



# Listas



Prof. Bruno Travençolo

# Listas

---

- ▶ Estrutura de dados linear usada para armazenar e organizar dados
- ▶ Sequência de elementos do mesmo tipo
- ▶ Já foi visto alguma estrutura semelhante?



# Listas

---

- ▶ Qual a diferença da ED “Sequência de elementos do mesmo tipo” para um vetor?
- ▶ Temos que ver a lista como um TAD. Se é um TAD devemos definir operações sobre a lista
- ▶ Quais operações são esperadas em uma lista?



# Listas

---

- ▶ **Operações de manipulação da lista**
  - ▶ Inserir um elemento na lista
  - ▶ Remover um elemento da lista
  - ▶ Buscar um elemento da lista
  - ▶ Verificar o tamanho da lista
- ▶ **Operações relacionadas a estrutura da lista**
  - ▶ Criar a lista
  - ▶ Destruir a lista



# Exemplo: Lista de alunos

---

- ▶ Suponha que queremos guardar diferentes listas de alunos que participam de diferentes atividades na universidade
  - ▶ Lista de alunos que participam do grupo PET
  - ▶ Lista de alunos que participam de maratonas de programação
  - ▶ Lista de alunos da atlética
  - ▶ Lista de alunos de grupo de estudo em empreendedorismo

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



# Lista de alunos

---

- ▶ Devemos criar um TAD para armazenar essas listas
- ▶ Para criar o TAD vamos precisar definir os dados e as operações que serão suportadas pelo TAD

ListaSequencial.h

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

```
//Arquivo ListaSequencial.h
#define MAX 100
struct aluno{
    int matricula;
    char nome[30];
    float n1,n2,n3;
};

typedef struct lista Lista;
Lista* cria_lista();
void libera_lista(Lista* li);
int consulta_lista_pos(Lista* li, int pos, struct aluno *al);
int consulta_lista_mat(Lista* li, int mat, struct aluno *al);
int insere_lista_final(Lista* li, struct aluno al);
int insere_lista_inicio(Lista* li, struct aluno al);
int insere_lista_ordenada(Lista* li, struct aluno al);
int remove_lista(Lista* li, int mat);
int remove_lista_inicio(Lista* li);
int remove_lista_final(Lista* li);
int tamanho_lista(Lista* li);
int lista_cheia(Lista* li);
int lista_vazia(Lista* li);
void imprime_lista(Lista* li);
int remove_lista_otimizado(Lista* li, int mat);
```

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include "ListaSequencial.h" //inclui os Protótipos
4
5  //Definição do tipo lista
6  struct lista{
7      int qtd;
8      struct aluno dados[MAX];
9  };
10
```





```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include "ListaSequencial.h" //inclui os Protótipos
4
5  //Definição do tipo lista
6  struct lista{
7      int qtd;
8      struct aluno dados[MAX];
9  };
10
```

Variável quantidade (qtd) é quem diferencia a lista de um vetor. Com ela controlamos *em tempo de execução* quantos elementos nossa lista possui



## ListaSequencial.h

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

```
1  #include <stdio.h>  
2  #include <stdlib.h>  
3  #include "ListaSequencial.h" //inclui os Protótipos  
4  
5  //Definição do tipo lista  
6  struct lista{  
7      int qtd;  
8      struct aluno dados[MAX];  
9  };  
10
```

Nossa lista suporta elementos do tipo *struct aluno*. Não é genérica para outros tipos (assunto a ser abordado em outros cursos)

ListaSequencial.h

```
#define MAX 100
```

```
//Arquivo ListaSequencial.c
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include "ListaSequencial.h" //inlui os Protótipos
4
5  //Definição do tipo lista
6  struct lista{
7      int qtd;
8      struct aluno dados[MAX];
9  };
10
```

Nossa lista pode ter no máximo MAX elementos (ou seja MAX *struct aluno*)

## ListaSequencial.c

```
struct lista{  
    int qtd;  
    struct aluno dados[MAX];  
};
```

## ListaSequencial.h

```
struct aluno{  
    int matricula;  
    char nome[30];  
    float n1,n2,n3;  
};  
typedef struct lista Lista;  
Lista* cria_lista();
```

//Arquivo ListaSequencial.c

// cria\_lista - realiza a inicialização do TAD lista

// Parameters:

// Return value: endereço de memória da lista criada dinamicamente

10

11 Lista\* cria\_lista(){

12 Lista \*li;

13 li = (Lista\*) malloc(sizeof(struct lista));

14 if(li != NULL)

15 li->qtd = 0;

16 return li;

17 }

## ListaSequencial.c

```
struct lista{
    int qtd;
    struct aluno dados[MAX];
};
```

## ListaSequencial.h

```
struct aluno{
    int matricula;
    char nome[30];
    float n1,n2,n3;
};
typedef struct lista Lista;
Lista* cria_lista();
```

```
//Arquivo ListaSequencial.c
// cria_lista - realiza a inicialização do TAD lista
// Parameters:
// Return value: endereço de memória da lista criada dinamicamente
10
11 Lista* cria_lista(){
12     Lista *li;
13     li = (Lista*) ma
14     if(li != NULL)
15         li->qtd = 0;
16     return li;
17 }
```

**Parameters:** sem entrada

**Return value:** endereço de memória da lista criada dinamicamente. Em caso de erro retorna NULL

### ListaSequencial.c

```
struct lista{
    int qtd;
    struct aluno dados[MAX];
};
```

### ListaSequencial.h

```
struct aluno{
    int matricula;
    char nome[30];
    float n1,n2,n3;
};
typedef struct lista Lista;
Lista* cria_lista();
```

```
//Arquivo ListaSequencial.c
// cria_lista - realiza a inicialização do TAD lista
// Parameters:
// Return value: endereço de memória da lista criada dinamicamente
10
11 Lista* cria_lista(){
12     Lista *li;
13     li = (Lista*) ma
14     if(li != NULL)
15         li->qtd = 0;
16     return li;
17 }
```

**Cria uma variável local para armazenar um ponteiro para a lista**

## ListaSequencial.c

```
struct lista{
    int qtd;
    struct aluno dados[MAX];
};
```

## ListaSequencial.h

```
struct aluno{
    int matricula;
    char nome[30];
    float n1,n2,n3;
};
typedef struct lista Lista;
Lista* cria_lista();
```

```
//Arquivo ListaSequencial.c
// cria_lista - realiza a inicialização do TAD lista
// Parameters:
// Return value: endereço de memória da lista criada dinamicamente
10
11 Lista* cria_lista(){
12     Lista *li;
13     li = (Lista*) malloc(sizeof(struct lista));
14     if(li != NULL)
15         li->qtd = 0;
16     return li;
17 }
```

Aloca a estrutura Lista na memória para ser usada pelo TAD

### ListaSequencial.c

```
struct lista{  
    int qtd;  
    struct aluno dados[MAX];  
};
```

### ListaSequencial.h

```
struct aluno{  
    int matricula;  
    char nome[30];  
    float n1,n2,n3;  
};  
typedef struct lista Lista;  
Lista* cria_lista();
```

//Arquivo ListaSequencial.c

// cria\_lista - realiza a inicialização do TAD lista

// Parameters:

// Return value: endereço de memória da lista criada dinamicamente

10

11 Lista\* cria\_lista(){

12 Lista \*li;

13 li = (Lista\*) malloc(sizeof(struct lista));

14 if(li != NULL)

15 li->qtd = 0;

16 return li;

17 }

Quantos bytes são alocados na *heap*?



```
13      li = (Lista*) malloc(sizeof(struct lista));
```

ListaSequencial.h

```
#define MAX 100
```

Quantos bytes são alocados na *heap*?

ListaSequencial.c

```
struct lista{  
    int qtd;  
    struct aluno dados[MAX];  
};
```

qtd: 4 bytes  
dados: 100xsizeof(struct aluno)

ListaSequencial.h

```
struct aluno{  
    int matricula;  
    char nome[30];  
    float n1,n2,n3;  
};  
typedef struct lista Lista;  
Lista* cria_lista();
```

matricula: 4 bytes  
n1 , n2 , n3 : 12 bytes  
nome: 30 bytes  
Total: 46  
\*\* obs: assumindo que não há  
alinhamento. Com alinhamento o  
tamanho será 48 bytes

Total *malloc*: 4 + 100x46 = 4604 bytes

```
13      li = (Lista*) malloc(sizeof(struct lista));
```

ListaSequencial.c

```
struct lista{  
    int qtd;  
    struct aluno dados[MAX];  
};
```

Total malloc:  $4 + 100 \times 46 = 4604$  bytes

Qual o valor de 'li'?

Resposta: é o endereço retornado pelo malloc. Neste exemplo é 48. Se não houvesse alocação li receberia NULL

Endereço	Nome variável	Tipo	bytes	Conteúdo
48	li→qtd	int	4	lixo
52	li→dados[0]	struct aluno	46	lixo
98	li→dados[1]	struct aluno	46	lixo
144	li→dados[2]	struct aluno	46	lixo
190	li→dados[3]	struct aluno	46	lixo
236	li→dados[4]	struct aluno	46	lixo
....	....	....	....	....
4652	li→dados[99]	struct aluno	46	lixo

## ListaSequencial.c

```
struct lista{
    int qtd;
    struct aluno dados[MAX];
};
```

## ListaSequencial.h

```
struct aluno{
    int matricula;
    char nome[30];
    float n1,n2,n3;
};
typedef struct lista Lista;
Lista* cria_lista();
```

```
//Arquivo ListaSequencial.c
// cria_lista - realiza a inicialização do TAD lista
// Parameters:
// Return value: endereço de memória da lista criada dinamicamente
10
11 Lista* cria_lista(){
12     Lista *li;
13     li = (Lista*) malloc(sizeof(struct lista));
14     if(li != NULL)
15         li->qtd = 0;
16     return li;
17 }
```

Testa se o malloc funcionou. Caso ele não funcione ele retornará NULL

## ListaSequencial.c

```
struct lista{  
    int qtd;  
    struct aluno dados[MAX];  
};
```

## ListaSequencial.h

```
struct aluno{  
    int matricula;  
    char nome[30];  
    float n1,n2,n3;  
};  
typedef struct lista Lista;  
Lista* cria_lista();
```

```
//Arquivo ListaSequencial.c  
// cria_lista - realiza a inicialização do TAD lista  
// Parameters:  
// Return value: endereço de memória da lista criada dinamicamente  
10  
11 Lista* cria_lista(){  
12     Lista *li;  
13     li = (Lista*) malloc(sizeof(struct lista));  
14     if(li != NULL)  
15         li->qtd = 0;  
16     return li;  
17 }
```

Sabendo que a alocação funcionou, devemos iniciar o TAD indicando que a lista possui 0 elementos

## Inicializando a Lista

```
14     if(li != NULL)
15         li->qtd = 0;
16     return li;
17 }
```



Endereço	Nome variável	Tipo	bytes	Conteúdo
48	li→qtd	Int	4	0
52	li→dados[0]	struct aluno	46	lixo
98	li→dados[1]	struct aluno	46	lixo
144	li→dados[2]	struct aluno	46	lixo
190	li→dados[3]	struct aluno	46	lixo
236	li→dados[4]	struct aluno	46	lixo
....	....	....	....	....
4652	li→dados[99]	struct aluno	46	lixo

ListaSequencial.c

```
struct lista{
    int qtd;
    struct aluno dados[MAX];
};
```



## ListaSequencial.c

```
struct lista{
    int qtd;
    struct aluno dados[MAX];
};
```

## ListaSequencial.h

```
struct aluno{
    int matricula;
    char nome[30];
    float n1,n2,n3;
};
typedef struct lista Lista;
Lista* cria_lista();
```

```
//Arquivo ListaSequencial.c
// cria_lista - realiza a inicialização do TAD lista
// Parameters:
// Return value: endereço de memória da lista criada dinamicamente
10
11 Lista* cria_lista(){
12     Lista *li;
13     li = (Lista*) malloc(sizeof(struct lista));
14     if(li != NULL)
15         li->qtd = 0;
16     return li;
17 }
```

Retornamos o ponteiro para a Lista criada; ou NULL em caso de falha (vem do próprio *malloc*)

# No programa principal (que usa o TAD)

---

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

```
int main(){
    Lista* lista_alunos_facom;
    lista_alunos_facom = cria_lista();
```



# No programa principal (que usa o TAD)

---

```
#include <stdio.h>
#include <stdlib.h>
→ #include "ListaSequencial.h"
```

```
int main(){
    Lista* lista_alunos_facom = cria_lista();
    lista_alunos_facom = cria_lista();
```

Incluir o .h do TAD para ter acesso aos tipos de dados e operações do TAD



# No programa principal (que usa o TAD)

---

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

```
int main(){
    Lista* lista_alunos_facom;
    lista_alunos_facom = cria_lista();
```

Cria uma ponteiro para o TAD. Note que ainda não houve alocação de memória para o TAD em si



# No programa principal (que usa o TAD)

---

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

```
int main(){
    Lista* lista_alunos_facom;
    lista_alunos_facom = cria_lista();
```



Cria a lista, alocando a memória pré-definida e retorna o ponteiro (endereço) para a região alocada para o main.c



```
10
11  Lista* cria_lista(){
12      Lista *li;
13      li = (Lista*) malloc(sizeof(struct lista));
14      if(li != NULL)
15          li->qtd = 0;
16      return li;
17  }
18
19  void libera_lista(Lista* li){
20      free(li);
21  }
```



```
42
43  int insere_lista_final(Lista* li, struct aluno al){
44      if(li == NULL)
45          return -1;
46      if(li->qtd == MAX) //lista cheia
47          return -1;
48      li->dados[li->qtd] = al;
49      li->qtd++;
50      return 0;
51  }
```

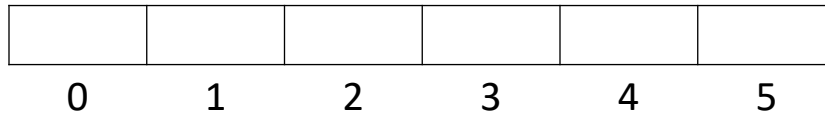


# Lista Sequencial

---

- ▶ Insere um elemento no final da lista
- ▶ Inicialmente vazia

MAX = 6



qtd = 0



# Lista Sequencial

---

- ▶ Insere um elemento no final da lista
- ▶ Insere “A” - `insere_lista_final(A)`
  - ▶ insere no final da lista

<b>A</b>					
0	1	2	3	4	5

qtd = 1



# Lista Sequencial

---

- ▶ Insere um elemento no final da lista
- ▶ Insere “A”
- ▶ Insere “B”

<b>A</b>	<b>B</b>				
0	1	2	3	4	5

qtd = 2



# Lista Sequencial

---

- ▶ Insere um elemento no final da lista
- ▶ Insere “A”
- ▶ Insere “B”
- ▶ Insere “C”

<b>A</b>	<b>B</b>	<b>C</b>			
0	1	2	3	4	5

qtd = 3





```
53  int insere_lista_inicio(Lista* li, struct aluno al){
54      if(li == NULL)
55          return -1;
56      if(li->qtd == MAX) //lista cheia
57          return -1;
58      int i;
59      for(i=li->qtd-1; i>=0; i--)
60          li->dados[i+1] = li->dados[i];
61      li->dados[0] = al;
62      li->qtd++;
63      return 0;
64  }
```



```
53  int insere_lista_inicio(Lista* li, struct aluno al){
54      if(li == NULL)
55          return -1;
56      if(li->qtd == MAX) //lista cheia
57          return -1;
58      int i;
59      for(i=li->qtd-1; i>=0; i--)
60          li->dados[i+1] = li->dados[i];
61      li->dados[0] = al;
62      li->qtd++;
63      return 0;
64  }
```



# Lista Sequencial

---

- ▶ **int** insere\_lista\_inicio(Lista\* li, **struct** aluno al){
- ▶ Insere “D”

<b>A</b>	<b>B</b>	<b>C</b>			
0	1	2	3	4	5

qtd = 3



# Lista Sequencial

---

- ▶ **int** insere\_lista\_inicio(Lista\* li, **struct** aluno al) {
- ▶ Insere “D”

i = 2

<b>A</b>	<b>B</b>	<b>C</b>			
0	1	2	3	4	5

qtd = 3

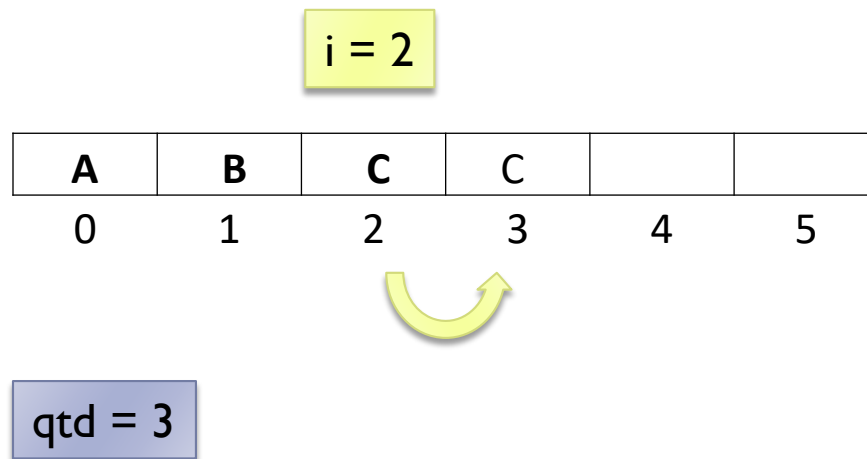
```
59     for(i=li->qtd-1; i>=0; i--)  
60         li->dados[i+1] = li->dados[i];
```



# Lista Sequencial

---

- ▶ `int` insere\_lista\_inicio(`Lista*` li, `struct` aluno al) {
- ▶ Inserir “D”

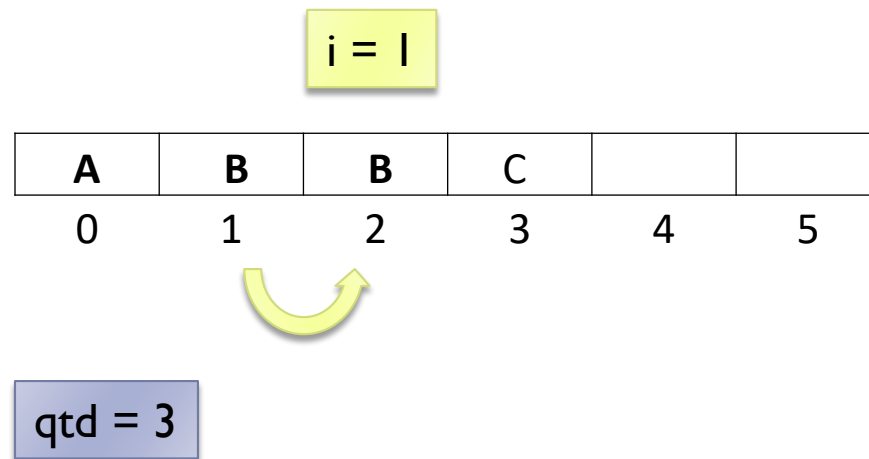


```
59     for(i=li->qtd-1; i>=0; i--)  
60         li->dados[i+1] = li->dados[i];
```

# Lista Sequencial

---

- ▶ **int** insere\_lista\_inicio(Lista\* li, **struct** aluno al) {
- ▶ Insere “D”

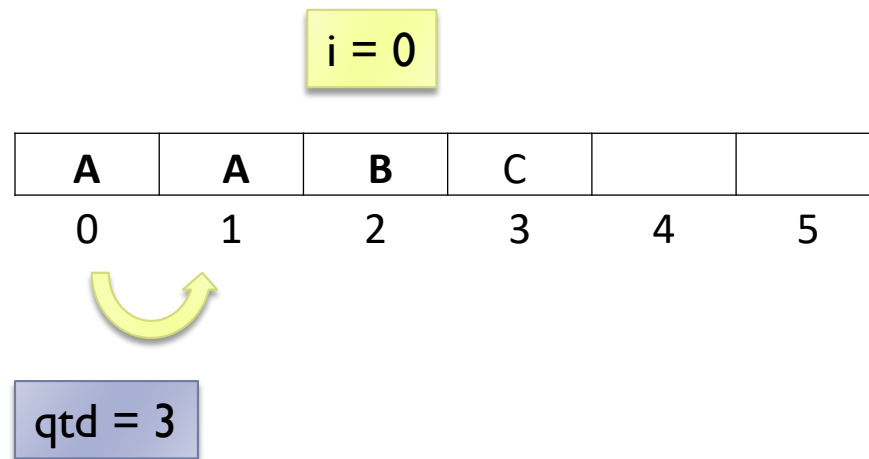


```
59     for(i=li->qtd-1; i>=0; i--)  
60         li->dados[i+1] = li->dados[i];
```

# Lista Sequencial

---

- ▶ **int** insere\_lista\_inicio(Lista\* li, **struct** aluno al) {
- ▶ Insere “D”



```
59     for(i=li->qtd-1; i>=0; i--)  
60         li->dados[i+1] = li->dados[i];
```

```
53  int insere_lista_inicio(Lista* li, struct aluno al){
54      if(li == NULL)
55          return -1;
56      if(li->qtd == MAX) //lista cheia
57          return -1;
58      int i;
59      for(i=li->qtd-1; i>=0; i--)
60          li->dados[i+1] = li->dados[i];
61      li->dados[0] = al;
62      li->qtd++;
63      return 0;
64  }
```





# Lista Sequencial

---

- ▶ `int` insere\_lista\_inicio(`Lista*` li,  
`struct` aluno al) {
- ▶ Insere “D”

Insere D na primeira posição e aumenta qtd

<b>D</b>	<b>A</b>	<b>B</b>	<b>C</b>		
0	1	2	3	4	5

qtd = 4

```
61      li->dados[0] = al;  
62      li->qtd++;
```



```
22
23  int consulta_lista_pos(Lista* li, int pos, struct aluno *al){
24      if(li == NULL || pos <= 0 || pos > li->qtd)
25          return -1;
26      *al = li->dados[pos-1];
27      return 0;
28  }
29
```



```
22
23  int consulta_lista_pos(Lista* li, int pos, struct aluno *al){
24      if(li == NULL || pos <= 0 || pos > li->qtd)
25          return -1;
26      *al = li->dados[pos-1];
27      return 0;
28  }
29
```



# Lista Sequencial

---

- ▶ `consulta_lista_pos(Lista* li, int pos, struct aluno *al)`
- ▶ A lista inicia em 1 (um), mas o vetor em zero
- ▶ Exemplo:
  - ▶ Consulta o 2º elemento da lista

pos = 2

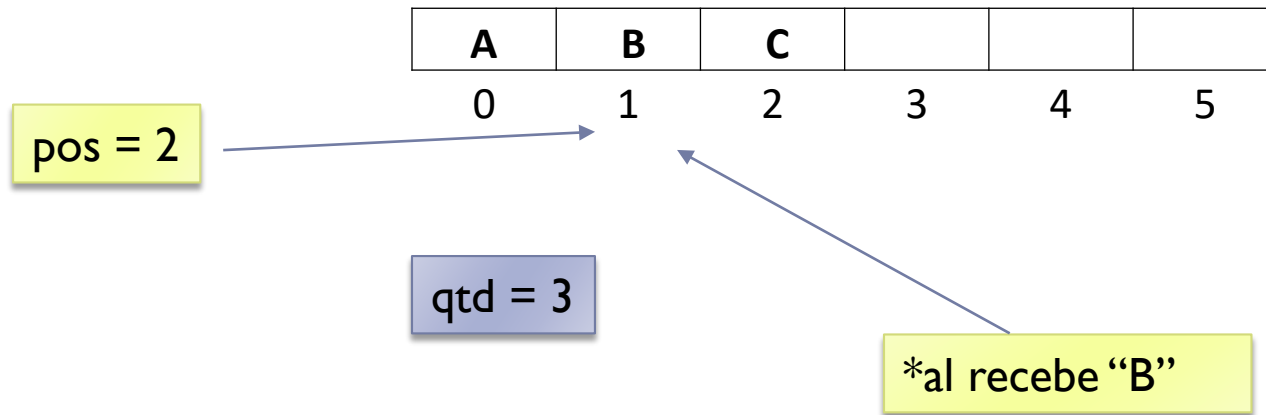
A	B	C			
0	1	2	3	4	5

qtd = 3



# Lista Sequencial

- ▶ `consulta_lista_pos(Lista* li, int pos, struct aluno *al)`
- ▶ A lista inicia em 1 (um), mas o vetor em zero
- ▶ Exemplo:
  - ▶ Consulta o 2º elemento da lista



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

# Lista Sequencial

---

- ▶ `consulta_lista_pos(Lista* li, int pos, struct aluno *al)`
- ▶ Cuidado com o parâmetro `struct aluno *al`
- ▶ O aluno já deve estar alocado no programa principal
  - ▶ Exemplo de chamada no programa principal

```
struct aluno aluno; // aloca variável no main()  
consulta_lista_pos(li, 2, &aluno);
```



Passagem por referência de aluno

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

# Lista Sequencial

---

- ▶ `consulta_lista_pos(Lista* li, int pos, struct aluno *al)`
- ▶ Cuidado com o parâmetro `struct aluno *al`
- ▶ O aluno já deve estar alocado no programa principal
  - ▶ Exemplo de chamada no programa principal

```
struct aluno aluno; // aloca variável no main()  
consulta_lista_pos(li, 2, &aluno);
```

**\*al** referencia a variável “**aluno**” que chamou a função consulta

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

```
30  int consulta_lista_mat(Lista* li, int mat, struct aluno *al){
31      if(li == NULL)
32          return -1;
33      int i = 0;
34      while(i < li->qtd && li->dados[i].matricula != mat)
35          i++;
36      if(i == li->qtd) //elemento nao encontrado
37          return -1;
38
39      *al = li->dados[i];
40      return 0;
41  }
```





```
30  int consulta_lista_mat(Lista* li, int mat, struct aluno *al){
31      if(li == NULL)
32          return -1;
33      int i = 0;
34      while(i < li->qtd && li->dados[i].matricula != mat)
35          i++;
36      if(i == li->qtd) //elemento nao encontrado
37          return -1;
38
39      *al = li->dados[i];
40      return 0;
41  }
```



- ▶ O loop faz a busca pelo número de matrícula desejado
- ▶ Exemplo:
  - ▶ Buscar aluno “B” (supor que a busca foi por número de matrícula)

```
33  int i = 0;
```

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

```
35      i++;
```

i = 0

A	B	C			
0	1	2	3	4	5

qtd = 3

li->dados[0]  
A != B ? True  
Vai pro próximo

- ▶ O loop faz a busca pelo número de matrícula desejado
- ▶ Exemplo:
  - ▶ Buscar aluno “B” (supor que a busca foi por número de matrícula)

```
33  int i = 0;
```

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

```
35      i++;
```

i = 1

A	B	C			
0	1	2	3	4	5

qtd = 3

li->dados[1]

B != B ? False

Achou o elemento, sai do loop

```
30  int consulta_lista_mat(Lista* li, int mat, struct aluno *al){
31      if(li == NULL)
32          return -1;
33      int i = 0;
34      while(i < li->qtd && li->dados[i].matricula != mat)
35          i++;
36      if(i == li->qtd) //elemento nao encontrado
37          return -1;
38
39      *al = li->dados[i];
40      return 0;
41  }
```



---

**i = 1 ; qtd = 3**

```
36     if(i == li->qtd)//elemento nao encontrado
37         return -1;
38
39     *al = li->dados[i];
```

**i = 1**

<b>A</b>	<b>B</b>	<b>C</b>			
0	1	2	3	4	5

**qtd = 3**



```

30  int consulta_lista_mat(Lista* li, int mat, struct aluno *al){
31      if(li == NULL)
32          return -1;
33      int i = 0;
34      while(i < li->qtd && li->dados[i].matricula != mat)
35          i++;
36      if(i == li->qtd) //elemento nao encontrado
37          return -1;
38
39      *al = li->dados[i];
40      return 0;
41  }

```

Passagem por referência de aluno

**al** é uma passagem por referência de uma variável já alocada

```

struct aluno aluno;
consulta_lista_mat(li, "B", &aluno)

```

```
65
66  int insere_lista_ordenada(Lista* li, struct aluno al){
67      if(li == NULL)
68          return -1;
69      if(li->qtd == MAX) //lista cheia
70          return -1;
71      int k, i = 0;
72      while(i < li->qtd && li->dados[i].matricula < al.matricula)
73          i++;
74
75      for(k = li->qtd - 1; k >= i; k--)
76          li->dados[k + 1] = li->dados[k];
77
78      li->dados[i] = al;
79      li->qtd++;
80      return 0;
81  }
```



# Lista Sequencial

---

- ▶ Remover um elemento
- ▶ `int remove_lista_inicio(Lista* li) {`
- ▶ Como remover do início?

<b>A</b>	<b>B</b>	<b>C</b>			
0	1	2	3	4	5

qtd = 3

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





# Lista Sequencial

---

- ▶ Remover um elemento
- ▶ `int remove_lista_inicio(Lista* li) {`
- ▶ Como remover do início?

<b>A</b>	<b>B</b>	<b>C</b>			
0	1	2	3	4	5

qtd = 3

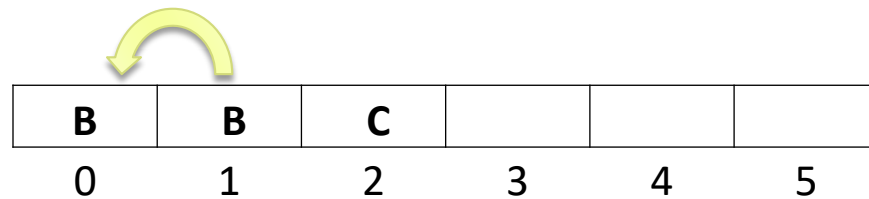
```
122     for(k=0; k< li->qtd-1; k++)
123         li->dados[k] = li->dados[k+1];
124     li->qtd--;
```

k = 0

# Lista Sequencial

---

- ▶ Remover um elemento
- ▶ `int remove_lista_inicio(Lista* li) {`
- ▶ Como remover do início?



qtd = 3

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

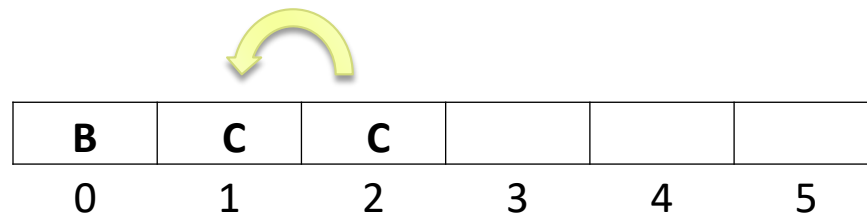
0

1

k = 0

# Lista Sequencial

- ▶ Remover um elemento
- ▶ `int remove_lista_inicio(Lista* li) {`
- ▶ Como remover do início?



qtd = 3

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

1

2

k = 1

# Lista Sequencial

---

- ▶ Remover um elemento
- ▶ `int remove_lista_inicio(Lista* li) {`
- ▶ Como remover do início?

<b>B</b>	<b>C</b>	<b>C</b>			
0	1	2	3	4	5

qtd = 2

2

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

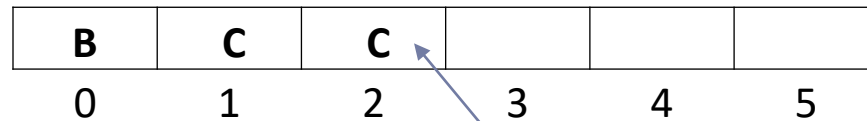
k = 2

qtd = 2

# Lista Sequencial

---

- ▶ Remover um elemento
- ▶ `int remove_lista_inicio(Lista* li) {`
- ▶ Como remover do início?



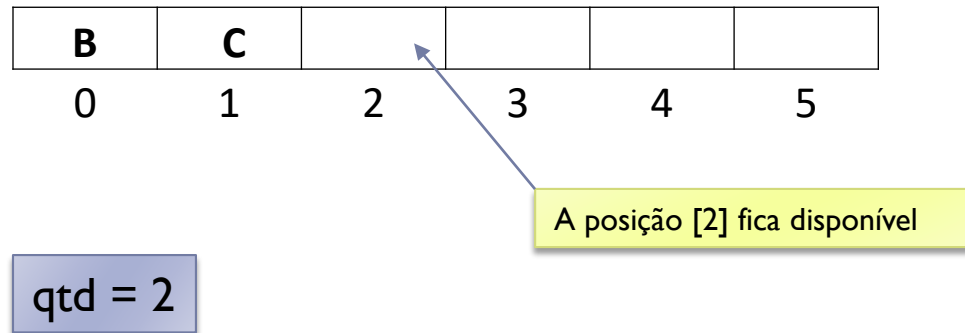
qtd = 2

O conteúdo não é apagado, somente é feito o deslocamento. A posição [2] do vetor nunca será lida pois qtd = 2

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

# Lista Sequencial

- ▶ Remover um elemento
- ▶ `int remove_lista_inicio(Lista* li) {`
- ▶ Como remover do início?



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

# Lista Sequencial

- ▶ Remover um elemento
- ▶ `int remove_lista_inicio(Lista* li) {`
- ▶ Como remover do início?

<b>B</b>	<b>C</b>	<b>C</b>			
0	1	2	3	4	5

qtd = 2

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

Custo computacional alto para uma operação simples (um remoção gera o deslocamento de todo o vetor)

```
127
128  int remove_lista_final(Lista* li){
129      if(li == NULL)
130          return -1;
131      if(li->qtd == 0)
132          return -1;
133      li->qtd--;
134      return 0;
135  }
```





```
127
128  int remove_lista_final(Lista* li){
129      if(li == NULL)
130          return -1;
131      if(li->qtd == 0)
132          return -1;
133      li->qtd--;
134      return 0;
135  }
```



# Lista Sequencial

---

- ▶ Insere um elemento na fila
- ▶ Insere “A”
- ▶ Insere “B”
- ▶ Insere “C”

<b>A</b>	<b>B</b>	<b>C</b>			
0	1	2	3	4	5

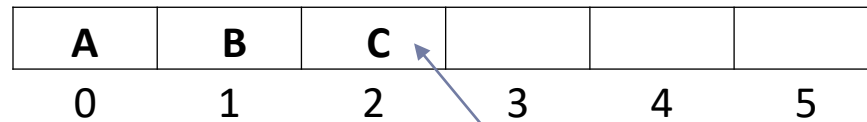
qtd = 3



# Lista Sequencial

---

► **int** remove\_lista\_final(Lista\* li){



qtd = 2

O conteúdo não é apagado, somente é diminuído o valor de qte. Após isso, a posição [2] do vetor nunca será lida pois qtd = 2

li->qtd--;

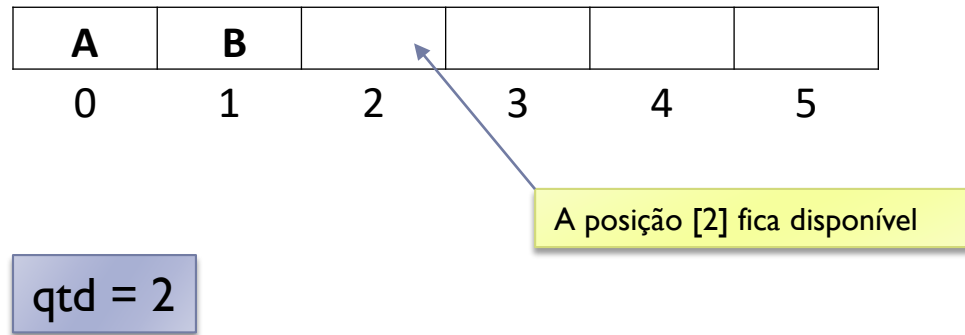
---



# Lista Sequencial

---

► **int** remove\_lista\_final(Lista\* li){



li->qtd--;

---

```
82
83  int remove_lista(Lista* li, int mat){
84      if(li == NULL)
85          return -1;
86      if(li->qtd == 0)
87          return -1;
88      int k,i = 0;
89      while(i<li->qtd && li->dados[i].matricula != mat)
90          i++;
91      if(i == li->qtd)//elemento nao encontrado
92          return 0;
93
94      for(k=i; k< li->qtd-1; k++)
95          li->dados[k] = li->dados[k+1];
96      li->qtd--;
97      return 0;
98  }
```



```
100  int remove_lista_otimizado(Lista* li, int mat){
101      if(li == NULL)
102          return -1;
103      if(li->qtd == 0)
104          return -1;
105      int i = 0;
106      while(i < li->qtd && li->dados[i].matricula != mat)
107          i++;
108      if(i == li->qtd) //elemento nao encontrado
109          return 0;
110
111      li->qtd--;
112      li->dados[i] = li->dados[li->qtd];
113      return 0;
114  }
```



```
115
116  int remove_lista_inicio(Lista* li){
117      if(li == NULL)
118          return -1;
119      if(li->qtd == 0)
120          return -1;
121      int k = 0;
122      for(k=0; k< li->qtd-1; k++)
123          li->dados[k] = li->dados[k+1];
124      li->qtd--;
125      return 0;
126  }
```



```
136
137  int tamanho_lista(Lista* li){
138      if(li == NULL)
139          return -1;
140      else
141          return li->qtd;
142  }
143
144  int lista_cheia(Lista* li){
145      if(li == NULL)
146          return -1;
147      return (li->qtd == MAX);
148  }
149
150  int lista_vazia(Lista* li){
151      if(li == NULL)
152          return -1;
153      return (li->qtd == 0);
154  }
```

---





```
155
156  int imprime_lista(Lista* li){
157      if(li == NULL)
158          return -1;
159      int i;
160      for(i=0; i< li->qtd; i++){
161          printf("Matricula: %d\n",li->dados[i].matricula);
162          printf("Nome: %s\n",li->dados[i].nome);
163          printf("Notas: %f %f %f\n",li->dados[i].n1,
164                                     li->dados[i].n2,
165                                     li->dados[i].n3);
166          printf("-----\n");
167      } return 0;
168  }
```



```
#include <stdio.h>
#include <stdlib.h>
#include "ListaSequencial.h"
int main(){
    struct aluno a[4] = {{2, "Andre", 9.5, 7.8, 8.5},
                          {4, "Ricardo", 7.5, 8.7, 6.8},
                          {1, "Bianca", 9.7, 6.7, 8.4},
                          {3, "Ana", 5.7, 6.1, 7.4}};

    Lista* li = cria_lista();
    int i;
    for(i=0; i < 4; i++)
        insere_lista_ordenada(li, a[i]);

    imprime_lista(li);
    printf("\n\n\n\n");

    for(i=0; i < 5; i++){
        if (remove_lista_otimizado(li, i)==-1)
            printf("Erro\n");

        imprime_lista(li);
        printf("\n\n\n\n");
    }

    libera_lista(li);
    system("pause");
    return 0;
}
```

---



# Lista Dinâmica Encadeada

---

- ▶ Lista que utiliza alocação dinâmica (individual) para cada elemento
  - ▶ Ex: para cada aluno inserido, um malloc é feito.
- ▶ Se cada elemento é alocado dinamicamente, cada um retornará um endereço de memória indicando seu local de alocação
- ▶ Quantos endereços teremos?
  - ▶ Um para cada elemento
  - ▶ Lista com 1000 elementos, 1000 endereços
- ▶ Como lidar com esses endereços? Onde armazená-los?



# Lista Dinâmica Encadeada

---

- ▶ **Vetor de ponteiros??**
  - ▶ Implicaria em gerenciar uma lista de ponteiros
- ▶ **Solução**
  - ▶ Incluir, para cada elemento da lista, o endereço do seu “vizinho”, ou seja, do próximo elemento da lista



# Exemplo: Lista de alunos

---

- ▶ Suponha que queremos guardar lista de alunos. As informações que queremos dos alunos está na estrutura *struct aluno*

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

- ▶ Além do aluno, devemos armazenar o endereço do próximo aluno da lista

```
struct aluno *prox; // endereço do próximo aluno
```




# Listas encadeadas

---

- ▶ A informação que queremos armazenar, juntamente com o endereço (ponteiro) pra próxima informação forma o que chamamos de **nó da lista**
- ▶ Outros nomes: **nó**; *node*; elemento
- ▶ Note que fazemos então um ponteiro para o próximo elemento **da lista** e não para o próximo dado

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

```
struct aluno *prox;
```



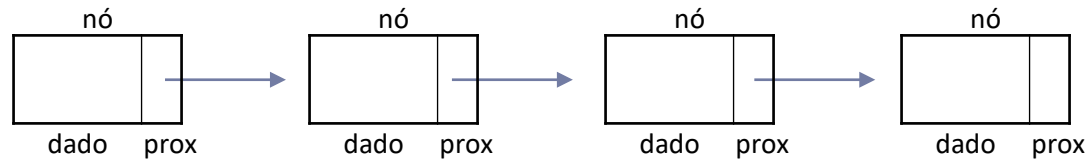
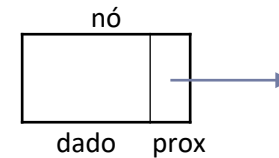
```
struct lista_no{  
    struct aluno dado;  
    struct lista_no *prox;  
};
```

# Listas encadeadas

---

## ► Representação gráfica

```
struct lista_no{  
    struct aluno dado;  
    struct lista_no *prox;  
};
```



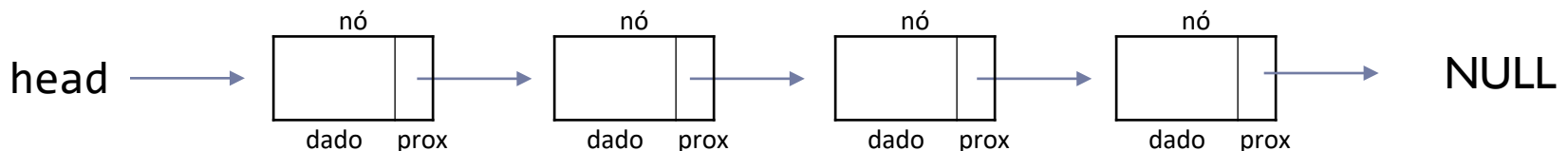
# Listas encadeadas

---

- ▶ Após a definição do nó, para definir a lista para indicar quem é o primeiro nó (nó cabeça – *head*).

```
struct lista{  
    struct lista_no *head;  
};
```

- ▶ O último nó deve ter como “próximo” o NULL, pois este é o final da lista





# Listas encadeadas

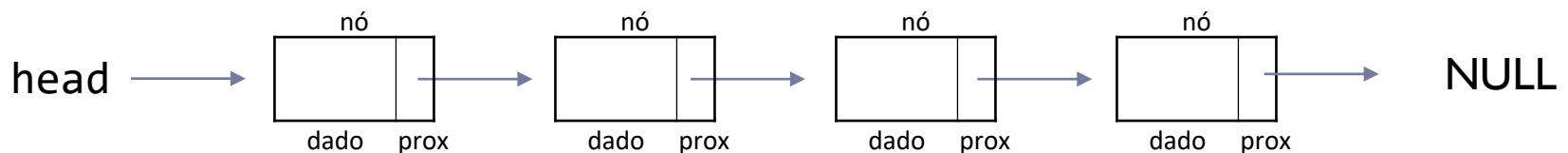
---

## ► Definição (com typedefs)

```
typedef struct lista Lista;  
typedef struct lista_no Lista_no;
```

```
struct lista{  
    Lista_no *head;  
};
```

```
struct Lista_no{  
    struct aluno dado;  
    Lista_no *prox;  
};
```



# In English

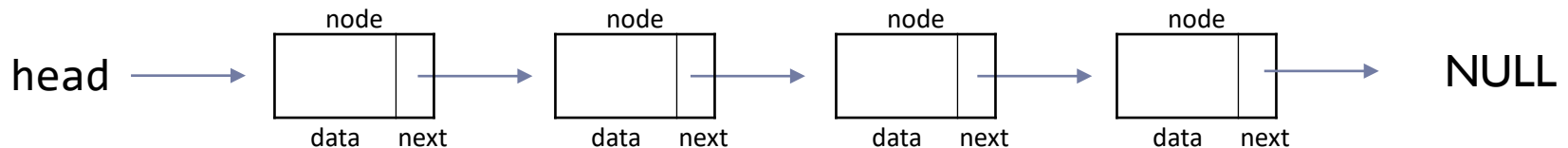
---

## ► Definição (com typedefs)

```
typedef struct list List;  
typedef struct list_node List_node;
```

```
struct list{  
    List_node *head;  
};
```

```
struct List_node{  
    struct aluno data;  
    List_node *next;  
};
```



# Lista Dinâmica Encadeada

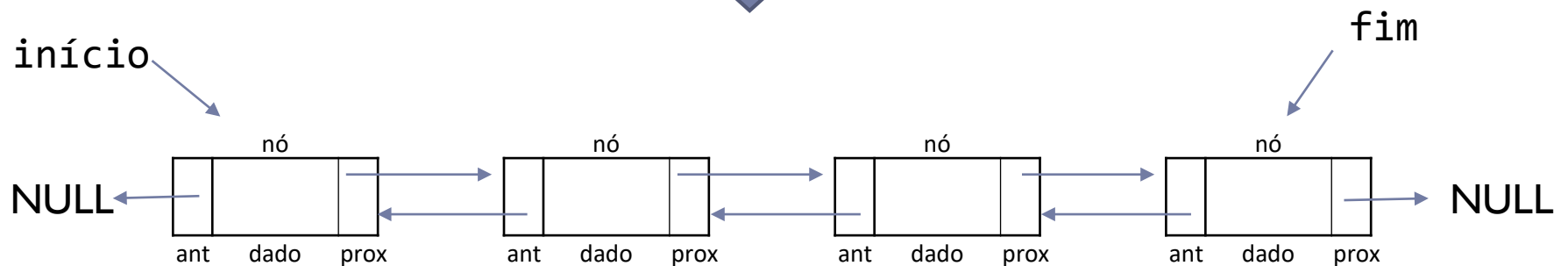
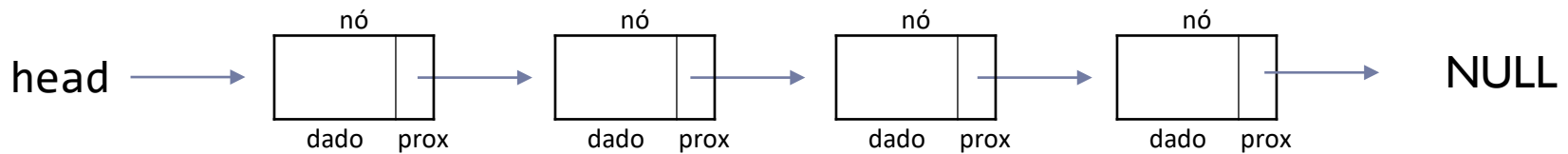
---

- ▶ Devido à alocação ser dinâmica para cada elemento os elementos da lista não ocuparão espaço contíguo
- ▶ Por não ser contíguo, não é possível saber de antemão em que local da memória foi alocado um determinado elemento da lista
  - ▶ (lembre que em vetores o espaço era contíguo, permitindo todos os elementos de um vetor sejam igualmente acessíveis)



# Listas dinâmica duplamente encadeada

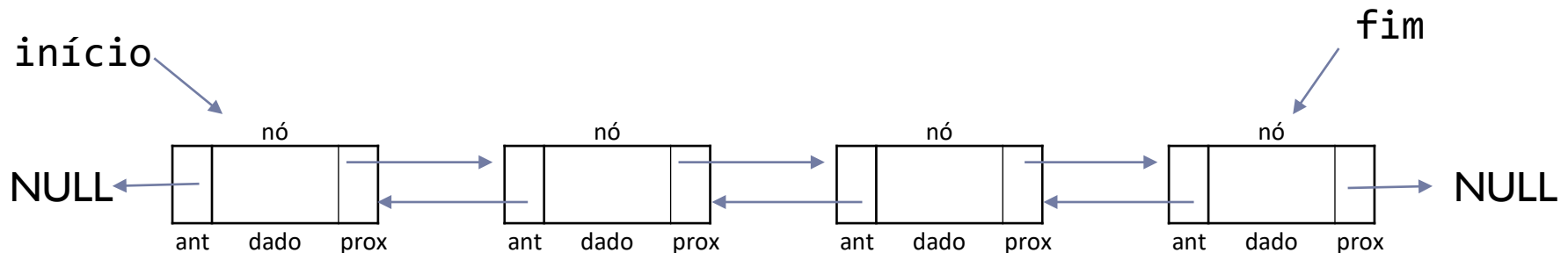
- Quais implicações e fazer a seguinte mudança:



# Listas dinâmica duplamente encadeada

---

## ► Listas dinâmica duplamente encadeada

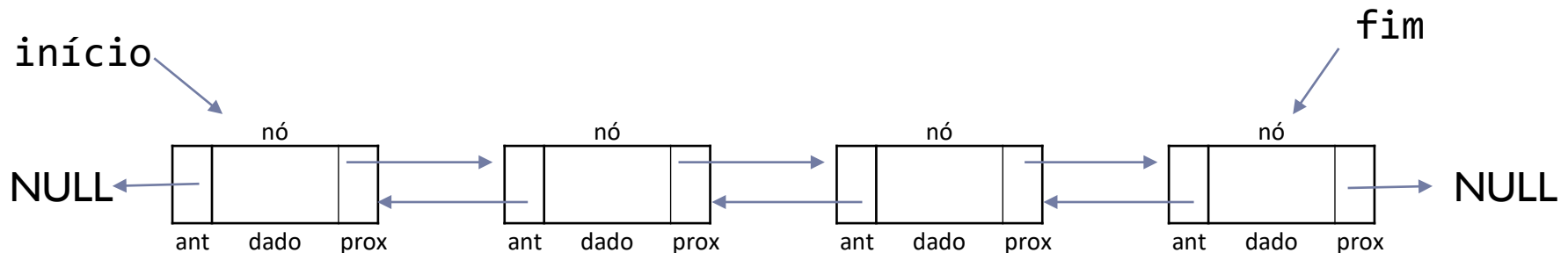


## ► Quais mudanças teremos nos códigos?

- Cuidar dos ponteiros para o próximo/anterior
- Avaliar sempre se está no início, meio ou final da lista nas operações
- Atualizar a quantidade de elementos (se houver)

# Listas dinâmica duplamente encadeada

## ▶ Listas dinâmica duplamente encadeada

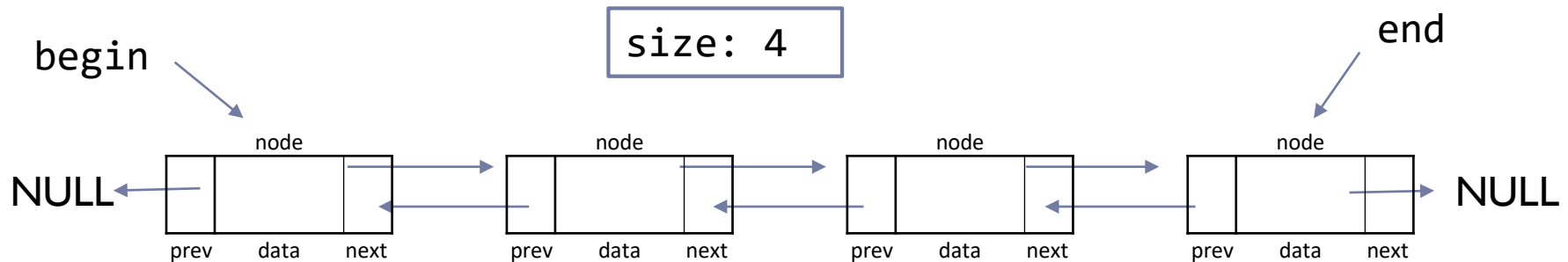


- ▶ Usa alocação dinâmica
- ▶ Usa acesso encadeado dos elementos (ponteiros 'prox' e 'ant')
- ▶ Último elemento tem como sucessor o NULL
- ▶ Primeiro elemento tem como antecessor o NULL
- ▶ Quais mudanças teremos nos códigos?
  - ▶ Cuidar dos ponteiros para o próximo/anterior
  - ▶ Avaliar sempre se está no início, meio ou final da lista nas operações
  - ▶ Atualizar a quantidade de elementos (se houver)

# In English...

---

## ► Doubly-linked lists



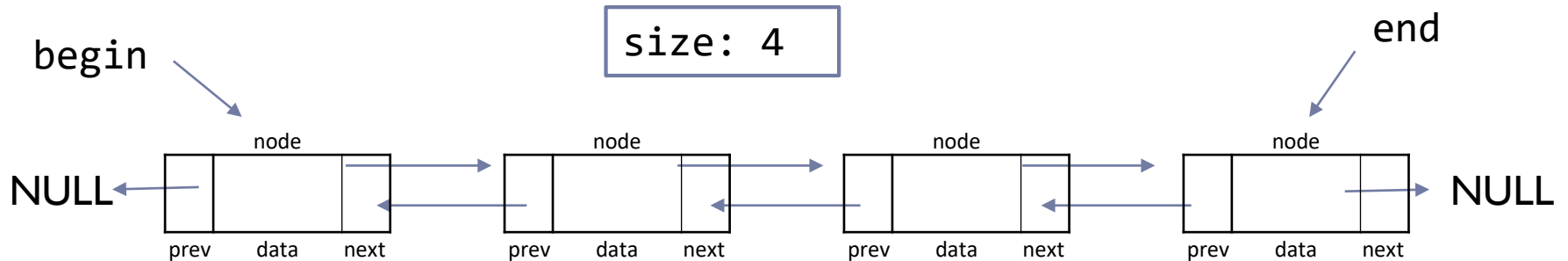
- Node: nó
  - Data: dados – estrutura com os dados
  - Prev (previous): anterior – ponteiro para o nó anterior
  - Next: próximo – ponteiro para o próximo nó
  - Begin: início – ponteiro para a cabeça da lista (início)
  - End: fim – ponteiro para o último nó
  - size: tamanho – tamanho da lista
- 



# Inserir no início (push\_front)

---

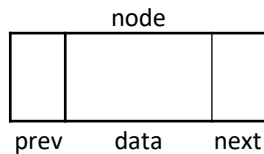
- Criar novo nó



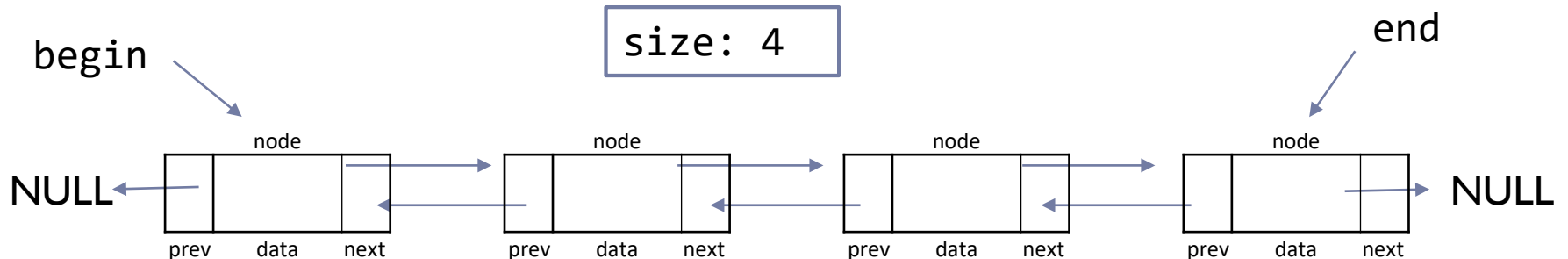


# Inserir no início (push\_front)

## ► Criar novo nó

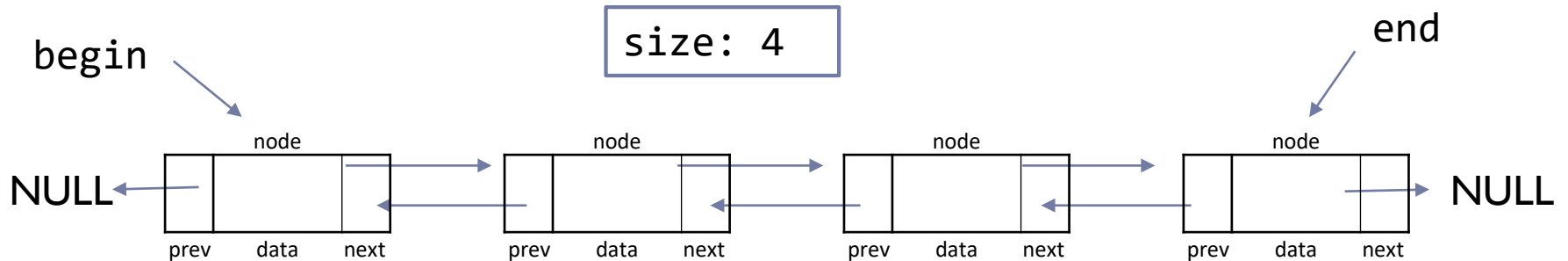
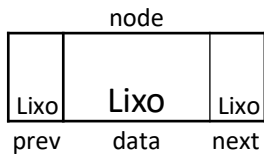


```
DLNode *node;  
node = malloc(sizeof(DLNode));
```



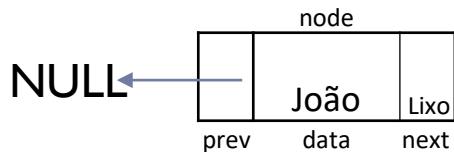
# Inserir no início (push\_front)

## ► Atualizar nó com informações

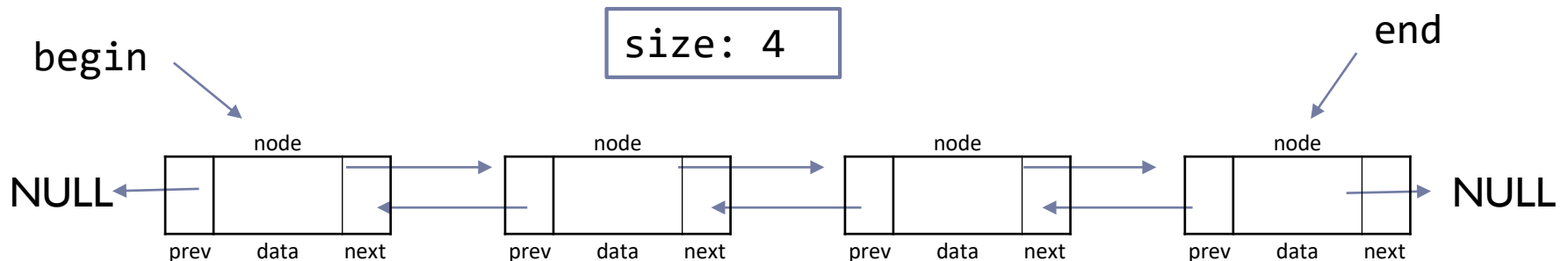


# Inserir no início (push\_front)

- ▶ Atualizar nó com informações
  - ▶ Copia informações do aluno
  - ▶ Já define ponteiro anterior como NULL, pois será inserido no início da lista



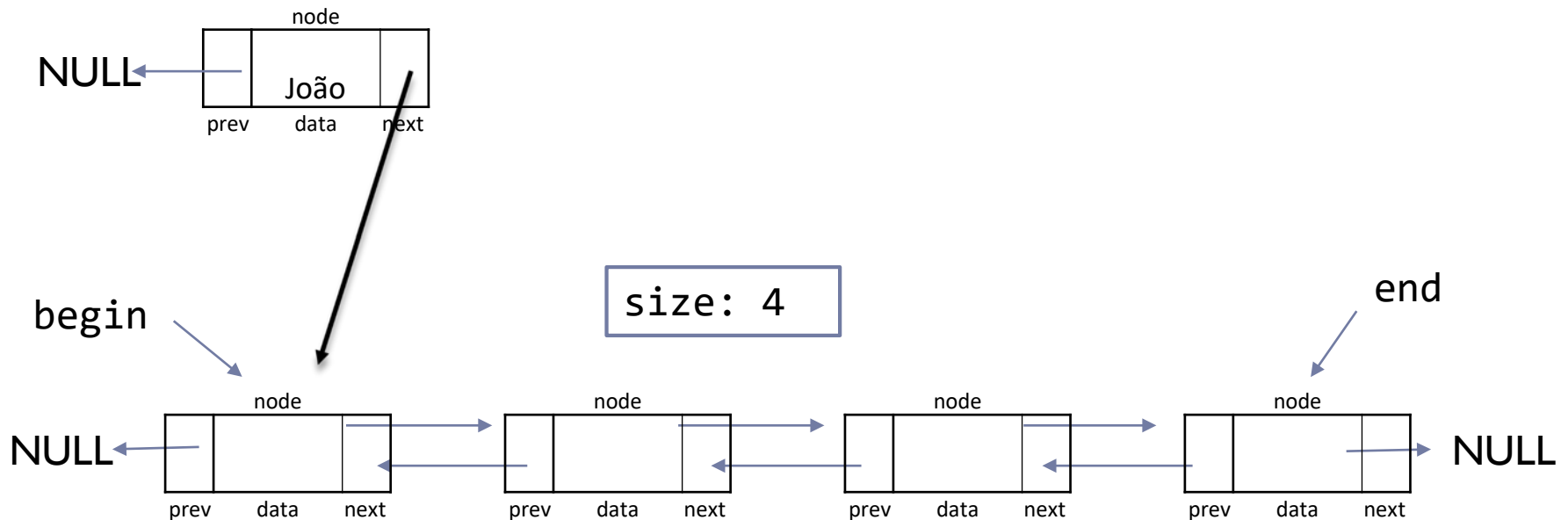
```
node->data = al;  
node->prev = NULL;
```



# Inserir no início (push\_front)

## ► Inserir na lista

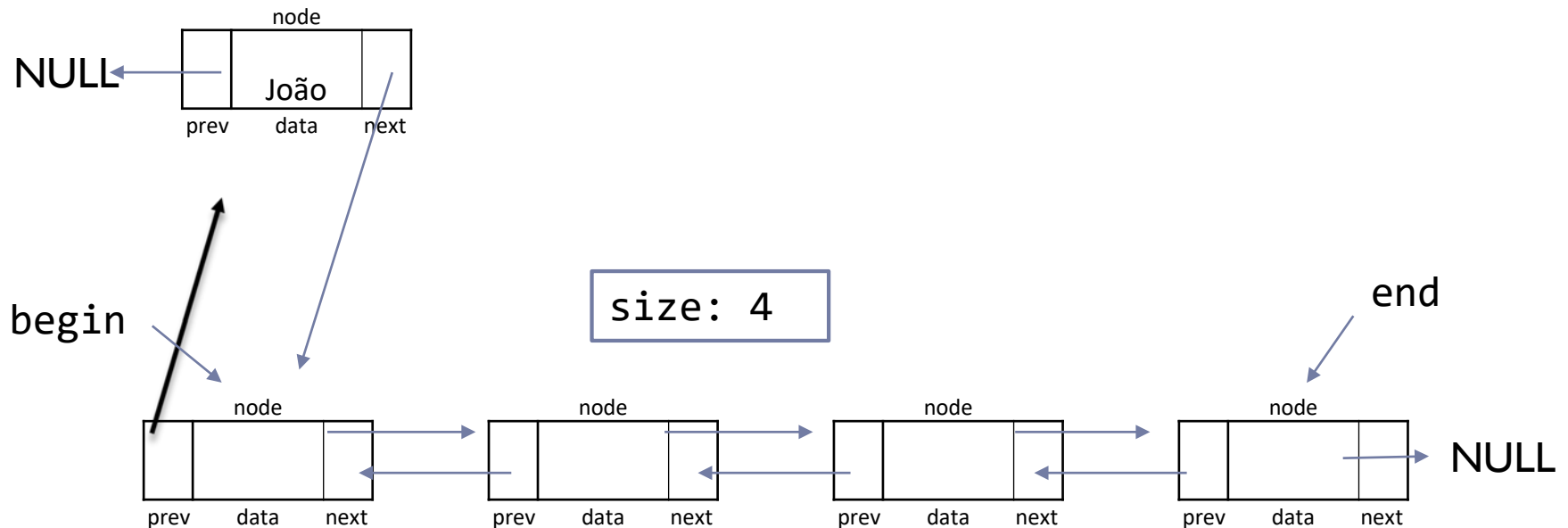
- Inicializar o ponteiro next do novo nó
- `node->next = li->begin;`



# Inserir no início (push\_front)

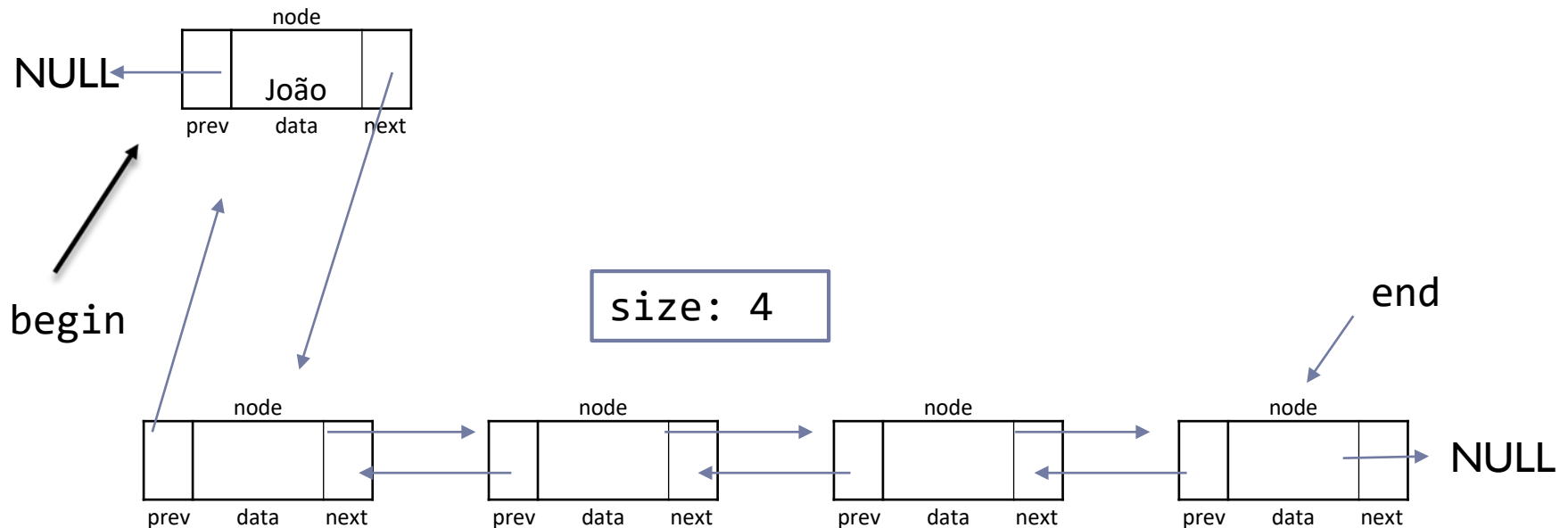
## ► Inserir na lista

- Antiga cabeça deve apontar para o novo nó
- `li->begin->prev = node;`



# Inserir no início (push\_front)

- ▶ Inserir na lista
  - ▶ Atualiza o begin da lista
  - ▶ `li->begin = node;`

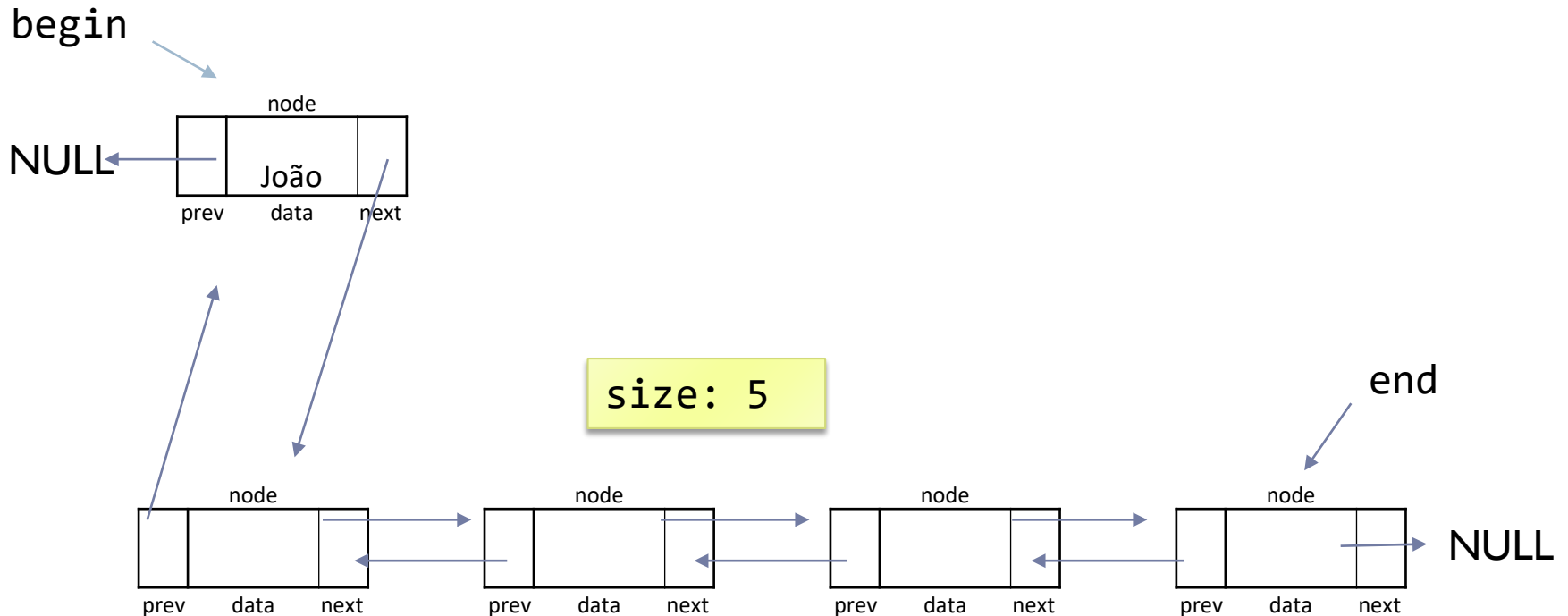


# Inserir no início (push\_front)

## ► Inserir na lista

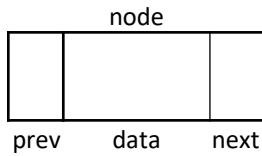
► Atualiza a quantidade

► `li->size = li->size + 1;`



# Inserir no início (push\_front)

---



begin



size: 0

end

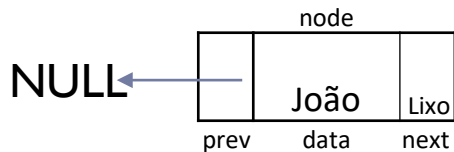




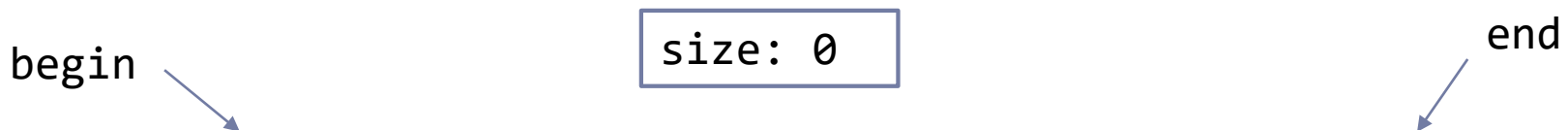
# Inserir no início (push\_front)

---

- ▶ Atualizar nó com informações
  - ▶ Copia informações do aluno
  - ▶ Já define ponteiro anterior como NULL, pois será inserido no início da lista



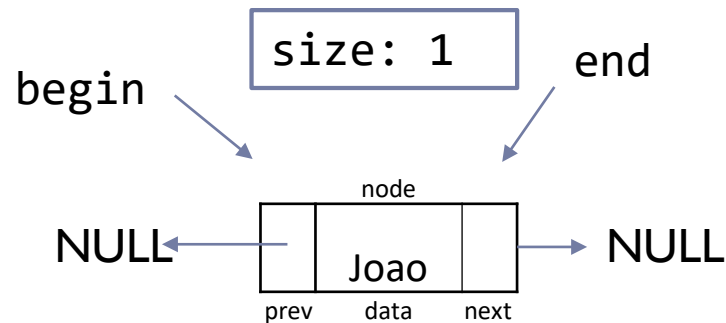
```
node->data = al;  
node->prev = NULL;
```



# Inserir no início (push\_front)

---

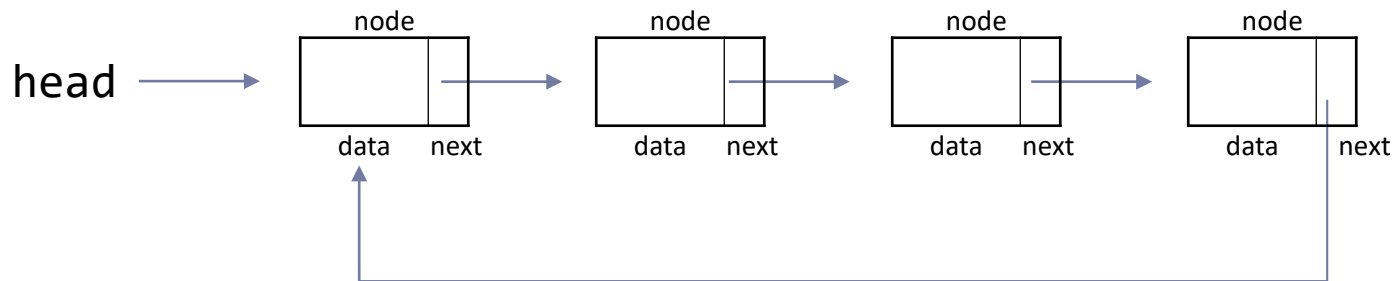
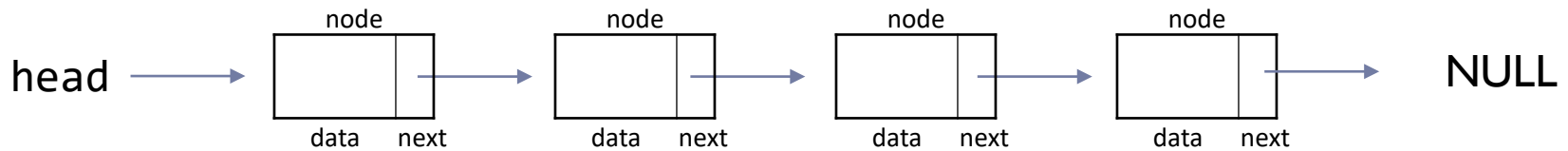
```
if (li->begin == NULL){  
    node->next = NULL;  
    li->begin = node;  
    li->end = node;  
    li->size = li->size + 1;  
}
```



# Listas circulares

---

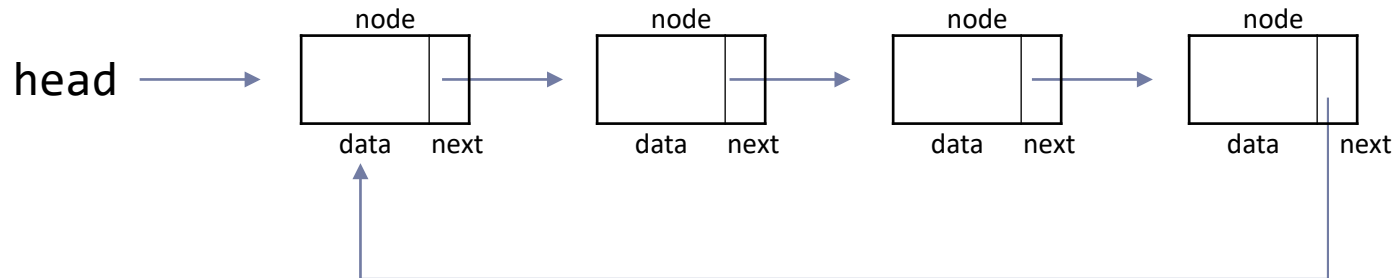
- Quais implicações e fazer a seguinte mudança:



# Listas circulares

---

## ► Lista dinâmica encadeada circular

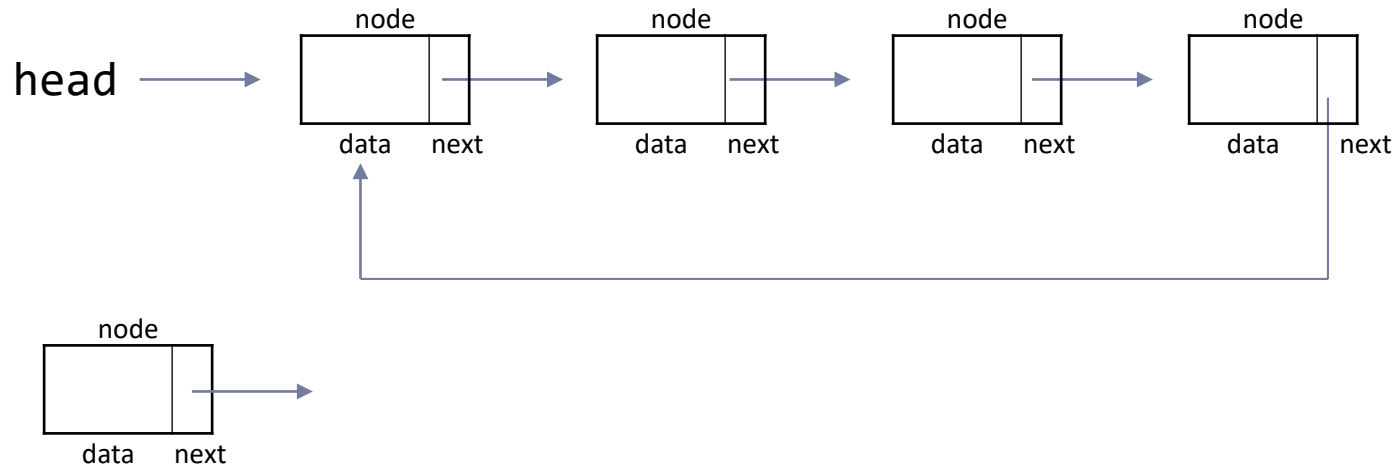


- Usa alocação dinâmica
- Usa acesso encadeado dos elementos (ponteiros 'next')
- Último elemento tem como sucessor o primeiro
- Quais mudanças teremos nos códigos?
  - Tudo que utilizava o NULL para atribuir/consultar o fim da lista

# Listas circulares - operações

---

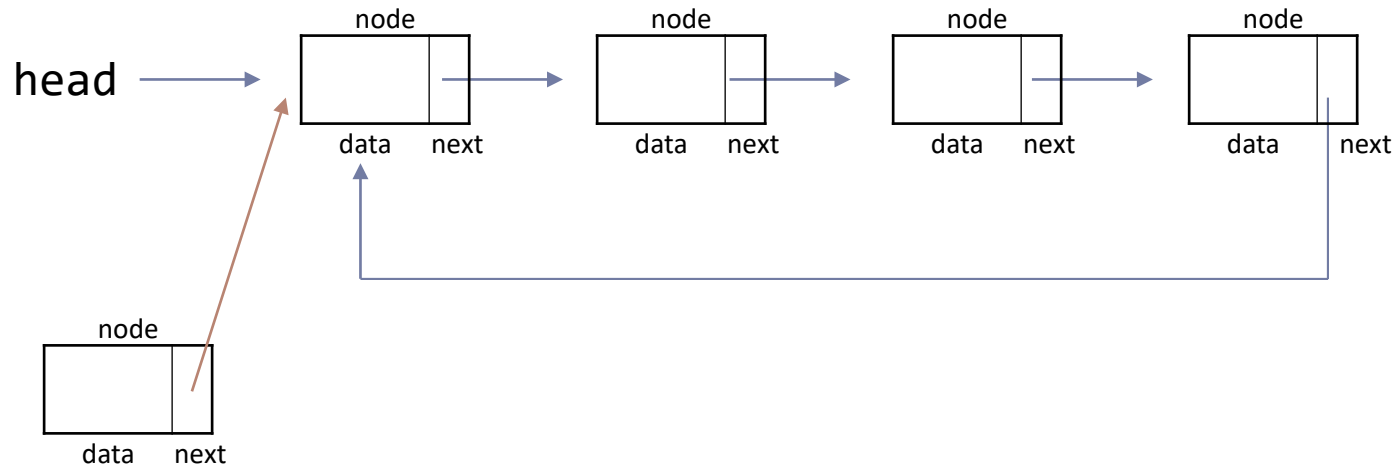
## ► Inserir no início



# Listas circulares - operações

---

## ► Inserir no início



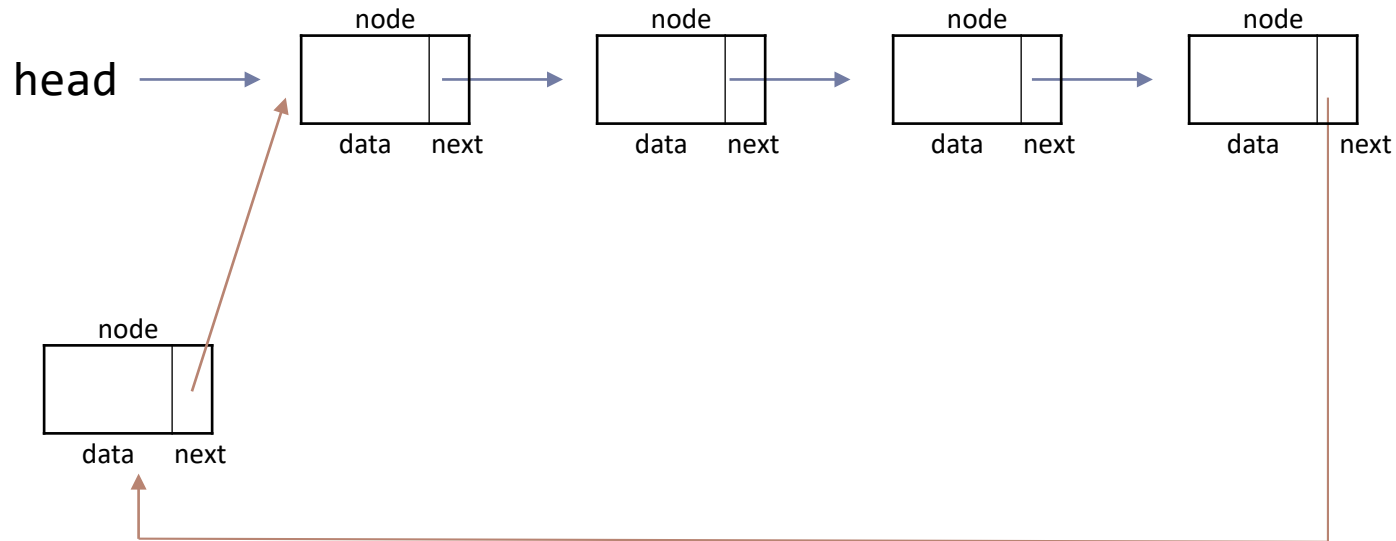
## ► Novo nó aponta para a cabeça



# Listas circulares - operações

---

## ► Inserir no início



## ► Último nó aponta para o novo nó

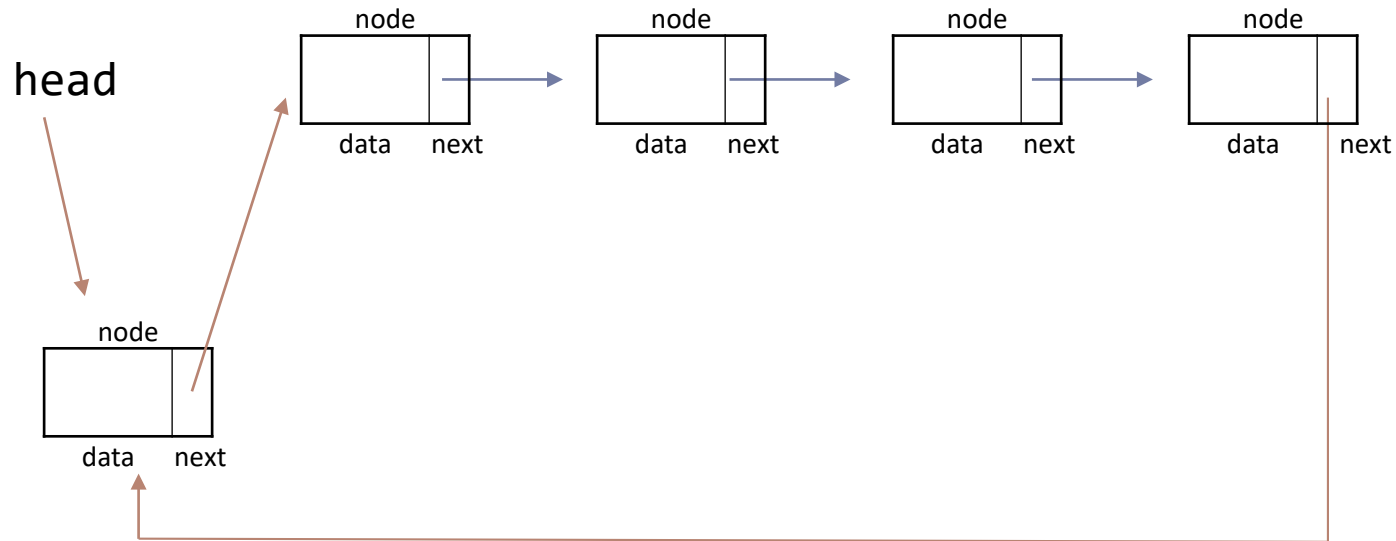
### ► Envolver percorrer toda a lista!



# Listas circulares - operações

---

## ► Inserir no início



## ► Cabeça aponta para o novo nó

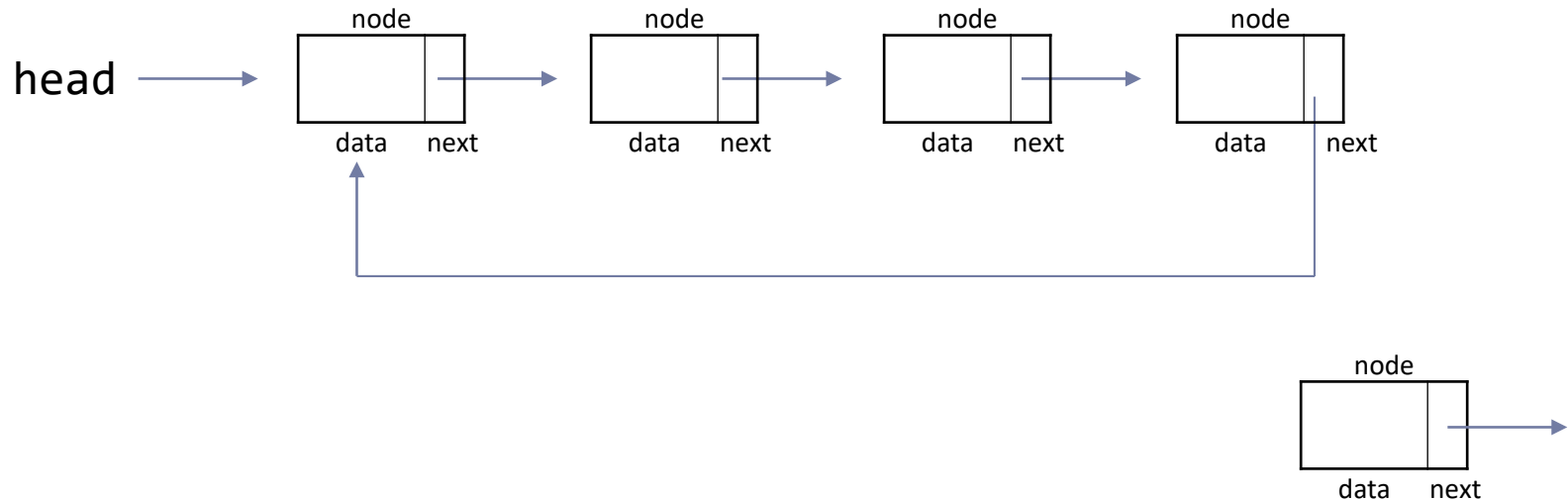




# Listas circulares - operações

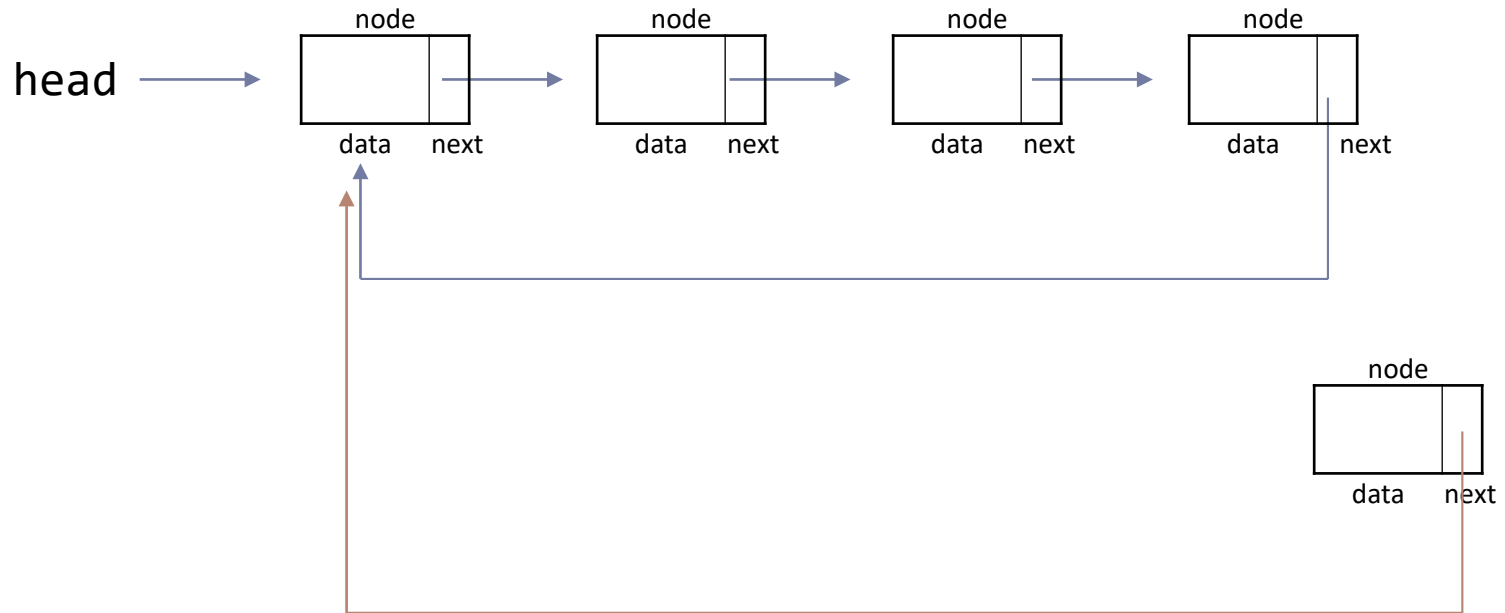
---

## ► Inserir no final



# Listas circulares - operações

## ► Inserir no final

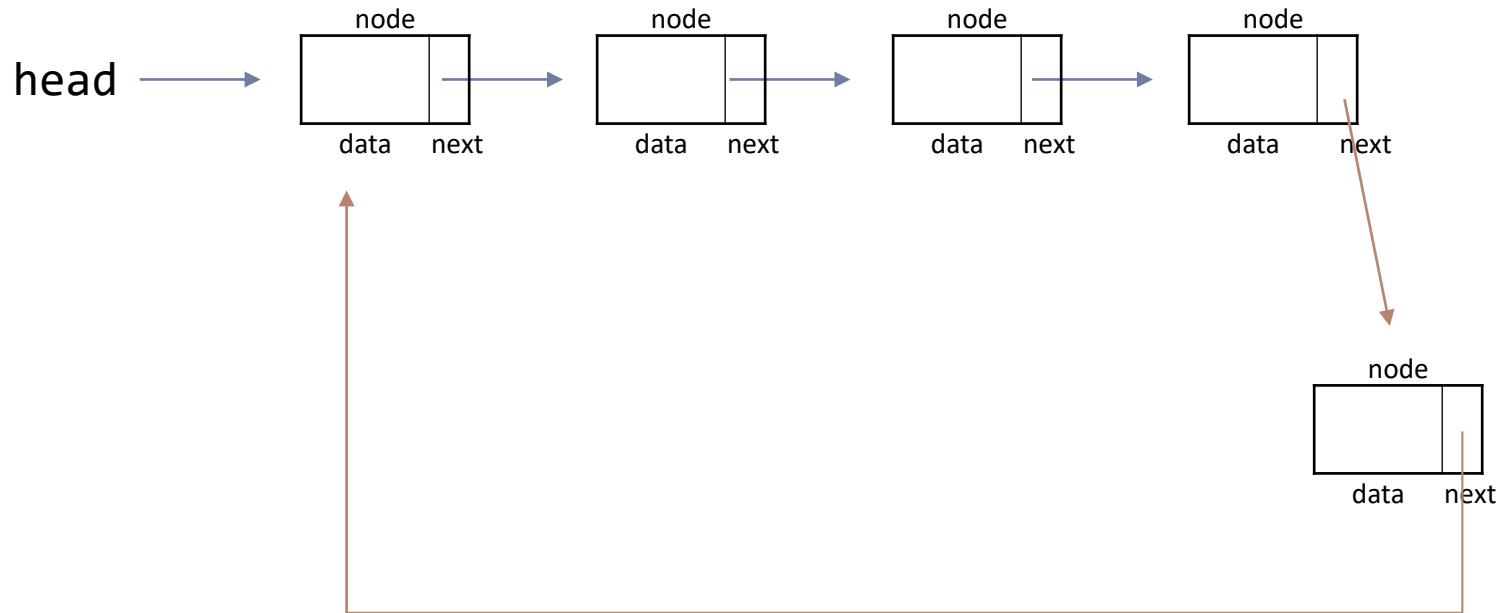


## ► Faz o novo nó apontar para a cabeça

# Listas circulares - operações

---

## ► Inserir no final



- Faz último nó apontar para o novo nó
  - Envolve percorrer toda a lista



# Listas circulares - operações

---

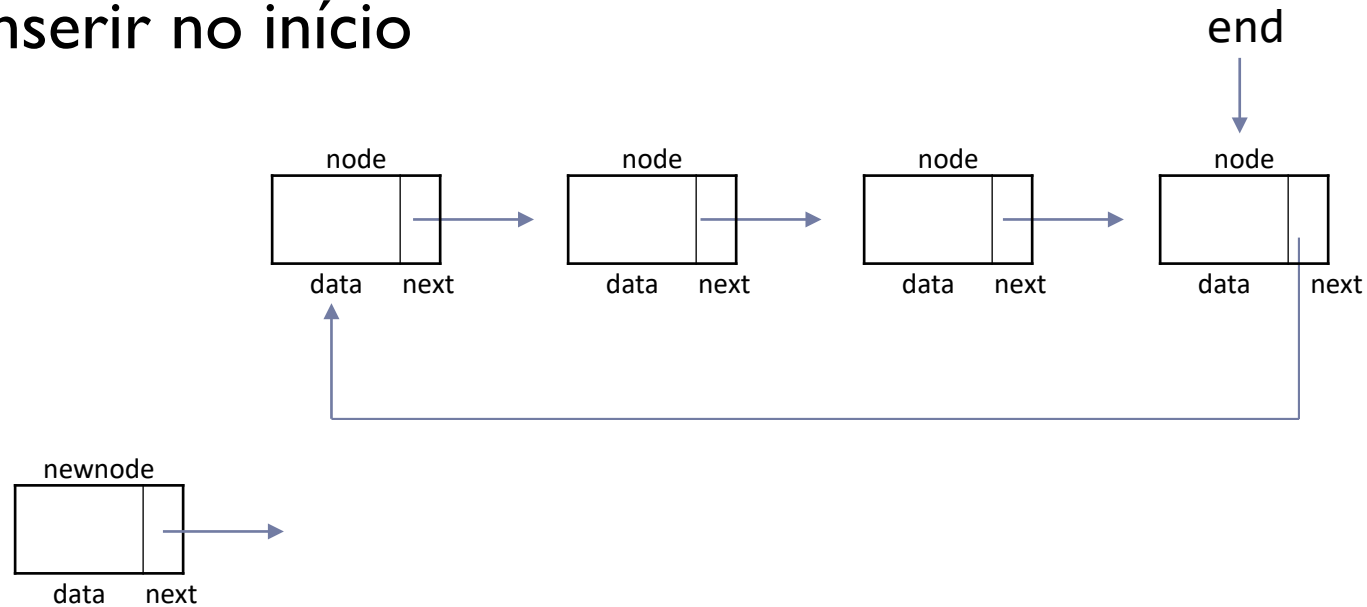
- ▶ Observe que ambas operações para inserir na lista circular, tanto no início quanto no final, envolve percorrer toda a lista
- ▶ Isso aumenta o custo computacional da operação de inserção
- ▶ Com uma simples modificação podemos reduzir esse custo
- ▶ Ao invés de termos um ponteiro apontando para o início da lista, colocamos um ponteiro para o último elemento



# Listas circulares - operações

---

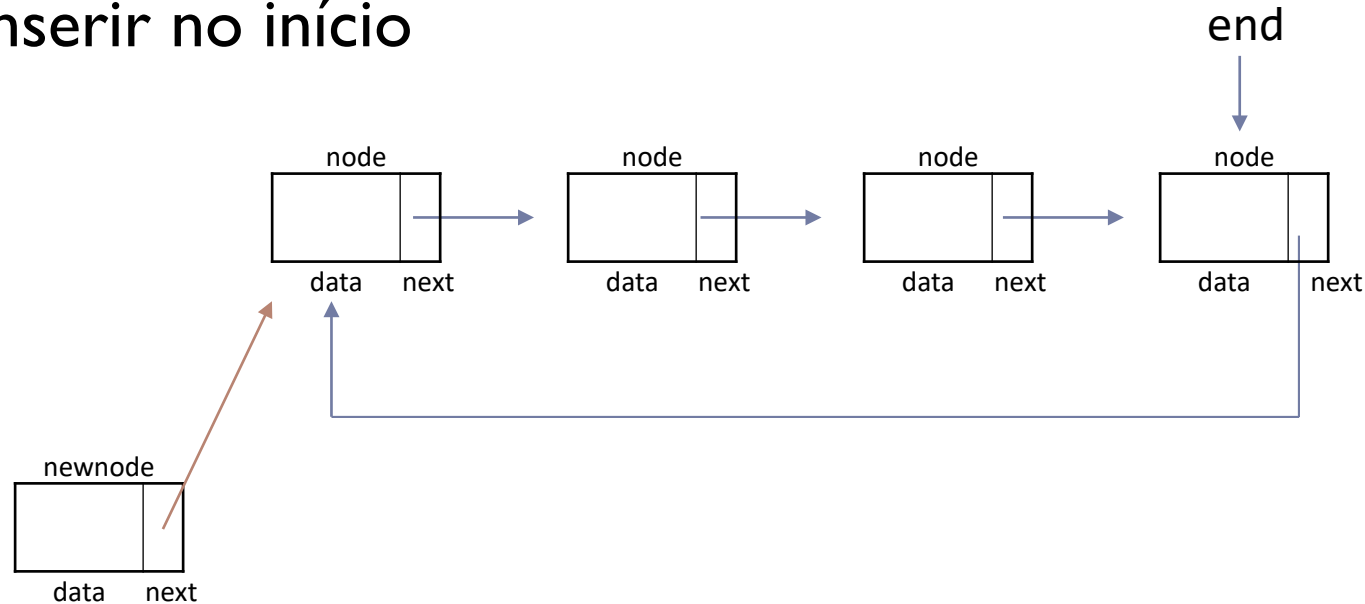
## ► Inserir no início



# Listas circulares - operações

---

## ► Inserir no início



## ► Faz o *next* do novo nó apontar para o início da lista

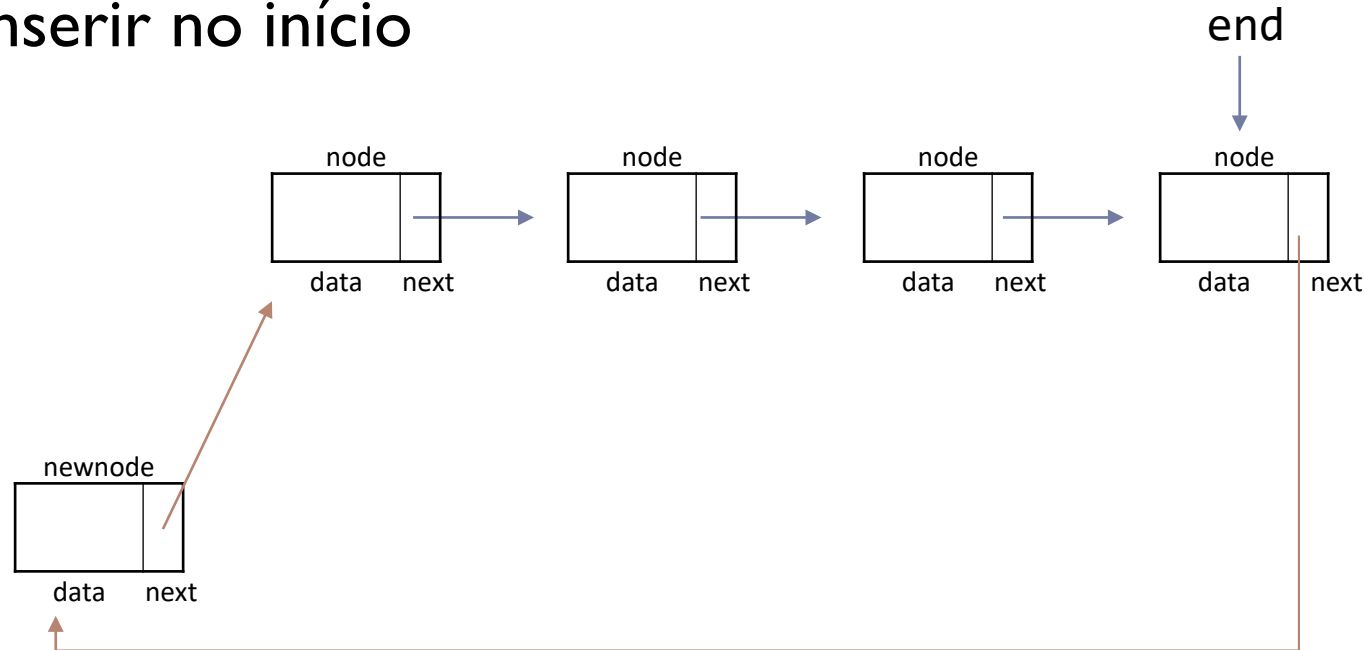
`newnode->next = end->next;`



# Listas circulares - operações

---

## ► Inserir no início



## ► Faz o próximo do final apontar para o novo nó

`end->next = newnode;`

## ► Pronto! Sem necessidade de percorrer a lista

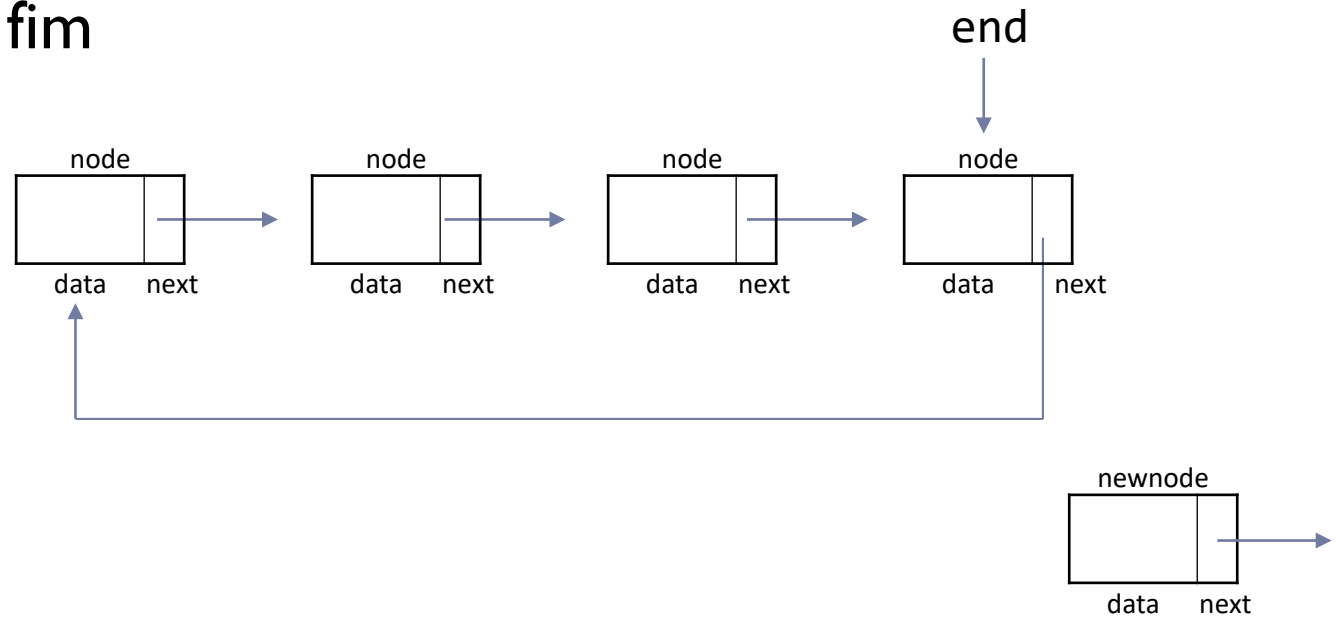
---



# Listas circulares - operações

---

## ► Inserir no fim

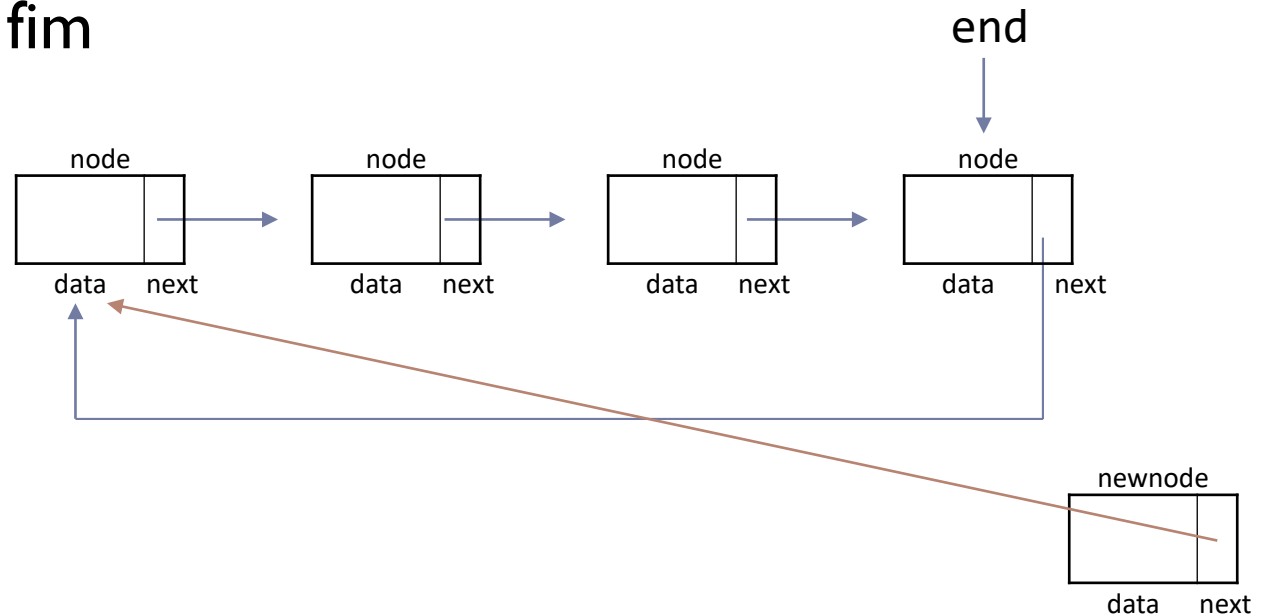




# Listas circulares - operações

---

## ► Inserir no fim



## ► Faz o *next* do novo nó apontar para o início da lista

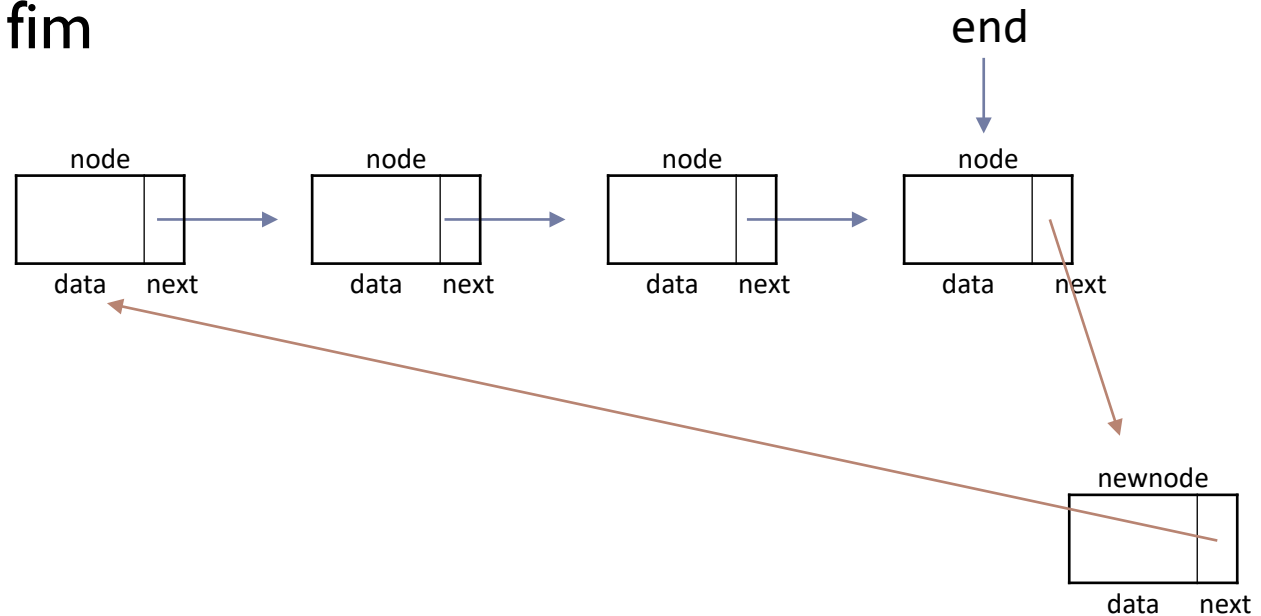
```
newnode->next = end->next;
```



# Listas circulares - operações

---

## ► Inserir no fim



## ► Faz o *next* do *end* apontar para o novo nó

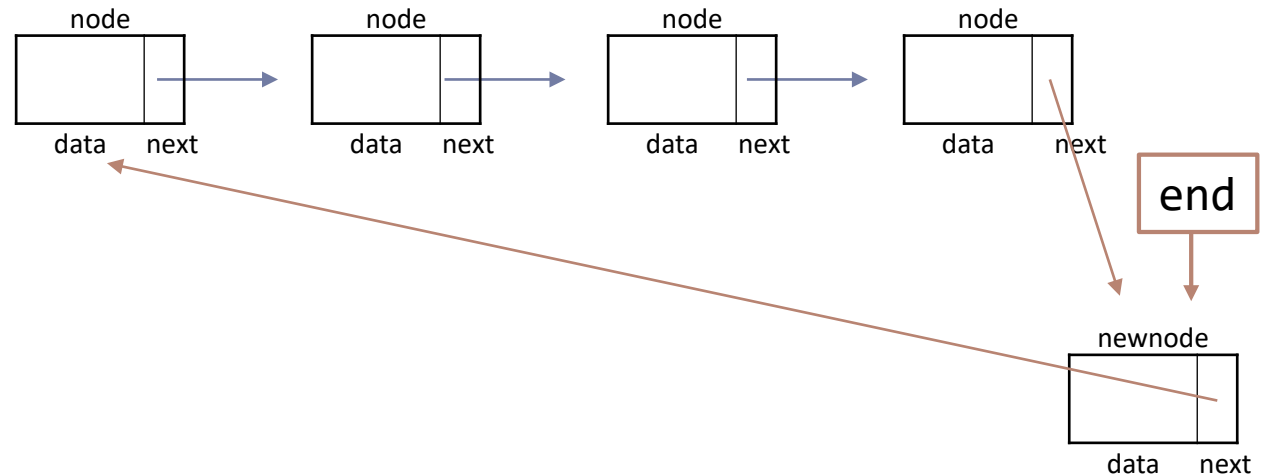
`end->next = newnode;`



# Listas circulares - operações

---

## ► Inserir no fim



## ► Altera o ponteiro 'end'

`end = newnode;`



# Listas dinâmica circulares

---

## ► Definição (com typedefs)

```
typedef struct circlist Circlist;  
typedef struct clistnode CList_node;
```

```
struct circlist{  
    CList_node *end;  
};
```

```
struct clistnode{  
    struct aluno dado;  
    CList_node *prox;  
};
```



# Observações

---

- ▶ Uma lista circular é diferente de um fila estática sequencial.
- ▶ A fila estática sequencial não é circular.
- ▶ Apenas o vetor que armazena os dados da fila possui um comportamento circular, mas a fila em si não é circular



# Listas sem nó descritor

- ▶ É possível construirmos uma lista encadeada sem utilizar nó descritor
- ▶ Para isso, devemos tratar nossa lista como um ponteiro para um nó direto, sem ter uma estrutura intermediária

## Com nó descritor

```
// no .h
typedef struct lista Lista;
// no .c
typedef struct lista_no Lista_no;

struct lista{
    Lista_no *head;
};

struct Lista_no{
    struct aluno dado;
    Lista_no *prox;
};
```

## SEM nó descritor

```
// no .h
typedef struct lista_no* Lista;
// no .c
typedef struct lista_no Lista_no;

struct lista_no{
    struct aluno dado;
    Lista_no *prox;
};
```

# Listas sem nó descritor

---

- ▶ A maior implicação dessa decisão é que as funções do TAD vão precisar desreferenciar esse ponteiro para acesso à lista

## Com nó descritor

```
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->inicio;
    if(li->inicio == NULL)
        li->final = no;
    li->inicio = no;
    li->qtd++;
    return 1;
}
```

## SEM nó descritor

```
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;
}
```



# Listas sem nó descritor

- ▶ A maior implicação dessa decisão é que as funções do TAD vão precisar desreferenciar esse ponteiro para acesso à lista

## Com nó descritor

```
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->inicio;
    if(li->inicio == NULL)
        li->final = no;
    li->inicio = no;
    li->qtd++;
    return 1;
}
```

## SEM nó descritor

```
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;
}
```

O cabeçalho é o mesmo, mas há uma grande diferença. Com nó descritor:

**Lista\* => struct lista\***

Sem nó descriptor:

**Lista\* => struct lista\_no\*\***



# Listas sem nó descritor

- ▶ A maior implicação dessa decisão é que as funções do TAD vão precisar desreferenciar esse ponteiro para acesso à lista

## Com nó descritor

```
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->inicio;
    if(li->inicio == NULL)
        li->final = no;
    li->inicio = no;
    li->qtd++;
    return 1;
}
```

## SEM nó descritor

```
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;
}
```

Comparação é a mesma, mas os tipos comparados são diferentes. Com nó:

**Lista\* => struct lista\***

Sem nó descritor:

**Lista\* => struct lista\_no\*\***

# Listas sem nó descritor

- ▶ A maior implicação dessa decisão é que as funções do TAD vão precisar desreferenciar esse ponteiro para acesso à lista

## Com nó descritor

```
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->inicio;
    if(li->inicio == NULL)
        li->final = no;
    li->inicio = no;
    li->qtd++;
    return 1;
}
```

## SEM nó descritor

```
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;
}
```

Tudo igual aqui

# Listas sem nó descritor

- ▶ A maior implicação dessa decisão é que as funções do TAD vão precisar desreferenciar esse ponteiro para acesso à lista

## Com nó descritor

```
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->inicio;
    if(li->inicio == NULL)
        li->final = no;
    li->inicio = no;
    li->qtd++;
    return 1;
}
```

## SEM nó descritor

```
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;
}
```

Inseriu no início, então o novo nó aponta para o início da lista atual  
Com nó descritor, mudamos o ponteiro início, sem nó descritor, mudamos a lista em si

# Listas sem nó descritor

- ▶ A maior implicação dessa decisão é que as funções do TAD vão precisar desreferenciar esse ponteiro para acesso à lista

## Com nó descritor

```
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->inicio;
    if(li->inicio == NULL)
        li->final = no;
    li->inicio = no;
    li->qtd++;
    return 1;
}
```

## SEM nó descritor

```
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;
}
```

O início da lista passa a ser o novo nó.  
Com nó descritor já temos um ponteiro para isso. Sem nó descritor mudamos o ponteiro para onde a lista aponta

# Listas sem nó descritor

- ▶ A maior implicação dessa decisão é que as funções do TAD vão precisar desreferenciar esse ponteiro para acesso à lista

## Com nó descritor

```
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->inicio;
    if(li->inicio == NULL)
        li->final = no;
    li->inicio = no;
    li->qtd++;
    return 1;
}
```

## SEM nó descritor

```
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;
}
```

Ter o nó descritor permite colocar outras variáveis na estrutura para facilitar outras operações

# Listas sem nó descritor

## ► Como funciona o ponteiro de ponteiro

```
// no .c
Lista* cria_lista(){
    Lista* li;
    li = (Lista*) malloc(sizeof(Lista));
    if(li != NULL)
        *li = NULL;
    return li;
}

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;
}
```

```
// no .h
typedef struct lista_no* Lista;
```

```
// no main.c
Lista* li;
li = cria_lista();
```

```
// no .h
typedef struct lista_no* Lista;
```

```
// no main.c
Lista* li;
li = cria_lista();
```

Endereço	Blocos (1 byte)	Nome variável	Tipo
0 / NULL	indefinido	----	----
1			
2			
3			
4	lx	li	Lista*
5			
6			
7			
8			
9			
10			
11			

Endereço	Blocos (1 byte)	Nome variável	Tipo
47			
48			
49			
50			
51			
52			
53			
54			
55			
56			
57			
58			



```
// no .h
typedef struct lista_no* Lista;
```

```
// no main.c
Lista* li;
li = cria_lista();
```

Endereço	Blocos (1 byte)	Nome variável	Tipo
0 / NULL	indefinido	----	----
1			
2			
3			
4	lx	li	Lista*
5			
6			
7			
8			
9			
10			
11			

Endereço	Blocos (1 byte)	Nome variável	Tipo
47			
48			
49			
50			
51			
52			
53			
54			
55			
56			
57			
58			

=  
struct lista\_no\*\*

li → LX



```
// no .h
typedef struct lista_no* Lista;
```

```
// no main.c
Lista* li;
li = cria_lista();
```

Endereço	Blocos (1 byte)	Nome variável	Tipo
0 / NULL	indefinido	----	----
1			
2			
3			
4	lx	li	Lista*
5			
6			
7			
8			
9			
10			
11			

Escopo local da função  
(stack)

Endereço	Blocos (1 byte)	Nome variável	Tipo
47			
48			
49			
50	lx	li	Lista*
51			
52			
53			
54			
55			
56			
57			
58			

```
// no .c
Lista* cria_lista(){
    Lista* li;
    li = (Lista*) malloc(sizeof(Lista));
    if(li != NULL)
        *li = NULL;
    return li;
}
```

li

→ LX

```
// no .h
typedef struct lista_no* Lista;
```

```
// no main.c
Lista* li;
li = cria_lista();
```

Endereço	Blocos (1 byte)	Nome variável	Tipo
0 / NULL	indefinido	----	----
1			
2			
3			
4	lx	li	Lista*
5			
6			
7			
8			
9			
10			
11			

Escopo local da função  
(stack)

Endereço	Blocos (1 byte)	Nome variável	Tipo
47			
48			
49			
50	370	li	Lista*
51			
52			
53			
54			
55			
56			
57			
58			

```
// no .c
Lista* cria_lista(){
    Lista* li;
    li = (Lista*) malloc(sizeof(Lista));
    if(li != NULL)
        *li = NULL;
    return li;
}
```

367			
368			
369			
370	lx	----	struct lista_no*
371			
372			
373			
374			
375			

heap

li → LX

```
// no .h
typedef struct lista_no* Lista;
```

```
// no main.c
Lista* li;
li = cria_lista();
```

Endereço	Blocos (1 byte)	Nome variável	Tipo
0 / NULL	indefinido	----	----
1			
2			
3			
4	lx	li	Lista*
5			
6			
7			
8			
9			
10			
11			

Escopo local da função  
(stack)

Endereço	Blocos (1 byte)	Nome variável	Tipo
47			
48			
49			
50	370	li	Lista*
51			
52			
53			
54			
55			
56			
57			
58			

```
// no .c
Lista* cria_lista(){
    Lista* li;
    li = (Lista*) malloc(sizeof(Lista));
    if(li != NULL)
        *li = NULL;
    return li;
}
```

struct lista\_no\*

367			
368			
369			
370	lx	----	struct lista_no*
371			
372			
373			
374			
375			

heap


li

→ LX

```
struct lista_no{
    struct aluno dado;
    Lista_no *prox;
};
```

Escrevendo de outra forma

```
// no .c
Lista* cria_lista(){
    struct lista_no** li;
    li = (struct lista_no**) malloc(sizeof(struct lista_no*));
    if(li != NULL)
        *li = NULL;
    return li;
}
```



```
// no .h
typedef struct lista_no* Lista;
```

```
// no main.c
Lista* li;
li = cria_lista();
```

Endereço	Blocos (1 byte)	Nome variável	Tipo
0 / NULL	indefinido	----	----
1			
2			
3			
4	lx	li	Lista*
5			
6			
7			
8			
9			
10			
11			

Escopo local da função  
(stack)

Endereço	Blocos (1 byte)	Nome variável	Tipo
47			
48			
49			
50	370	li	Lista*
51			
52			
53			
54			
55			
56			
57			
58			

```
// no .c
Lista* cria_lista(){
    Lista* li;
    li = (Lista*) malloc(sizeof(Lista));
    if(li != NULL)
        *li = NULL;
    return li;
}
```

367			
368			
369			
370	NULL	----	struct lista_no*
371			
372			
373			
374			
375			

heap

li

→ LX

```
// no .h
typedef struct lista_no* Lista;
```

```
// no main.c
Lista* li;
li = cria_lista();
```

Endereço	Blocos (1 byte)	Nome variável	Tipo
0 / NULL	indefinido	----	----
1			
2			
3			
4	370	li	Lista*
5			
6			
7			
8			
9			
10			
11			

Endereço	Blocos (1 byte)	Nome variável	Tipo
47			
48			
49			
50			
51			
52			
53			
54			
55			
56			
57			
58			

```
// no .c
Lista* cria_lista(){
    Lista* li;
    li = (Lista*) malloc(sizeof(Lista));
    if(li != NULL)
        *li = NULL;
    return li;
}
```

367			
368			
369			
370	NULL	----	struct lista_no*
371			
372			
373			
374			
375			

li → 370

heap

```
// no .h
typedef struct lista_no* Lista;
```

```
// no main.c
Lista* li;
li = cria_lista();
...
insere_lista_inicio(
li, al)
```

Endereço	Blocos (1 byte)	Nome variável	Tipo
0 / NULL	indefinido	----	----
1			
2			
3			
4	370	li	Lista*
5			
6			
7			
8..54	João	al	struct aluno
55			

Endereço	Blocos (1 byte)	Nome variável	Tipo
47			
48			
49			
50			
51			
52			
53			
54			
55			
56			

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

367			
368			
369			
370	NULL	----	struct lista_no*
371			
372			
373			
374			
375			

heap

```
// no .h
typedef struct lista_no* Lista;
```

```
// no main.c
Lista* li;
li = cria_lista();
...
insere_lista_inicio(
li, al)
```

Endereço	Blocos (1 byte)	Nome variável	Tipo
0 / NULL	indefinido	----	----
1			
2			
3			
4	370	li	Lista*
5			
6			
7			
8..54	João	al	struct aluno

Endereço	Blocos (1 byte)	Nome variável	Tipo
47	370	li	Lista*
48			
49			
50			
57..97	João	al	struct aluno
98			
99			
100			
101			

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

367			
368			
369			
370	NULL	----	struct lista_no*
371			
372			
373			
374			
375			

heap



```
// no .h
typedef struct lista_no* Lista;
```

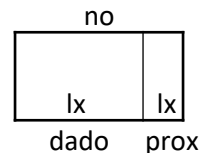
```
struct lista_no{
    struct aluno dado;
    Lista_no *prox;
};
```

```
// no main.c
Lista* li;
li = cria_lista();
...
insere_lista_inicio(
li, al)
```

Endereço	Blocos (1 byte)	Nome variável	Tipo
0 / NULL	indefinido	----	----
1			
2			
3			
4	370	li	Lista*
5			
6			
7			
8..54	João	al	struct aluno

Endereço	Blocos (1 byte)	Nome variável	Tipo
47	370	li	Lista*
48			
49			
50			
57..97	João	al	struct aluno
98			
99			
100			
101			

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



heap

370	NULL		struct lista_no*
371			
372			
373			
374	lx	prox	struct lista_no*
375			
376			
377			
378..424	lx	dado	struct aluno

```
// no .h
typedef struct lista_no* Lista;
```

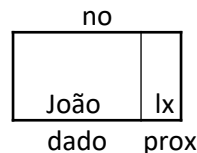
```
struct lista_no{
    struct aluno dado;
    Lista_no *prox;
};
```

```
// no main.c
Lista* li;
li = cria_lista();
...
insere_lista_inicio(
li, al)
```

Endereço	Blocos (1 byte)	Nome variável	Tipo
0 / NULL	indefinido	----	----
1			
2			
3			
4	370	li	Lista*
5			
6			
7			
8..54	João	al	struct aluno

Endereço	Blocos (1 byte)	Nome variável	Tipo
47	370	li	Lista*
48			
49			
50			
57..97	João	al	struct aluno
98			
99			
100			
101			

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



heap

370	NULL		struct lista_no*
371			
372			
373			
374		prox	struct lista_no*
375			
376			
377			
378..424	João	dado	struct aluno

```
// no .h
typedef struct lista_no* Lista;
```

```
struct lista_no{
    struct aluno dado;
    Lista_no *prox;
};
```

```
// no main.c
Lista* li;
li = cria_lista();
...
insere_lista_inicio(
li, al)
```

Endereço	Blocos (1 byte)	Nome variável	Tipo
0 / NULL	indefinido	----	----
1			
2			
3			
4	370	li	Lista*
5			
6			
7			
8..54	João	al	struct aluno

Endereço	Blocos (1 byte)	Nome variável	Tipo
47	370	li	Lista*
48			
49			
50			
57..97	João	al	struct aluno
98			
99			
100			
101			

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

no	
João	Null
dado	prox

heap

370	NULL		struct lista_no*
371			
372			
373			
374	NULL	prox	struct lista_no*
375			
376			
377			
378..424	João	dado	struct aluno

```
// no .h
typedef struct lista_no* Lista;
```

```
struct lista_no{
    struct aluno dado;
    Lista_no *prox;
};
```

```
// no main.c
Lista* li;
li = cria_lista();
...
insere_lista_inicio(
li, al)
```

Endereço	Blocos (1 byte)	Nome variável	Tipo
0 / NULL	indefinido	----	----
1			
2			
3			
4	370	li	Lista*
5			
6			
7			
8..54	João	al	struct aluno

Endereço	Blocos (1 byte)	Nome variável	Tipo
47	370	li	Lista*
48			
49			
50			
57..97	João	al	struct aluno
98			
99			
100			
101			

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

no	
João	Null
dado	prox

heap

370	374		struct lista_no*
371			
372			
373			
374	NULL	prox	struct lista_no*
375			
376			
377			
378..424	João	dado	struct aluno

```
// no .h
typedef struct lista_no* Lista;
```

```
struct lista_no{
    struct aluno dado;
    Lista_no *prox;
};
```

```
// no main.c
Lista* li;
li = cria_lista();
...
insere_lista_inicio(
li, al)
```

Endereço	Blocos (1 byte)	Nome variável	Tipo
0 / NULL	indefinido	----	----
1			
2			
3			
4	370	li	Lista*
5			
6			
7			
8..54	João	al	struct aluno
55			

Endereço	Blocos (1 byte)	Nome variável	Tipo
47			
48			
49			
50			
51			
52			
53			
54			
55			
56			

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

heap

370	374		struct lista_no*
371			
372			
373			
374	NULL	prox	struct lista_no*
375			
376			
377			
378..424	João	dado	struct aluno

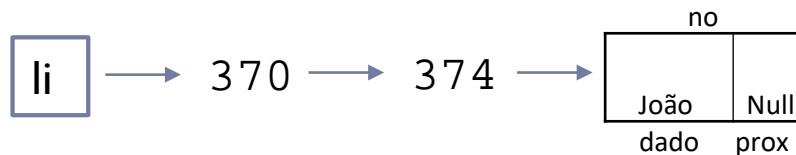
```
// no .h
typedef struct lista_no* Lista;
```

```
struct lista_no{
    struct aluno dado;
    Lista_no *prox;
};
```

```
// no main.c
Lista* li;
li = cria_lista();
...
insere_lista_inicio(
li, al)
```

Endereço	Blocos (1 byte)	Nome variável	Tipo
0 / NULL	indefinido	----	----
1			
2			
3			
4	370	li	Lista*
5			
6			
7			
8..54	João	al	struct aluno
55			

Endereço	Blocos (1 byte)	Nome variável	Tipo
47			
48			
49			
50			
51			
52			
53			
54			
55			
56			



heap

370	374		struct lista_no*
371			
372			
373			
374	NULL	prox	struct lista_no*
375			
376			
377			
378..424	João	dado	struct aluno