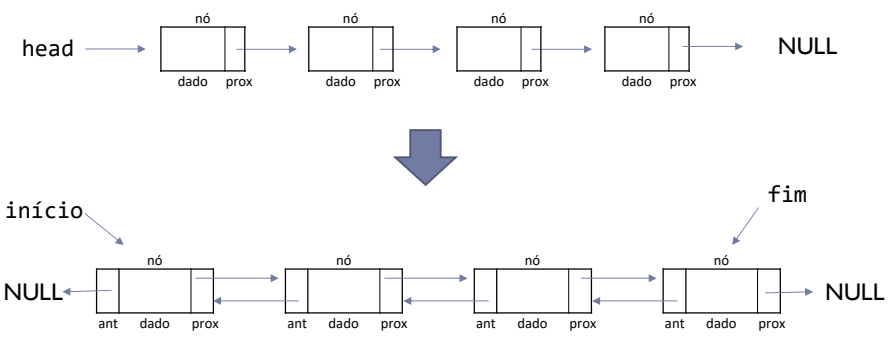
Variações de Listas Encadeadas

Prof. Bruno Travençolo
(com adaptações feitas por Paulo H. R. Gabriel)

91

Listas dinâmica duplamente encadeada

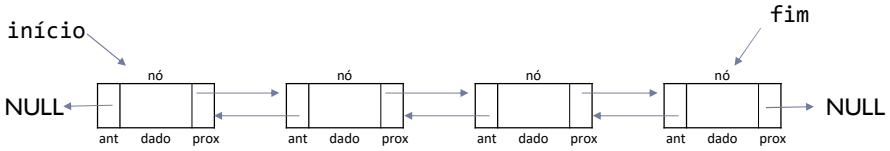
- Quais implicações e fazer a seguinte mudança:



92

Listas dinâmica duplamente encadeada

- Listas dinâmica duplamente encadeada

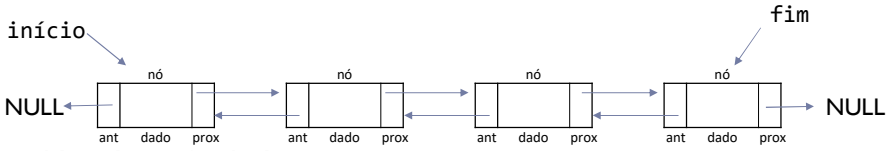


- Quais mudanças teremos nos códigos?
  - Cuidar dos ponteiros para o próximo/anterior
  - Avaliar sempre se está no início, meio ou final da lista nas operações
  - Atualizar a quantidade de elementos (se houver)

93

Listas dinâmica duplamente encadeada

- Listas dinâmica duplamente encadeada

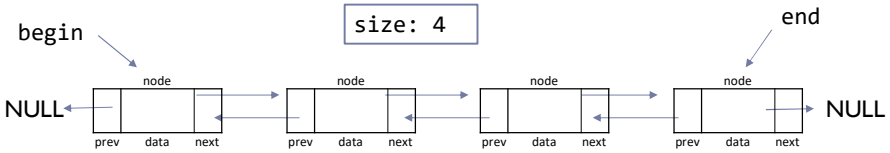


- Usa alocação dinâmica
- Usa acesso encadeado dos elementos (ponteiros 'prox' e 'ant')
- Último elemento tem como sucessor o NULL
- Primeiro elemento tem como antecessor o NULL
- Quais mudanças teremos nos códigos?
  - Cuidar dos ponteiros para o próximo/anterior
  - Avaliar sempre se está no início,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

94

In English...

- Doubly-linked lists

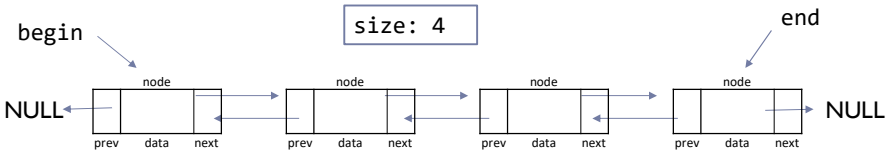


- Node: nó
- Data: dados – estrutura com os dados
- Prev (previous): anterior – ponteiro para o nó anterior
- Next: próximo – ponteiro para o próximo nó
- Begin: início – ponteiro para a cabeça da lista (início)
- End: fim – ponteiro para o último nó
- size: tamanho – tamanho da lista

95

Insere no início (push_front)

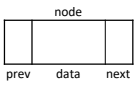
- Criar novo nó



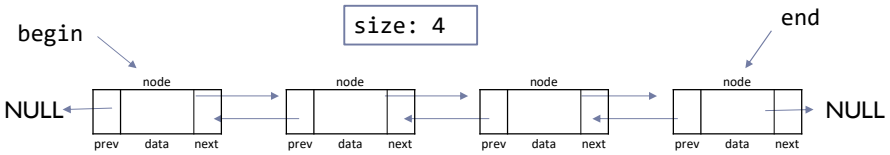
96

Insere no início (push_front)

- Criar novo nó



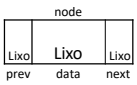

```
DLNode *node;
node = malloc(sizeof(DLNode));
```

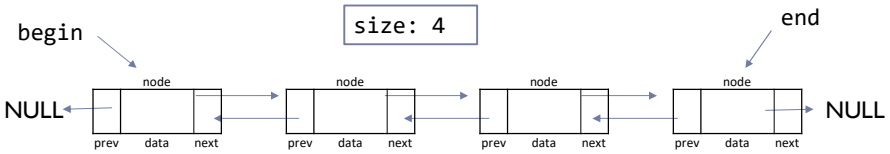


97

Insere no início (push_front)

- Atualizar nó com informações

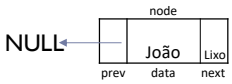




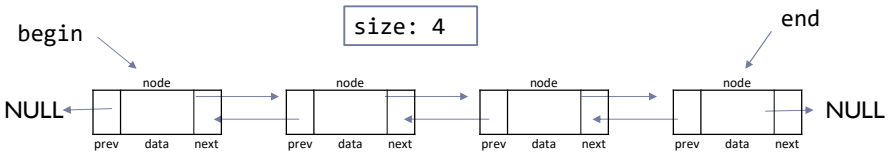
98

Insere 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



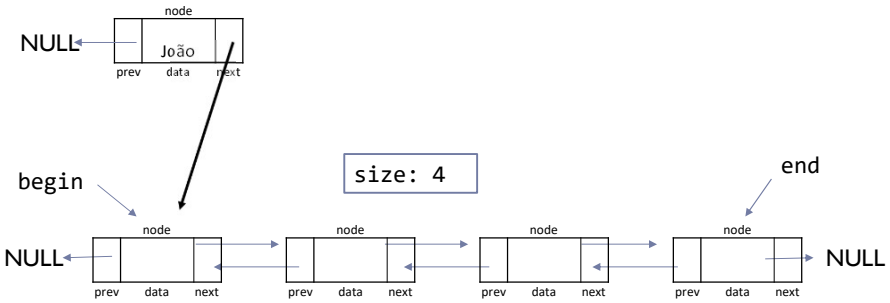

```
node->data = al;
node->prev = NULL;
```



99

Insere no início (push_front)

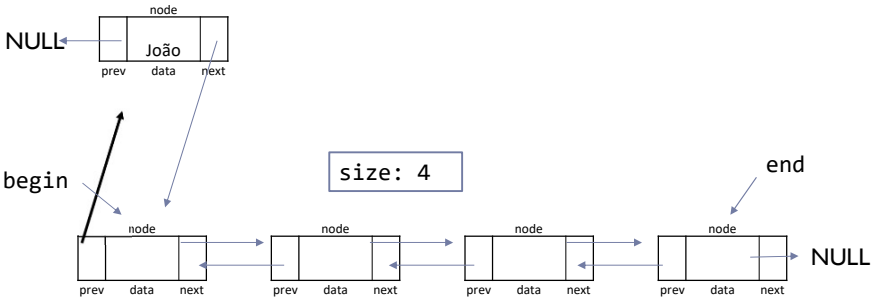
- Insere na lista
  - Inicializar o ponteiro next do novo nó
  - `node->next = li->begin;`



100

Insere no início (push_front)

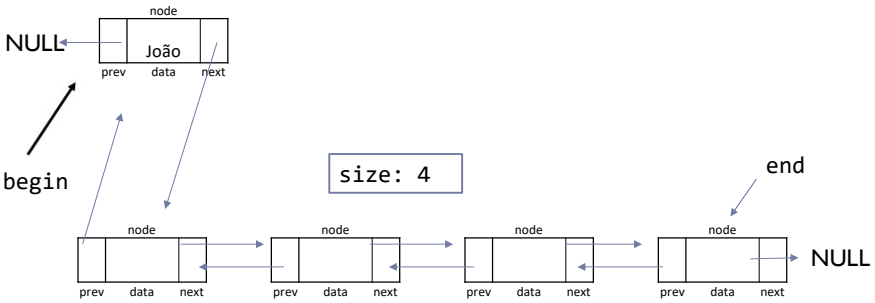
- Insere na lista
  - Antiga cabeça deve apontar para o novo nó
  - `li->begin->prev = node;`



101

Insere no início (push_front)

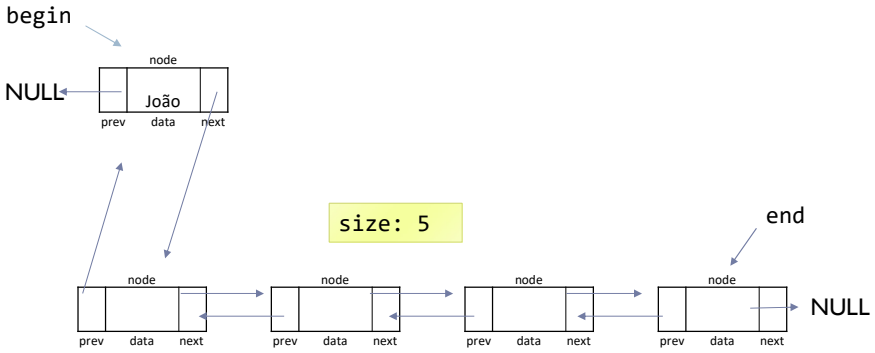
- Insere na lista
  - Atualiza o begin da lista
  - `li->begin = node;`



102

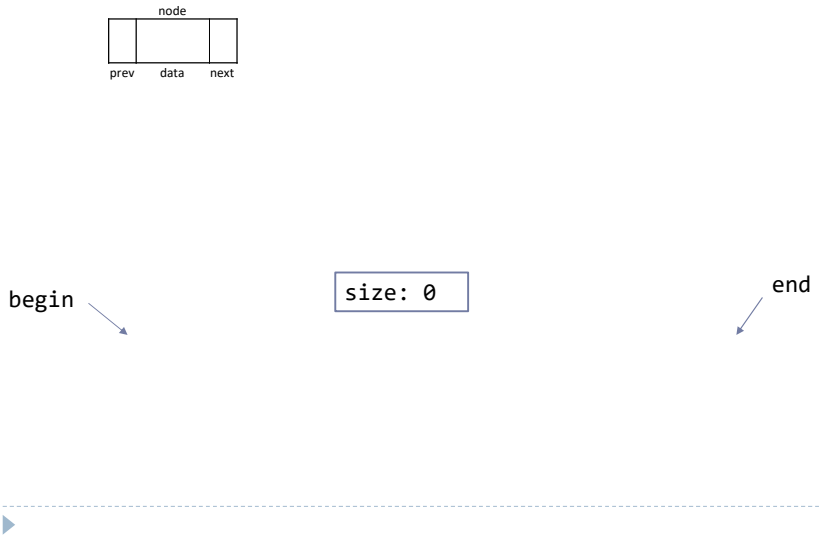
Insere no início (push_front)

- Insere na lista
  - Atualiza a quantidade
  - `li->size = li->size + 1;`



103

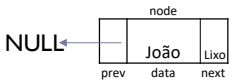
Insere no início (push_front)



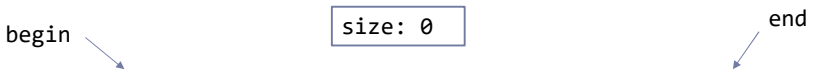
104

Insere 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



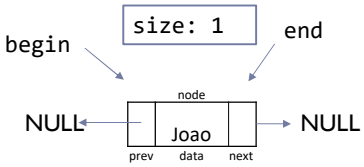

```
node->data = al;
node->prev = NULL;
```



105

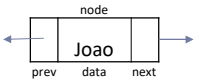
Insere no início (push_front)

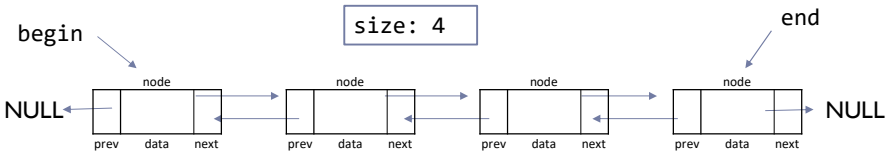
```
if (li->begin == NULL){
        node->next = NULL;
        li->begin = node;
        li->end = node;
        li->size = li->size + 1;
}
```



106

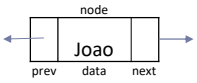
Insert - Posição 6

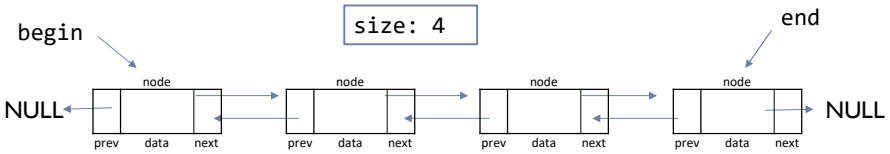




107

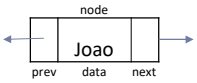
Insert - Posição 3

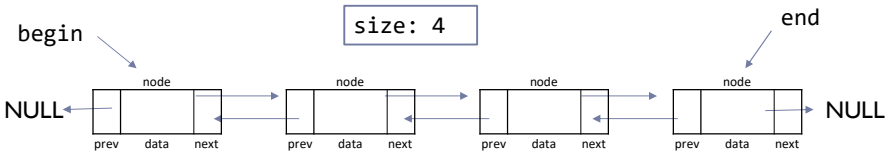




108

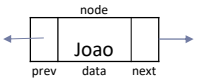
Insert - Posição 1

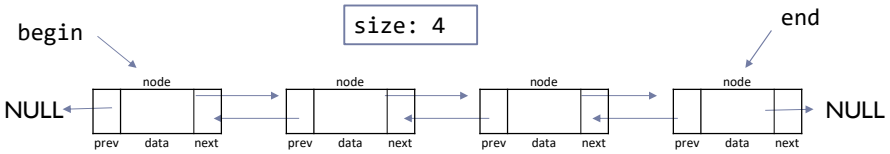




109

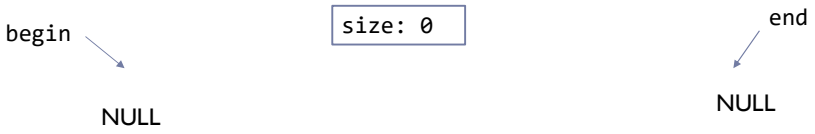
Insert - Posição 5





110

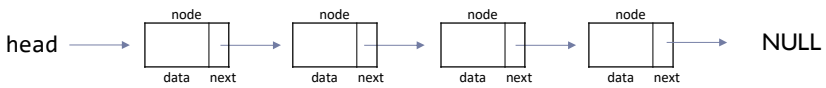
Insert - Posição 1

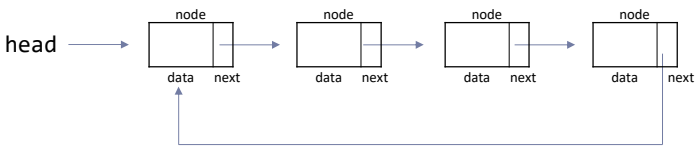


111

Listas circulares

- Quais implicações e fazer a seguinte mudança:

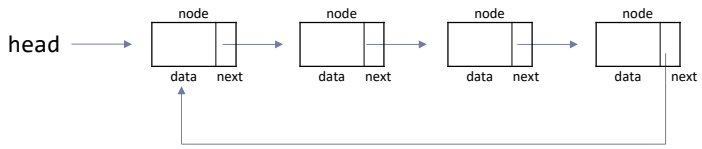




112

Listas circulares

- Lista dinâmica encadeada circular

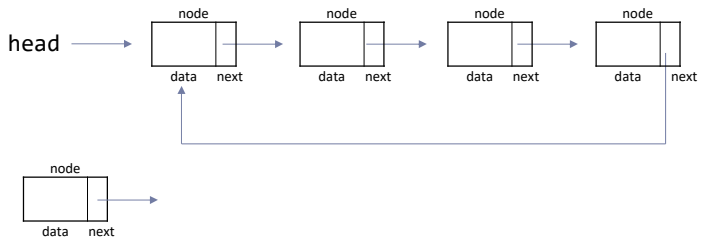


- Usa alocação dinâmica
- Usa acesso encadeado dos elementos (ponteiros 'next')
- Último elemento tem como sucessor o primeiro
- Quais mudanças teremos nos códigos?
  - Tudo que utilizava o NULL para atribuir/consultar o fim da lista

113

Listas circulares - operações

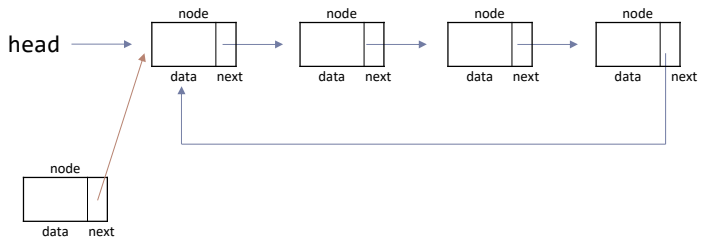
- Inserir no início



114

Listas circulares - operações

- Inserir no início

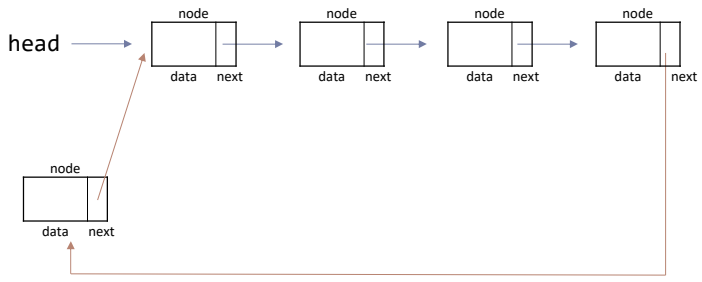


- Novo nó aponta para a cabeça

115

Listas circulares - operações

- Inserir no início

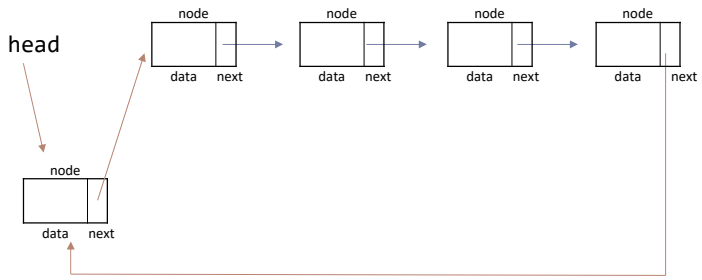


- Último nó aponta para o novo nó
  - Envolver percorrer toda a lista!

116

Listas circulares - operações

- Inserir no início

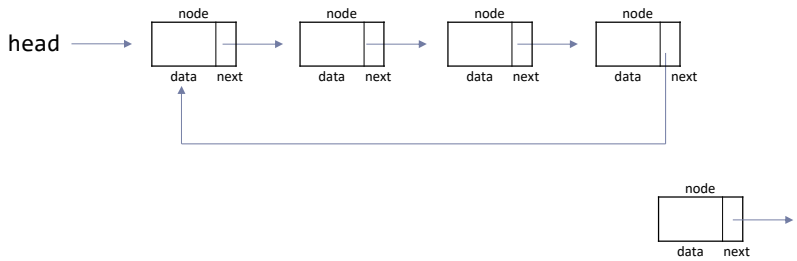


- Cabeça aponta para o novo nó

117

Listas circulares - operações

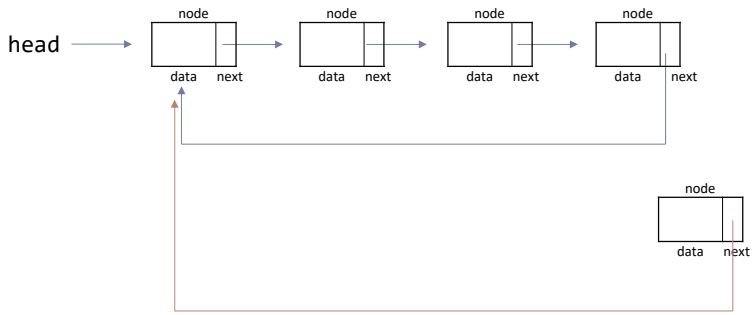
- Inserir no final



118

Listas circulares - operações

- Inserir no final

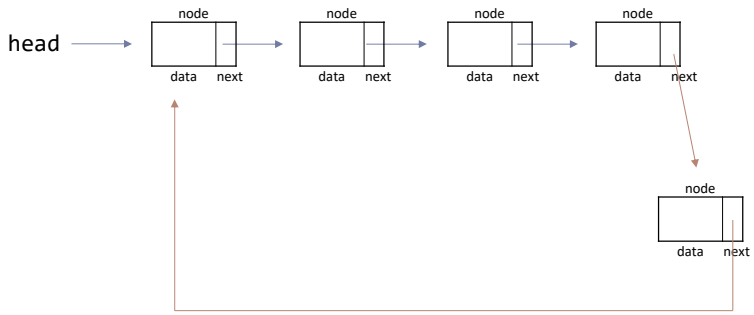


- Faz o novo nó apontar para a cabeça

119

Listas circulares - operações

- Inserir no final



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

120

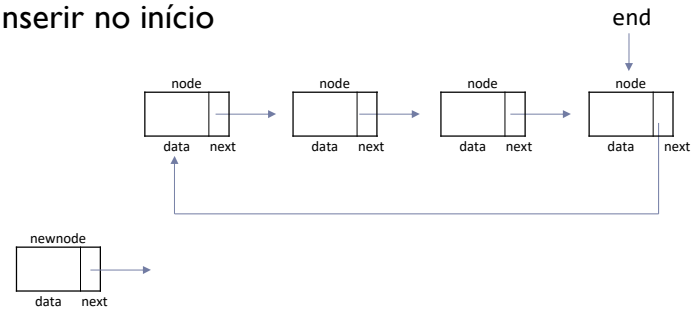
Listas circulares - operações

- 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

121

Listas circulares - operações

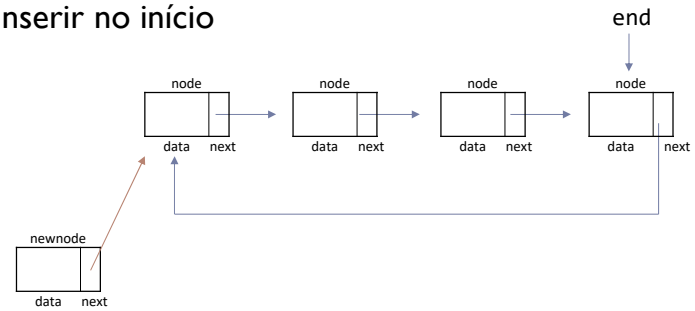
- Inserir no início



122

Listas circulares - operações

- Inserir no início



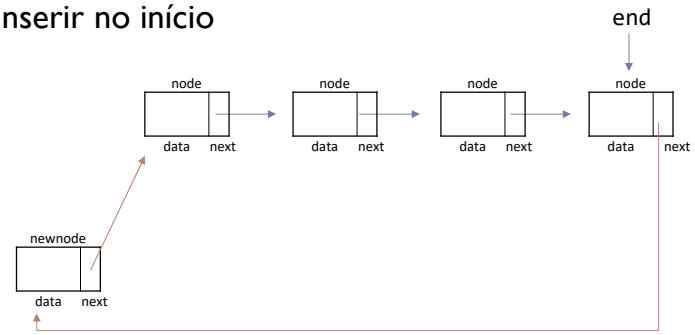
- Faz o *next* do novo nó apontar para o início da lista

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

123

Listas circulares - operações

- Inserir no início



- Faz o próximo do final apontar para o novo nó

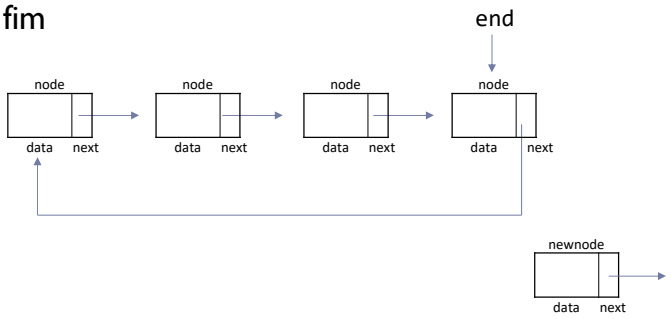
```
end->next = newnode;
```

- Pronto! Sem necessidade de percorrer a lista

124

Listas circulares - operações

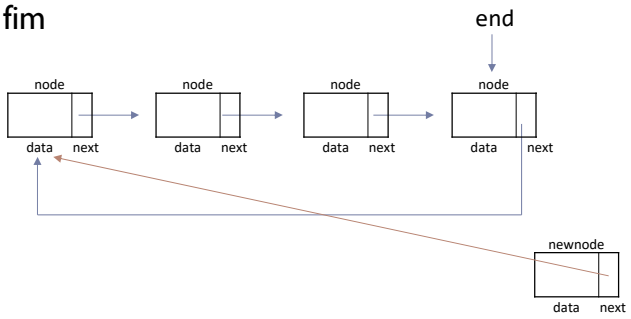
- Inserir no fim



125

Listas circulares - operações

- Inserir no fim



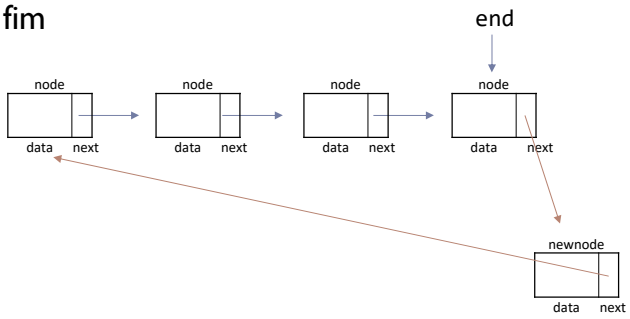
- Faz o *next* do novo nó apontar para o início da lista

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

126

Listas circulares - operações

- Inserir no fim



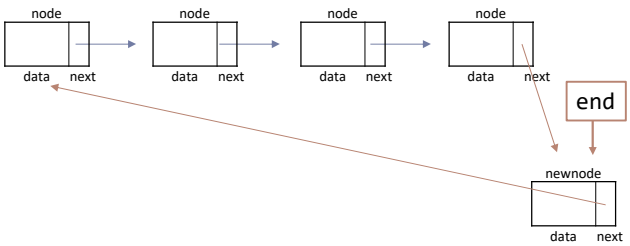
- Faz o *next* do *end* apontar para o novo nó

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

127

Listas circulares - operações

- Inserir no fim



- Altera o ponteiro '*end*'

```
end = newnode;
```

128

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;
};
```

129

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

130

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;
};
```

131

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;
}
```

132

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**`

133

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**`

134

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

135

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

136

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

137

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

138

Listas sem nó descritor

- Como funciona o ponteiro de ponteiro

```c
// 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;
}
```

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

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

139

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

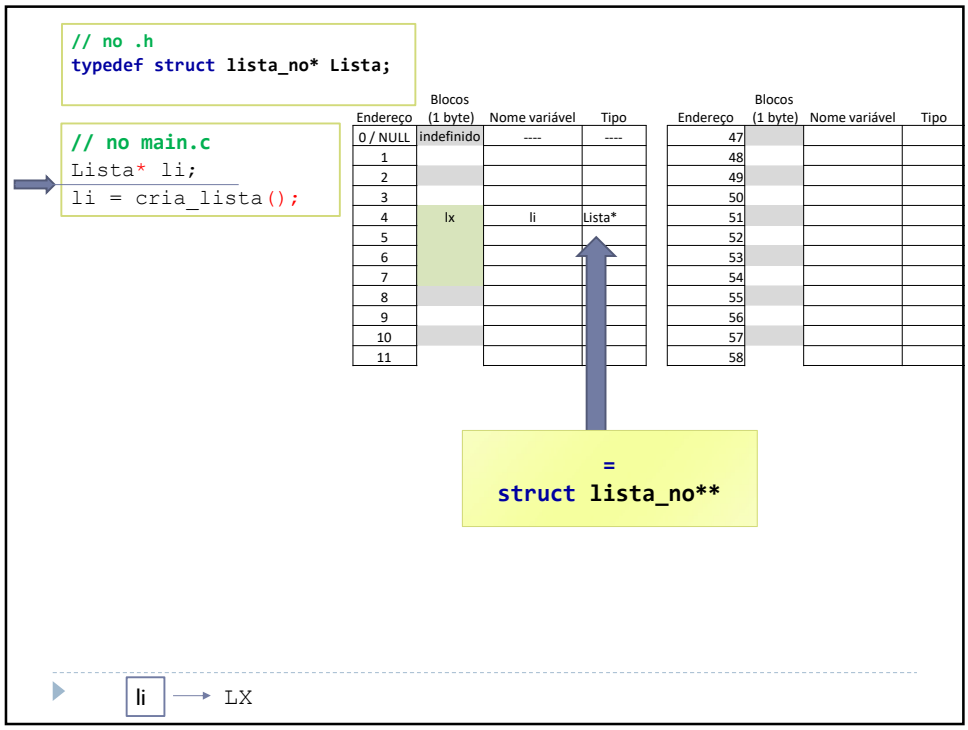
```c
// no main.c
Lista* li;
li = cria_lista();
```

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

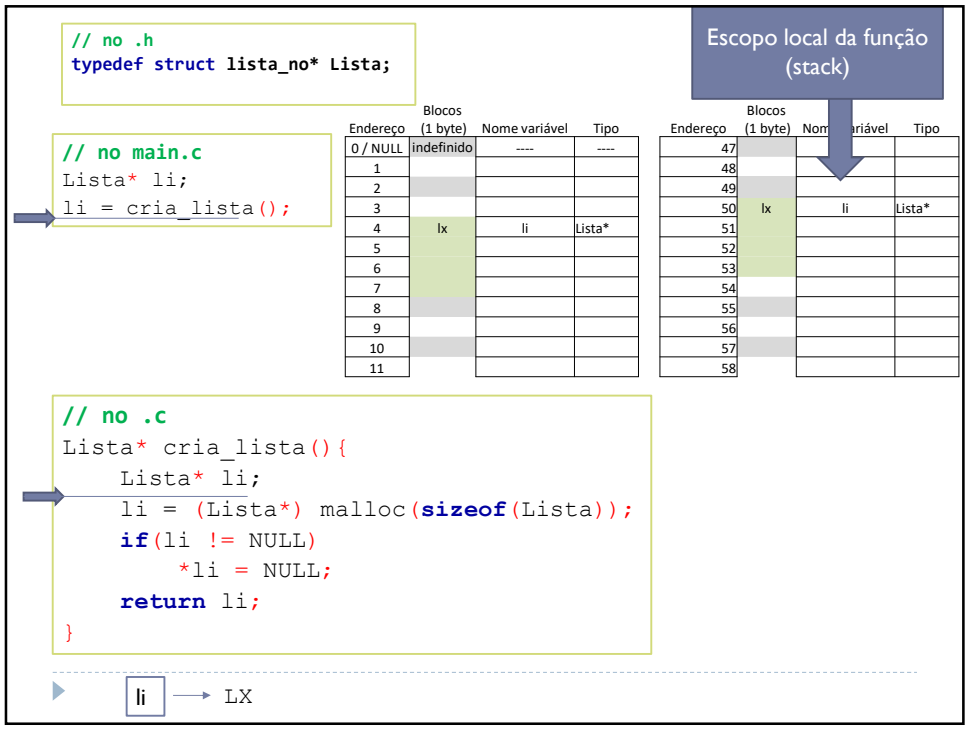
Endereço	Blocos (1 byte)	Nome variável	Tipo
47			
48			
49			
50			
51			
52			
53			
54			
55			
56			
57			
58			

li → LX

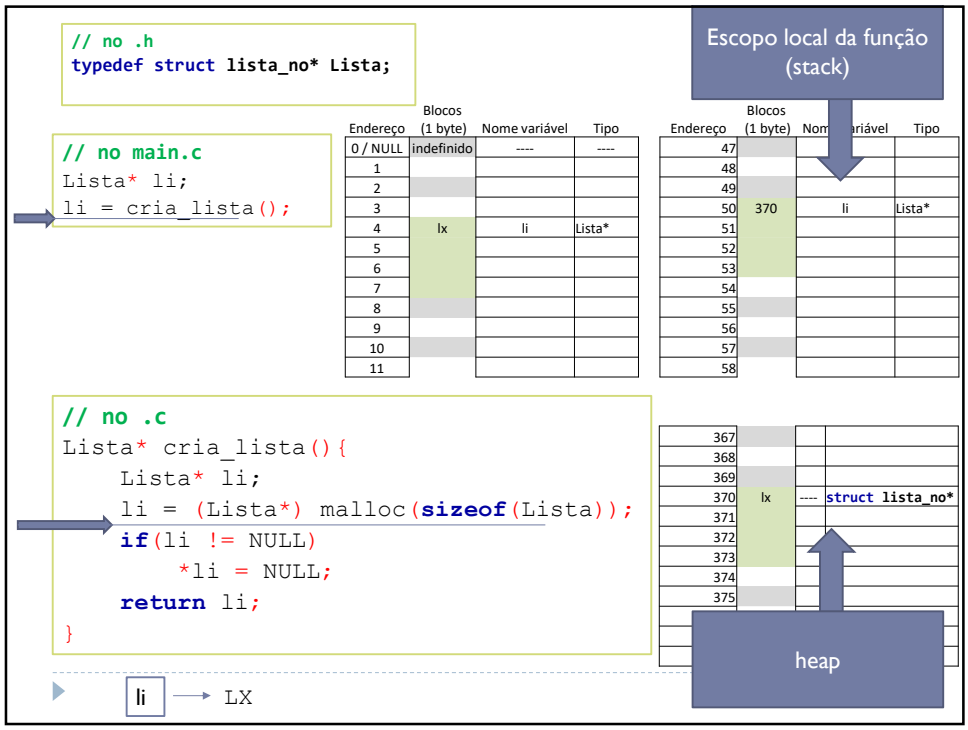
140


// 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
lx
li
Lista*
= struct lista_no**
li → LX

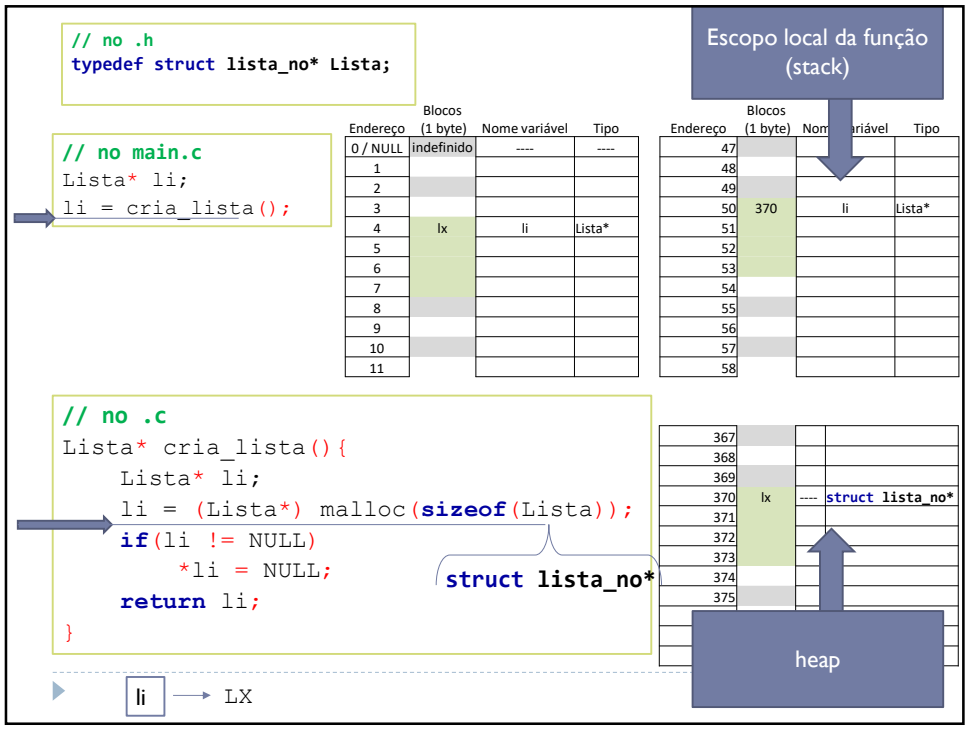
141


// no .h
typedef struct lista_no* Lista;
Escopo local da função (stack)
// no main.c
Lista* li;
li = cria_lista();
Endereço
Blocos (1 byte)
Nome variável
Tipo
0 / NULL
indefinido
lx
li
Lista*
// no .c
Lista* cria_lista(){
Lista* li;
li = (Lista*) malloc(sizeof(Lista));
if(li != NULL)
*li = NULL;
return li;
}
li → LX

142


// no .h
typedef struct lista_no* Lista;
// no main.c
Lista* li;
li = cria_lista();
// no .c
Lista* cria_lista(){
Lista* li;
li = (Lista*) malloc(sizeof(Lista));
if(li != NULL)
*li = NULL;
return li;
}
Escopo local da função (stack)
Blocos (1 byte)
Endereço
Nome variável
Tipo
0 / NULL
indefinido
4
lx
li
Lista*
370
struct lista_no*
heap
li
LX

143


// no .h
typedef struct lista_no* Lista;
// no main.c
Lista* li;
li = cria_lista();
// no .c
Lista* cria_lista(){
Lista* li;
li = (Lista*) malloc(sizeof(Lista));
if(li != NULL)
*li = NULL;
return li;
}
struct lista_no*
Escopo local da função (stack)
Blocos (1 byte)
Endereço
Nome variável
Tipo
0 / NULL
indefinido
4
lx
li
Lista*
370
struct lista_no*
heap
li
LX

144

```
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;
}
```

145

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

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

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

Escopo local da função (stack)

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

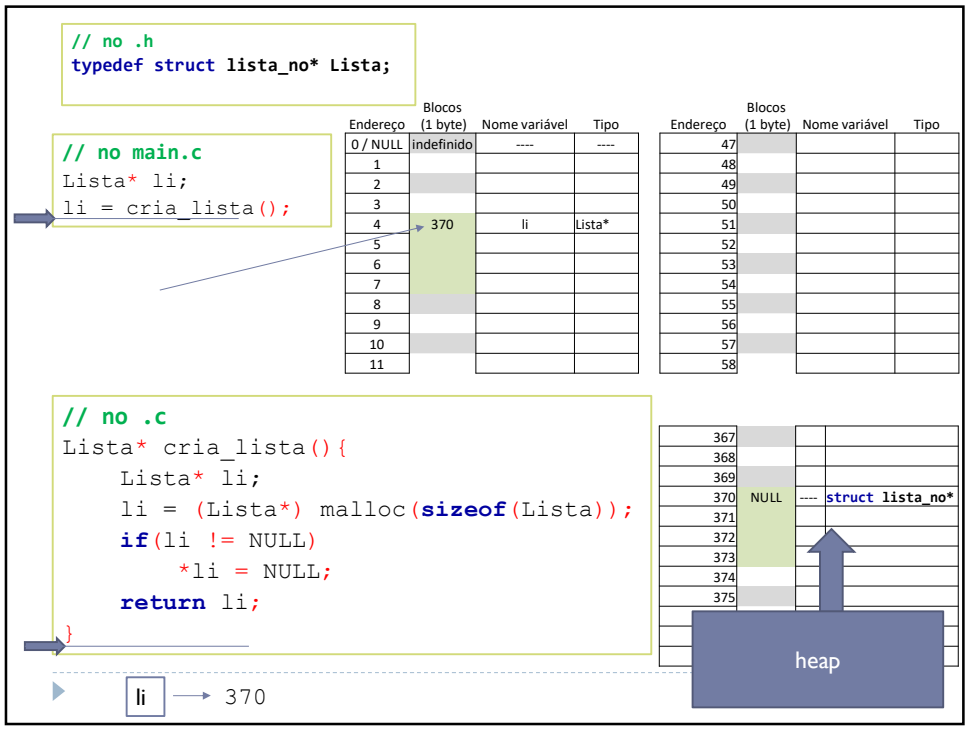
Endereço	Blocos (1 byte)	Nom…riável	Tipo
47			
48			
49			
50	370	li	Lista*
51			
52			
53			
54			
55			
56			
57			
58			

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

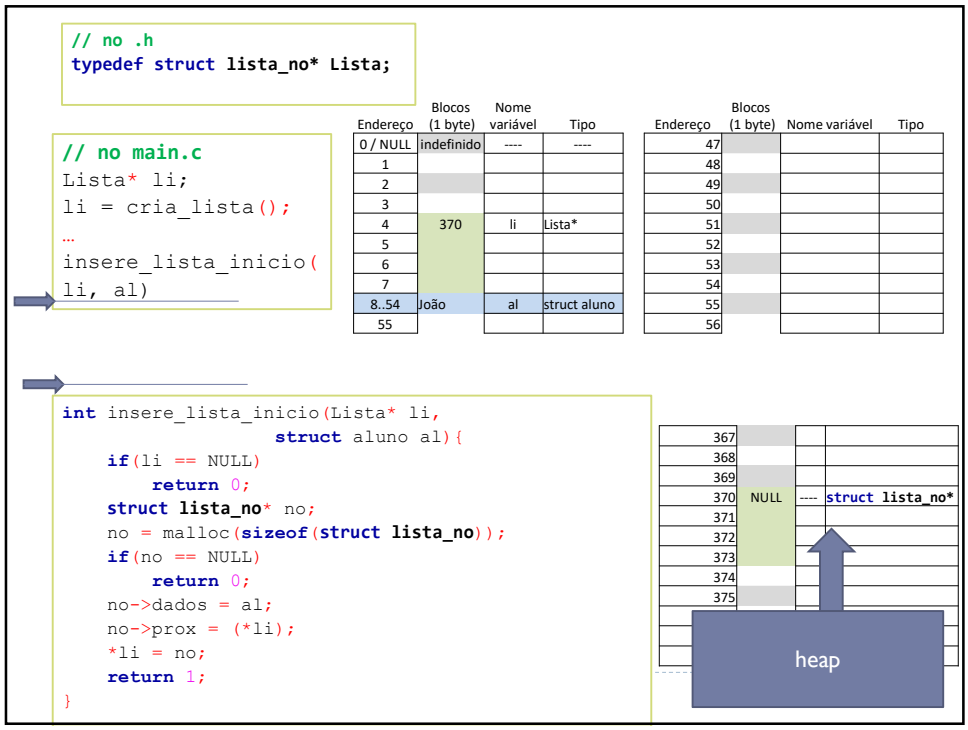
heap

li → LX

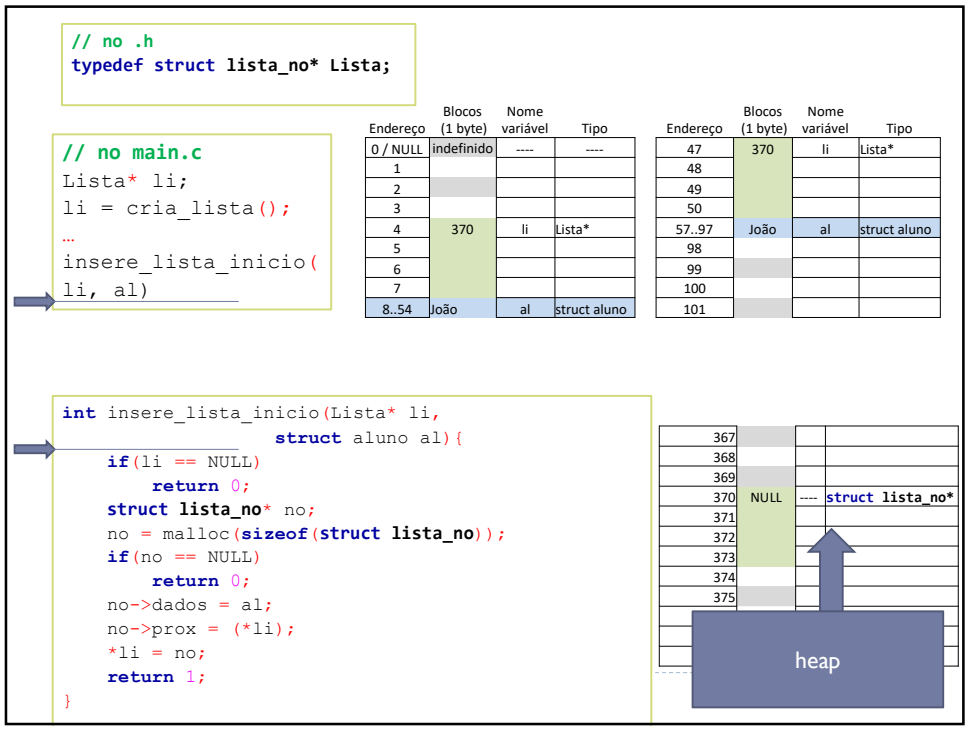
146


// no .h
typedef struct lista_no* Lista;
// no main.c
Lista* li;
li = cria_lista();
// no .c
Lista* cria_lista(){
Lista* li;
li = (Lista*) malloc(sizeof(Lista));
if(li != NULL)
*li = NULL;
return li;
}
Blocos (1 byte)
Endereço
Nome variável
Tipo
0 / NULL
indefinido
370
li
Lista*
NULL
struct lista_no*
heap
li
370

147

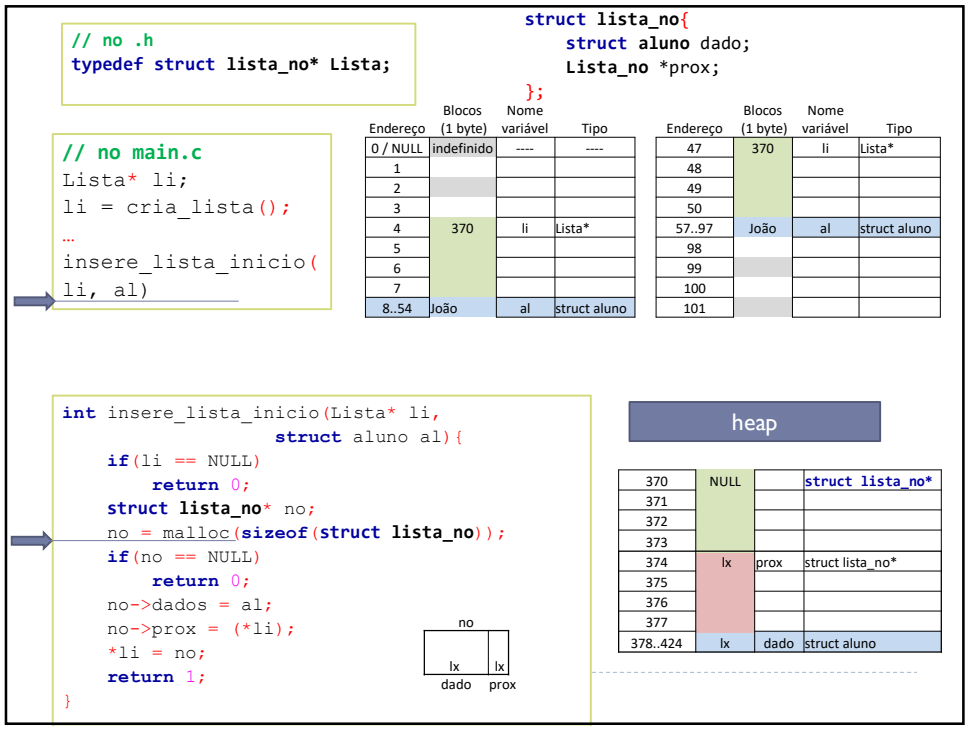

// no .h
typedef struct lista_no* Lista;
// no main.c
Lista* li;
li = cria_lista();
…
insere_lista_inicio(
li, al)
Blocos (1 byte)
Endereço
Nome variável
Tipo
0 / NULL
indefinido
370
li
Lista*
8..54
João
al
struct aluno
int insere_lista_inicio(Lista* li,
struct aluno al){
if(li == NULL)
return 0;
struct lista_no* no;
no = malloc(sizeof(struct lista_no));
if(no == NULL)
return 0;
no->dados = al;
no->prox = (*li);
*li = no;
return 1;
}
NULL
struct lista_no*
heap

148


// no .h
typedef struct lista_no* Lista;
// no main.c
Lista* li;
li = cria_lista();
…
insere_lista_inicio(
li, al)
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;
}
Endereço
Blocos (1 byte)
Nome variável
Tipo
0 / NULL
indefinido

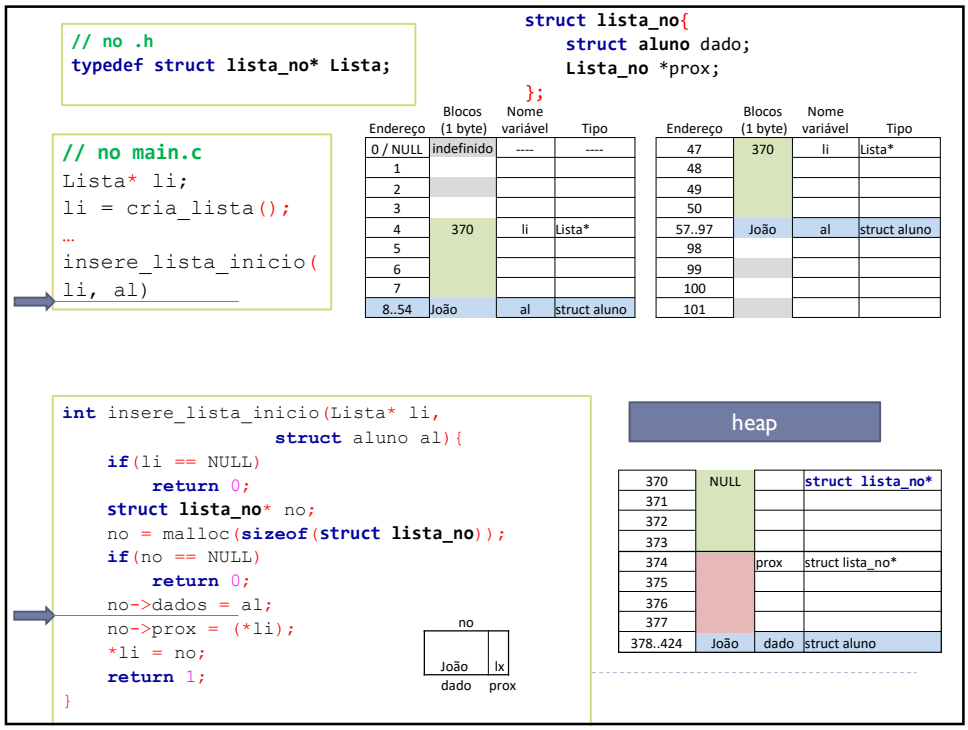
1
2
3
4
370
li
Lista*
5
6
7
8..54
João
al
struct aluno
47
48
49
50
57..97
98
99
100
101
367
368
369
370
NULL
struct lista_no*
371
372
373
374
375
heap

149


// 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)
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
lx
lx
dado
prox
heap
Endereço
Blocos (1 byte)
Nome variável
Tipo
0 / NULL
indefinido

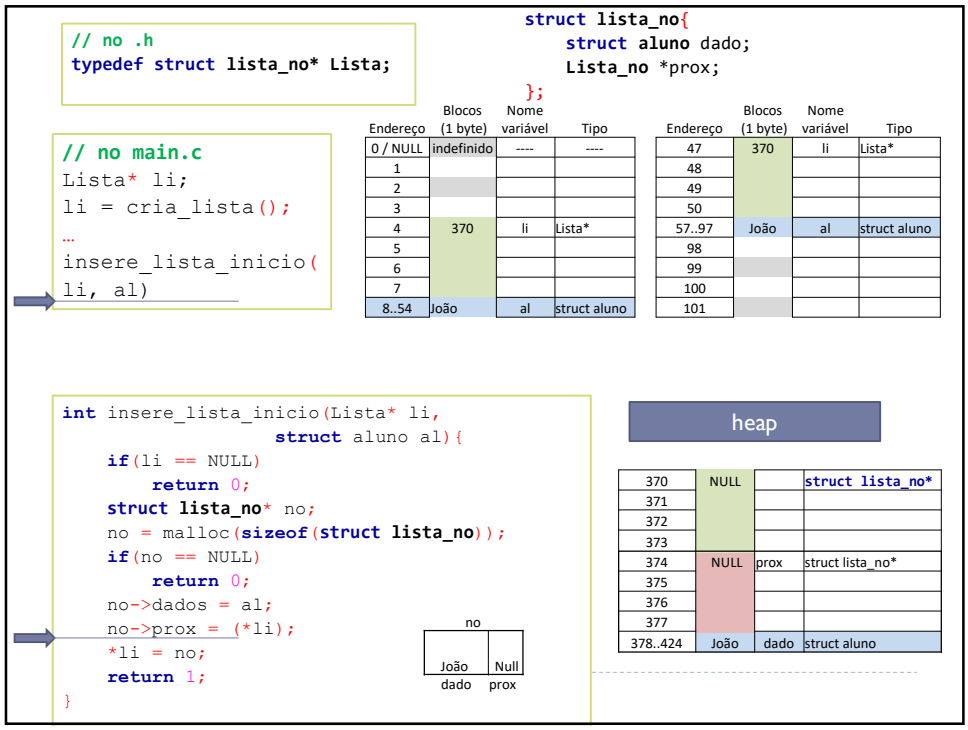
1
2
3
4
370
li
Lista*
5
6
7
8..54
João
al
struct aluno
47
48
49
50
57..97
98
99
100
101
370
NULL
struct lista_no*
371
372
373
374
lx
prox
struct lista_no*
375
376
377
378..424
lx
dado
struct aluno

150


// no .h
typedef struct lista_no* Lista;
// no main.c
Lista* li;
li = cria_lista();
…
insere_lista_inicio(
li, al)
struct lista_no{
struct aluno dado;
Lista_no *prox;
};
Endereço
Blocos (1 byte)
Nome variável
Tipo
0 / NULL
indefinido

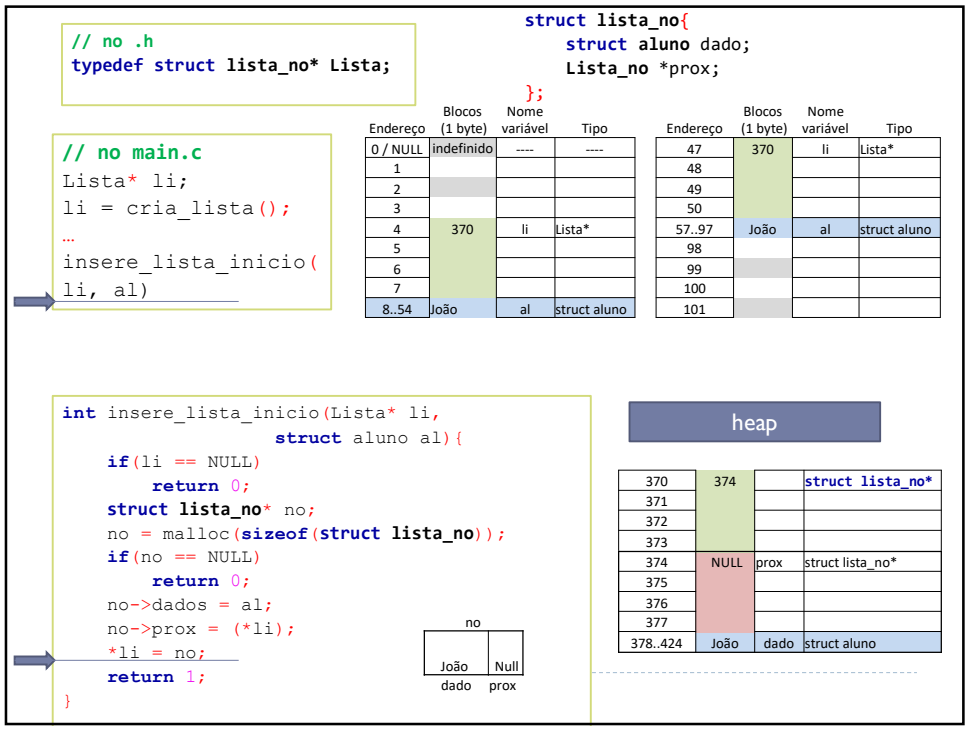
1
2
3
4
370
li
Lista*
5
6
7
8..54
João
al
struct aluno
47
48
49
50
57..97
98
99
100
101
int insere_lista_inicio(Lista* li,
struct aluno al){
if(li == NULL)
return 0;
struct lista_no* no;
no = malloc(sizeof(struct lista_no));
if(no == NULL)
return 0;
no->dados = al;
no->prox = (*li);
*li = no;
return 1;
}
no
João
dado
lx
prox
heap
370
NULL
struct lista_no*
371
372
373
374
prox
struct lista_no*
375
376
377
378..424
João
dado
struct aluno

151


// no .h
typedef struct lista_no* Lista;
// no main.c
Lista* li;
li = cria_lista();
…
insere_lista_inicio(
li, al)
struct lista_no{
struct aluno dado;
Lista_no *prox;
};
Endereço
Blocos (1 byte)
Nome variável
Tipo
0 / NULL
indefinido

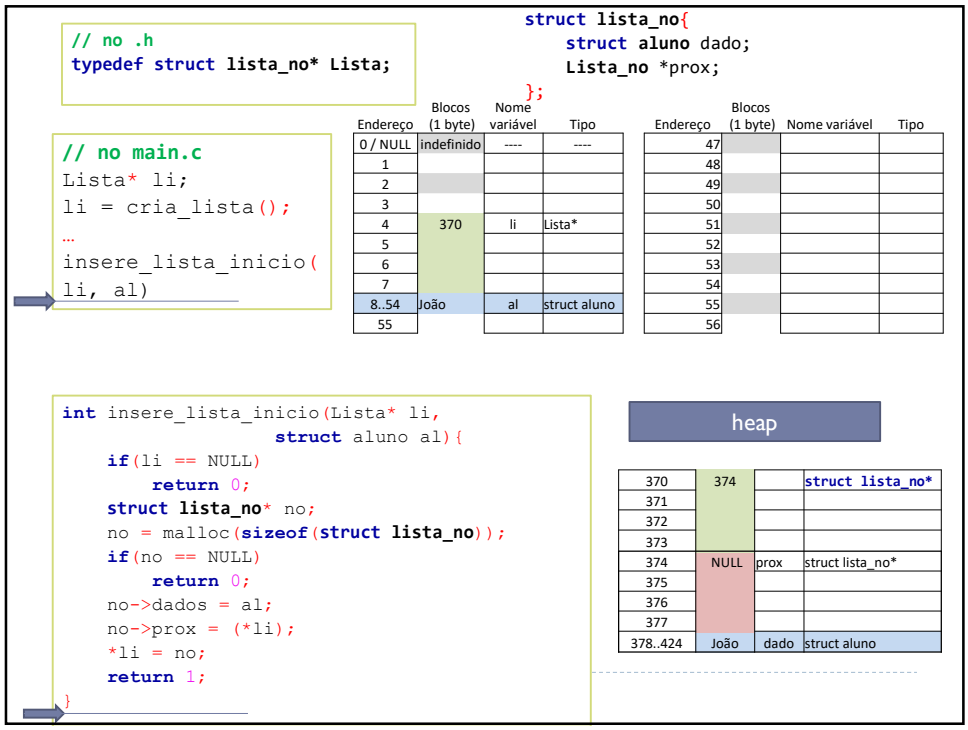
1
2
3
4
370
li
Lista*
5
6
7
8..54
João
al
struct aluno
47
48
49
50
57..97
98
99
100
101
int insere_lista_inicio(Lista* li,
struct aluno al){
if(li == NULL)
return 0;
struct lista_no* no;
no = malloc(sizeof(struct lista_no));
if(no == NULL)
return 0;
no->dados = al;
no->prox = (*li);
*li = no;
return 1;
}
no
João
dado
Null
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

152


// no .h
typedef struct lista_no* Lista;
// no main.c
Lista* li;
li = cria_lista();
…
insere_lista_inicio(
li, al)
struct lista_no{
struct aluno dado;
Lista_no *prox;
};
Endereço
Blocos (1 byte)
Nome variável
Tipo
0 / NULL
indefinido

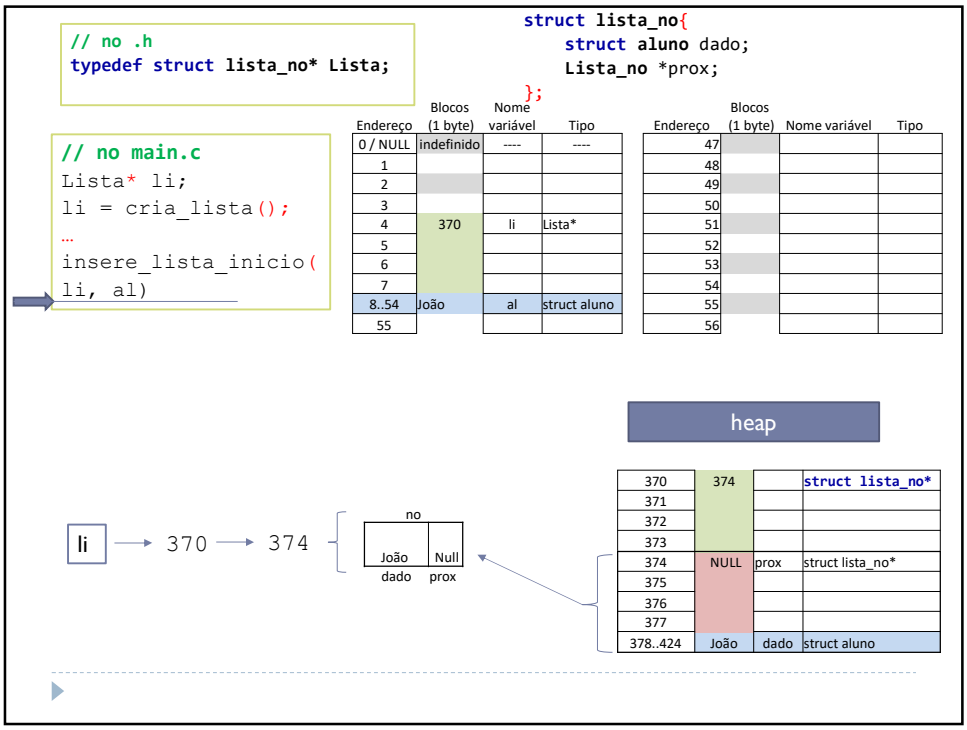
4
370
li
Lista*
8..54
João
al
struct aluno
47
57..97
98
99
100
101
int insere_lista_inicio(Lista* li,
struct aluno al){
if(li == NULL)
return 0;
struct lista_no* no;
no = malloc(sizeof(struct lista_no));
if(no == NULL)
return 0;
no->dados = al;
no->prox = (*li);
*li = no;
return 1;
}
no
João
Null
dado
prox
heap
370
374
struct lista_no*
371
372
373
NULL
prox
struct lista_no*
375
376
377
378..424
dado

153


// no .h
typedef struct lista_no* Lista;
// no main.c
Lista* li;
li = cria_lista();
…
insere_lista_inicio(
li, al)
struct lista_no{
struct aluno dado;
Lista_no *prox;
};
Endereço
Blocos (1 byte)
Nome variável
Tipo
0 / NULL
indefinido

4
370
li
Lista*
8..54
João
al
struct aluno
55
47
51
52
53
54
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
NULL
prox
struct lista_no*
375
376
377
378..424
João
dado
struct aluno

154


// no .h
typedef struct lista_no* Lista;
// no main.c
Lista* li;
li = cria_lista();
…
insere_lista_inicio(
li, al)
struct lista_no{
struct aluno dado;
Lista_no *prox;
};
Endereço
Blocos (1 byte)
Nome variável
Tipo
0 / NULL
indefinido

4
370
li
Lista*
8..54
João
al
struct aluno
55
47
48
49
50
51
52
53
54
56
heap
li
370
374
no
João
dado
Null
prox
struct lista_no*
NULL
struct lista_no*
378..424
João
dado
struct aluno

155