarefa simples, mas trabalhosa
 - Envolve percorrer a lista toda e liberar a memória do elemento removido
- Primeiro, verificamos se
 - a lista existe
 - a lista possui elementos
- Em seguida
 - encontrar o final da lista
 - verificar se lista fica vazia
 - liberar a memória do nó

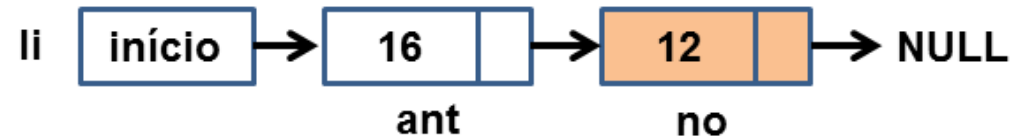
```
int remove_lista_final(Lista* li){  
    if(li == NULL)  
        return 0;  
    if((*li) == NULL) //lista vazia  
        return 0;  
  
    Elem *ant, *no = *li;  
    while(no->prox != NULL) {  
        ant = no;  
        no = no->prox;  
    }  
  
    if(no == (*li)) //remover o primeiro?  
        *li = no->prox;  
    else  
        ant->prox = no->prox;  
    free(no);  
    return 1;  
}
```

Lista Dinâmica | Remoção do final

Lista inicial

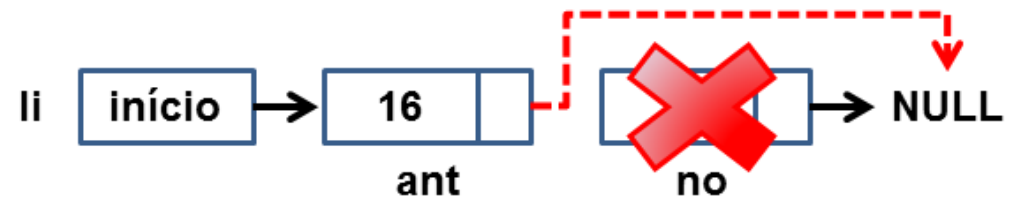
Busca o último elemento:

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

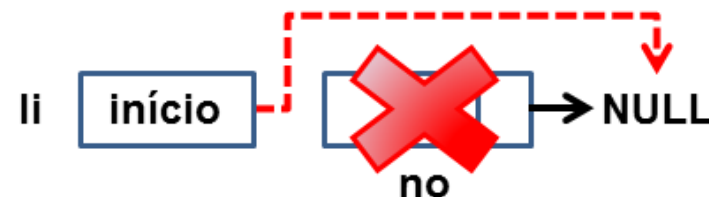


`ant->prox = no->prox;`
`free(no);`

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



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



Lista Dinâmica | Remoção ordenada

- Envolve procurar o local de remoção
 - pode ser no início, meio ou final da lista
 - a lista pode ficar vazia
- Se a lista existe e possui elementos
 - procurar o elemento a ser removido
 - É o primeiro da lista?
 - ajustar o início
 - Não é o primeiro?
 - o anterior dele aponta para seu próximo
 - liberar a memória do nó

```
int remove_lista(Lista* li, int mat){
    if(li == NULL)
        return 0;
    if((*li) == NULL) //lista vazia
        return 0;
    Elem *ant, *no = *li;
    while(no != NULL && no->dados.matricula != mat){
        ant = no;
        no = no->prox;
    }
    if(no == NULL) //não encontrado
        return 0;

    if(no == *li) //remover o primeiro?
        *li = no->prox;
    else
        ant->prox = no->prox;
    free(no);
    return 1;
}
```

Lista Dinâmica | Remoção ordenada

Lista inicial

Busca qual remover:



```
no = *li;
```

```
while(no != NULL && no->dados.matricula != mat) {
```

```
    ant = no;
```

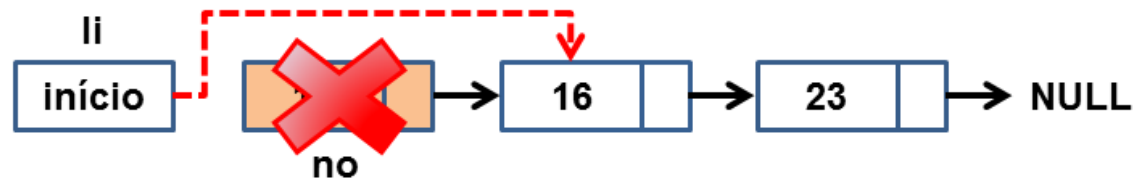
```
    no = no->prox;
```

```
}
```

Remover do início:

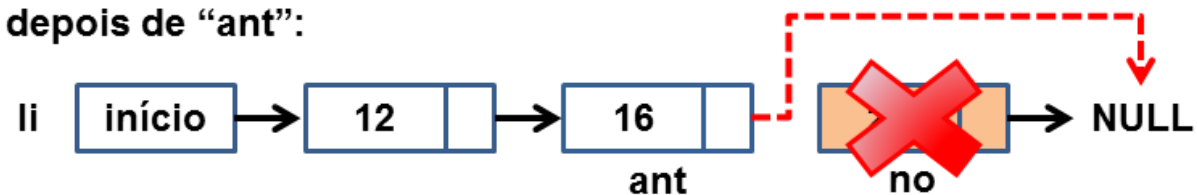
```
*li = no->prox;
```

```
free(no);
```



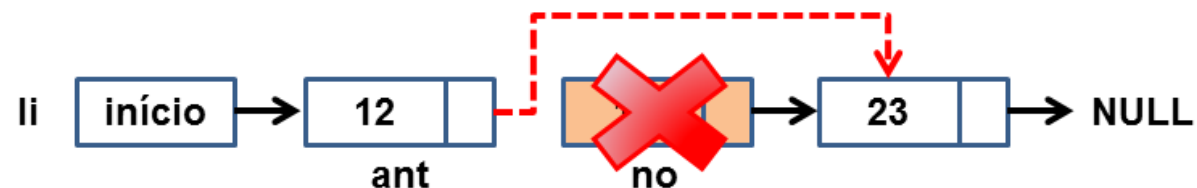
Remover do meio ou do fim

significa remover depois de “ant”:



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

```
free(no);
```



Lista Dinâmica | Acesso

- Podemos acessar qualquer elemento
- A busca pode ser por
 - posição
 - conteúdo

```
int busca_lista_pos(Lista* li, int pos, struct aluno *al){  
    if(li == NULL || pos <= 0) return 0;
```

```
    Elem *no = *li;  
    int i = 1;  
    while(no != NULL && i < pos){  
        no = no->prox;  
        i++;  
    }  
    if(no == NULL)  
        return 0;  
    else{  
        *al = no->dados;  
        return 1;  
    }  
}
```

```
int busca_lista_mat(Lista* li, int mat, struct aluno *al){  
    if(li == NULL) return 0;
```

```
    Elem *no = *li;  
    while(no != NULL && no->dados.matricula != mat){  
        no = no->prox;  
    }  
    if(no == NULL)  
        return 0;  
    else{  
        *al = no->dados;  
        return 1;  
    }  
}
```

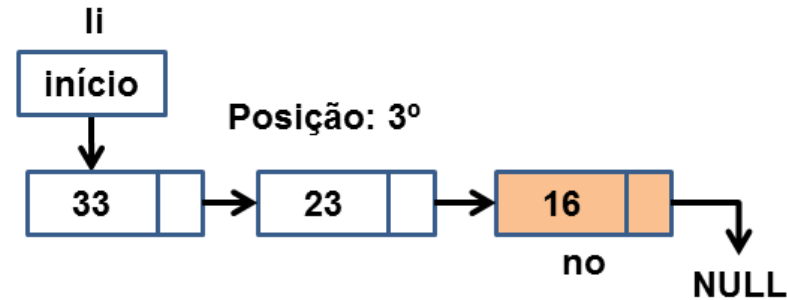
Lista Dinâmica | Acesso

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;  
}
```

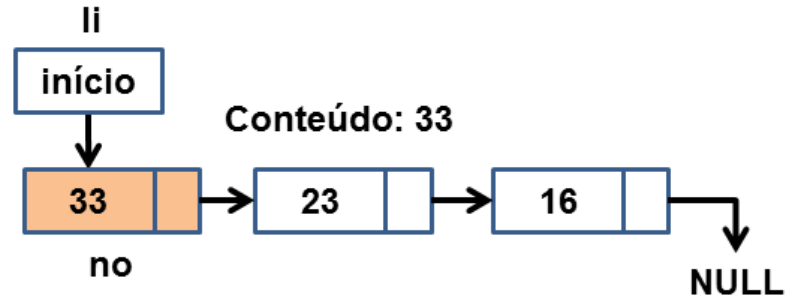


Busca pelo conteúdo do elemento

```
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;  
}
```



Lista Dinâmica | Complexidade

- Ao lado são mostradas as complexidades computacionais das principais operações na Lista Dinâmica Encadeada

Operação	Início	Final	Ordenada
Inserção	$O(1)$	$O(N)$	$O(N)$
Remoção	$O(1)$	$O(N)$	$O(N)$
Busca	$O(N)$		

LISTA DINÂMICA DUPLAMENTE ENCADEADA

Lista Dinâmica Dupla

- Similar a Lista Dinâmica Encadeada, é utilizada quando existe a necessidade de acessar a informação de um elemento antecessor
 - Presença de dois ponteiros, prox e ant, garantem que a lista seja encadeada em dois sentidos
 - sentido normal, do seu início até o seu final
 - sentido inverso, do final até o seu início



Lista Dinâmica Dupla

- Vantagens
 - Acesso em ambas as direções (para frente e para trás)
 - Inserções e remoções eficientes em qualquer posição
- Desvantagens
 - Usa mais memória (dois ponteiros por nó)
 - Inserções e remoções são um pouco mais complexas que na lista simplesmente encadeada
- Aplicações
 - Implementação de editores de texto
 - navegadores (histórico de páginas)
 - caches (LRU)

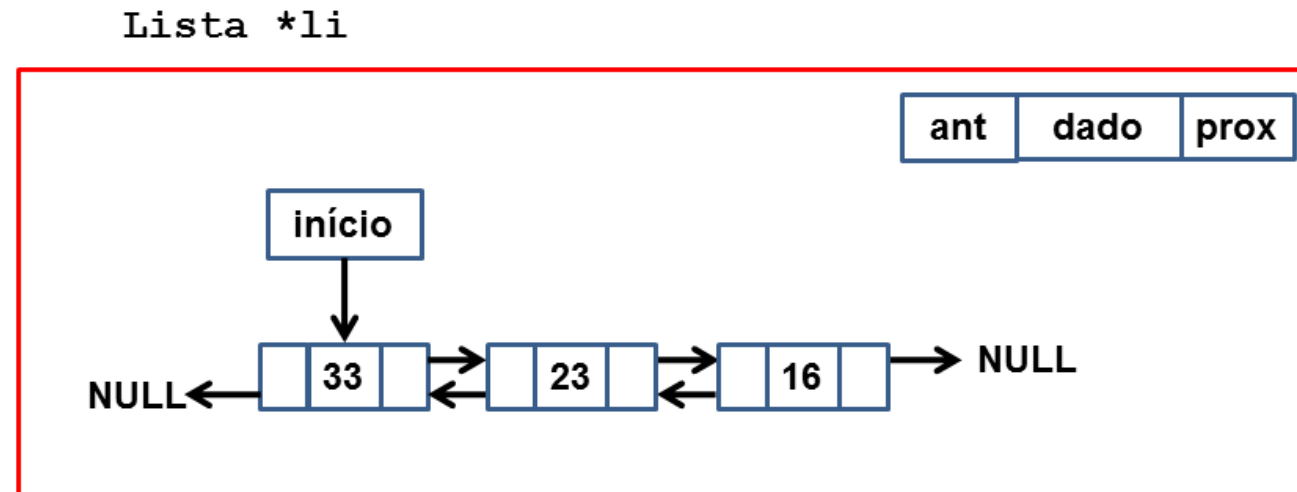
Lista Dinâmica Dupla | TAD

- Implementação similar a da Lista Dinâmica
 - Baseada num ponteiro para ponteiro
 - A diferença é a existência de um ponteiro para o anterior em cada nó, além dos dados e um ponteiro para o próximo

```
//Definição do tipo lista
struct elemento{
    struct elemento *ant;
    struct aluno dados;
    struct elemento *prox;
};
typedef struct elemento Elem;
typedef struct elemento* Lista;
```

Lista Dinâmica Dupla | TAD

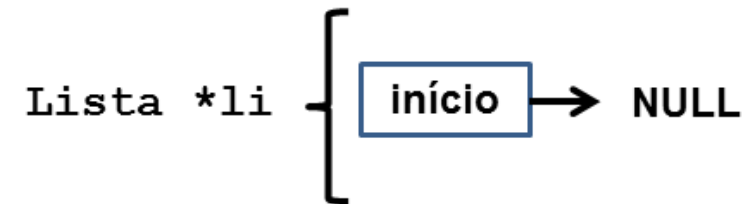
- Diferente da Lista Dinâmica, essa lista possui dois finais
 - Tanto o sentido normal quanto o sentido inverso devem sinalizar o final da lista com NULL



Lista Dinâmica Dupla | Criação

- Faz a alocação de uma área de memória para armazenar o endereço do início da lista
 - Equivale a criar uma lista vazia

```
Lista* cria_lista(){  
    Lista* li = (Lista*) malloc(sizeof(Lista));  
    if(li != NULL)  
        *li = NULL;  
    return li;  
}
```



Lista Dinâmica Dupla | Liberação

- Para liberar uma lista dinâmica é preciso percorrer toda a lista liberando a memória alocada para cada elemento inserido
 - Equivale a remoção do início
 - Processo termina ao encontrar NULL
- Ao final, liberamos a memória da lista em si

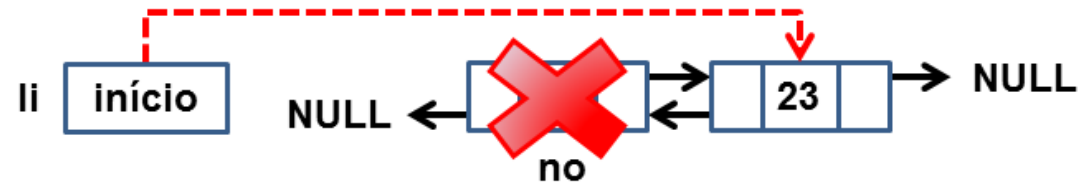
```
void libera_lista(Lista* li) {  
    if(li != NULL) {  
        Elem* no;  
        while((*li) != NULL) {  
            no = *li;  
            *li = (*li)->prox;  
            free(no);  
        }  
        free(li);  
    }  
}
```

Lista Dinâmica Dupla | Liberação



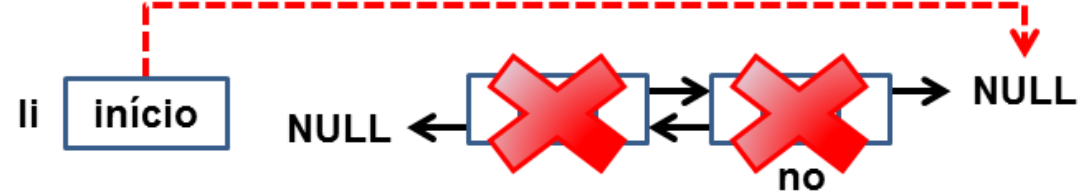
Passo 1:

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



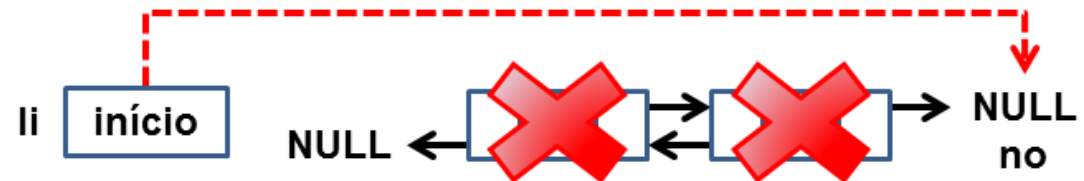
Passo 2:

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



Fim:

```
no == NULL
```

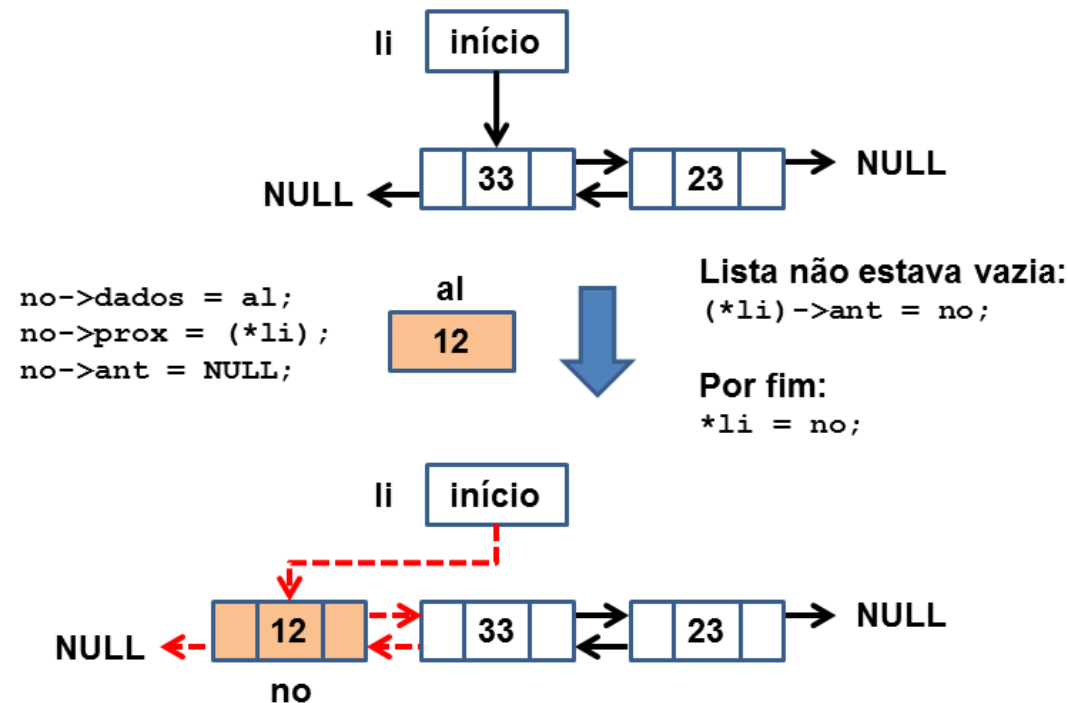
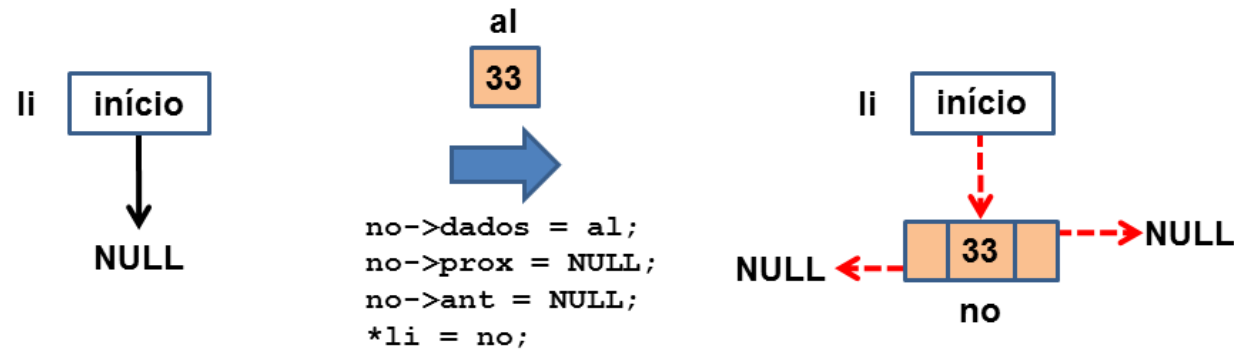


Lista Dinâmica Dupla | Inserção no início

- Tarefa simples
 - Envolve alocar espaço para o novo elemento e ajustar o início
- Basicamente, se a lista existe
 - alocar memória para o novo nó
 - copiar os dados
 - ajustar anterior e próximo
 - mudar o início

```
int insere_lista_inicio(Lista* li, struct aluno al){  
    if(li == NULL)  
        return 0;  
    Elem* no = (Elem*) malloc(sizeof(Elem));  
    if(no == NULL)  
        return 0;  
    no->dados = al;  
    no->prox = (*li);  
    no->ant = NULL;  
    //lista não vazia: apontar para o anterior!  
    if(*li != NULL)  
        (*li)->ant = no;  
    *li = no;  
    return 1;  
}
```

Lista Dinâmica Dupla | Inserção no início

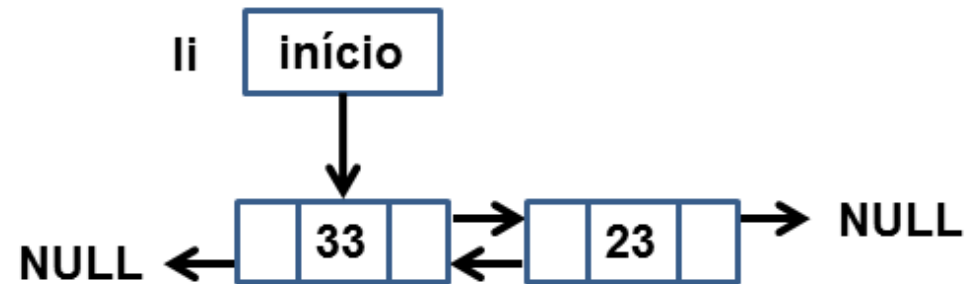


Lista Dinâmica Dupla | Inserção no final

- Tarefa simples, mas trabalhosa
 - Envolve alocar espaço para o novo elemento e ajustar o início
- Basicamente, se a lista existe
 - alocar memória para o novo nó
 - copiar os dados
 - procurar o último elemento
 - apontar o último para o novo e o novo para o último

```
int insere_lista_final(Lista* li, struct aluno al){
    if(li == NULL) return 0;
    Elem *no = (Elem*) malloc(sizeof(Elem));
    if(no == NULL)
        return 0;
    no->dados = al;
    no->prox = NULL;
    if((*li) == NULL){//lista vazia: insere início
        no->ant = NULL;
        *li = no;
    }else{
        Elem *aux = *li;
        while(aux->prox != NULL){
            aux = aux->prox;
        }
        aux->prox = no;
        no->ant = aux;
    }
    return 1;
}
```


Lista Dinâmica Dupla | Inserção no final



Busca onde Inserir:

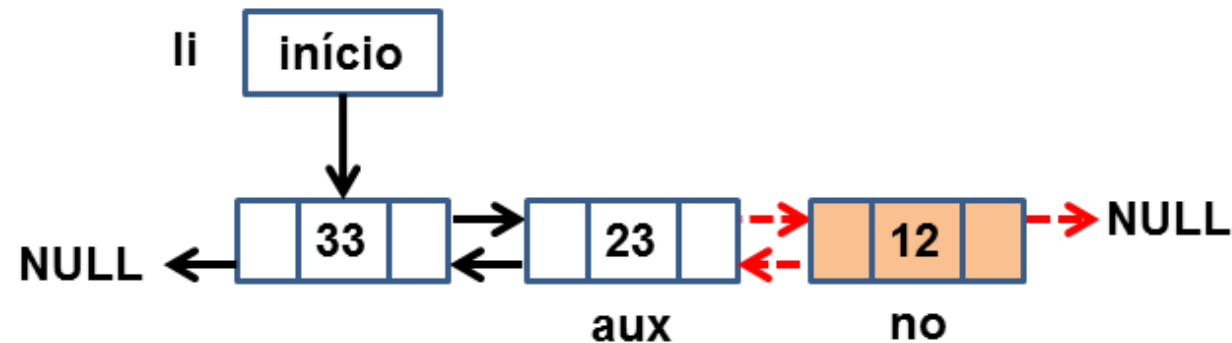
```
aux = *li;  
while(aux->prox != NULL) {  
    aux = aux->prox;  
}
```

al
12



Inserir depois de "aux":

```
no->dados = al;  
no->prox = NULL;  
aux->prox = no;  
no->ant = aux;
```



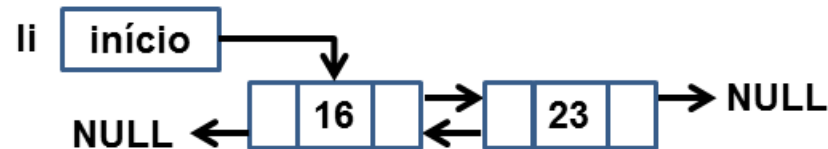
Lista Dinâmica Dupla | Inserção ordenada

- Envolve procurar o local de inserção
 - pode ser no início, meio ou final da lista
 - também pode ser uma lista vazia
- Basicamente, se a lista existe
 - alocar memória para o novo nó
 - copiar os dados
 - Achou último elemento com matricula menor?
 - aponta-lo para o novo
 - Caso contrário
 - Sou maior que todos? Insere no final
 - Sou menor que todos? Insere no início

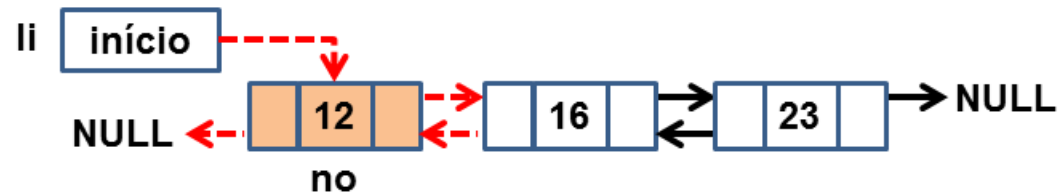
```
int insere_lista_ordenada(Lista* li, struct aluno al){
    if(li == NULL) return 0;
    Elem *no = (Elem*) malloc(sizeof(Elem));
    if(no == NULL)
        return 0;
    no->dados = al;
    if((*li) == NULL){//lista vazia: insere início
        no->prox = NULL;
        no->ant = NULL;
        *li = no;
        return 1;
    }
    else{
        Elem *ante, *atual = *li;
        while(atual != NULL && atual->dados.matricula < al.matricula){
            ante = atual;
            atual = atual->prox;
        }
        if(atual == *li){//insere início
            no->ant = NULL;
            (*li)->ant = no;
            no->prox = (*li);
            *li = no;
        }else{
            no->prox = ante->prox;
            no->ant = ante;
            ante->prox = no;
            if(atual != NULL)
                atual->ant = no;
        }
        return 1;
    }
}
```

Lista Dinâmica Dupla | Inserção ordenada

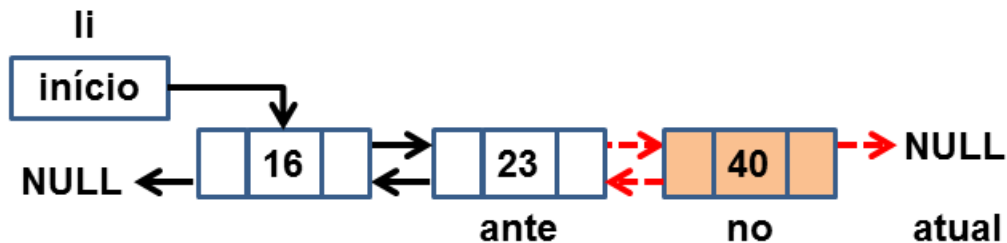
Lista inicial `atual = *li;`
Busca onde `while(atual!=NULL && atual->dados.matricula < al.matricula){`
inserir: `ante = atual;`
`atual = atual->prox;`
`}`



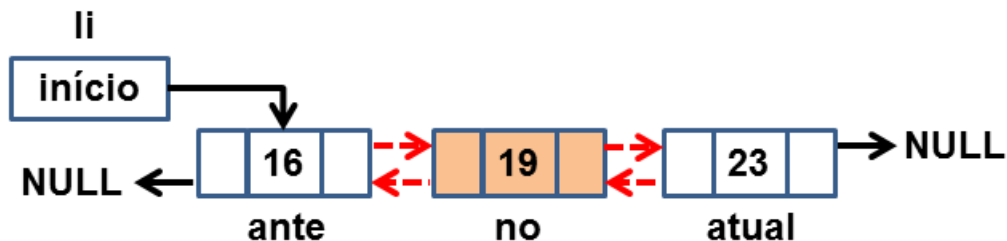
Inserir no início:
`no->ant = NULL;`
`(*li)->ant = no;`
`no->prox = (*li);`
`*li = no;`



Inserir depois de “ante”:
`no->prox=ante->prox;`
`no->ant = ante;`
`ante->prox = no;`



Não é final da lista:
`atual->ant = no;`



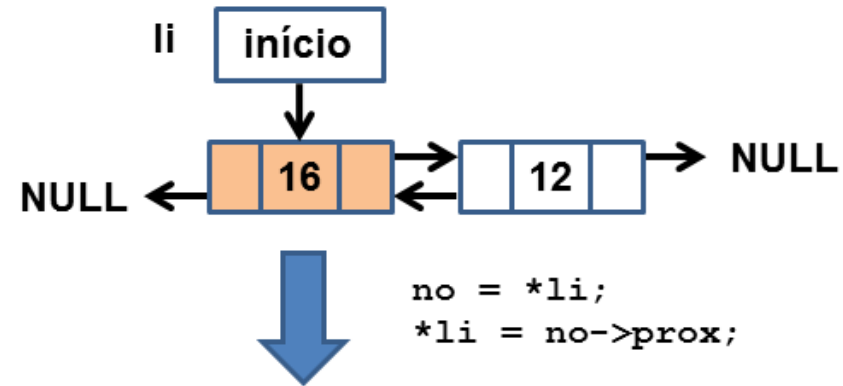
Lista Dinâmica Dupla | Remoção do início

- Envolve liberar a memória do elemento removido e ajustar alguns ponteiros
- Primeiro, verificamos se
 - a lista existe
 - a lista possui elementos
- Em seguida
 - mudar o início da lista
 - liberar a memória do nó
 - Se existe um nó próximo, seu anterior passar a ser NULL

```
int remove_lista_inicio(Lista* li){  
    if(li == NULL)  
        return 0;  
    if((*li) == NULL) //lista vazia  
        return 0;  
  
    Elem *no = *li;  
    *li = no->prox;  
    if(no->prox != NULL)  
        no->prox->ant = NULL;  
  
    free(no);  
    return 1;  
}
```

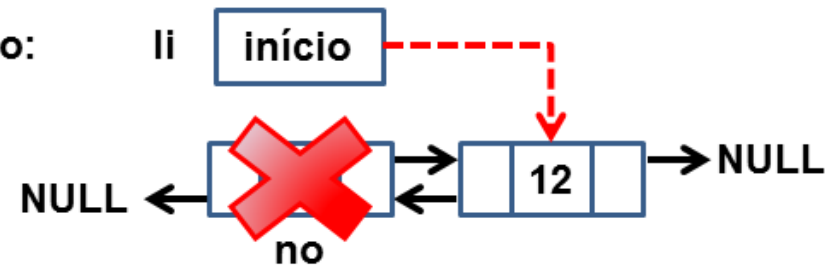
Lista Dinâmica Dupla | Remoção do início

Lista inicial

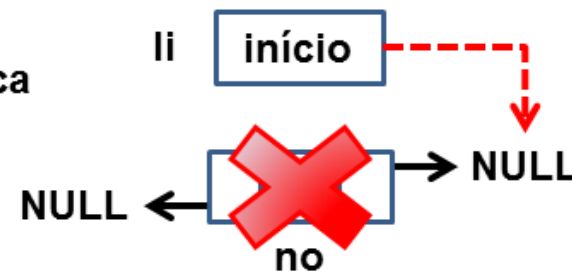


Se a lista possui mais de um elemento:
`no->prox->ant = NULL;`

Por fim:
`free(no);`



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



Lista Dinâmica Dupla | Remoção do final

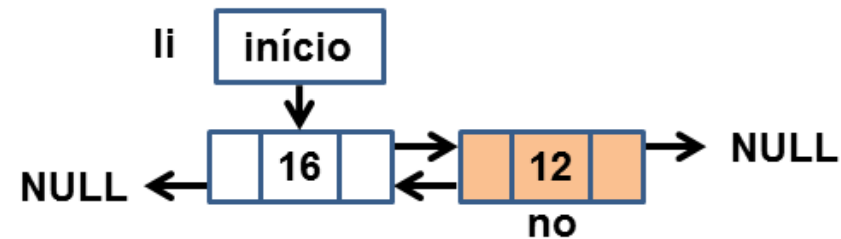
- Tarefa simples, mas trabalhosa
 - Envolve percorrer a lista toda e liberar a memória do elemento removido
- Primeiro, verificamos se
 - a lista existe
 - a lista possui elementos
- Em seguida
 - encontrar o final da lista
 - verificar se lista fica vazia
 - apontar penúltimo nó para NULL
 - liberar a memória do nó

```
int remove_lista_final(Lista* li){  
    if(li == NULL)  
        return 0;  
    if((*li) == NULL)//lista vazia  
        return 0;  
  
    Elem *no = *li;  
    while(no->prox != NULL)  
        no = no->prox;  
  
    if(no->ant == NULL)//remover o primeiro e único  
        *li = no->prox;  
    else  
        no->ant->prox = NULL;  
  
    free(no);  
    return 1;  
}
```

Lista Dinâmica Dupla | Remoção do final

Procura último elemento da lista:

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

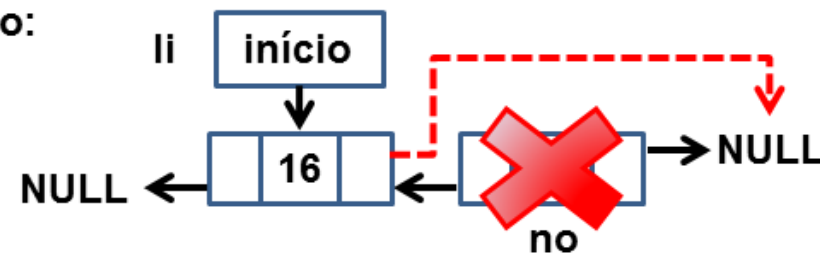


Se a lista possui mais de um elemento:

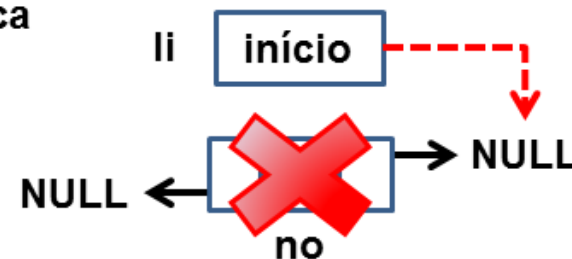
```
no->ant->prox = NULL;
```

Por fim:

```
free(no);
```



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



Lista Dinâmica Dupla | Remoção ordenada

- Envolve procurar o local de remoção
 - pode ser no início, meio ou final da lista
 - a lista pode ficar vazia
- Se a lista existe e possui elementos
 - procurar o elemento a ser removido
 - É o primeiro da lista?
 - ajustar o início
 - Não é o primeiro?
 - o anterior dele aponta para seu próximo
 - o próximo aponta para seu anterior
 - liberar a memória do nó

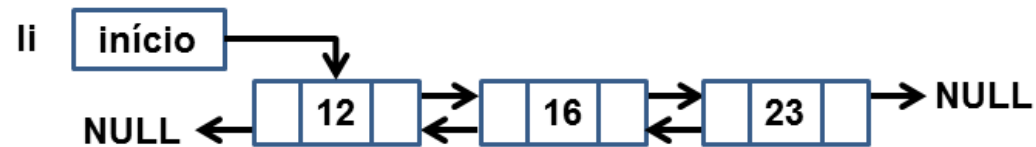
```
int remove_lista(Lista* li, int mat){  
    if(li == NULL)  
        return 0;  
    if((*li) == NULL) //lista vazia  
        return 0;  
    Elem *no = *li;  
    while(no != NULL && no->dados.matricula != mat)  
        no = no->prox;  
  
    if(no == NULL) //não encontrado  
        return 0;  
  
    if(no->ant == NULL) //remover o primeiro  
        *li = no->prox;  
    else  
        no->ant->prox = no->prox;  
  
    if(no->prox != NULL) //não é o último  
        no->prox->ant = no->ant;  
  
    free(no);  
    return 1;  
}
```


Lista Dinâmica Dupla | Remoção ordenada

Lista inicial

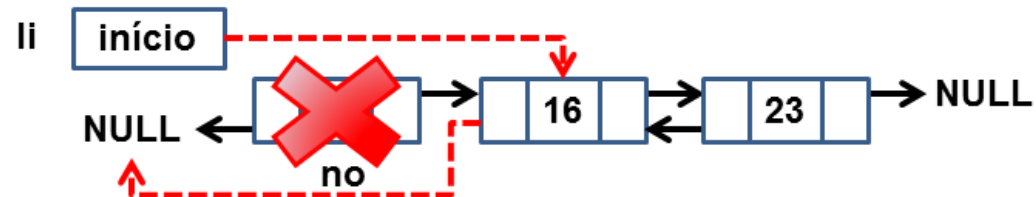
Busca qual remover:

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



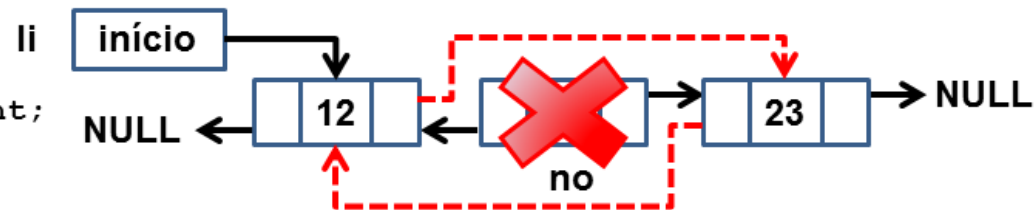
Remover do início:

*li = no->prox;



Não está removendo
do final:

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

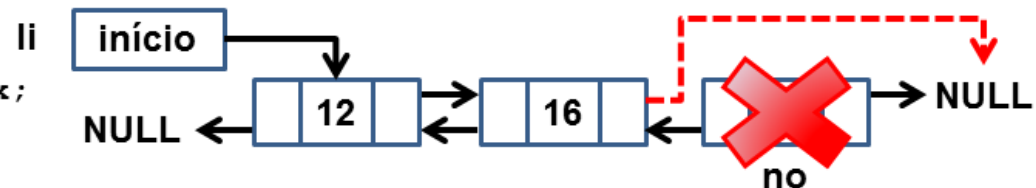


Remover do meio
ou do final:

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

Por fim:

free(no);



Lista Dinâmica Dupla | Acesso

- Podemos acessar qualquer elemento
- A busca pode ser por
 - posição
 - conteúdo

```
int busca_lista_pos(Lista* li, int pos, struct aluno *al){
    if(li == NULL || pos <= 0) return 0;

    Elem *no = *li;
    int i = 1;
    while(no != NULL && i < pos){
        no = no->prox;
        i++;
    }
    if(no == NULL)
        return 0;
    else{
        *al = no->dados;
        return 1;
    }
}

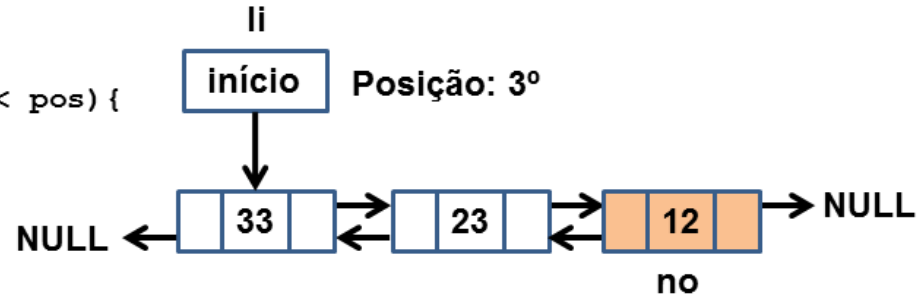
int busca_lista_mat(Lista* li, int mat, struct aluno *al){
    if(li == NULL) return 0;
    Elem *no = *li;
    while(no != NULL && no->dados.matricula != mat)
        no = no->prox;

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

Lista Dinâmica Dupla | Acesso

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

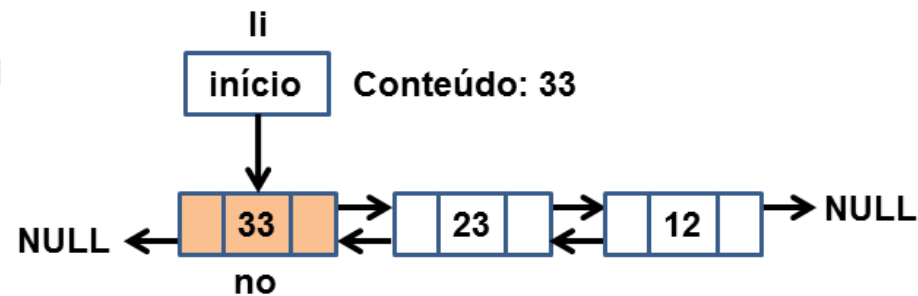
```
if(no == NULL) return 0;  
else{  
    *al = no->dados;  
    return 1;  
}
```

Busca 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;  
}
```



Lista Dinâmica Dupla | Complexidade

- Ao lado são mostradas as complexidades computacionais das principais operações na Lista Dinâmica Duplamente Encadeada

Operação	Início	Final	Ordenada
Inserção	$O(1)$	$O(N)$	$O(N)$
Remoção	$O(1)$	$O(N)$	$O(N)$
Busca	$O(N)$		

LISTA CIRCULAR

Lista Circular

- Similar a Lista Dinâmica Encadeada, é utilizada quando existe a necessidade de voltar ao primeiro elemento depois de percorrer a lista
 - Possibilidade de percorrer a lista diversas vezes



Lista Circular

- Vantagens

- Ciclicidade natural, útil em cenários que exigem repetição (acesso contínuo)
- Inserções e remoções eficientes (especialmente no início e fim), dependendo da implementação

- Desvantagens

- Pode ser mais complexa para implementar e gerenciar
- Acesso sequencial, como na lista simplesmente encadeada

- Aplicações

- Sistemas operacionais (gestão de processos em *round-robin*)
- buffers circulares

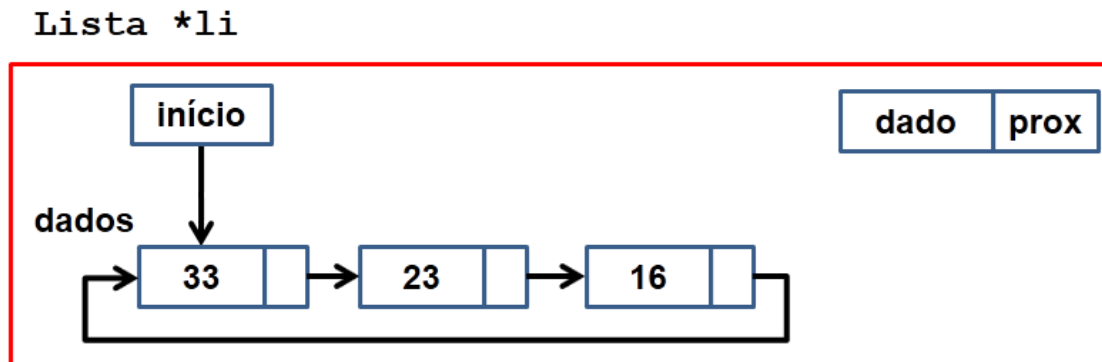
Lista Circular | TAD

- Segue a mesma implementação da Lista Dinâmica
 - Cada nó armazena os dados e um ponteiro para o próximo
 - Baseada num ponteiro para ponteiro
- A diferença é apenas na lógica de funcionamento
 - Na Lista Dinâmica não existe nenhum nó após o último, o qual aponta para NULL

```
//Definição do tipo lista
struct elemento{
    struct aluno dados;
    struct elemento *prox;
};
typedef struct elemento Elem;
typedef struct elemento* Lista;
```


Lista Circular | TAD

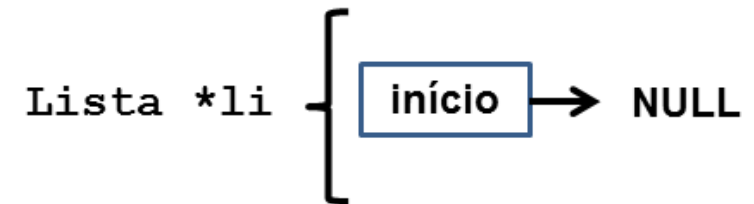
- Na Lista Dinâmica Circular, o último nó tem como sucessor o primeiro da lista
 - Na Lista Circular não existe posição final, não existe o NULL
 - Depois do último elemento voltamos para o primeiro, como em um círculo



Lista Circular | Criação

- Faz a alocação de uma área de memória para armazenar o endereço do início da lista
 - Equivale a criar uma lista vazia
 - Processo idêntico ao da Lista Dinâmica

```
Lista* cria_lista(){  
    Lista* li = (Lista*) malloc(sizeof(Lista));  
    if(li != NULL)  
        *li = NULL;  
    return li;  
}
```

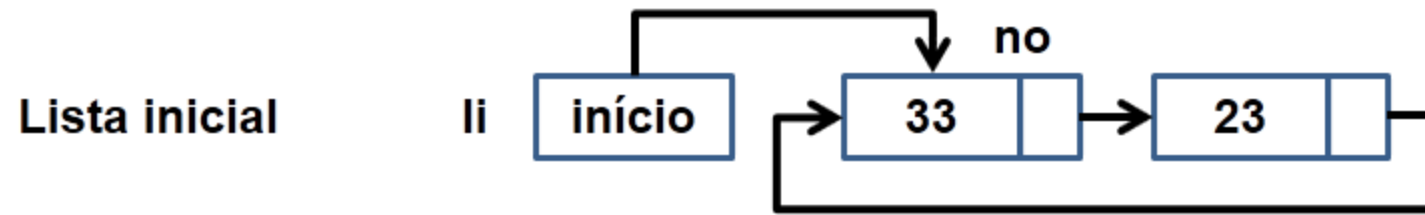


Lista Circular | Liberação

- Para liberar uma lista circular é preciso percorrer toda a lista liberando a memória alocada para cada elemento inserido
 - Diferente das outras listas, não temos o NULL
 - Processo termina ao encontra um nó que aponta para si mesmo
- Ao final, liberamos a memória da lista em si

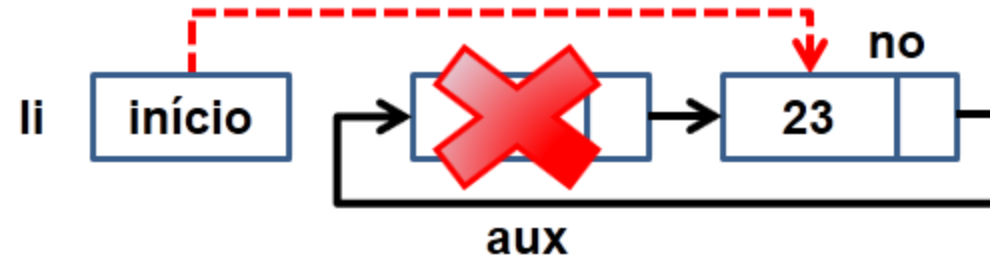
```
void libera_lista(Lista* li) {  
    if(li != NULL && (*li) != NULL) {  
        Elem *aux, *no = *li;  
        while((*li) != no->prox) {  
            aux = no;  
            no = no->prox;  
            free(aux);  
        }  
        free(no);  
        free(li);  
    }  
}
```

Lista Circular | Liberação



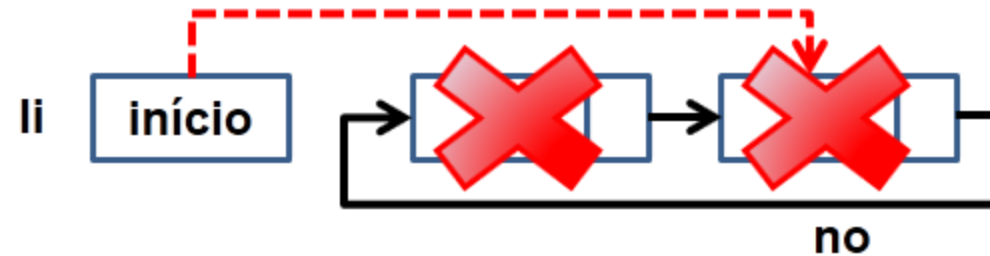
Passo 1:

```
aux = no;  
no = no->prox;  
free(aux);
```



Fim:

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



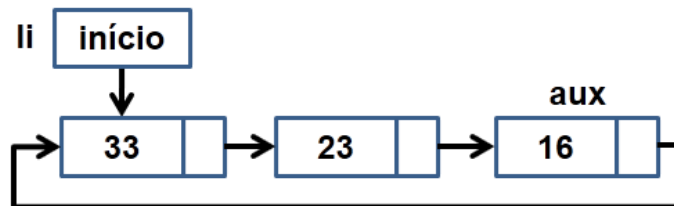
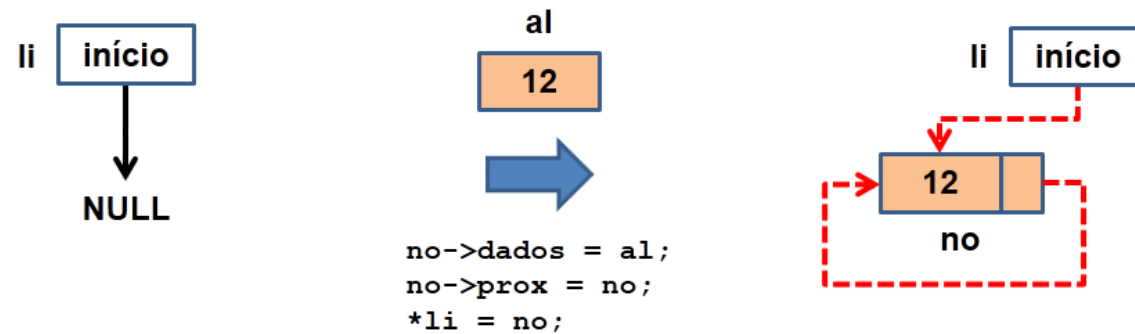
Lista Circular | Inserção no início

- Tarefa simples, mas trabalhosa
 - Envolve alocar espaço para o novo elemento e ajustar o início
- Basicamente, se a lista existe
 - alocar memória para o novo nó
 - copiar os dados
 - procurar o último elemento
 - colocar o novo como próximo do último
 - mudar o início

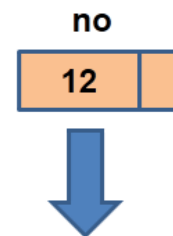
```
int insere_lista_inicio(Lista* li, struct aluno al){
    if(li == NULL) return 0;

    Elem *no = (Elem*) malloc(sizeof(Elem));
    if(no == NULL)
        return 0;
    no->dados = al;
    if((*li) == NULL){//lista vazia: insere início
        *li = no;
        no->prox = no;
    }else{
        Elem *aux = *li;
        while(aux->prox != (*li)){
            aux = aux->prox;
        }
        aux->prox = no;
        no->prox = *li;
        *li = no;
    }
    return 1;
}
```

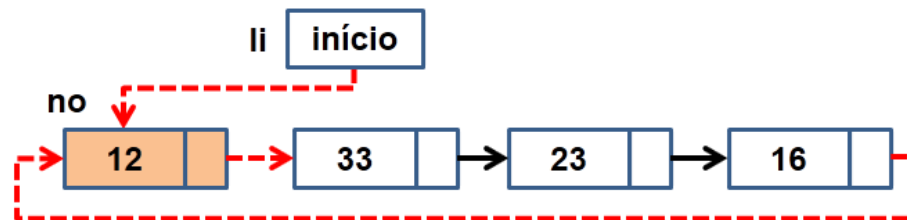
Lista Circular | Inserção no início



Busca último elemento "aux":
`aux = *li;`
`while(aux->prox != (*li)){`
 `aux = aux->prox;`
`}`



Inserir depois de "aux":
`aux->prox = no;`
`no->prox = *li;`
`*li = no;`

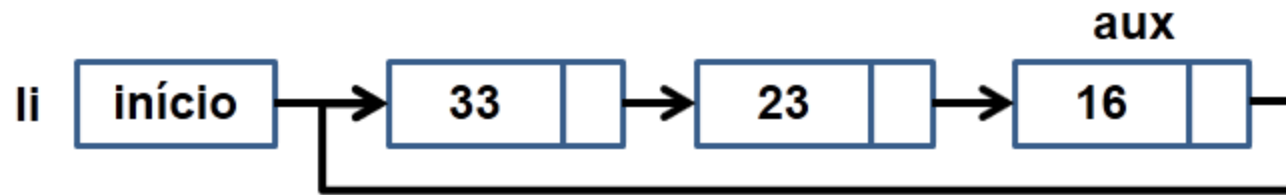


Lista Circular | Inserção no final

- Tarefa simples, mas trabalhosa
 - Envolve alocar espaço para o novo elemento e ajustar o início
- Basicamente, se a lista existe
 - alocar memória para o novo nó
 - copiar os dados
 - procurar o último elemento
 - ele aponta para o início da lista
 - apontar o último para o novo e o novo para o início

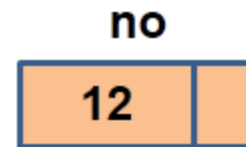
```
int insere_lista_final(Lista* li, struct aluno al){
    if(li == NULL) return 0;
    Elem *no = (Elem*) malloc(sizeof(Elem));
    if(no == NULL)
        return 0;
    no->dados = al;
    if((*li) == NULL){//lista vazia: insere início
        *li = no;
        no->prox = no;
    }else{
        Elem *aux = *li;
        while(aux->prox != (*li)){
            aux = aux->prox;
        }
        aux->prox = no;
        no->prox = *li;
    }
    return 1;
}
```

Lista Circular | Inserção no final



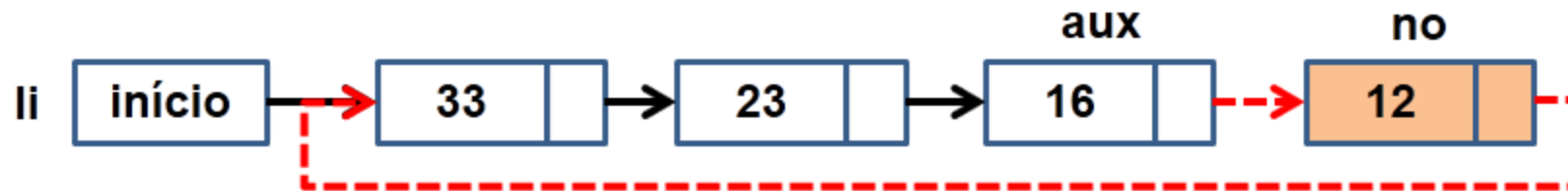
Busca último elemento "aux":

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



Inserir depois de "aux":

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



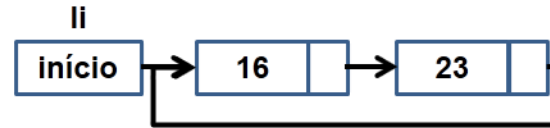
Lista Circular | Inserção ordenada

- Envolve procurar o local de inserção
 - pode ser no início, meio ou final da lista
 - também pode ser uma lista vazia
- Basicamente, se a lista existe
 - alocar memória para o novo nó
 - copiar os dados
 - Achou último elemento com matricula menor?
 - aponta-lo para o novo
 - Caso contrário
 - Sou maior que todos? Insere no final
 - Sou menor que todos? Insere no início

```
int insere_lista_ordenada(Lista* li, struct aluno al){
    if(li == NULL) return 0;
    Elem *no = (Elem*) malloc(sizeof(Elem));
    if(no == NULL) return 0;
    no->dados = al;
    if((*li) == NULL){//insere início
        *li = no;
        no->prox = no;
        return 1;
    }
    else{
        if((*li)->dados.matricula > al.matricula){//insere início
            Elem *atual = *li;
            while(atual->prox != (*li)){//procura o último
                atual = atual->prox;
            }
            no->prox = *li;
            atual->prox = no;
            *li = no;
            return 1;
        }
        Elem *ant = *li, *atual = (*li)->prox;
        while(atual != (*li) && atual->dados.matricula < al.matricula){
            ant = atual;
            atual = atual->prox;
        }
        ant->prox = no;
        no->prox = atual;
        return 1;
    }
}
```

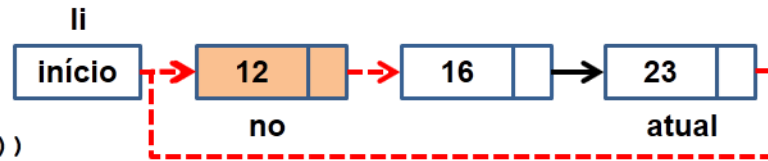
Lista Circular | Inserção ordenada

Lista inicial



Inserir no início

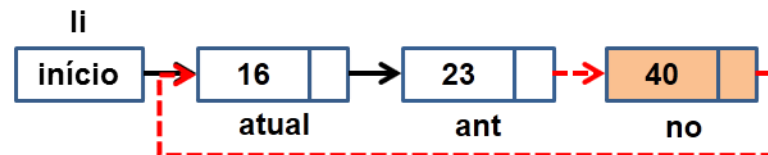
```
Elem *atual = *li;  
while(atual->prox != (*li))  
    atual = atual->prox;  
no->prox = *li;  
atual->prox = no;  
*li = no;
```



Inserir no meio ou no final

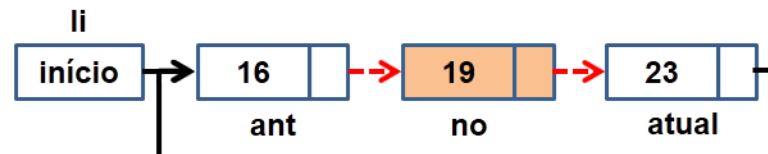
Busca onde inserir

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



Inserir depois de "ant":

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



Lista Circular | Remoção do início

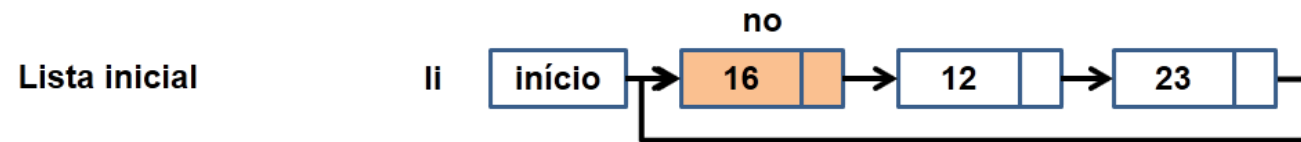
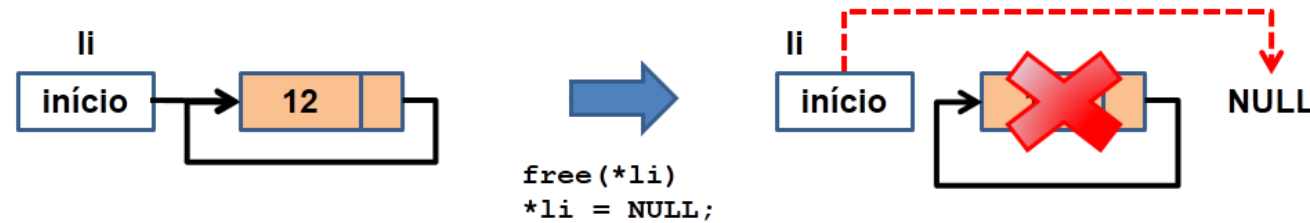
- Tarefa simples, mas trabalhosa
 - Envolve liberar elemento removido e ajustar o início
- Primeiro, verificamos se
 - a lista existe
 - a lista possui elementos
- Em seguida
 - verificar se é caso de lista ficar vazia
 - caso contrário, procurar o último elemento
 - apontar o próximo do último para o próximo do início
 - mudar o início

```
int remove_lista_inicio(Lista* li) {
    if(li == NULL)
        return 0;
    if((*li) == NULL) //lista vazia
        return 0;

    if((*li) == (*li)->prox) { //lista fica vazia
        free(*li);
        *li = NULL;
        return 1;
    }
    Elem *atual = *li;
    while(atual->prox != (*li)) //procura o último
        atual = atual->prox;

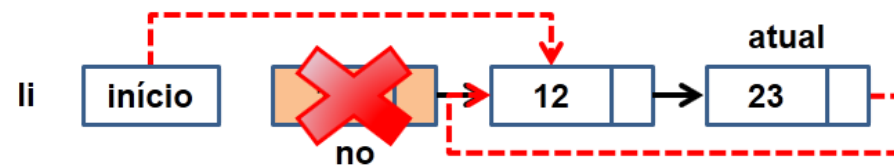
    Elem *no = *li;
    atual->prox = no->prox;
    *li = no->prox;
    free(no);
    return 1;
}
```

Lista Circular | Remoção do início



Busca último elemento: “atual”

```
atual = *li;  
while(atual->prox != (*li))  
    atual = atual->prox;
```

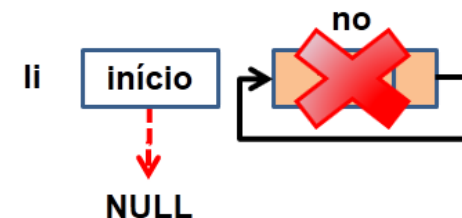


Remove o elemento

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

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

```
free(*li);  
*li = NULL;
```



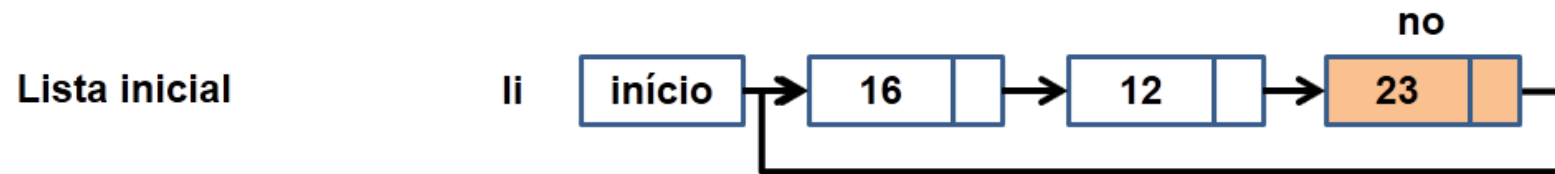
Lista Circular | Remoção do final

- Tarefa simples, mas trabalhosa
 - Envolve percorrer a lista toda e liberar a memória do elemento removido
- Primeiro, verificamos se
 - a lista existe
 - a lista possui elementos
- Em seguida
 - verificar se lista fica vazia
 - encontrar o final da lista
 - apontar penúltimo nó para o início
 - liberar a memória do nó

```
int remove_lista_final(Lista* li){
    if(li == NULL)
        return 0;
    if((*li) == NULL) //lista vazia
        return 0;

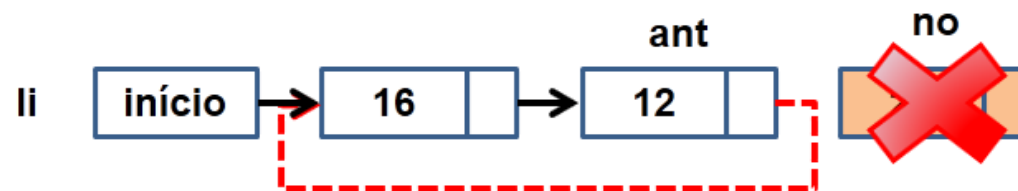
    if((*li) == (*li)->prox) { //lista fica vazia
        free(*li);
        *li = NULL;
        return 1;
    }
    Elem *ant, *no = *li;
    while(no->prox != (*li)) { //procura o último
        ant = no;
        no = no->prox;
    }
    ant->prox = no->prox;
    free(no);
    return 1;
}
```

Lista Circular | Remoção do final



Busca último elemento: "no"

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

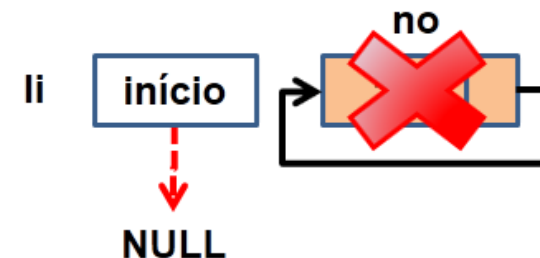


Remove o elemento

```
ant->prox = no->prox;  
free(no);
```

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

```
free(*li);  
*li = NULL;
```



Lista Circular | Remoção ordenada

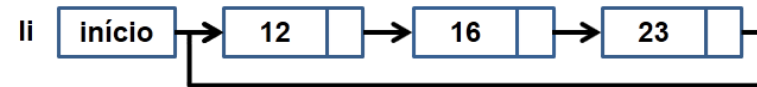
- Envolve procurar o local de remoção
 - pode ser no início, meio ou final da lista
 - a lista pode ficar vazia
- Se a lista existe e possui elementos
 - É o primeiro da lista?
 - ajustar o início, isso envolve achar o último
 - verificar se fica vazia
 - Não é o primeiro?
 - procurar o elemento a ser removido
 - o anterior dele aponta para seu próximo
 - liberar a memória do nó

```
int remove_lista(Lista* li, int mat){
    if(li == NULL) return 0;
    if((*li) == NULL)//lista vazia
        return 0;
    Elem *no = *li;
    if(no->dados.matricula == mat){//remover do início
        if(no == no->prox){//lista fica vazia
            free(no);
            *li = NULL;
            return 1;
        }else{
            Elem *ult = *li;
            while(ult->prox != (*li))//procura o último
                ult = ult->prox;
            ult->prox = (*li)->prox;
            *li = (*li)->prox;
            free(no);
            return 1;
        }
    }
    Elem *ant = no;
    no = no->prox;
    while(no != (*li) && no->dados.matricula != mat){
        ant = no;
        no = no->prox;
    }
    if(no == *li) return 0; //não encontrado

    ant->prox = no->prox;
    free(no);
    return 1;
}
```

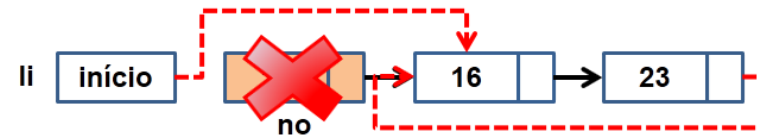
Lista Circular | Remoção ordenada

Lista Inicial



Remover do início:

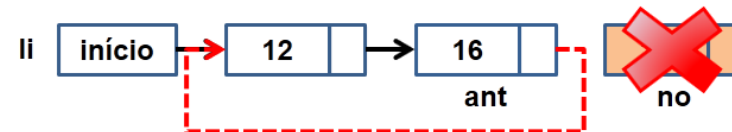
```
if(no == no->prox){ //lista fica vazia
    free(no);
    *li = NULL;
    return 1;
}else{
    Elem *ult = *li;
    while(ult->prox != (*li))
        ult = ult->prox;
    ult->prox = (*li)->prox;
    *li = (*li)->prox;
    free(no);
    return 1;
}
```



Remover do meio ou do final

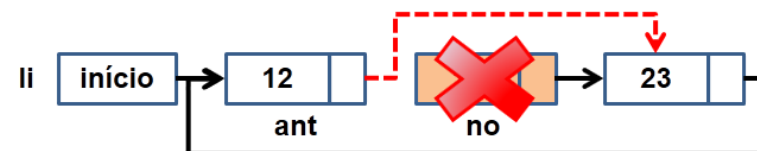
Buscar onde remover:

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



Remover depois de "ant":

```
ant->prox = no->prox;
free(no);
```



Lista Circular | Acesso

- Podemos acessar qualquer elemento
- A busca pode ser por
 - posição
 - conteúdo

```
int busca_lista_pos(Lista* li, int pos, struct aluno *al){
    if(li == NULL || (*li) == NULL || pos <= 0)
        return 0;
    Elem *no = *li;
    int i = 1;
    while(no->prox != (*li) && i < pos){
        no = no->prox;
        i++;
    }
    if(i != pos)
        return 0;
    else{
        *al = no->dados;
        return 1;
    }
}

int busca_lista_mat(Lista* li, int mat, struct aluno *al){
    if(li == NULL || (*li) == NULL) return 0;

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

    if(no->dados.matricula != mat)
        return 0;
    else{
        *al = no->dados;
        return 1;
    }
}
```

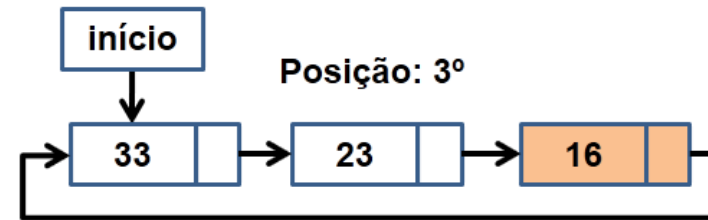
Lista Circular | Acesso

Busca pela posição do elemento

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

Verifica se a posição foi encontrada e a retorna

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

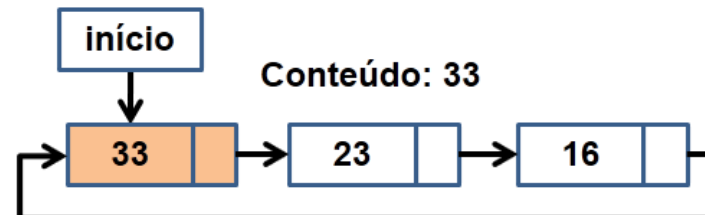


Busca pelo conteúdo do elemento

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

Verifica se o elemento foi encontrado e o retorna

```
if(no->dados.matricula != mat) {  
    return 0;  
} else {  
    *al = no->dados;  
    return 1;  
}
```



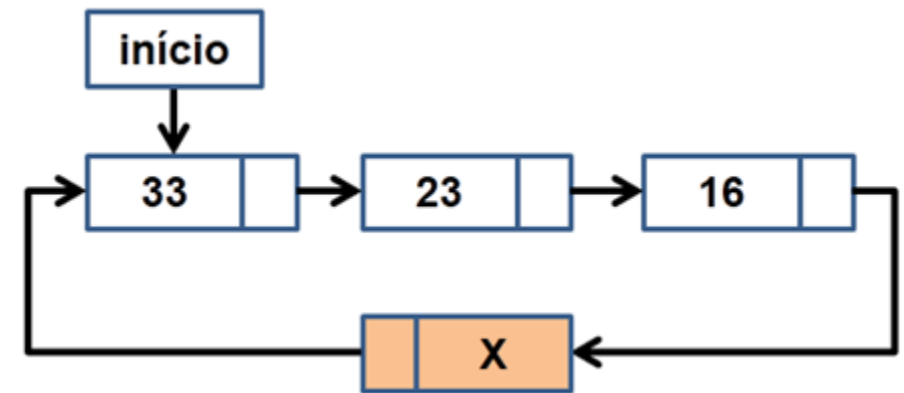
Lista Circular | Complexidade

- Ao lado são mostradas as complexidades computacionais das principais operações na Lista Dinâmica Encadeada Circular

Operação	Início	Final	Ordenada
Inserção	$O(N)$	$O(N)$	$O(N)$
Remoção	$O(N)$	$O(N)$	$O(N)$
Busca	$O(N)$		

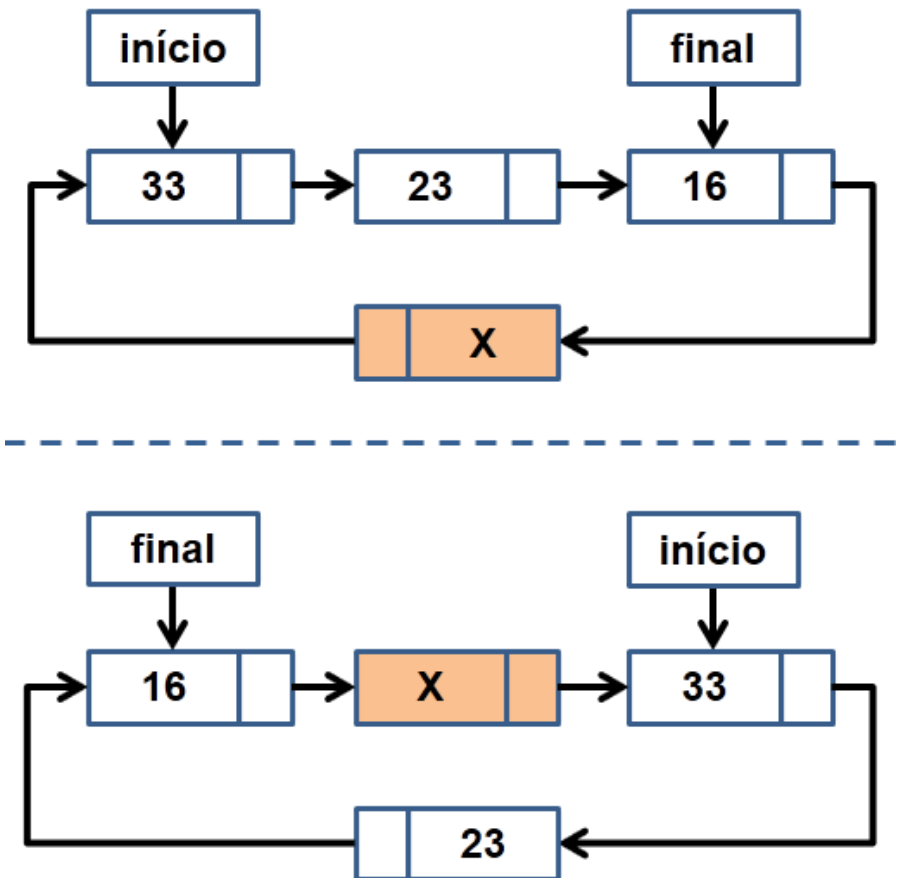
Lista Circular | Aumentando o desempenho

- Inserção e remoção no início ou final da lista são bastante trabalhosas
 - É preciso percorrer a lista toda para descobrir o último elemento, aquele que aponta para o primeiro da lista



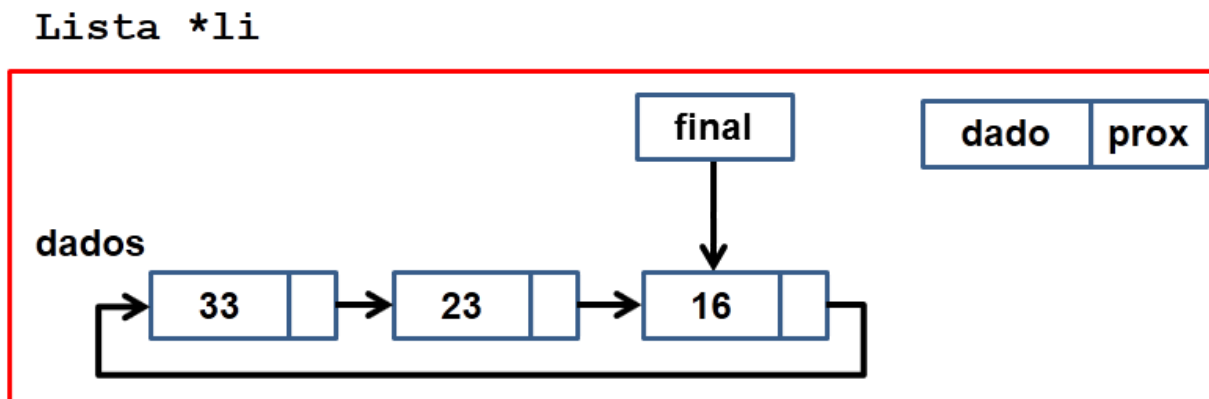
Lista Circular | Aumentando o desempenho

- No entanto, inserir no início ou no final equivale a colocar um novo elemento entre o último e o primeiro
 - Podemos tirar proveito disso e mudar a representação da nossa lista circular



Lista Circular | Aumentando o desempenho

- Ao invés de guarda a posição de início da lista, guardamos a posição final da lista
 - Isso não altera o armazenamento de elementos na lista
 - É apenas uma mudança na lógica de operação da lista
 - Simplifica algumas operações

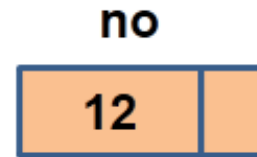
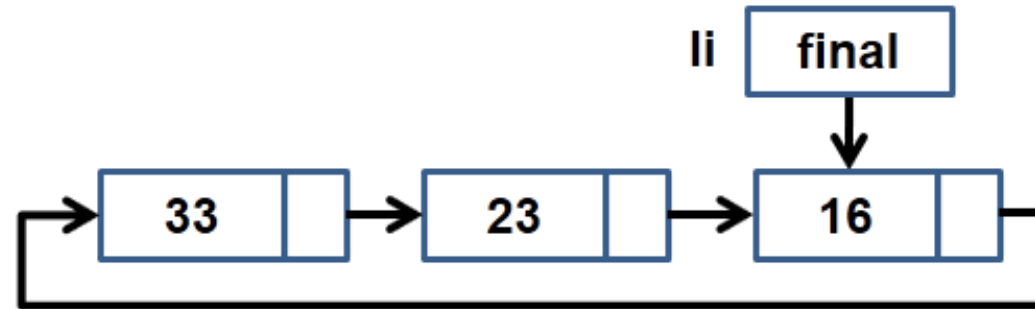


Lista Circular | Aumentando o desempenho

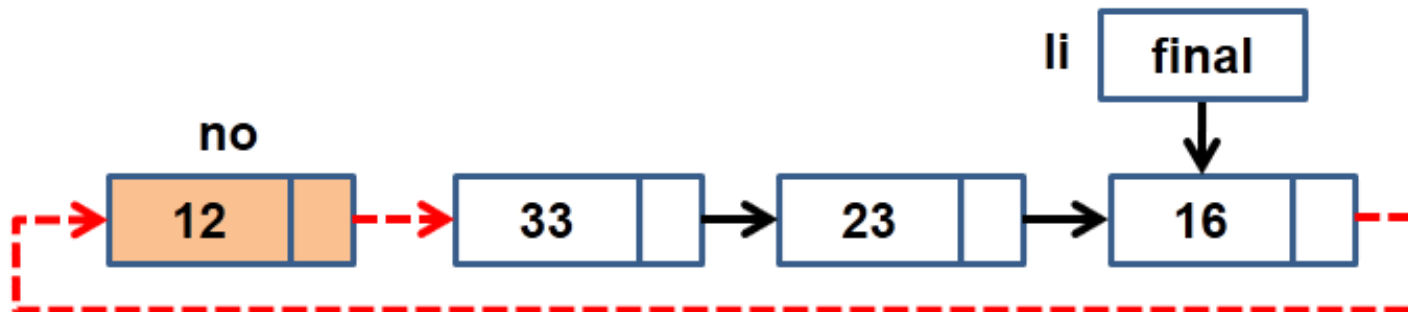
- Inserindo no início da lista
 - Não é mais necessário percorrer a lista
- Se a lista existe, precisamos
 - alocar memória para o novo nó
 - copiar os dados
 - tratar o caso de lista vazia
 - apontar o final para o novo e o novo para o início

```
int insere_lista_inicio(Lista* li, struct aluno al){  
    if(li == NULL)  
        return 0;  
    Elem *no = (Elem*) malloc(sizeof(Elem));  
    if(no == NULL)  
        return 0;  
    no->dados = al;  
    if((*li) == NULL){//lista vazia: insere início  
        *li = no;  
        no->prox = no;  
    }else{  
        no->prox = (*li)->prox;  
        (*li)->prox = no;  
    }  
    return 1;  
}
```

Lista Circular | Aumentando o desempenho



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

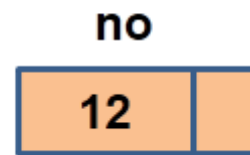
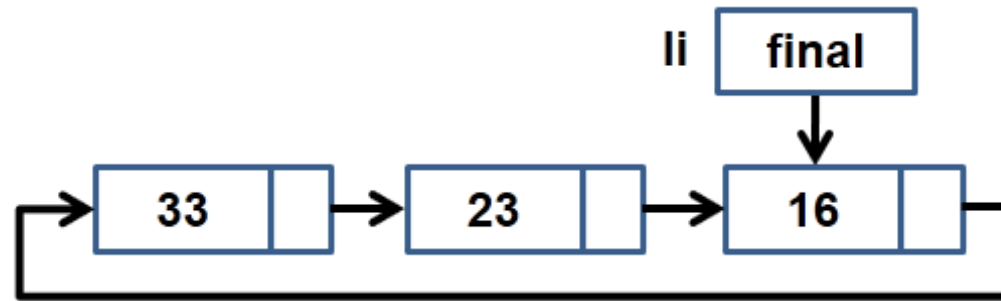


Lista Circular | Aumentando o desempenho

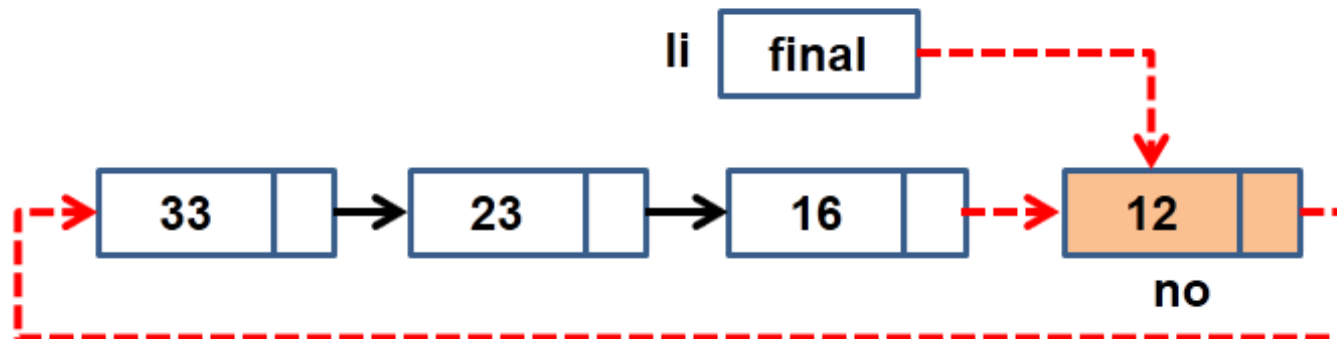
- Inserindo no final da lista
 - Não é mais necessário percorrer a lista
- Se a lista existe, precisamos
 - alocar memória para o novo nó
 - copiar os dados
 - tratar o caso de lista vazia
 - apontar o final para o novo e o novo para o início
 - mudar o final

```
int insere_lista_final(Lista* li, struct aluno al){  
    if(li == NULL) return 0;  
  
    Elem *no = (Elem*) malloc(sizeof(Elem));  
    if(no == NULL)  
        return 0;  
  
    no->dados = al;  
    if((*li) == NULL){//lista vazia: insere início  
        *li = no;  
        no->prox = no;  
    }else{  
        no->prox = (*li)->prox;  
        (*li)->prox = no;  
        *li = no;  
    }  
    return 1;  
}
```

Lista Circular | Aumentando o desempenho



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



Lista Circular | Aumentando o desempenho

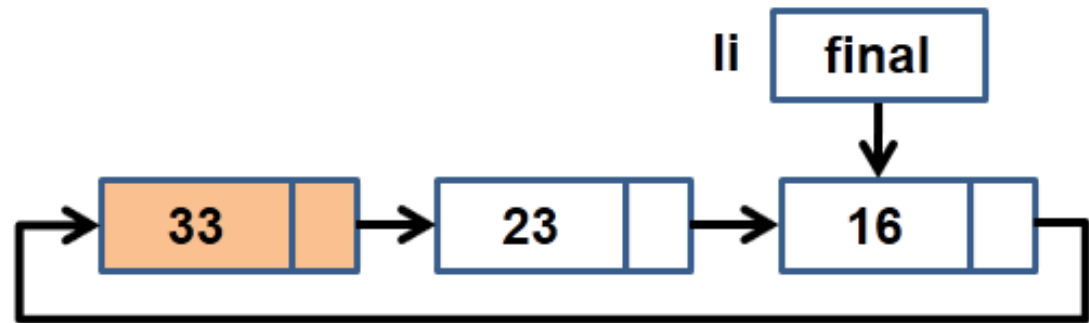
- Remoção do início da lista
 - Não é mais necessário percorrer a lista
- Se a lista existe e não está vazia, precisamos
 - Verificar se a lista fica vazia
 - apontar o final para o nó seguinte ao início

```
int remove_lista_inicio(Lista* li){
    if(li == NULL)
        return 0;

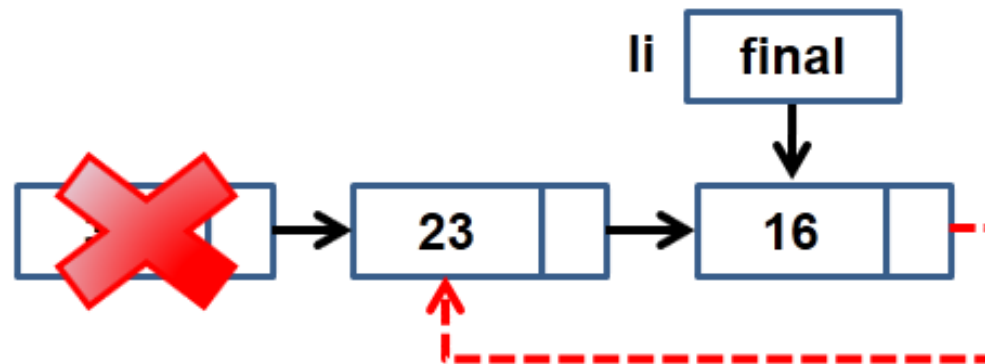
    if((*li) == NULL)//lista vazia
        return 0;

    if((*li) == (*li)->prox){//lista fica vazia
        free(*li);
        *li = NULL;
        return 1;
    }
    Elem *no = (*li)->prox;
    (*li)->prox = no->prox;
    free(no);
    return 1;
}
```

Lista Circular | Aumentando o desempenho



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



Lista Circular | Complexidade

- Ao lado são mostradas as complexidades computacionais das principais operações na Lista Dinâmica Encadeada Circular após a mudança

Operação	Início	Final	Ordenada
Inserção	$O(1)$	$O(1)$	$O(N)$
Remoção	$O(1)$	$O(N)$	$O(N)$
Busca	$O(N)$		

Material Complementar | Vídeo Aulas

- Aula 03 – Listas – Definição
 - <http://youtu.be/S6rOYN-UiAA>
- Aula 04 – Lista Estática Sequencial
 - <http://youtu.be/rxVrRdF0MTE>
- Aula 05 – Implementação da Listas Estáticas
 - <http://youtu.be/UCDCEjRDYrE>
- Aula 06 – Informações da Lista Estática
 - <http://youtu.be/zO8JAxb1GmA>
- Aula 07 – Inserção na Lista Estática
 - <http://www.youtube.com/watch?v=IpL31ZkVZSI>
- Aula 08 – Remoção na Lista Estática
 - http://www.youtube.com/watch?v=3KwG_OAB98g
- Aula 09 - Consulta na Lista Estática
 - <http://www.youtube.com/watch?v=xFN6Nefpx0k>

Material Complementar | Vídeo Aulas

- Aula 10 - Lista Dinâmica Encadeada
 - <http://www.youtube.com/watch?v=0BDMqra4D94>
- Aula 11 - Implementação da Lista Dinâmica Encadeada
 - <http://www.youtube.com/watch?v=wfC61zUVaos>
- Aula 12 - Informações da Lista Dinâmica Encadeada
 - <http://www.youtube.com/watch?v=WvmBhiQjPZ0>
- Aula 13 - Inserção na Lista Dinâmica Encadeada
 - <http://www.youtube.com/watch?v=fNP1GHLLKuY>
- Aula 14 - Remoção da Lista Dinâmica Encadeada
 - http://www.youtube.com/watch?v=67KZx_Rcfgw
- Aula 15 - Consulta na Lista Dinâmica Encadeada
 - <http://www.youtube.com/watch?v=rzPsfHZllek>

Material Complementar | Vídeo Aulas

- Aula 16 - Lista Dinâmica Duplamente Encadeada
 - http://www.youtube.com/watch?v=pWh_nJ66Rrk
- Aula 17 - Implementação da Lista Dinâmica Duplamente Encadeada
 - <http://www.youtube.com/watch?v=QU0TponoeZ0>
- Aula 18 - Informações da Lista Dinâmica Duplamente Encadeada
 - <http://www.youtube.com/watch?v=CokNvkDNB1k>
- Aula 19 - Inserção na Lista Dinâmica Duplamente Encadeada
 - <http://www.youtube.com/watch?v=cC-UWJssr30>
- Aula 20 - Remoção da Lista Dinâmica Duplamente Encadeada
 - <http://www.youtube.com/watch?v=30097hte7ys>
- Aula 21 – Consulta em Lista Duplamente Encadeada
 - <http://youtu.be/3iCG86067pQ>

Material Complementar | Vídeo Aulas

- Aula 22 – Lista Circular
 - <http://youtu.be/p8OxiV4FYK4>
- Aula 23 - Implementação da Lista Circular
 - <http://youtu.be/lhClGie5CEo>
- Aula 24 - Informações da Lista Circular
 - <http://youtu.be/iC9oH8ysoAU>
- Aula 25 - Inserção na Lista Circular
 - <http://youtu.be/iC9oH8ysoAU>
- Aula 26 - Remoção da Lista Circular
 - <http://youtu.be/A4Vz4Dcf9ww>
- Aula 27 - Consulta na Lista Circular
 - <http://youtu.be/bT2QSkNMecg>

Material Complementar | Vídeo Aulas

- Aula 28 - Lista com Nó Descritor
 - <http://youtu.be/2RHRjBcTy0A>
- Aula 29 - Manipulando uma Lista com Nó Descritor
 - <http://youtu.be/N901eJajCIM>
- Aula 30 - Remoção em uma Lista com Nó Descritor
 - <http://youtu.be/anwUiyXDB6o>
- Aula 130 - Criando uma estrutura de dados genérica em C com ponteiros genéricos
 - <https://youtu.be/oAXHhWAZAnU>

Material Complementar | GitHub

- <https://github.com/arbackes>

Popular repositories

Livro_Python

Public

☆ 118 🍴 12

Estrutura-de-Dados-em-C

Public

Códigos fontes de diversas estrutura de dados implementadas em linguagem C

● C ☆ 49 🍴 4

Estrutura-de-Dados-em-Python

Public

Códigos fontes de diversas estrutura de dados implementadas em linguagem Python

● Python ☆ 9 🍴 2

Codigos-Fontes-Diversos-em-C

Public

Códigos fontes de diferentes projetos implementados em linguagem C

● C ☆ 7 🍴 1