

Estrutura de Dados

**Técnicas de Encadeamento
(Encadeamento Duplo)**

**Prof. Luiz Gustavo Almeida Martins, com
adaptação Profa. Gina M. B. Oliveira**

Introdução

- Diferentes técnicas de encadeamento podem ser aplicadas a fim de **gerar algoritmos mais simples e/ou eficientes**

Introdução

- Diferentes técnicas de encadeamento podem ser aplicadas a fim de **gerar algoritmos mais simples e/ou eficientes**
- **Técnicas mais usuais:**
 - Uso do nó cabeçalho
 - Encadeamento circular
 - Encadeamento duplo

Introdução

- Diferentes técnicas de encadeamento podem ser aplicadas a fim de **gerar algoritmos mais simples e/ou eficientes**
- **Técnicas mais usuais:**
 - Uso do nó cabeçalho
 - Encadeamento circular
 - **Encadeamento duplo**

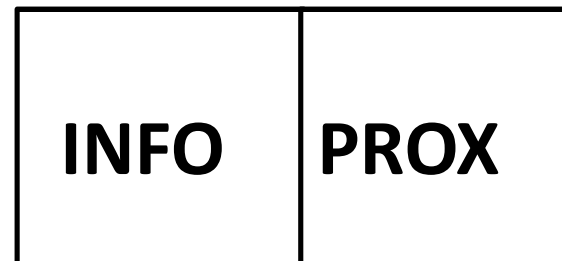
Encadeamento Duplo

- Cada nó conhece seu **sucessor** e **antecessor** na lista

Encadeamento Duplo

- Cada nó conhece seu **sucessor** e **antecessor** na lista
- Muda a **estrutura de representação do nó**:
 - Campo **info** guarda o **valor do elemento**
 - Campo **prox** aponta para o **sucessor do nó**

Nó =



Encadeamento Duplo

- Cada nó conhece seu **sucessor** e **antecessor** na lista
- Muda a **estrutura de representação do nó**:
 - Campo **info** guarda o **valor do elemento**
 - Campo **prox** aponta para o **sucessor do nó**
 - Campo **ant** aponta para o **antecessor do nó**



Lista Duplamente Encadeada

- Lista vazia (Ex: $L = \{ \}$)

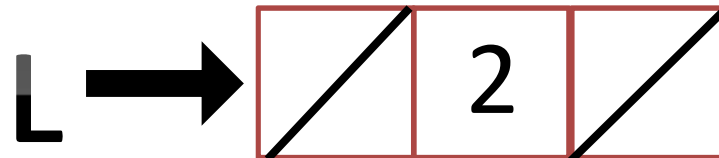
L → **NULL**

Lista Duplamente Encadeada

- Lista vazia (Ex: $L = \{ \}$)

$L \rightarrow \text{NULL}$

- Lista com 1 único elemento (Ex: $L = \{2\}$)



Lista Duplamente Encadeada

- Lista com mais de um elemento:

Ex: $L = \{ 5, 3, 9 \}$



Lista Duplamente Encadeada

- Lista com mais de um elemento:

Ex: $L = \{ 5, 3, 9 \}$



- A **LISTA** ainda é um **ponteiro** para a **estrutura nó-duplo**

Lista Duplamente Encadeada

- **Facilita:**
 - A construção de operações que necessitam **percorrer a lista nas duas direções**

Lista Duplamente Encadeada

- **Facilita:**
 - A construção de operações que necessitam **percorrer a lista nas duas direções**
 - Procurar por um determinado elemento
 - Analisa o campo *info* do **nó atual** e não do sucessor

Lista Duplamente Encadeada

- **Facilita:**
 - A construção de operações que necessitam **percorrer a lista nas duas direções**
 - Procurar por um determinado elemento
 - Analisa o campo *info* do **nó atual** e não do sucessor
 - Remoção utiliza um **ÚNICO** ponteiro auxiliar

Lista Duplamente Encadeada

- **Facilita:**
 - A construção de operações que necessitam **percorrer a lista nas duas direções**
 - Procurar por um determinado elemento
 - Analisa o campo *info* do **nó atual** e não do sucessor
 - Remoção utiliza um **ÚNICO** ponteiro auxiliar
- Requer **mais memória e código com mais linhas** (controle dos ponteiros)

Estrutura de Representação em C

- Declaração da estrutura nó inteiro no **lista.c**:

```
struct no {  
    int info;  
    struct no * prox;  
    struct no * ant;  
};
```

- Definição do tipo de dado lista no **lista.h**:

```
typedef struct no * Lista;
```


Encadeamento Duplo

- Operações *cria_lista* e *lista_vazia* também são **IDÊNTICAS** ao encadeamento simples

Encadeamento Duplo

- Operações *cria_lista* e *lista_vazia* também são **IDÊNTICAS** ao encadeamento simples

```
Lista cria_lista() {  
    return NULL;  
}
```

Encadeamento Duplo

- Operações *cria_lista* e *lista_vazia* também são **IDÊNTICAS** ao encadeamento simples

```
Lista cria_lista() {  
    return NULL;  
}
```

```
int lista_vazia (Lista lst) {  
    if (lst == NULL)  
        return 1;  
    else  
        return 0;  
}
```

Encadeamento Duplo

- Analisaremos as operações de **inserção** e **remoção**

Encadeamento Duplo

- Analisaremos as operações de **inserção** e **remoção**
- **TAD lista não-ordenada:**
 - **Inserir elemento**
 - Melhor local de inserção é no início da lista
 - Evita percorrimento da lista

Encadeamento Duplo

- Analisaremos as operações de **inserção** e **remoção**
- **TAD lista não-ordenada:**
 - **Inserir elemento**
 - Melhor local de inserção é no início da lista
 - Evita percorrimeto da lista
 - **Remove elemento**
 - Encontrar o elemento envolve percorrimeto
 - Deve percorrer até o final da lista para determinar que o elemento não existe

Inserir Elemento

- Existem **2 cenários** possíveis:
 - Lista vazia
 - Lista com elementos

Inserir Elemento

- Lista vazia:

Ex: inserir 5

L → **NULL**

Inserir Elemento

- **Lista vazia:**
 - Aloca o novo nó

Ex: inserir 5



Inserir Elemento

- **Lista vazia:**
 - Aloca o novo nó
 - Preenche os campos do novo nó:
 - Campo ***info*** com o valor do elemento

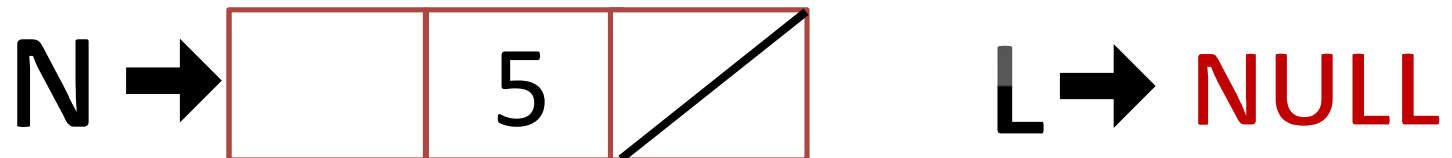
Ex: inserir 5



Inserir Elemento

- **Lista vazia:**
 - Aloca o novo nó
 - Preenche os campos do novo nó:
 - Campo ***info*** com o valor do elemento
 - Campo ***prox*** com ***NULL***

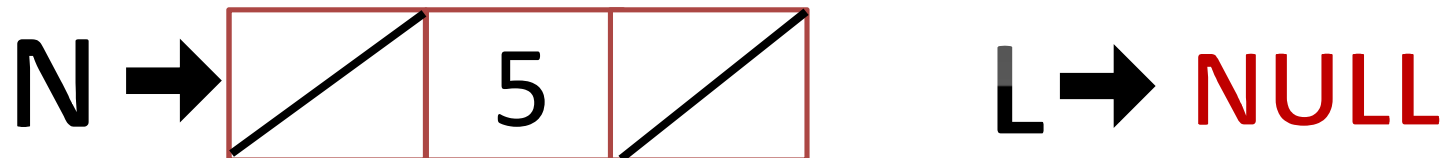
Ex: inserir 5



Inserir Elemento

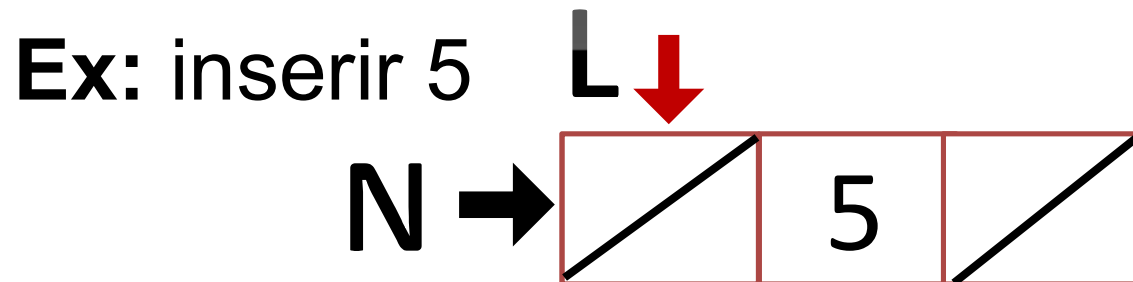
- **Lista vazia:**
 - Aloca o novo nó
 - Preenche os campos do novo nó:
 - Campo ***info*** com o valor do elemento
 - Campo ***prox*** com ***NULL***
 - Campo ***ant*** com ***NULL***

Ex: inserir 5



Inserir Elemento

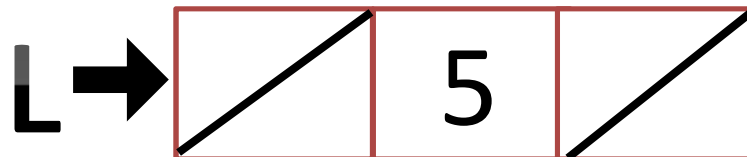
- **Lista vazia:**
 - Aloca o novo nó
 - Preenche os campos do novo nó:
 - Campo ***info*** com o valor do elemento
 - Campo ***prox*** com ***NULL***
 - Campo ***ant*** com ***NULL***
 - Faz a lista apontar para o **novo nó**



Inserir Elemento

- **Lista vazia:**
 - Aloca o novo nó
 - Preenche os campos do novo nó:
 - Campo **info** com o valor do elemento
 - Campo **prox** com **NULL**
 - Campo **ant** com **NULL**
 - Faz a lista apontar para o **novo nó**

Ex: inserir 5

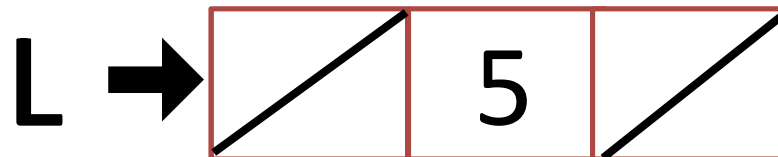


retorna 1

Inserir Elemento

- Lista com elementos:

Ex: inserir 2



Inserir Elemento

- Lista com elementos:
 - Aloca o novo nó

Ex: inserir 2



Inserir Elemento

- Lista com elementos:
 - Aloca o novo nó
 - Preenche os campos do novo nó:
 - Campo *info* com o valor do elemento

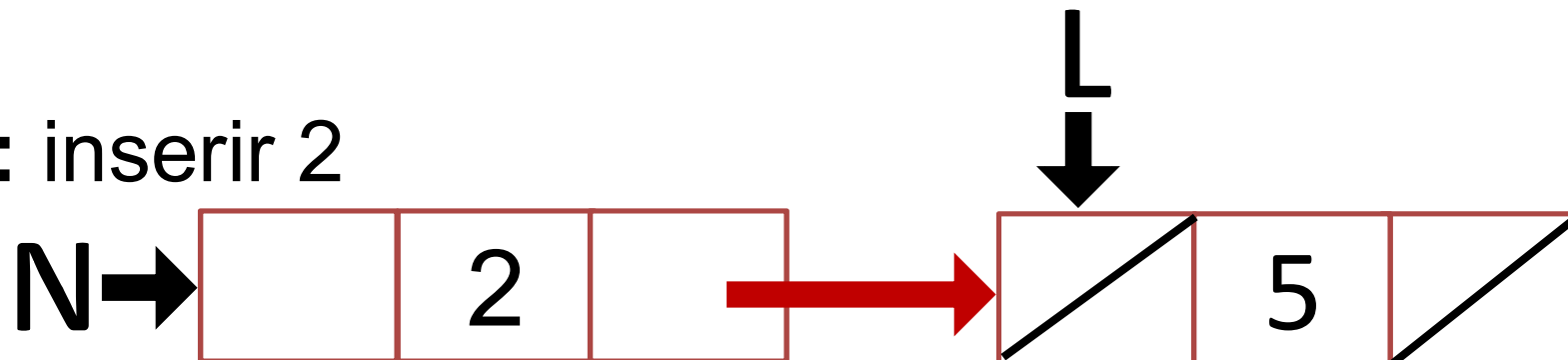
Ex: inserir 2



Inserir Elemento

- Lista com elementos:
 - Aloca o novo nó
 - Preenche os campos do novo nó:
 - Campo **info** com o valor do elemento
 - Campo **prox** aponta para o 1º nó ($N \rightarrow prox = L$)

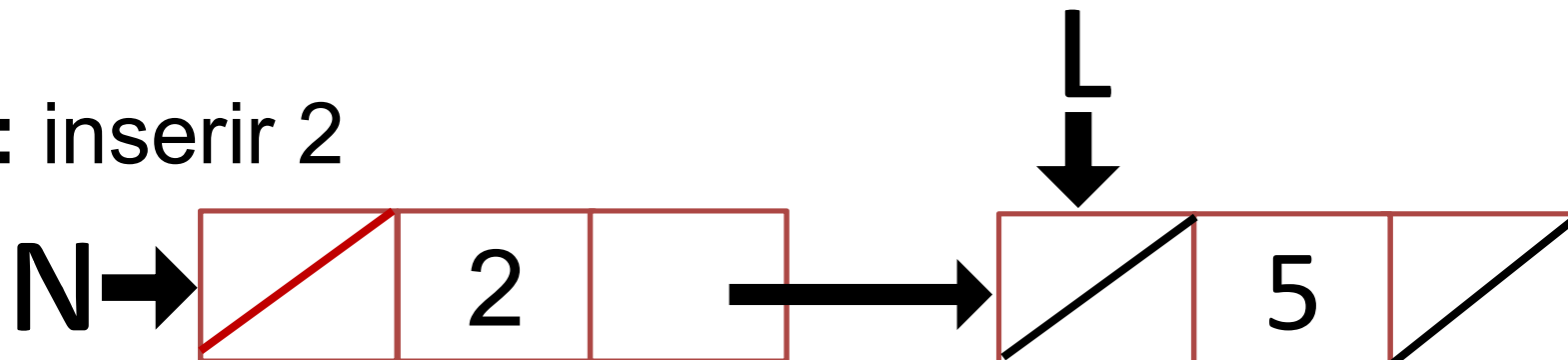
Ex: inserir 2



Inserir Elemento

- Lista com elementos:
 - Aloca o novo nó
 - Preenche os campos do novo nó:
 - Campo **info** com o valor do elemento
 - Campo **prox** aponta para o 1º nó ($N \rightarrow prox = L$)
 - Campo **ant** com **NULL** ($N \rightarrow ant = L \rightarrow ant$)

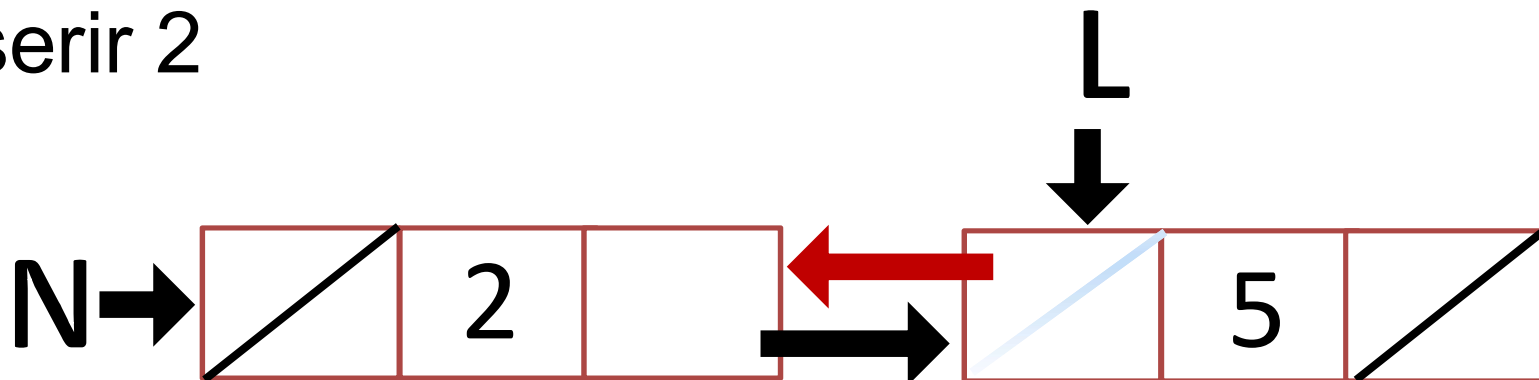
Ex: inserir 2



Inserir Elemento

- Lista com elementos:
 - Faz o campo *ant* do 1º nó da lista apontar para o novo nó ($L \rightarrow ant = N$)

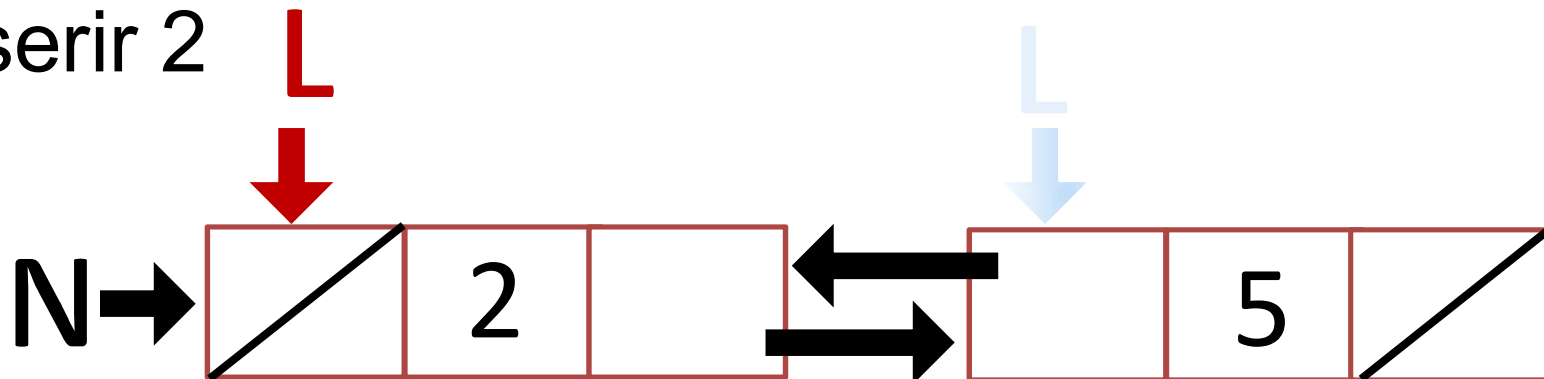
Ex: inserir 2



Inserir Elemento

- Lista com elementos:
 - Faz o campo *ant* do 1º nó da lista apontar para o novo nó ($L \rightarrow ant = N$)
 - Faz a Lista apontar para o novo nó ($L = N$)

Ex: inserir 2

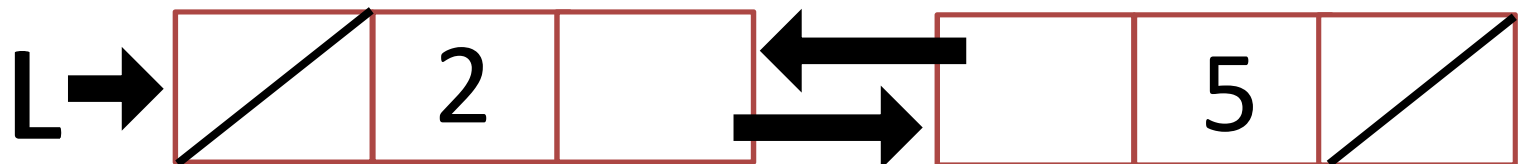


Inserir Elemento

- Lista com elementos:
 - Faz o campo *ant* do 1º nó da lista apontar para o novo nó ($L \rightarrow ant = N$)
 - Faz a Lista apontar para o novo nó ($L = N$)

Ex: inserir 2

retorna 1



Inserer Elemento

- Implementação em C:

```
int insere_elemento (Lista *lst, int elem) {  
    // Aloca um novo nó e preenche campo info  
    Lista N = (Lista) malloc(sizeof(struct no));  
    if (N == NULL) { return 0; } // Falha: nó não alocado  
    N->info = elem; // Insere o conteúdo (valor do elem)  
    N->ant = NULL; // Não tem antecessor do novo nó  
    N->prox = *lst; // Sucessor do novo nó recebe mesmo end. da lista  
    if (lista_vazia(*lst) == 0) // Se lista NÃO vazia  
        (*lst)->ant = N; // Faz o antecessor do 1º nó ser o novo nó  
    *lst = N; // Faz a lista apontar para o novo nó  
    return 1;  
}
```

Remove Elemento

- Existem **6 cenários** possíveis:
 - Lista vazia
 - Elemento não está na lista
 - Elemento está na lista:
 - Lista com um único nó
 - Elemento = 1º nó
 - Elemento = último nó
 - 1º nó < Elemento < último nó

Remove Elemento

- Lista vazia:

Ex: remover 5

L → **NULL**

Remove Elemento

- **Lista vazia:**
 - Não existe o elemento

Ex: remover 5

~~5~~

L → **NULL**

Remove Elemento

- **Lista vazia:**
 - Não existe o elemento
 - Retorna ZERO (**operação falha**)

Ex: remover 5

retorna 0

~~5~~

L → NULL

Remove Elemento

- Elemento não está na lista:

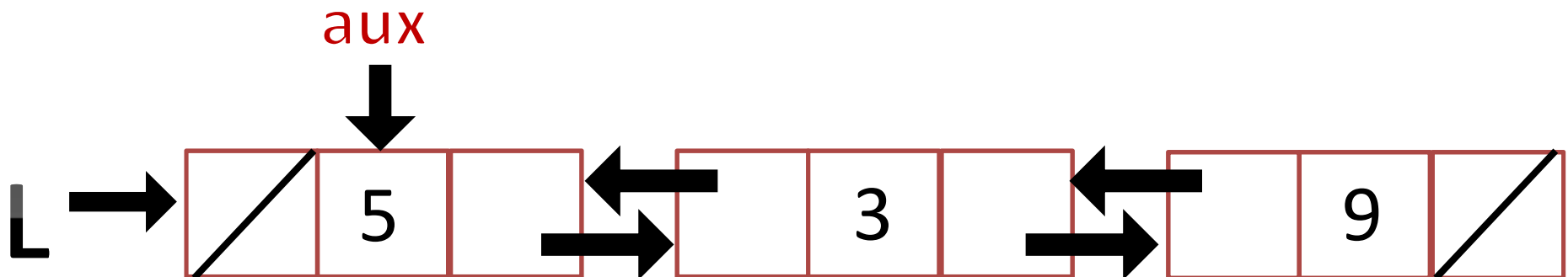
Ex: remover 7



Remove Elemento

- Elemento não está na lista:
 - Faz ponteiro **auxiliar** apontar para o 1º nó

Ex: remover 7



Remove Elemento

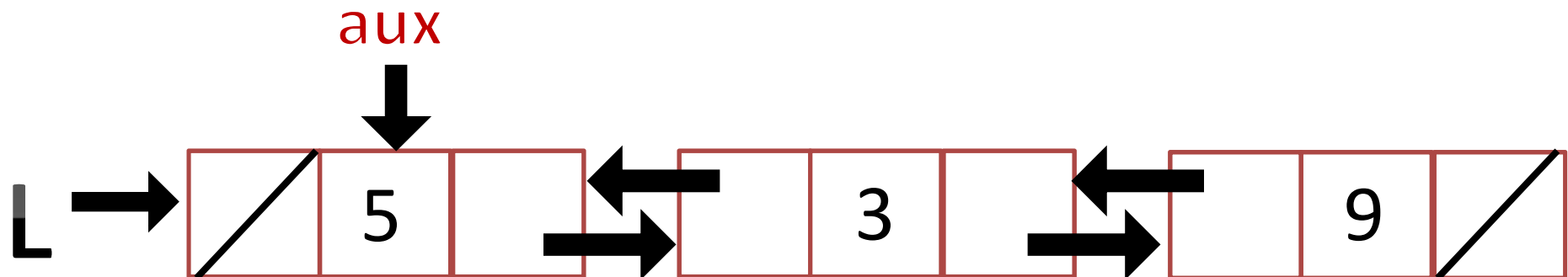
- Elemento não está na lista:
 - Faz ponteiro **auxiliar** apontar para o 1º nó
 - Percorre a lista **até o seu final**

5 ≠ 7

aux->prox ≠ NULL E aux->info ≠ elem?

Sim (avançar)

Ex: remover 7



Remove Elemento

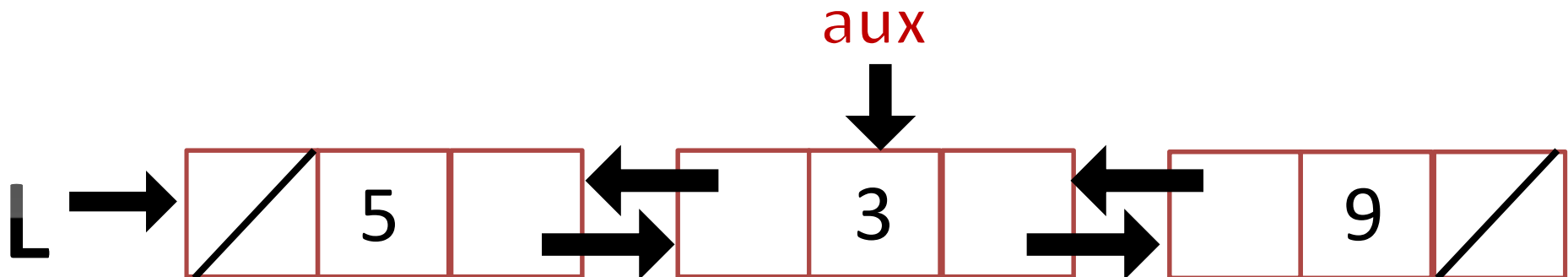
- Elemento não está na lista:
 - Faz ponteiro **auxiliar** apontar para o 1º nó
 - Percorre a lista **até o seu final**

3 \neq **7**

aux->prox \neq NULL E aux->info \neq elem?

Sim (avançar)

Ex: remover 7

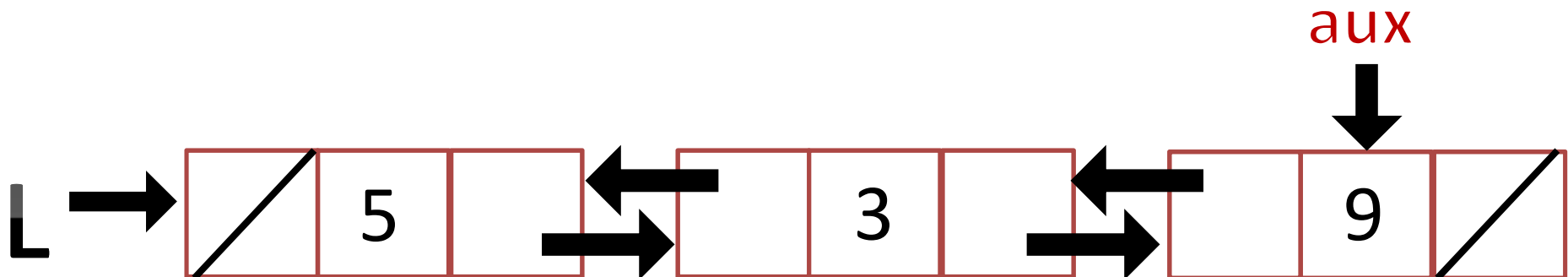


Remove Elemento

- Elemento não está na lista:
 - Faz ponteiro **auxiliar** apontar para o 1º nó
 - Percorre a lista **até o seu final**

Ex: remover 7

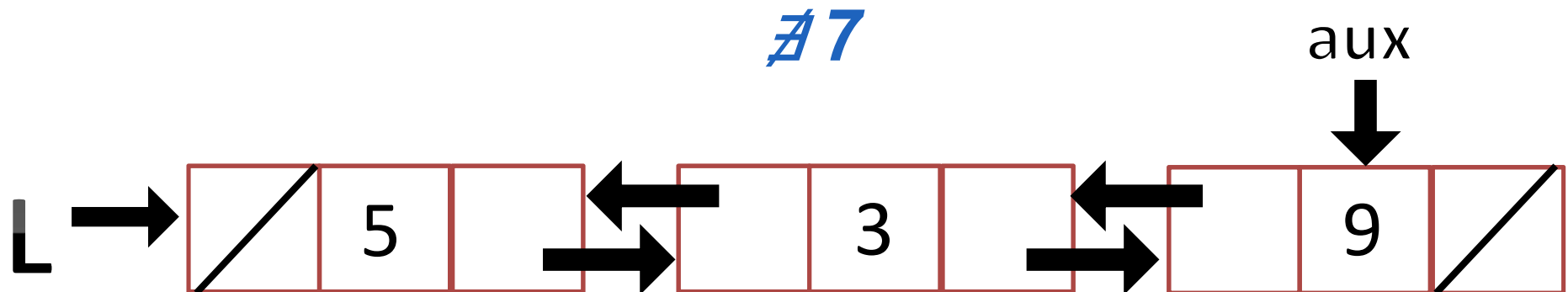
= $9 \neq 7$
 $aux \rightarrow prox \neq NULL \text{ E } aux \rightarrow info \neq elem?$
Não (parar)



Remove Elemento

- Elemento não está na lista:
 - Faz ponteiro **auxiliar** apontar para o 1º nó
 - Percorre a lista **até o seu final**
 - Se atingiu o final é porque **não existe o elemento**

Ex: remover 7



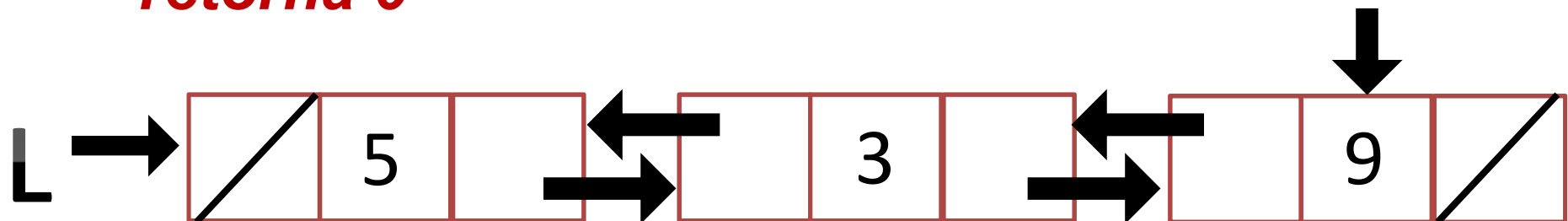
Remove Elemento

- Elemento não está na lista:
 - Faz ponteiro **auxiliar** apontar para o 1º nó
 - Percorre a lista **até o seu final**
 - Se atingiu o final é porque **não existe o elemento**
 - Retorna ZERO (**operação falha**)

Ex: remover 7

retorna 0

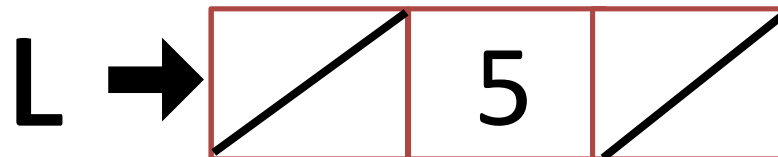
~~7~~



Remove Elemento

- Lista com um único nó:

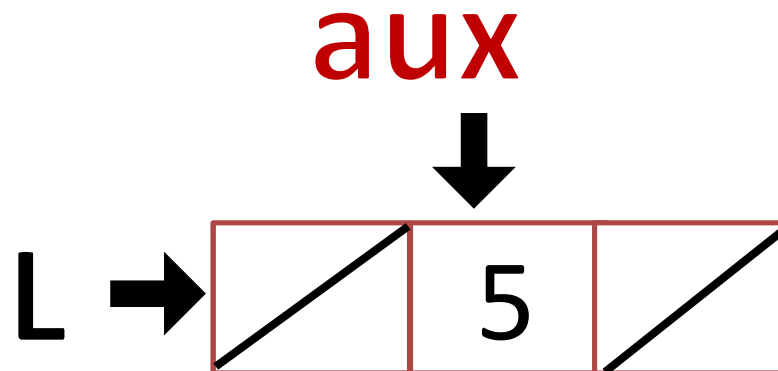
Ex: remover 5



Remove Elemento

- Lista com um único nó:
 - Faz ponteiro **auxiliar** apontar para o 1º nó

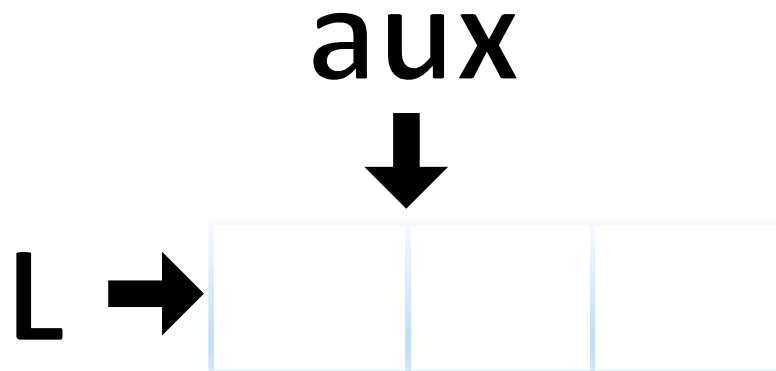
Ex: remover 5



Remove Elemento

- **Lista com um único nó:**
 - Faz ponteiro **auxiliar** apontar para o 1º nó
 - Se campo **info** do 1º nó = elemento
 - Libera memória alocada pelo nó (**free(aux)**)

Ex: remover 5



Remove Elemento

- Lista com um único nó:
 - Faz ponteiro **auxiliar** apontar para o 1º nó
 - Se campo **info** do 1º nó = elemento
 - Libera memória alocada pelo nó (**free(aux)**)
 - Lista retorna ao estado de vazia (**L = NULL**)

Ex: remover 5

L → **NULL**

Remove Elemento

- Lista com um único nó:
 - Faz ponteiro **auxiliar** apontar para o 1º nó
 - Se campo **info** do 1º nó = elemento
 - Libera memória alocada pelo nó (**free(aux)**)
 - Lista retorna ao estado de vazia (**L = NULL**)

Ex: remover 5

retorna 1

L → NULL

Remove Elemento

- Elemento = 1º nó da lista:

Ex: remover 5



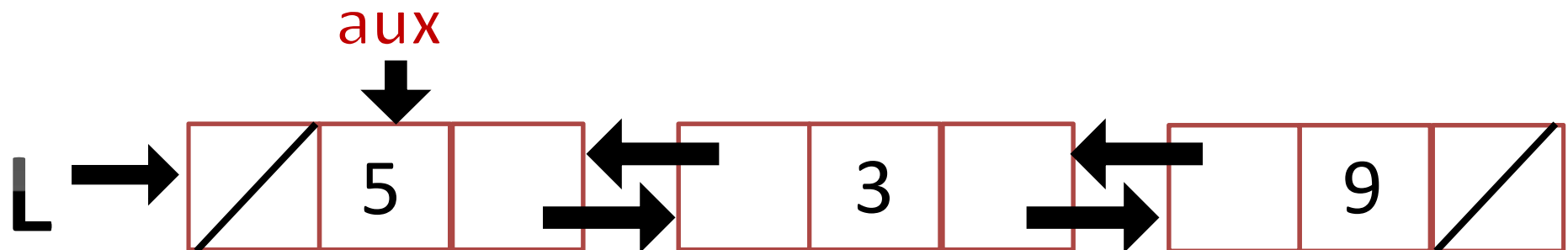
Remove Elemento

- Elemento = 1º nó da lista:
 - Faz ponteiro **auxiliar** apontar para o 1º nó

$$5 = 5$$

aux->prox \neq NULL E aux->info \neq elem?
Não (parar)

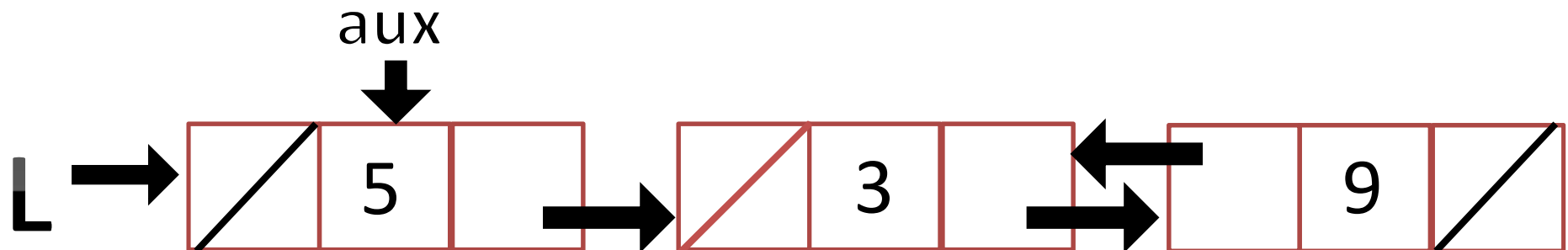
Ex: remover 5



Remove Elemento

- Elemento = 1º nó da lista:
 - Faz ponteiro **auxiliar** apontar para o 1º nó
 - Faz o antecessor do 2º nó igual a **NULL** (*$aux \rightarrow prox \rightarrow ant = NULL$*)

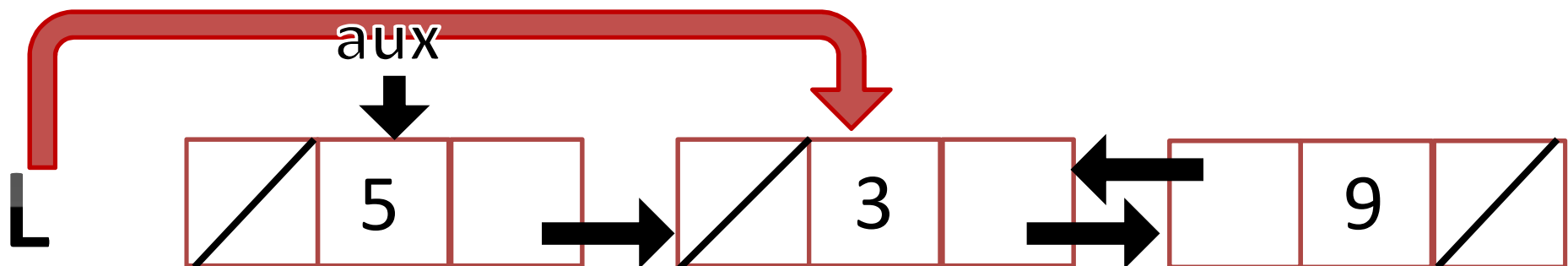
Ex: remover 5



Remove Elemento

- Elemento = 1º nó da lista:
 - Faz ponteiro **auxiliar** apontar para o 1º nó
 - Faz o antecessor do 2º nó igual a **NULL** ($aux \rightarrow prox \rightarrow ant = NULL$)
 - Faz a lista apontar para o 2º nó ($L = aux \rightarrow prox$)

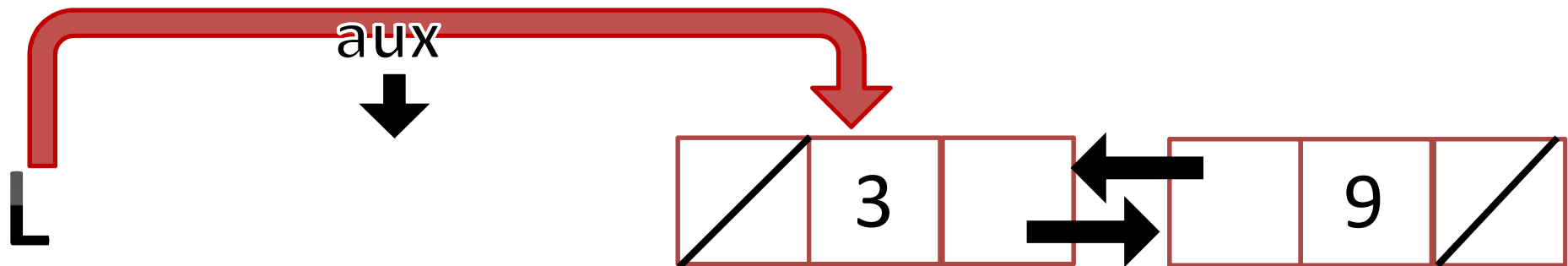
Ex: remover 5



Remove Elemento

- Elemento = 1º nó da lista:
 - Faz ponteiro **auxiliar** apontar para o 1º nó
 - Faz o antecessor do 2º nó igual a **NULL** ($aux \rightarrow prox \rightarrow ant = NULL$)
 - Faz a lista apontar para o 2º nó ($L = aux \rightarrow prox$)
 - Libera a memória alocada pelo nó removido

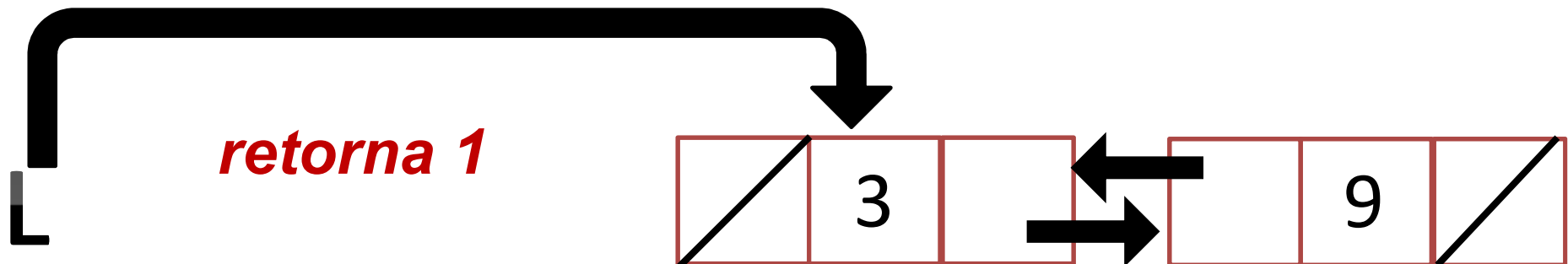
Ex: remover 5



Remove Elemento

- Elemento = 1º nó da lista:
 - Faz ponteiro **auxiliar** apontar para o 1º nó
 - Faz o antecessor do 2º nó igual a **NULL** (***aux->prox->ant = NULL***)
 - Faz a lista apontar para o 2º nó (***L = aux->prox***)
 - Libera a memória alocada pelo nó removido

Ex: remover 5



Remove Elemento

- Elemento = último nó da lista:

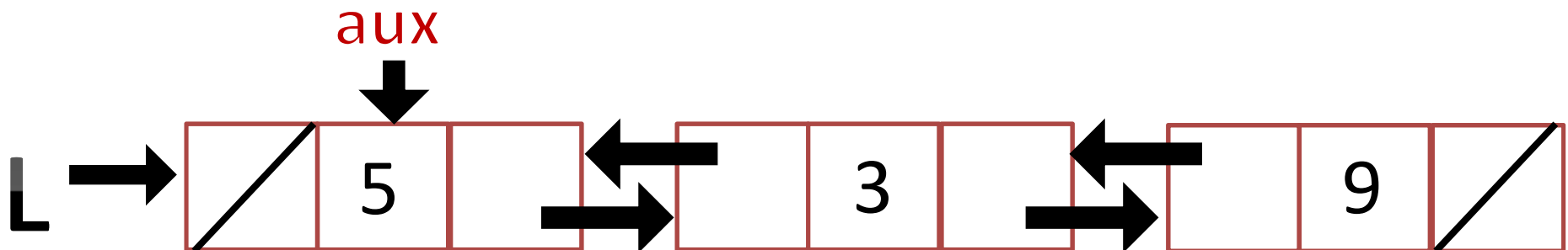
Ex: remover 9



Remove Elemento

- Elemento = último nó da lista:
 - Faz ponteiro **auxiliar** apontar para o 1º nó

Ex: remover 9

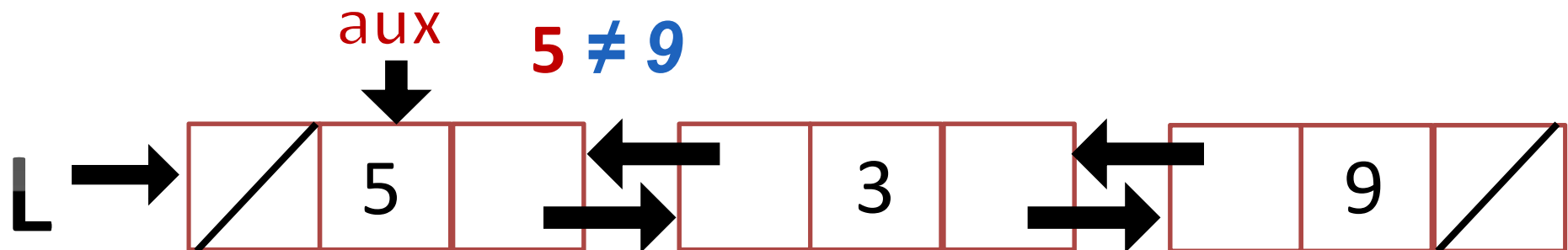


Remove Elemento

- Elemento = último nó da lista:
 - Faz ponteiro **auxiliar** apontar para o 1º nó
 - Percorre a lista **até encontrar o elemento** (*$aux \rightarrow info = elem$*)

$aux \rightarrow prox \neq NULL$ E $aux \rightarrow info \neq elem$?
Sim (avançar)

Ex: remover 9

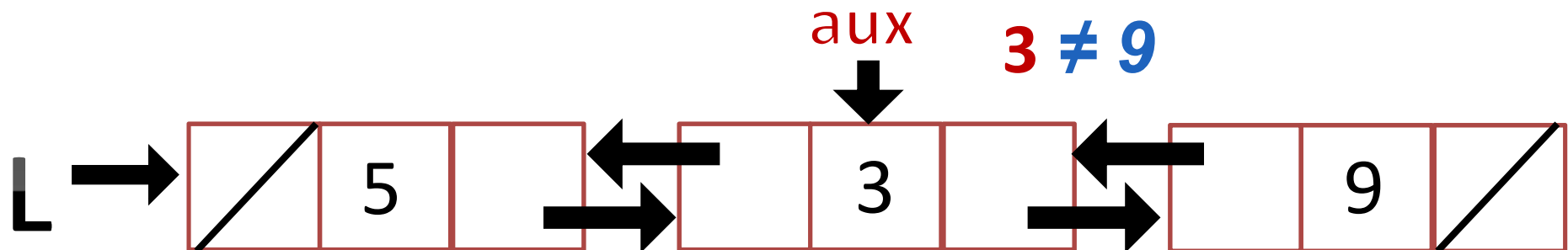


Remove Elemento

- Elemento = último nó da lista:
 - Faz ponteiro **auxiliar** apontar para o 1º nó
 - Percorre a lista **até encontrar o elemento** (***aux->info = elem***)

aux->prox ≠ NULL E aux->info ≠ elem?
Sim (avançar)

Ex: remover 9

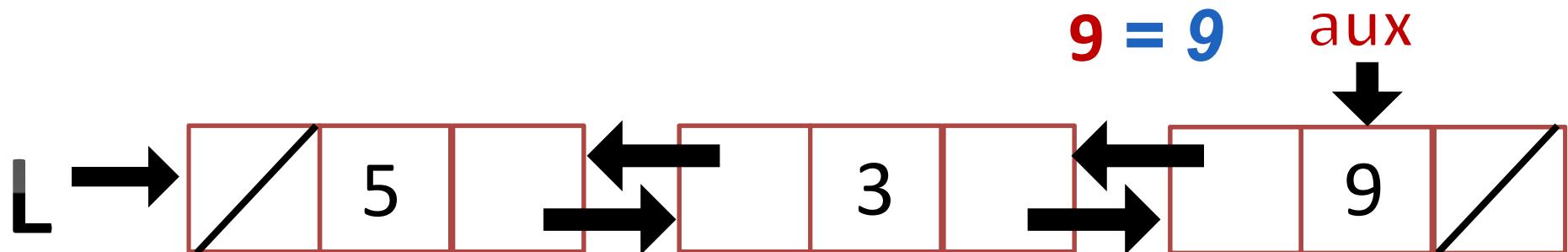


Remove Elemento

- Elemento = último nó da lista:
 - Faz ponteiro **auxiliar** apontar para o 1º nó
 - Percorre a lista **até encontrar o elemento** (*$aux \rightarrow info = elem$*)

=
 $aux \rightarrow prox \neq NULL$ E $aux \rightarrow info \neq elem$?
Não (parar)

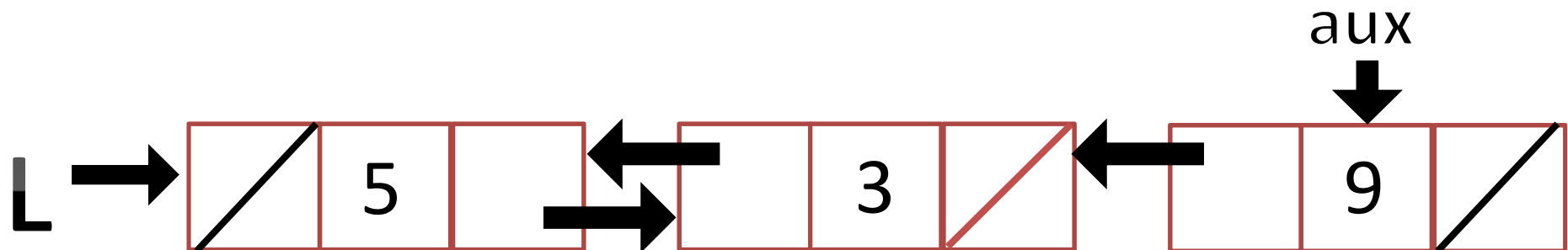
Ex: remover 9



Remove Elemento

- Elemento = último nó da lista:
 - Faz ponteiro **auxiliar** apontar para o 1º nó
 - Percorre a lista **até encontrar o elemento** (***aux->info = elem***)
 - Faz o antecessor do último nó apontar para **NULL** (***aux->ant->prox = NULL***)

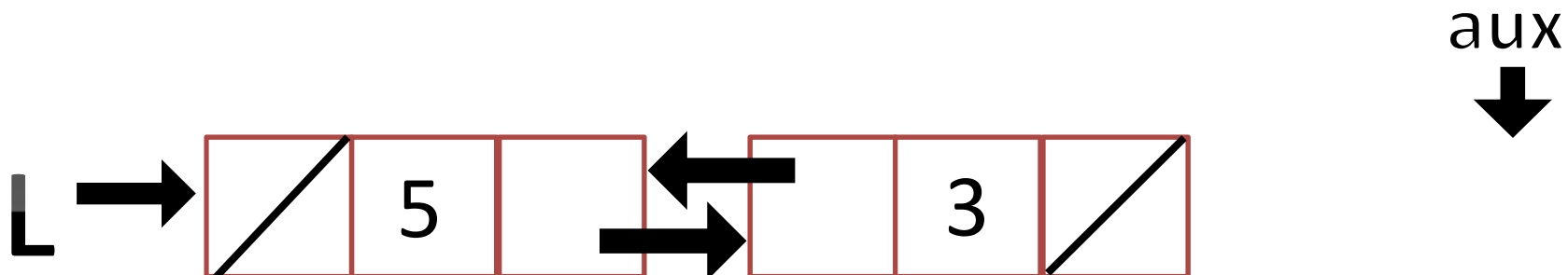
Ex: remover 9



Remove Elemento

- Elemento = último nó da lista:
 - Faz ponteiro **auxiliar** apontar para o 1º nó
 - Percorre a lista **até encontrar o elemento** (***aux->info = elem***)
 - Faz o antecessor do último nó apontar para **NULL** (***aux->ant->prox = NULL***)
 - Libera a memória alocada pelo nó removido

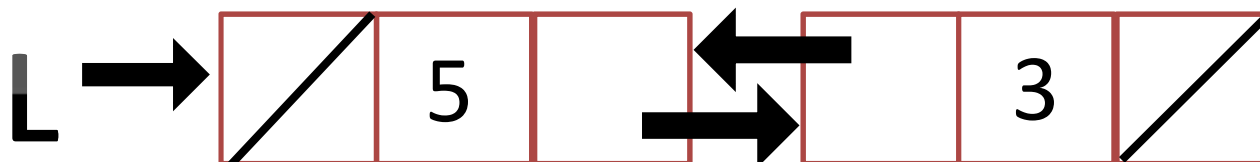
Ex: remover 9



Remove Elemento

- Elemento = último nó da lista:
 - Faz ponteiro **auxiliar** apontar para o 1º nó
 - Percorre a lista **até encontrar o elemento** (***aux->info = elem***)
 - Faz o antecessor do último nó apontar para **NULL** (***aux->ant->prox = NULL***)
 - Libera a memória alocada pelo nó removido

Ex: remover 9



retorna 1

Remove Elemento

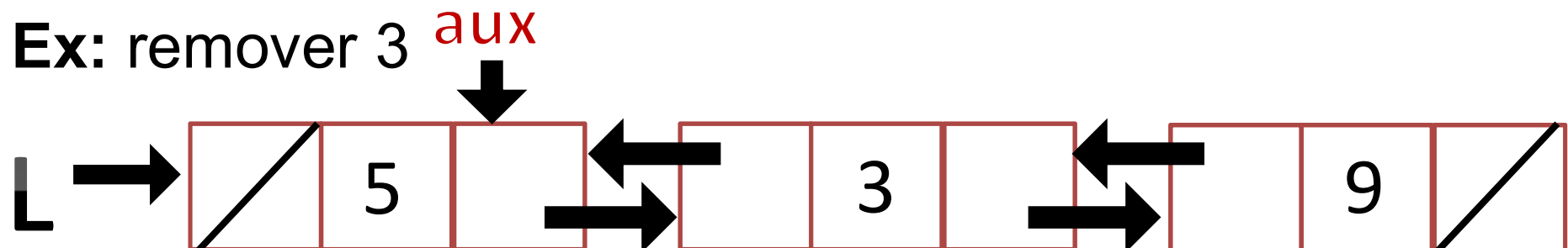
- 1º nó < Elemento < último nó da lista:

Ex: remover 3



Remove Elemento

- 1º nó < Elemento < último nó da lista:
 - Faz ponteiro **auxiliar** apontar para o 1º nó



Remove Elemento

- 1º nó < Elemento < último nó da lista:
 - Faz ponteiro **auxiliar** apontar para o 1º nó
 - Percorre a lista **até encontrar o elemento**
(*aux->info = elem*)

aux->prox ≠ NULL E aux->info ≠ elem?
Sim (avançar)

Ex: remover 3 *aux* **5 ≠ 3**

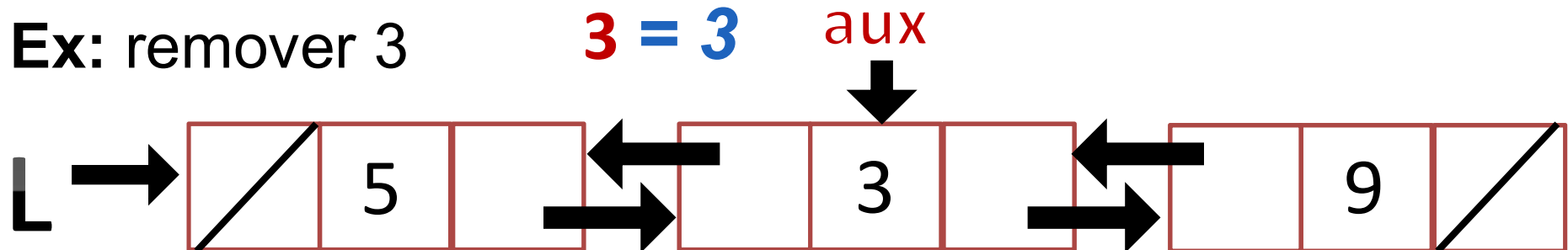


Remove Elemento

- 1º nó < Elemento < último nó da lista:
 - Faz ponteiro **auxiliar** apontar para o 1º nó
 - Percorre a lista **até encontrar o elemento**
(*aux->info = elem*)

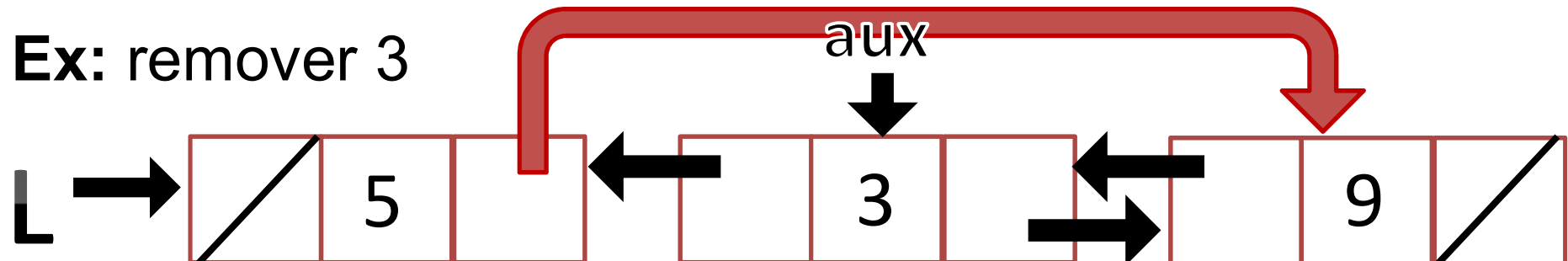
aux->prox ≠ NULL E aux->info ≠ elem?
Não (parar)

Ex: remover 3



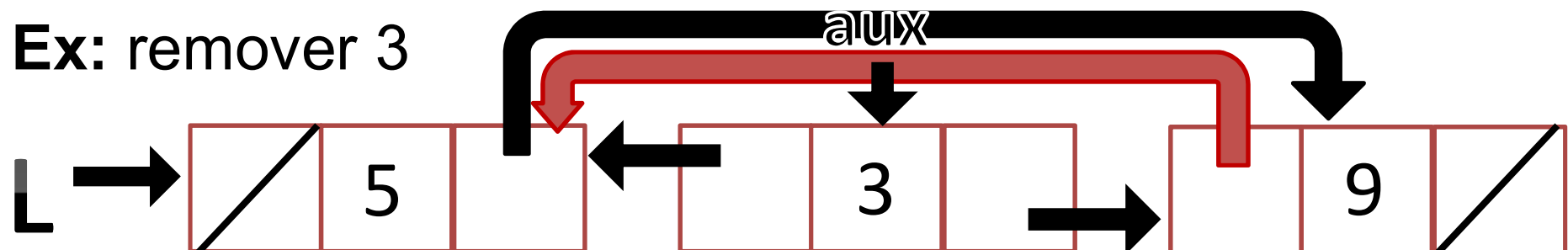
Remove Elemento

- 1º nó < Elemento < último nó da lista:
 - Faz ponteiro **auxiliar** apontar para o 1º nó
 - Percorre a lista **até encontrar o elemento** (*aux->info = elem*)
 - Faz o **antecessor do nó** apontar para o **sucessor** (*aux->ant->prox = aux->prox*)



Remove Elemento

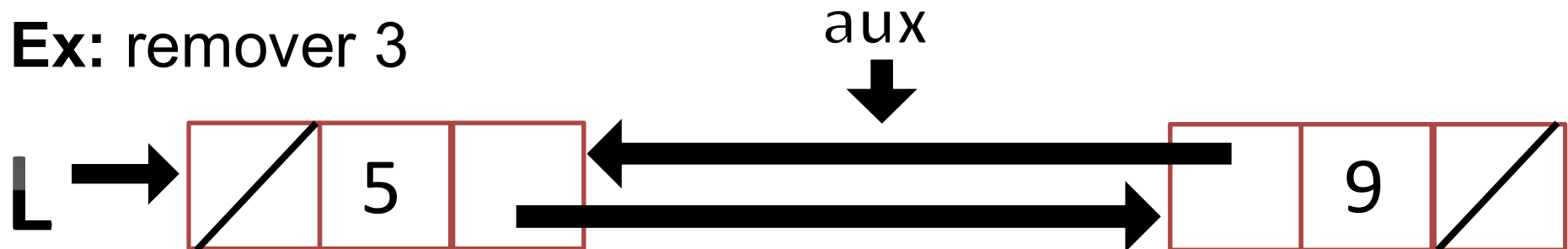
- 1º nó < Elemento < último nó da lista:
 - Faz ponteiro **auxiliar** apontar para o 1º nó
 - Percorre a lista **até encontrar o elemento** (*aux->info = elem*)
 - Faz o **antecessor do nó** apontar para o **sucessor** (*aux->ant->prox = aux->prox*)
 - Faz o **sucessor do nó** apontar para o **antecessor** (*aux->prox->ant = aux->ant*)



Remove Elemento

- 1º nó < Elemento < último nó da lista:
 - Faz ponteiro **auxiliar** apontar para o 1º nó
 - Percorre a lista **até encontrar o elemento** (*aux->info = elem*)
 - Faz o **antecessor do nó** apontar para o **sucessor** (*aux->ant->prox = aux->prox*)
 - Faz o **sucessor do nó** apontar para o **antecessor** (*aux->prox->ant = aux->ant*)
 - Libera a memória alocada pelo nó

Ex: remover 3



Remove Elemento

- 1º nó < Elemento < último nó da lista:
 - Faz ponteiro **auxiliar** apontar para o 1º nó
 - Percorre a lista **até encontrar o elemento** (***aux->info = elem***)
 - Faz o **antecessor do nó** apontar para o **sucessor** (***aux->ant->prox = aux->prox***)
 - Faz o **sucessor do nó** apontar para o **antecessor** (***aux->prox->ant = aux->ant***)
 - Libera a memória alocada pelo nó

Ex: remover 3

retorna 1



Remove Elemento

- Implementação em C:

```
int remove_elemento (Lista *lst, int elem) {  
    if (lista_vazia(*lst) // Trata lista vazia)  
        return 0;  
    Lista aux = *lst; // Faz aux apontar para 1º nó  
    while (aux->prox != NULL && aux->info != elem)  
        aux = aux->prox;  
    if (aux->info != elem) return 0; // Elemento não está na lista  
    if (aux->prox != NULL) (aux)->prox->ant = aux->ant;  
    if (aux->ant != NULL) (aux)->ant->prox = aux->prox;  
    if (aux == *lst) *lst = aux->prox;  
    free(aux);  
    return 1;  
}
```

Encadeamento Duplo

- **TAD lista ordenada:**
 - **Inserir ordenado**
 - Elemento deve ser colocado em uma posição específica para **manter ordenação**
 - Envolve percorrimento da lista

Encadeamento Duplo

- **TAD lista ordenada:**
 - **Insere ordenado**
 - Elemento deve ser colocado em uma posição específica para **manter ordenação**
 - Envolve percorrimeto da lista
 - **Remove ordenado**
 - Encontrar o elemento envolve percorrimeto
 - Inexistência do elemento é determinada por final de lista ou encontrar elemento maior

Encadeamento Duplo

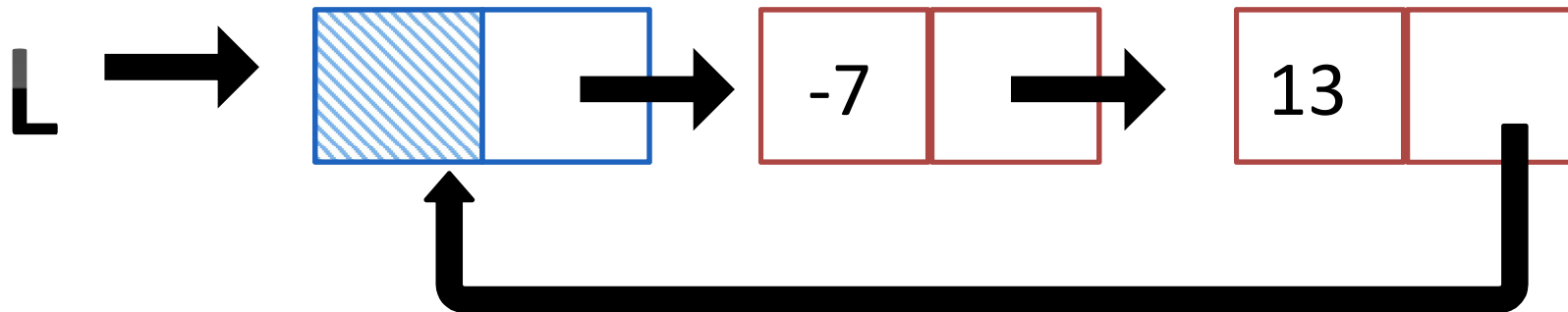
- **TAD lista ordenada:**
 - **Inserir ordenado**
 - Elemento deve ser colocado em uma posição específica para **manter ordenação**
 - Envolve percorrimento da lista
 - **Remover ordenado**
 - Encontrar o elemento envolve percorrimento
 - Inexistência do elemento é determinada por final de lista ou encontrar elemento maior
- As operações *insere_ord* e *remove_ord* ficam como exercício

Combinação das Técnicas de Encadeamento

- Pode-se combinar as várias técnicas a fim de obter **listas encadeadas ainda mais complexas**

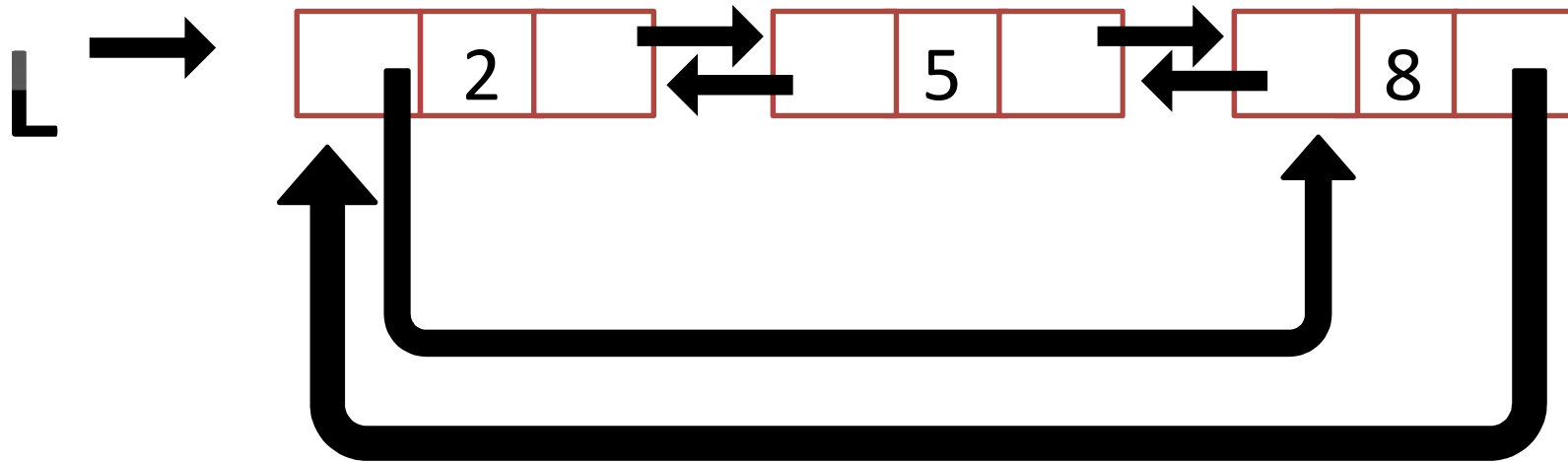
Exemplos

- **Lista circular com nó cabeçalho:**



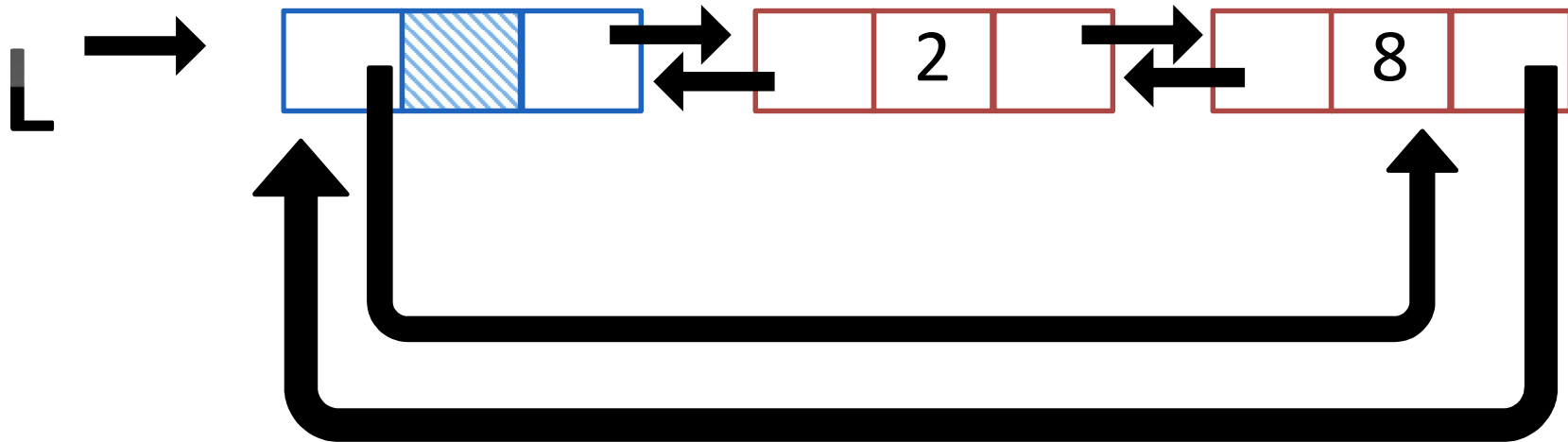
Exemplos

- **Lista circular duplamente encadeada:**



Exemplos

- **Lista circular duplamente encadeada com nó cabeçalho:**



Exercícios

1. Implementar, utilizando a implementação **dinâmica com encadeamento duplo**, o TAD lista linear não ordenada de números inteiros. Essa implementação deve contemplar as operações básicas: *criar_lista*, *lista_vazia*, *insere_elemento*, *remove_elemento* e *obtem_valor_elem*. Além disso, desenvolva um programa aplicativo que permita ao usuário inicializar uma lista, inserir e remover elementos e imprimir a lista.

Teste este programa com a seguinte seqüência de operações:

- *Inicialize a lista*
- *Imprima a lista*
- *Insira os elementos {4,8,-1,19,2,7,8,5,9,22,45};*
- *Imprima a lista*
- *Remova o elemento 8*
- *Imprima a lista*
- *Inicialize a lista*
- *Imprima a lista*

Referências

- *Backes, André, Linguagem C Descomplicada, portal de vídeo-aulas, <https://programacaodescomplicada.wordpress.com/>, acessado em 09/03/2016.*
- *Celes, W., Cerqueira, R. e Rangel, J. L. Introdução a estruturas de dados. Ed. Campus Elsevier, 2004.*