

# *Estrutura de Dados*

**Implementação Dinâmica/Encadeada de  
Listas Lineares**

**Prof. Luiz Gustavo Almeida Martins**

# Lista Dinâmica/Encadeada

- Implementada através da **alocação de elementos individualizados** que armazenam:
  - Informação desejada (**valor do elemento**)
  - Ponteiro para seu sucessor (**encademento**)
    - Garante a formação da lista
- Apresenta as seguintes características:
  - Elementos ocupam **posições NÃO consecutivas**
  - Espaço de memória ocupado pela lista é **proporcional ao número de elementos** armazenado

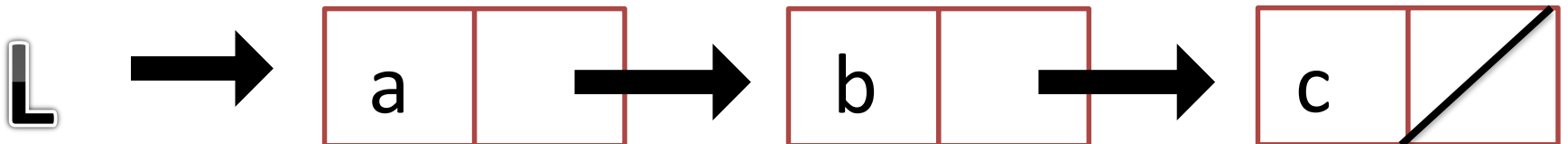
# Lista Dinâmica/Encadeada

- **Vantagens:**
  - Utilização otimizada da memória
  - Não requer movimentação na inserção/remoção
  - Não requer gerenciamento do espaço livre
- **Desvantagens:**
  - Não tem acesso direto (indexado) aos elementos da lista
- **Recomendação de uso:**
  - Listas grandes
  - Aplicações baseadas em inserções e remoções no meio da lista e poucas consultas
  - Tamanho máximo da lista não conhecido

# Lista Dinâmica/Encadeada

- A lista é representada pela estrutura a seguir:

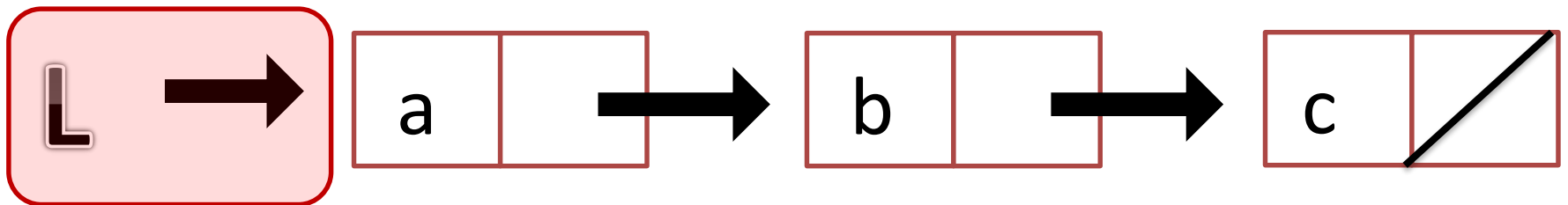
**Ex:**  $L = \{a, b, c\}$



# Lista Dinâmica/Encadeada

- A lista é representada pela estrutura a seguir:
  - O início da lista é representado por um ponteiro para o 1º nó

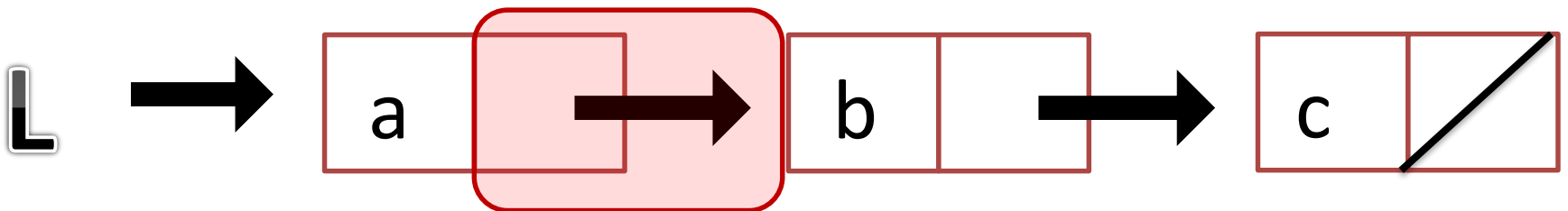
**Ex:**  $L = \{a, b, c\}$



# Lista Dinâmica/Encadeada

- A lista é representada pela estrutura a seguir:
  - O início da lista é representado por um ponteiro para o 1º nó
  - O 1º nó da lista aponta para o 2º

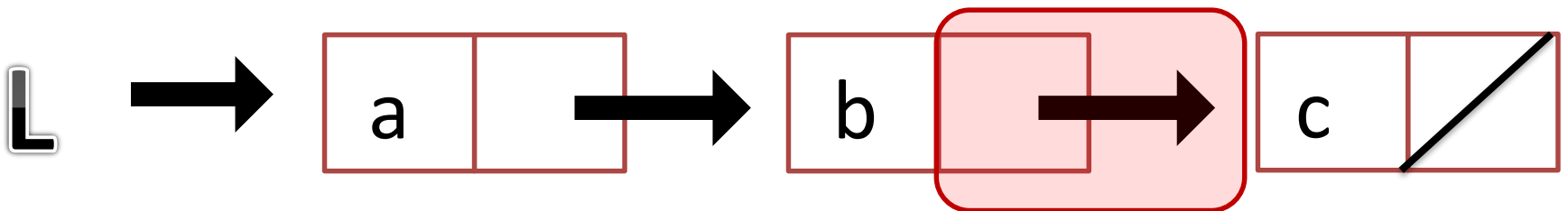
**Ex:**  $L = \{a, b, c\}$



# Lista Dinâmica/Encadeada

- A lista é representada pela estrutura a seguir:
  - O início da lista é representado por um ponteiro para o 1º nó
  - O 1º nó da lista aponta para o 2º
  - O 2º nó da lista aponta para o 3º

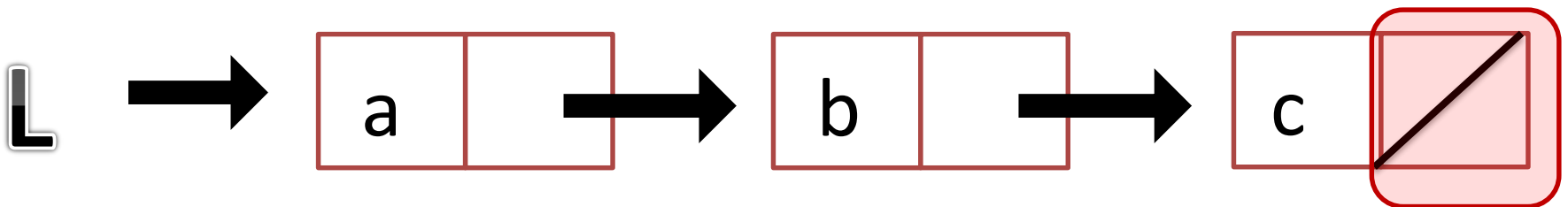
**Ex:**  $L = \{a, b, c\}$



# Lista Dinâmica/Encadeada

- A lista é representada pela estrutura a seguir:
  - O início da lista é representado por um ponteiro para o 1º nó
  - O 1º nó da lista aponta para o 2º
  - O 2º nó da lista aponta para o 3º
  - O 3º e último nó da lista aponta para **NULL**

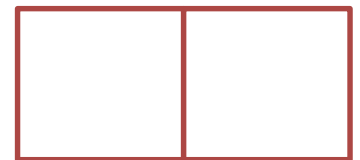
**Ex:**  $L = \{a, b, c\}$





# Lista Dinâmica/Encadeada

- Estruturas de representação:
  - Uma estrutura para representar o nó



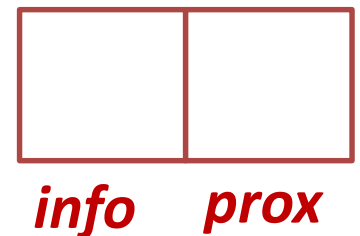
# Lista Dinâmica/Encadeada

- Estruturas de representação:
  - Uma estrutura para representar o **nó**
    - Campo para informação (**ex:** *int info*)



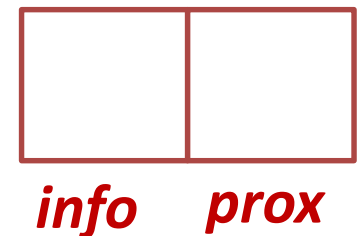
# Lista Dinâmica/Encadeada

- Estruturas de representação:
  - Uma estrutura para representar o **nó**
    - Campo para informação (**ex:** *int info*)
    - Campo para apontar o sucessor (**ex:** *struct no \*prox*)



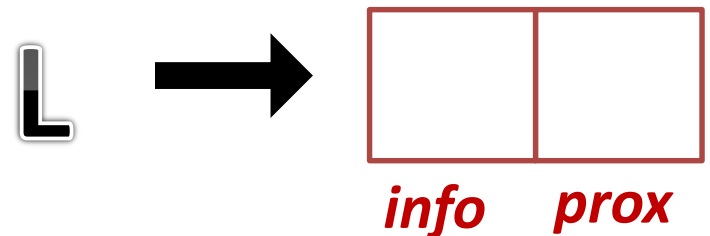
# Lista Dinâmica/Encadeada

- Estruturas de representação:
  - Uma estrutura para representar o **nó**
    - Campo para informação (**ex:** *int info*)
    - Campo para apontar o sucessor (**ex:** *struct no \*prox*)
    - É a estrutura que será **alocada** dinamicamente **na inserção** e **liberada na remoção**



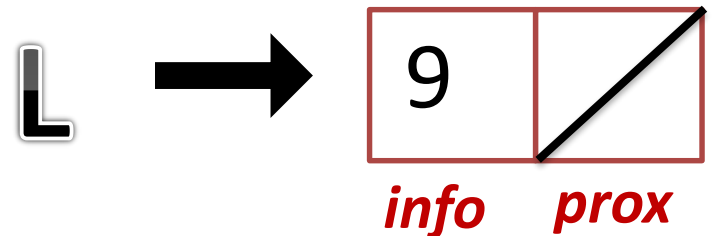
# Lista Dinâmica/Encadeada

- Estruturas de representação:
  - Uma estrutura para representar o **nó**
    - Campo para informação (**ex:** *int info*)
    - Campo para apontar o sucessor (**ex:** *struct no \*prox*)
    - É a estrutura que será **alocada** dinamicamente **na inserção** e **liberada na remoção**
  - Um ponteiro para o início da lista (**1º nó**)



# Lista Dinâmica/Encadeada

- Estruturas de representação:
  - Uma estrutura para representar o **nó**
    - Campo para informação (**ex:** *int info*)
    - Campo para apontar o sucessor (**ex:** *struct no \*prox*)
    - É a estrutura que será **alocada** dinamicamente **na inserção** e **liberada na remoção**
  - Um ponteiro para o início da lista (**1º nó**)
  - **Final da lista** é indicado por **prox = NULL**



# Estrutura de Representação em C

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

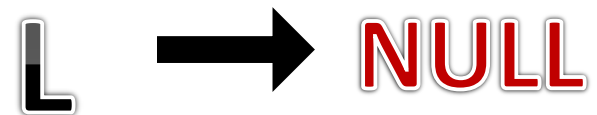
```
struct no {  
    int info;  
    struct no * prox;  
};
```

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

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

# Operação **cria\_lista**

- Colocar a lista no estado de vazia
  - Lista vazia é representada pelo ponteiro **NULL**

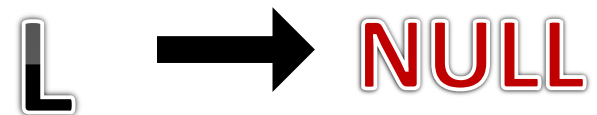




# Operação **cria\_lista**

- Colocar a lista no estado de vazia
  - Lista vazia é representada pelo ponteiro **NULL**
- Implementação em C:

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



# Operação **lista\_vazia()**

- Verifica se a lista está na condição de vazia:
  - Se a **variável do tipo lista** aponta para ***NULL***

# Operação **lista\_vazia()**

- Verifica se a lista está na condição de vazia:
  - Se a **variável do tipo lista** aponta para **NULL**
- **Implementação em C:**

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

# Operação **lista\_cheia()**

- **Teoricamente** a lista **NÃO** fica cheia na alocação dinâmica (lista infinita)
- **Na prática**, tamanho da lista é **limitado pelo espaço de memória**
  - Função ***malloc()*** retorna ***NULL*** quando não é possível alocar um novo nó

# Operação de Inserção

- Afetada pelo **critério de ordenação**
  - **Lista não ordenada:**
    - Inserção na ordem de chegada
    - Insere no **início** da lista (**mais simples**)
      - Evita o percorrimeto da lista
    - Função: ***insere\_elem()***
  - **Lista ordenada:**
    - Deve garantir que a lista permaneça ordenada
    - Inserção envolve **percorrimeto da lista** para buscar a posição correta (**mais complexo**)
    - Função: ***insere\_ord()***

# Operação de Inserção (**Lista NÃO Ordenada**)

- Existem **2 cenários** possíveis de inserção:
  - Lista sem elementos (**lista vazia**)
  - Lista com 1 ou mais elementos

# Operação de Inserção (**Lista NÃO Ordenada**)

- Existem **2 cenários** possíveis de inserção:
  - Lista sem elementos (**lista vazia**)
  - Lista com 1 ou mais elementos
- **Ambos são tratados da mesma forma:**
  - Alocação do novo nó

# Operação de Inserção (Lista NÃO Ordenada)

- Existem **2 cenários** possíveis de inserção:
  - Lista sem elementos (**lista vazia**)
  - Lista com 1 ou mais elementos
- **Ambos são tratados da mesma forma:**
  - Alocação do novo nó
  - Preenchimento dos campos do novo nó
    - ***info*** recebe o valor do novo elemento
    - ***prox*** recebe o endereço armazenado pela lista (**1º nó atual**)



# Operação de Inserção (**Lista NÃO Ordenada**)

- Existem **2 cenários** possíveis de inserção:
  - Lista sem elementos (**lista vazia**)
  - Lista com 1 ou mais elementos
- **Ambos são tratados da mesma forma:**
  - Alocação do novo nó
  - Preenchimento dos campos do novo nó
    - **info** recebe o valor do novo elemento
    - **prox** recebe o endereço armazenado pela lista (**1º nó atual**)
  - Alteração do endereço armazenado na lista
    - Exige que a variável seja **passada por referência**

# Operação de Inserção (Lista NÃO Ordenada)

- Implementação em C:

```
int insere_elem (Lista *lst, int elem) {  
    // Aloca um novo nó  
    Lista N = (Lista) malloc(sizeof(struct no));  
    if (N == NULL) { return 0; } // Falha: nó não alocado  
    // Preenche os campos do novo nó  
    N->info = elem; // Insere o conteúdo (valor do elem)  
    N->prox = *lst; // Aponta para o 1º nó atual da lista  
    *lst = N; // Faz a lista apontar para o novo nó  
    return 1;  
}
```

# Operação de Inserção (Lista NÃO Ordenada)

- Implementação em C: passagem por referência

```
int insere_elem (Lista *lst, int elem) {
```

```
    // Aloca um novo nó
```

```
    Lista N = (Lista) malloc(sizeof(struct no));
```

```
    if (N == NULL) { return 0; } // Falha: nó não alocado
```

```
    // Preenche os campos do novo nó
```

```
    N->info = elem; // Insere o conteúdo (valor do elem)
```

```
    N->prox = *lst; // Aponta para o 1º nó atual da lista
```

```
    *lst = N; // Faz a lista apontar para o novo nó
```

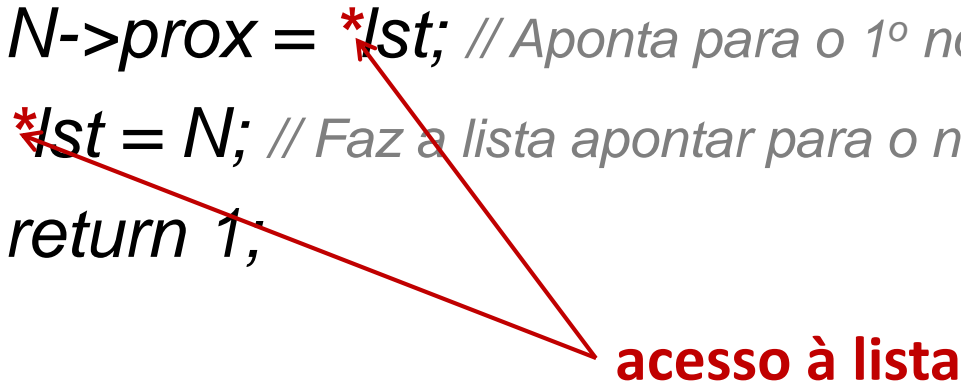
```
    return 1;
```

```
}
```

# Operação de Inserção (Lista NÃO Ordenada)

- Implementação em C:

```
int insere_elem (Lista *lst, int elem) {  
    // Aloca um novo nó  
    Lista N = (Lista) malloc(sizeof(struct no));  
    if (N == NULL) { return 0; } // Falha: nó não alocado  
    // Preenche os campos do novo nó  
    N->info = elem; // Insere o conteúdo (valor do elem)  
    N->prox = *lst; // Aponta para o 1º nó atual da lista  
    *lst = N; // Faz a lista apontar para o novo nó  
    return 1;  
}
```



**acesso à lista**

# Operação de Inserção (Lista Ordenada)

- Inserção na **posição correta**
  - Envolve **percorrimento**

# Operação de Inserção (Lista Ordenada)

- Inserção na **posição correta**
  - Envolve **percorrimento**
- Existem **4 cenários** possíveis de inserção:
  - Lista está vazia
  - Novo elemento  $\leq$  1º nó da lista
  - Novo elemento  $>$  último nó da lista
  - Novo elemento entre o 1º e o último nó da lista

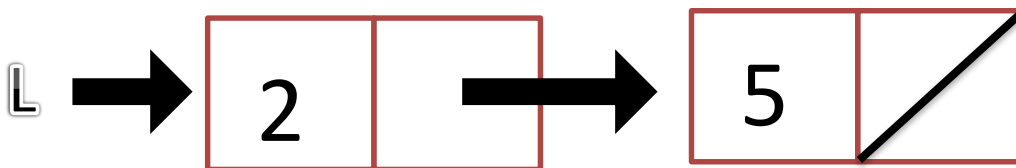
# Operação de Inserção (Lista Ordenada)

- Inserção na **posição correta**
    - Envolve **percorrimento**
  - Existem **4 cenários** possíveis de inserção:
    - Lista está vazia
    - Novo elemento  $\leq$  1º nó da lista
    - Novo elemento  $>$  último nó da lista
    - Novo elemento entre o 1º e o último nó da lista
- similar ao  
TAD Lista  
NÃO Ordenada**
- novos casos**

# Operação de Inserção (Lista Ordenada)

- **Elemento > último nó da lista:**

**Ex:** inserir 8

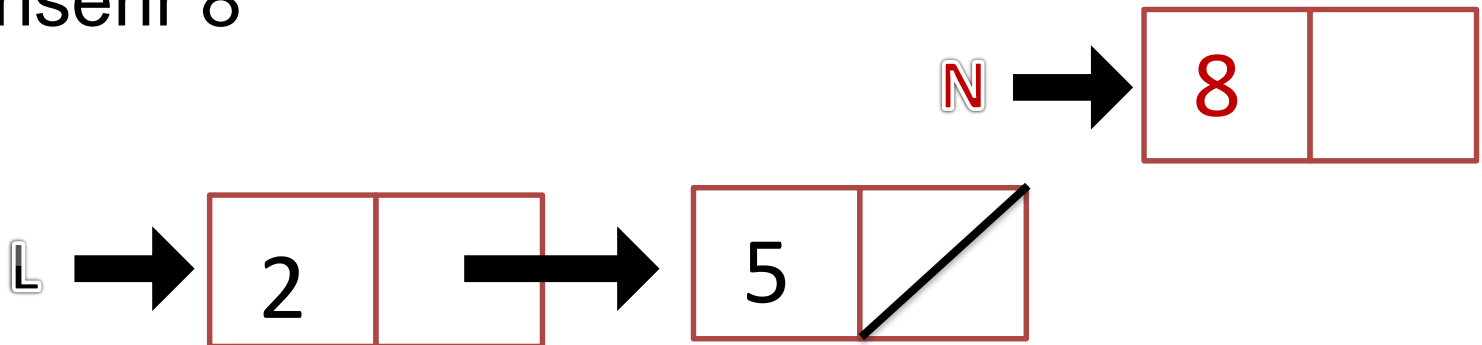




# Operação de Inserção (Lista Ordenada)

- **Elemento > último nó da lista:**
  - Alocar um novo nó e preencher o campo *info*

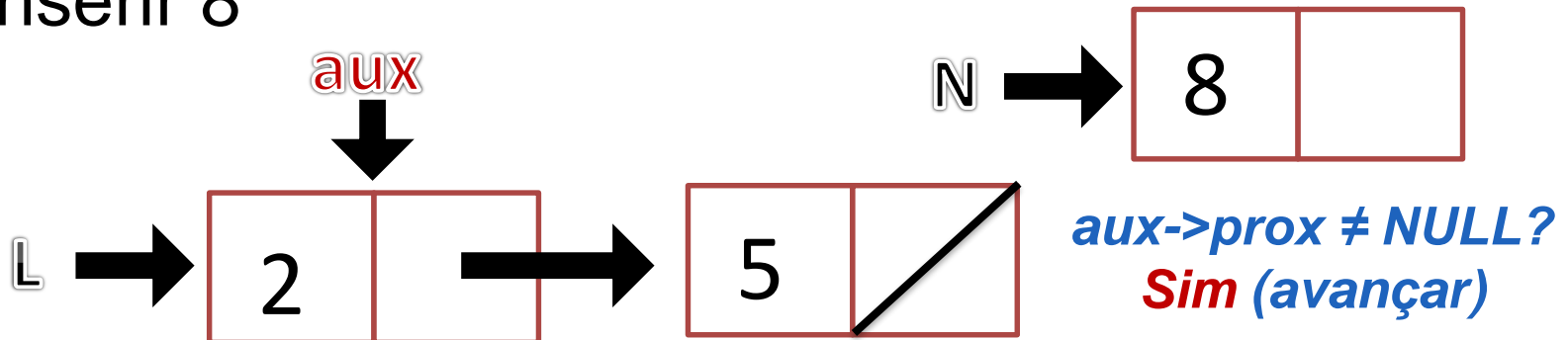
**Ex:** inserir 8



# Operação de Inserção (Lista Ordenada)

- **Elemento > último nó da lista:**
  - Alocar um novo nó e preencher o campo *info*
  - Percorrer a lista até seu final ( $\nexists$  **sucessor**)
    - Usar ponteiro auxiliar
    - Condição de parada: ***aux->prox = NULL***

**Ex:** inserir 8

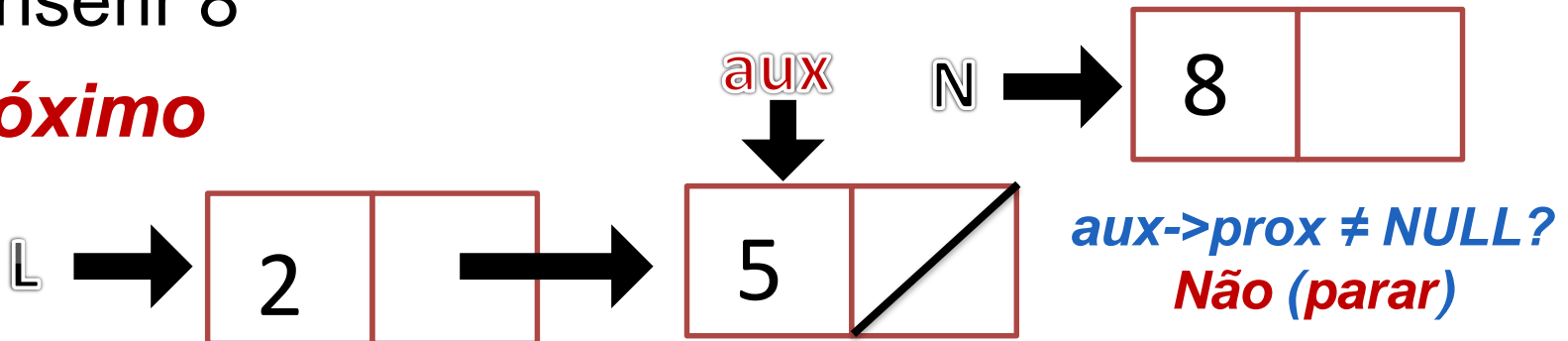


# Operação de Inserção (Lista Ordenada)

- **Elemento > último nó da lista:**
  - Alocar um novo nó e preencher o campo *info*
  - Percorrer a lista até seu final (~~N~~ **sucessor**)
    - Usar ponteiro auxiliar
    - Condição de parada: *aux->prox = NULL*

**Ex:** inserir 8

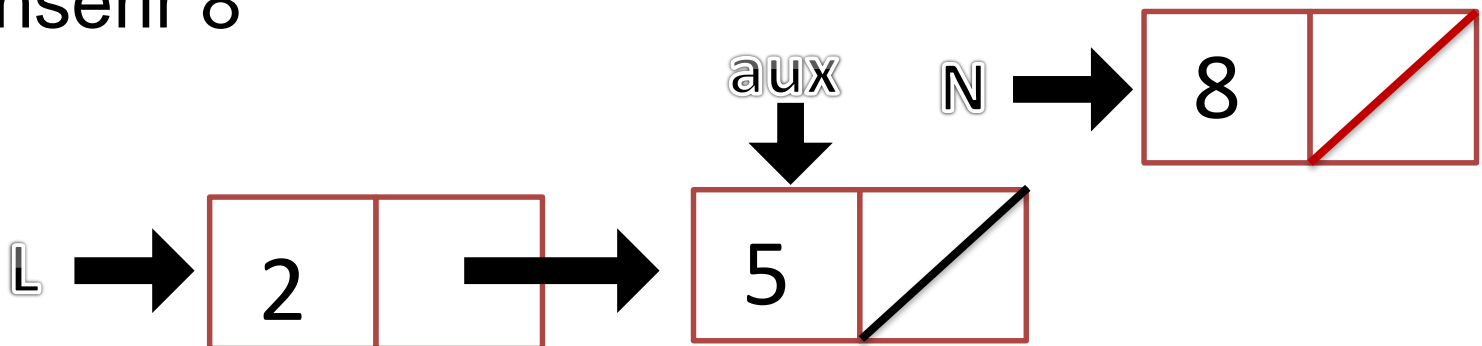
~~N~~ **próximo**



# Operação de Inserção (Lista Ordenada)

- **Elemento > último nó da lista:**
  - Alocar um novo nó e preencher o campo *info*
  - Percorrer a lista até seu final (~~N~~ **sucessor**)
    - Usar ponteiro auxiliar
    - Condição de parada: *aux->prox = NULL*
  - Preencher campo *prox* do novo nó

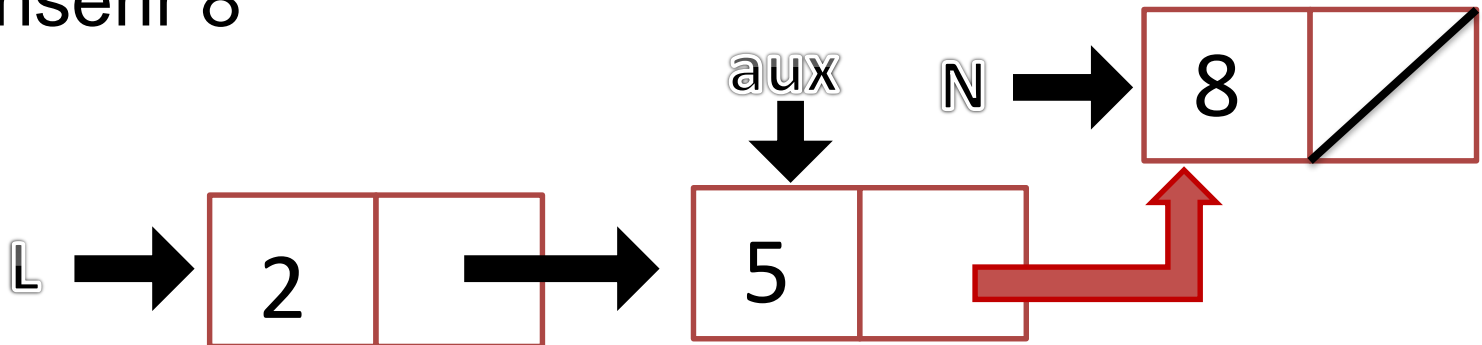
**Ex:** inserir 8



# Operação de Inserção (Lista Ordenada)

- **Elemento > último nó da lista:**
  - Alocar um novo nó e preencher o campo *info*
  - Percorrer a lista até seu final (~~N~~ **sucessor**)
    - Usar ponteiro auxiliar
    - Condição de parada: ***aux->prox = NULL***
  - Preencher campo ***prox*** do novo nó
  - Fazer o último nó apontar para o novo nó

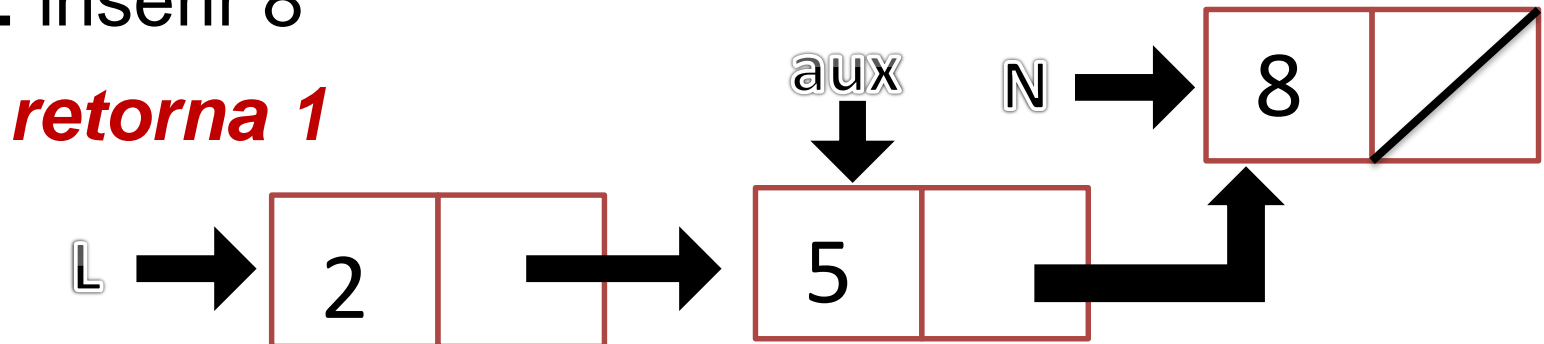
**Ex:** inserir 8



# Operação de Inserção (Lista Ordenada)

- **Elemento > último nó da lista:**
  - Alocar um novo nó e preencher o campo *info*
  - Percorrer a lista até seu final (~~N~~ **sucessor**)
    - Usar ponteiro auxiliar
    - Condição de parada: ***aux->prox = NULL***
  - Preencher campo ***prox*** do novo nó
  - Fazer o último nó apontar para o novo nó

**Ex:** inserir 8



# Operação de Inserção (Lista Ordenada)

- 1o nó < Elemento  $\leq$  último nó da lista:

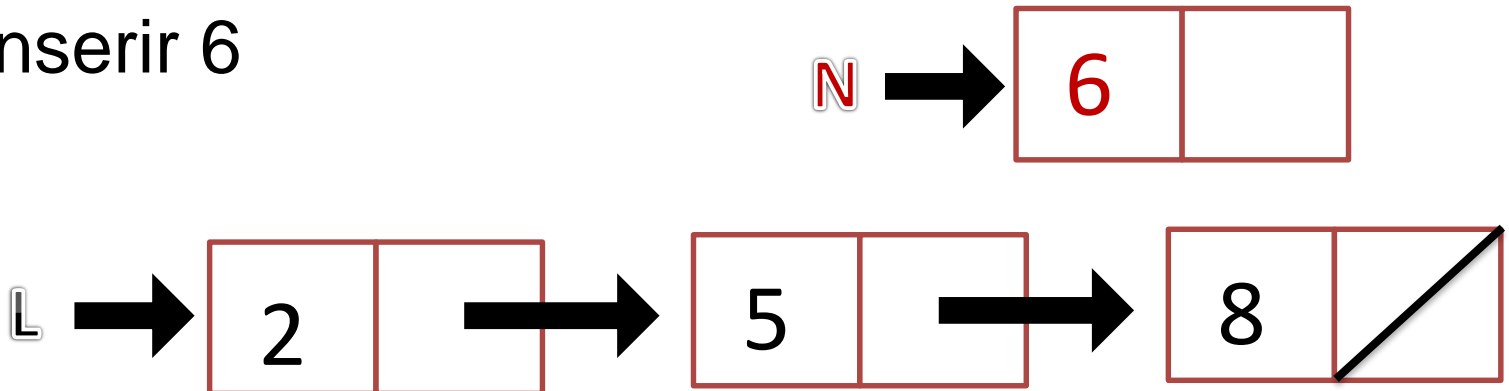
**Ex:** inserir 6



# Operação de Inserção (Lista Ordenada)

- **1o nó < Elemento ≤ último nó da lista:**
  - Alocar um novo nó e preencher o campo *info*

**Ex:** inserir 6



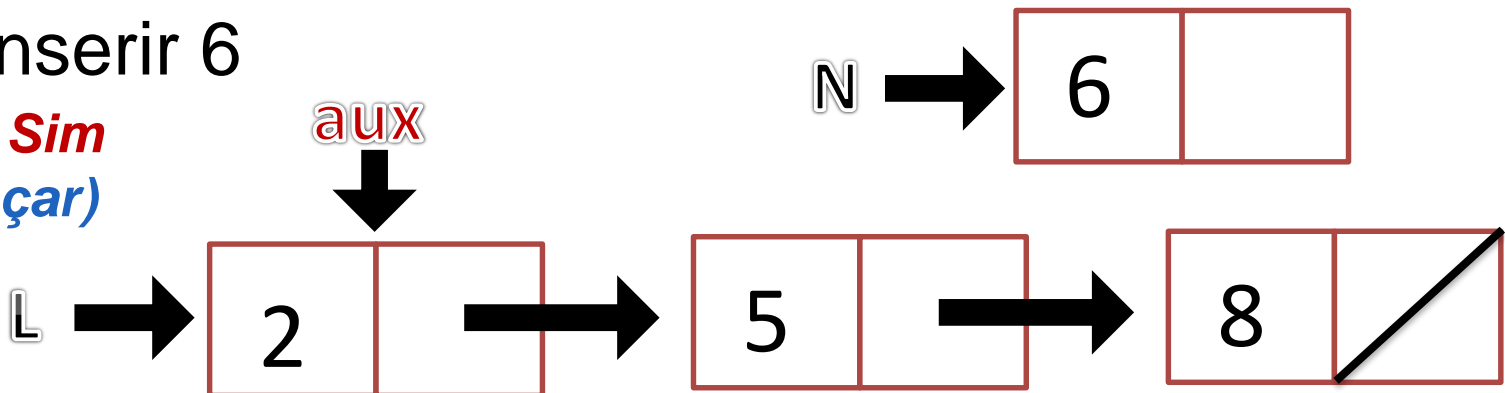


# Operação de Inserção (Lista Ordenada)

- **1o nó < Elemento ≤ último nó da lista:**
  - Alocar um novo nó e preencher o campo *info*
  - Percorrer a lista até achar nó maior ou igual
    - Usar ponteiro auxiliar
    - Verificar *info* do sucessor de *aux* (*aux->prox->info*)

**Ex:** inserir 6

**5 < 6? Sim**  
(*avancar*)

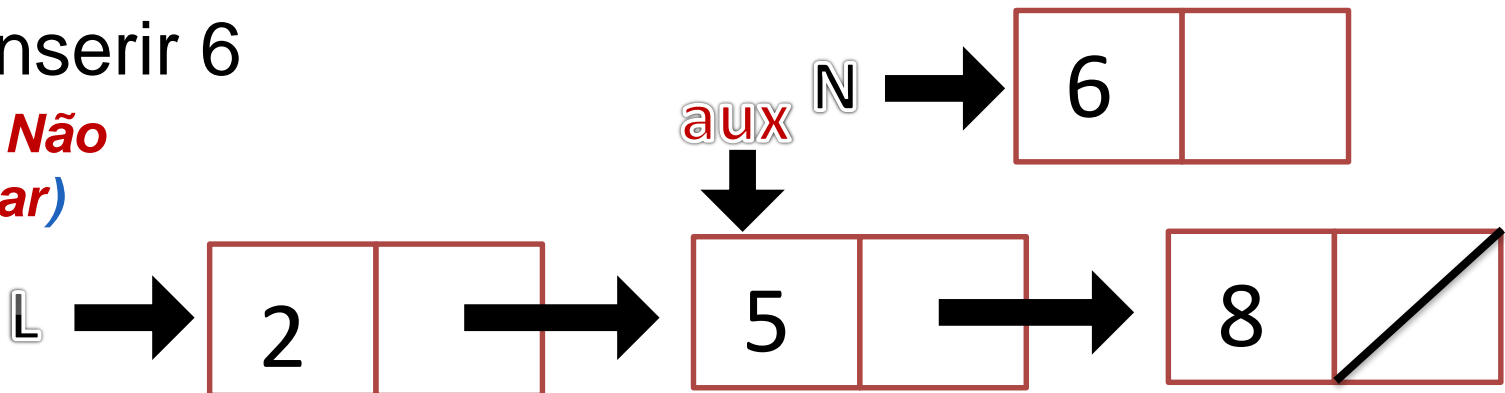


# Operação de Inserção (Lista Ordenada)

- **1o nó < Elemento ≤ último nó da lista:**
  - Alocar um novo nó e preencher o campo *info*
  - Percorrer a lista até achar nó maior ou igual
    - Usar ponteiro auxiliar
    - Verificar *info* do sucessor de *aux* (*aux->prox->info*)

**Ex:** inserir 6

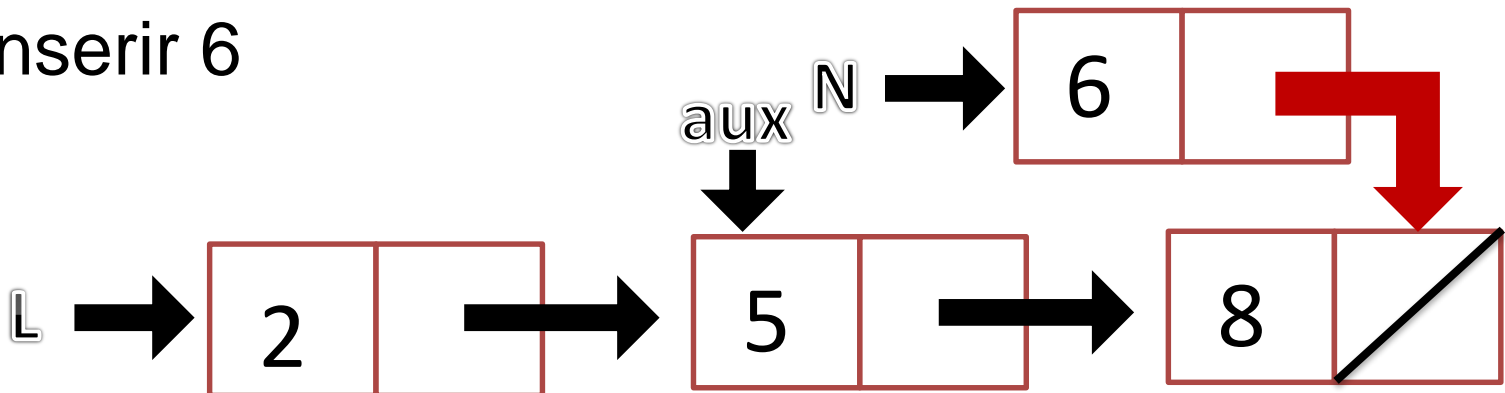
**8 < 6? Não**  
**(parar)**



# Operação de Inserção (Lista Ordenada)

- **1o nó < Elemento ≤ último nó da lista:**
  - Alocar um novo nó e preencher o campo *info*
  - Percorrer a lista até achar nó maior ou igual
    - Usar ponteiro auxiliar
    - Verificar *info* do **sucessor de *aux*** (*aux->prox->info*)
  - Preencher campo *prox* do novo nó

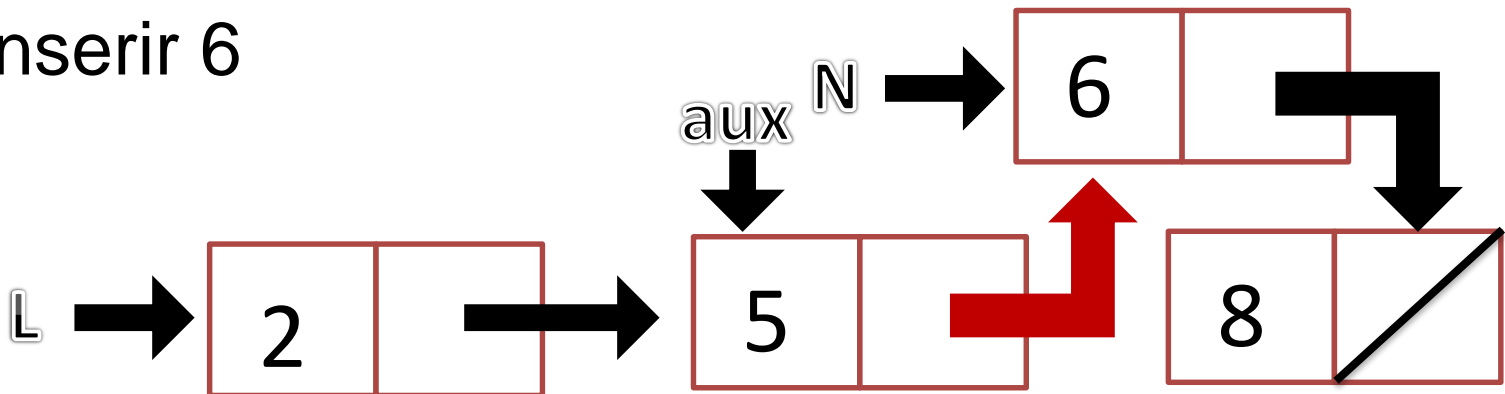
**Ex:** inserir 6



# Operação de Inserção (Lista Ordenada)

- **1o nó < Elemento ≤ último nó da lista:**
  - Alocar um novo nó e preencher o campo *info*
  - Percorrer a lista até achar nó maior ou igual
    - Usar ponteiro auxiliar
    - Verificar *info* do **sucessor de *aux*** (*aux->prox->info*)
  - Preencher campo *prox* do novo nó
  - Nó apontado por *aux* aponta para o novo nó

**Ex:** inserir 6

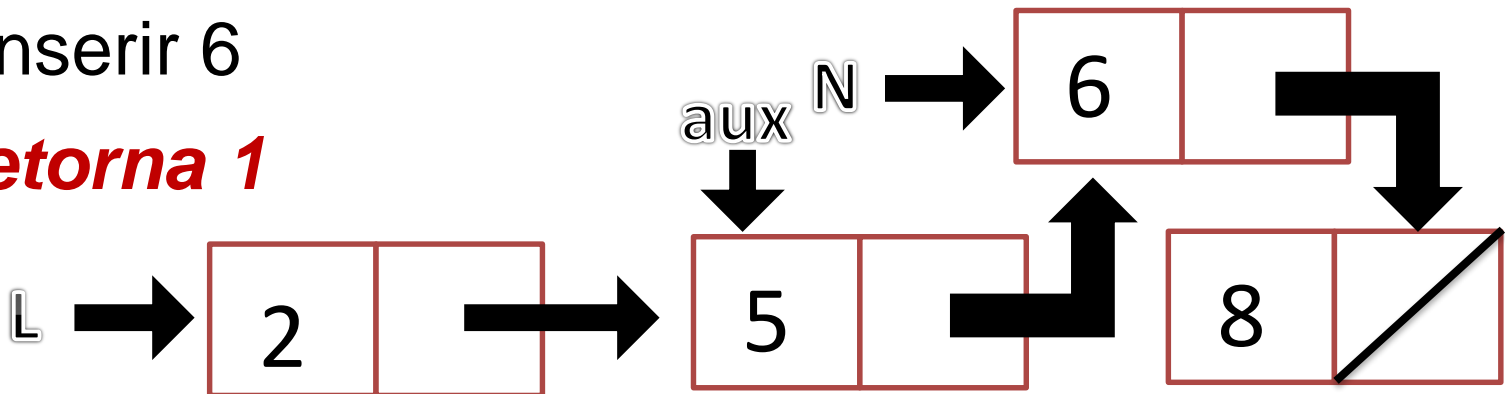


# Operação de Inserção (Lista Ordenada)

- **1o nó < Elemento ≤ último nó da lista:**
  - Alocar um novo nó e preencher o campo *info*
  - Percorrer a lista até achar nó maior ou igual
    - Usar ponteiro auxiliar
    - Verificar *info* do **sucessor de *aux*** (*aux->prox->info*)
  - Preencher campo *prox* do novo nó
  - Nó apontado por *aux* aponta para o novo nó

**Ex:** inserir 6

**retorna 1**



# Operação de Inserção (Lista Ordenada)

- Implementação em C:

```
int insere_ord (Lista *lst, int elem) {  
    // Aloca um novo nó  
    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)  
    if (lista_vazia(*lst) || elem <= (*lst)->info) {  
        N->prox = *lst; // Aponta para o 1º nó atual da lista  
        *lst = N; // Faz a lista apontar para o novo nó  
        return 1; }  
}
```

...

# Operação de Inserção (Lista Ordenada)

- Implementação em C:

```
int insere_ord (Lista *lst, int elem) {  
    // Aloca um novo nó  
    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)  
    if (lista_vazia(*lst) || elem <= (*lst)->info) {  
        N->prox = *lst; // Aponta para o 1º nó atual da lista  
        *lst = N; // Faz a lista apontar para o novo nó  
        return 1; }  
}
```

...

# Operação de Inserção (Lista Ordenada)

- Implementação em C:

...

*// Percorrimento da lista (elem > 1º nó da lista)*

*Lista aux = \*lst; // Faz aux apontar para 1º nó*

*while (aux->prox != NULL && aux->prox->info < elem)*

*aux = aux->prox; // Avança*

*// Insere o novo elemento na lista*

*N->prox = aux->prox;*

*aux->prox = N;*

*return 1;*



# Operação de Remoção

- Necessita de percorrimento da lista
  - Busca pelo elemento a ser removido
- Remoção no meio **NÃO** envolve movimentação dos nós
  - Apenas **mudança nos ponteiros**
- **Critério de ordenação** afeta quando não existe o elemento na lista
  - **Lista não ordenada:** tem que percorrer **até o final**
  - **Lista ordenada:** percorrer **até achar nó maior**

# Operação de Remoção (Lista NÃO Ordenada)

- Existem 4 casos possíveis de remoção:
  - Lista vazia
  - Elemento igual ao 1º nó da lista
  - Elemento entre o 2º e o último nó da lista
  - Elemento não está na lista

# Operação de Remoção (Lista NÃO Ordenada)

- Lista vazia:

Ex: remover 5

**L** → **NULL**

# Operação de Remoção (Lista NÃO Ordenada)

- **Lista vazia:**
  - Não existe elemento a ser removido

Ex: remover 5

~~5~~

L → NULL

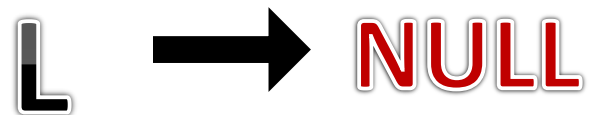
# Operação de Remoção (Lista NÃO Ordenada)

- **Lista vazia:**
  - Não existe elemento a ser removido
  - Retorna 0 (**operação falha**)

Ex: remover 5

~~5~~

*retorna 0*



# Operação de Remoção (Lista NÃO Ordenada)

- Elemento igual ao 1º nó da lista:

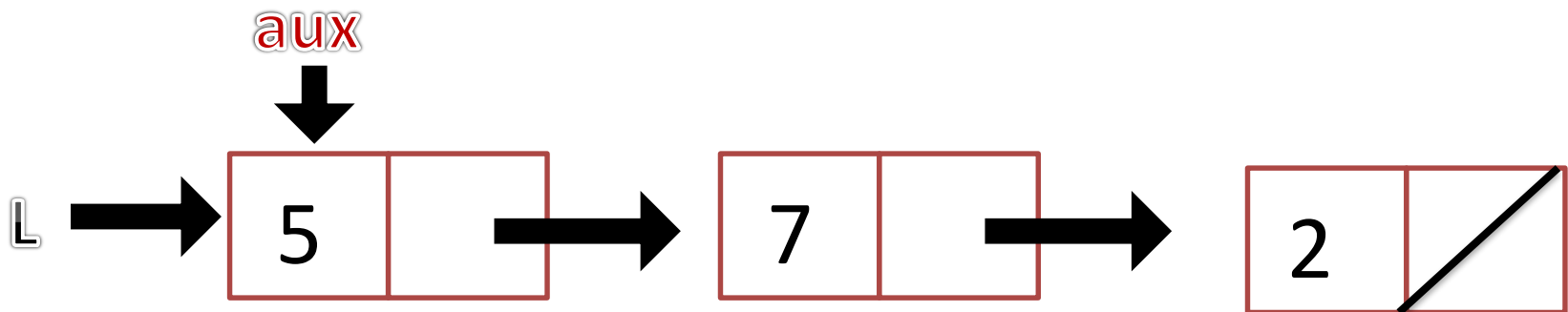
**Ex:** remover 5



# Operação de Remoção (Lista NÃO Ordenada)

- **Elemento igual ao 1º nó da lista:**
  - Colocar um ponteiro auxiliar no 1º nó

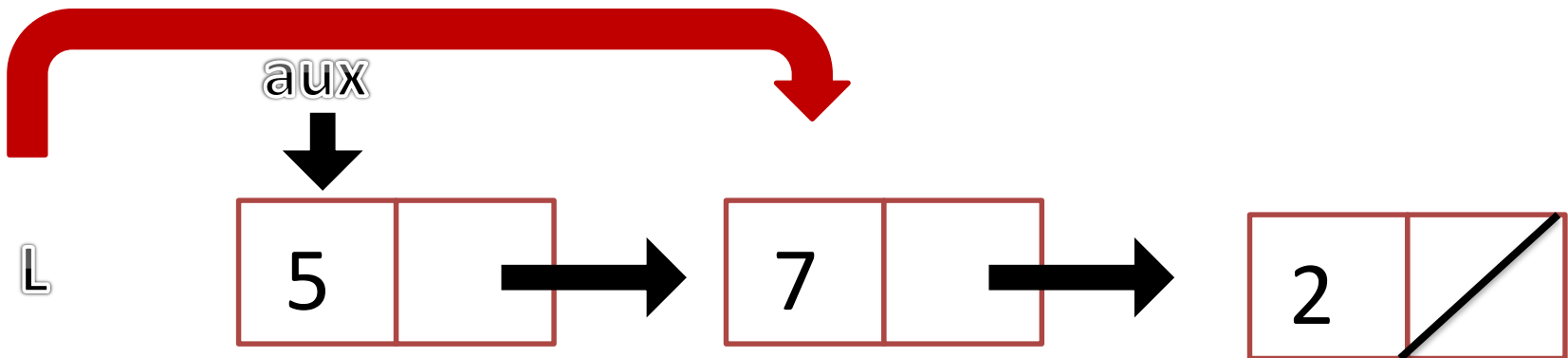
**Ex:** remover 5



# Operação de Remoção (Lista NÃO Ordenada)

- **Elemento igual ao 1º nó da lista:**
  - Colocar um ponteiro auxiliar no 1º nó
  - Mudar a lista para apontar para o 2º nó

Ex: remover 5

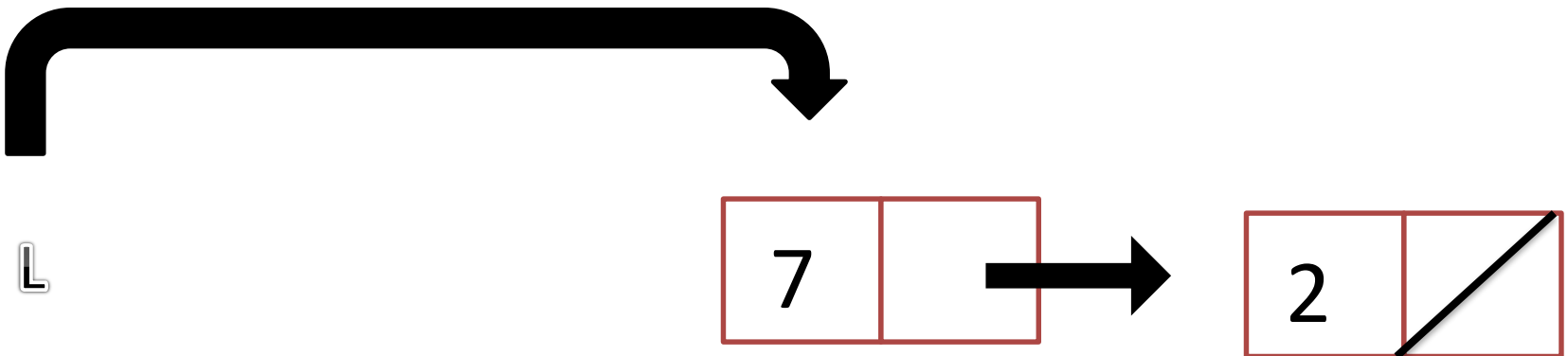




# Operação de Remoção (Lista NÃO Ordenada)

- **Elemento igual ao 1º nó da lista:**
  - Colocar um ponteiro auxiliar no 1º nó
  - Mudar a lista para apontar para o 2º nó
  - Liberar o espaço do nó apontado por *aux*

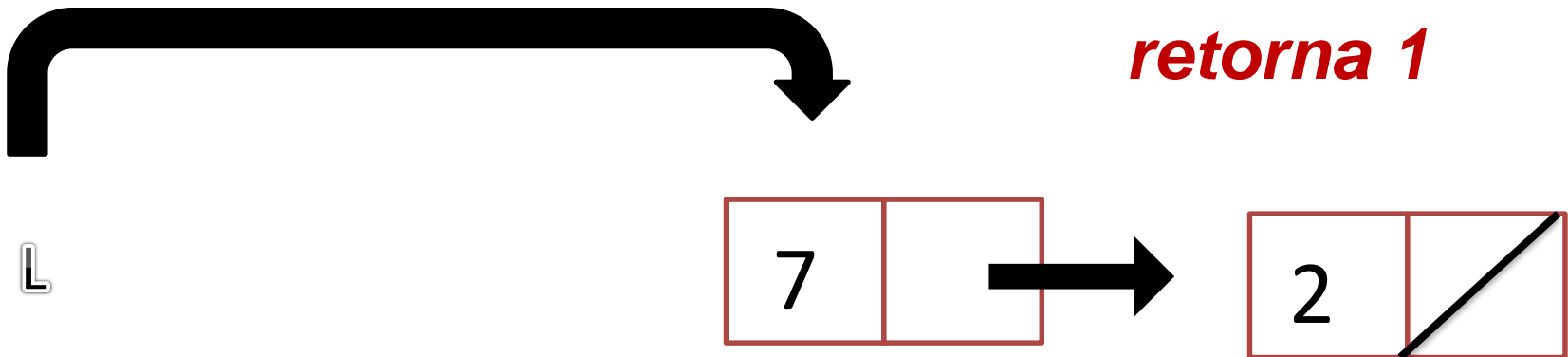
Ex: remover 5



# Operação de Remoção (Lista NÃO Ordenada)

- **Elemento igual ao 1º nó da lista:**
  - Colocar um ponteiro auxiliar no 1º nó
  - Mudar a lista para apontar para o 2º nó
  - Liberar o espaço do nó apontado por *aux*
  - Retornar 1 (**operação bem sucedida**)

Ex: remover 5



# Operação de Remoção (Lista NÃO Ordenada)

- Elemento entre o 1º e o último nó da lista:

Ex: remover 3

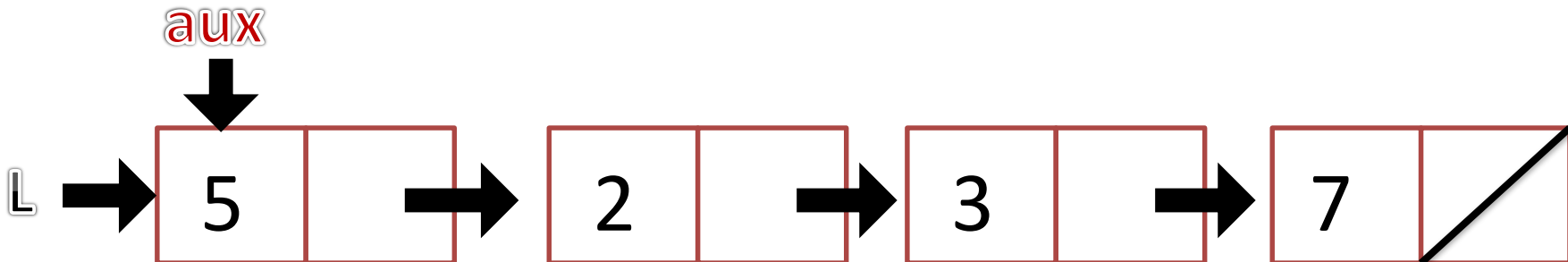


# Operação de Remoção

## (Lista NÃO Ordenada)

- **Elemento entre o 1º e o último nó da lista:**
  - Colocar um ponteiro auxiliar no 1º nó

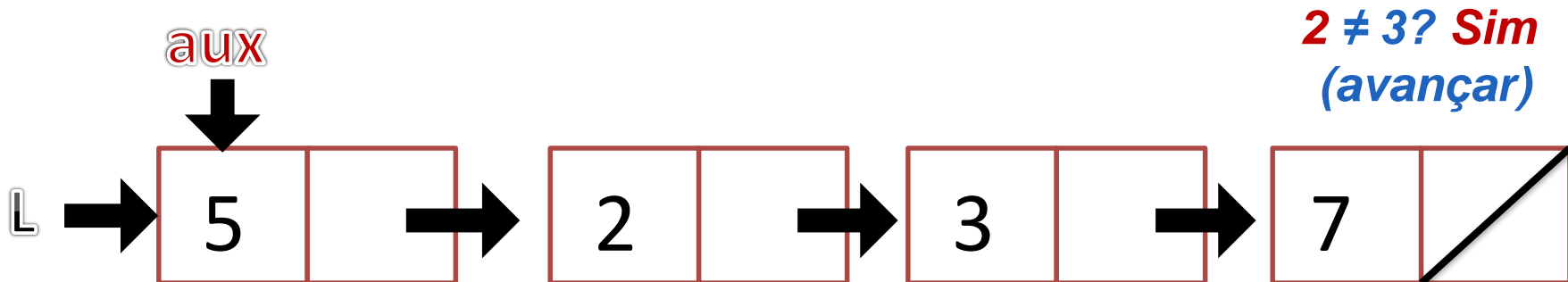
Ex: remover 3



# Operação de Remoção (Lista NÃO Ordenada)

- **Elemento entre o 1º e o último nó da lista:**
  - Colocar um ponteiro auxiliar no 1º nó
  - Percorrer a lista até encontrar o elemento
    - Avançar **aux** enquanto o **sucessor de aux**  $\neq$  elem

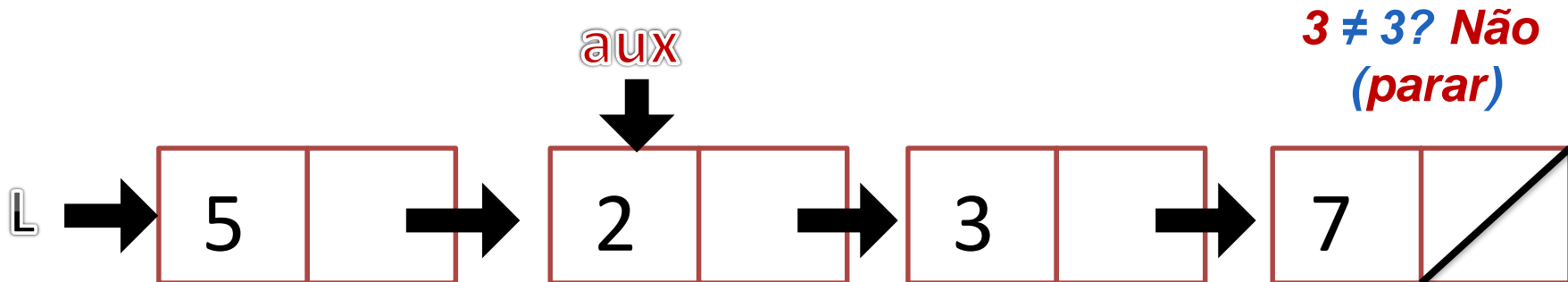
Ex: remover 3



# Operação de Remoção (Lista NÃO Ordenada)

- **Elemento entre o 1º e o último nó da lista:**
  - Colocar um ponteiro auxiliar no 1º nó
  - Percorrer a lista até encontrar o elemento
    - Avançar **aux** enquanto o **sucessor de aux**  $\neq$  elem

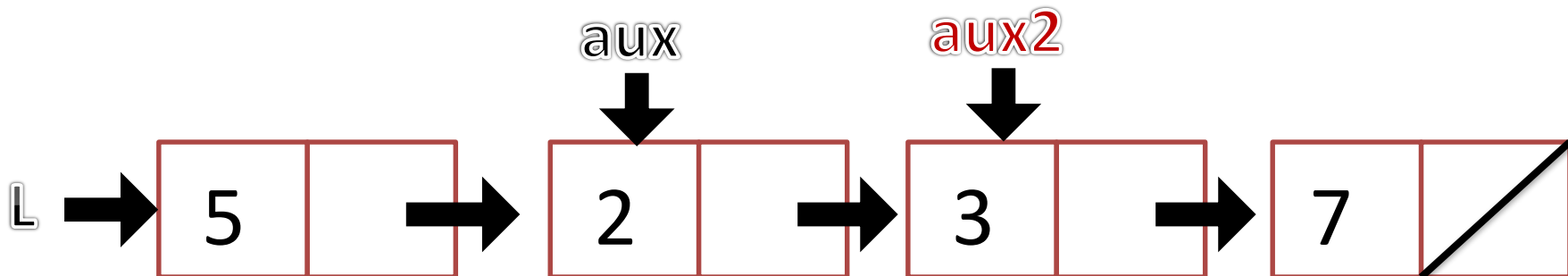
Ex: remover 3



# Operação de Remoção (Lista NÃO Ordenada)

- **Elemento entre o 1º e o último nó da lista:**
  - Colocar um ponteiro auxiliar no 1º nó
  - Percorrer a lista até encontrar o elemento
    - Avançar **aux** enquanto o **sucessor de aux**  $\neq$  elem
  - Colocar outro ponteiro auxiliar no nó a ser removido

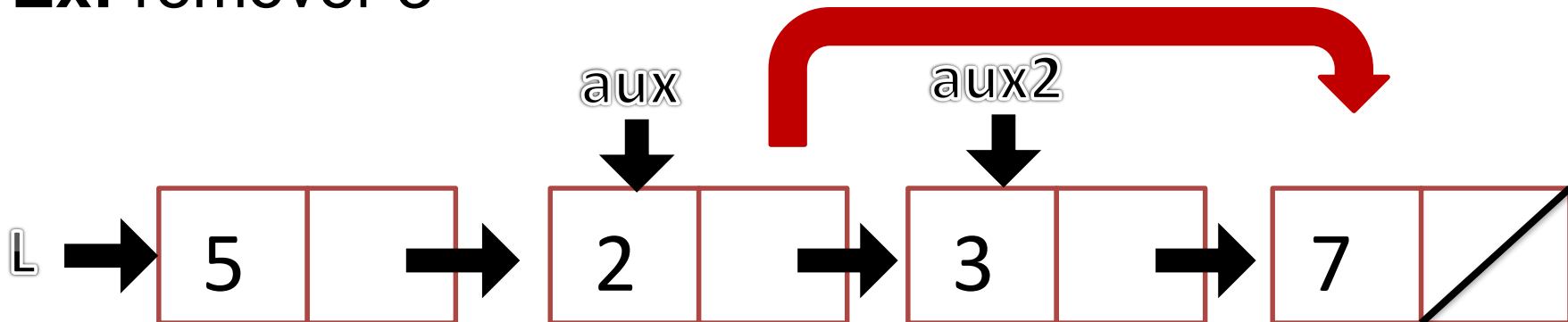
**Ex:** remover 3



# Operação de Remoção (Lista NÃO Ordenada)

- **Elemento entre o 1º e o último nó da lista:**
  - Fazer o nó apontado por ***aux*** apontar para o **sucessor** do nó apontado por ***aux2***

Ex: remover 3

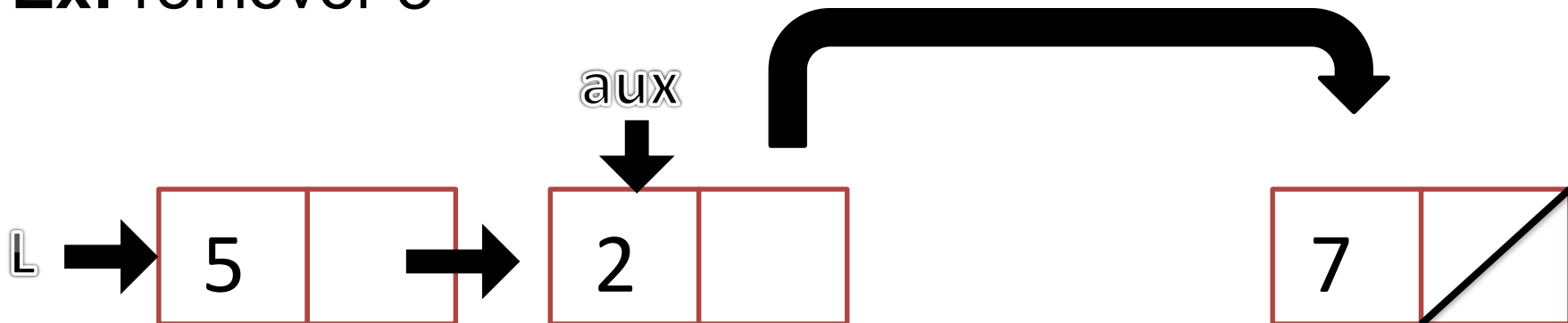




# Operação de Remoção (Lista NÃO Ordenada)

- **Elemento entre o 1º e o último nó da lista:**
  - Fazer o nó apontado por ***aux*** apontar para o **sucessor** do nó apontado por ***aux2***
  - Liberar o espaço do nó apontado por ***aux2***

Ex: remover 3

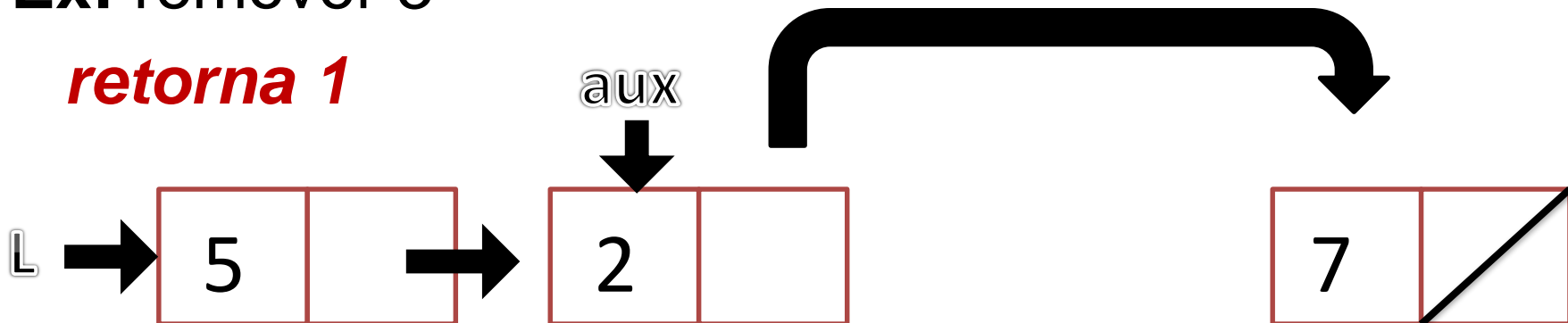


# Operação de Remoção (Lista NÃO Ordenada)

- **Elemento entre o 1º e o último nó da lista:**
  - Fazer o nó apontado por ***aux*** apontar para o **sucessor** do nó apontado por ***aux2***
  - Liberar o espaço do nó apontado por ***aux2***
  - Retornar 1 (**operação bem sucedida**)

Ex: remover 3

***retorna 1***



# Operação de Remoção (Lista NÃO Ordenada)

- **Elemento não está na lista:**

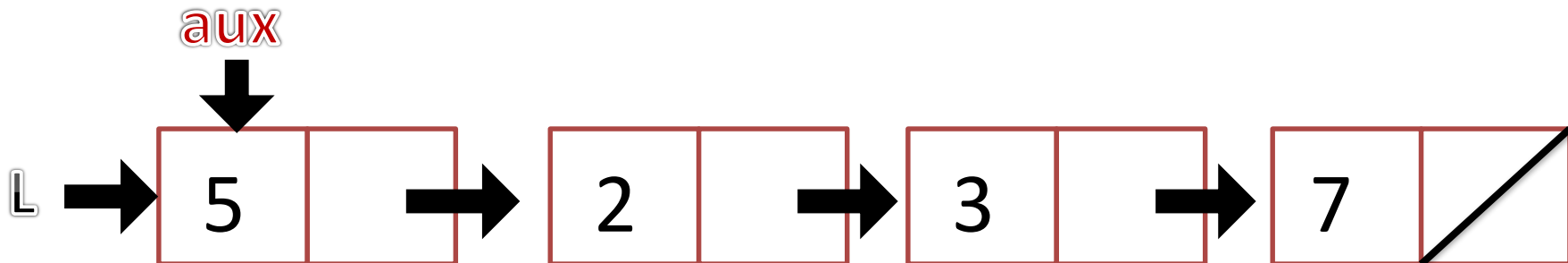
**Ex:** remover 1



# Operação de Remoção (Lista NÃO Ordenada)

- **Elemento não está na lista:**
  - Colocar um ponteiro auxiliar no 1º nó

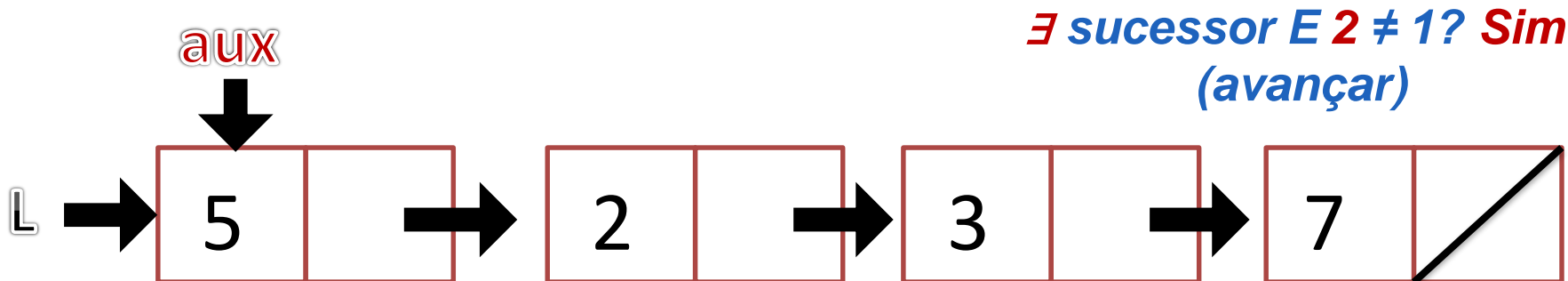
Ex: remover 1



# Operação de Remoção (Lista NÃO Ordenada)

- **Elemento não está na lista:**
  - Colocar um ponteiro auxiliar no 1º nó
  - Percorrer a lista até o seu final
    - Avançar **aux** enquanto  $\exists$  **sucessor de aux**

Ex: remover 1

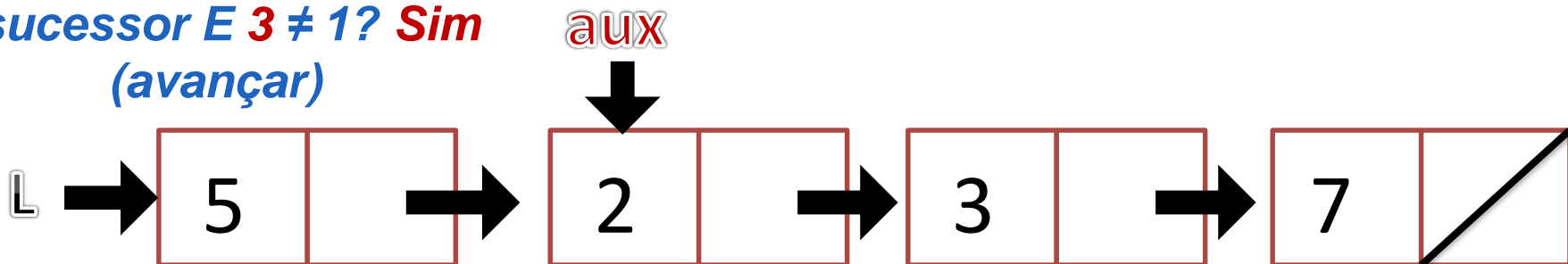


# Operação de Remoção (Lista NÃO Ordenada)

- **Elemento não está na lista:**
  - Colocar um ponteiro auxiliar no 1º nó
  - Percorrer a lista até o seu final
    - Avançar **aux** enquanto  $\exists$  **sucessor de aux**

Ex: remover 1

$\exists$  sucessor  $E$   $3 \neq 1$ ? **Sim**  
(*avançar*)

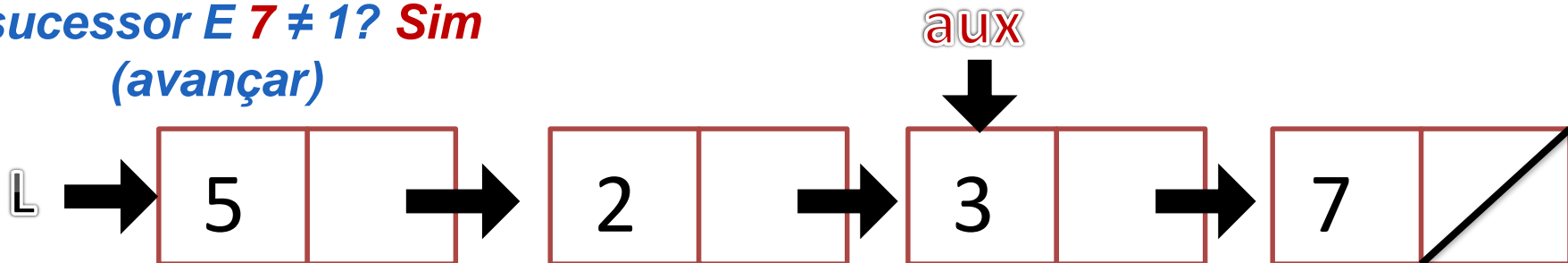


# Operação de Remoção (Lista NÃO Ordenada)

- **Elemento não está na lista:**
  - Colocar um ponteiro auxiliar no 1º nó
  - Percorrer a lista até o seu final
    - Avançar **aux** enquanto  $\exists$  **sucessor de aux**

**Ex:** remover 1

$\exists$  **sucessor** **E** **7**  $\neq$  **1**? **Sim**  
(*avançar*)



# Operação de Remoção (**Lista NÃO Ordenada**)

- **Elemento não está na lista:**
  - Colocar um ponteiro auxiliar no 1º nó
  - Percorrer a lista até o seu final
    - Avançar **aux** enquanto  $\exists$  **sucessor de aux**
  - Não existe o elemento

**Ex:** remover 1

$\exists$  **sucessor?** **Não**  
(*parar*)

~~1~~





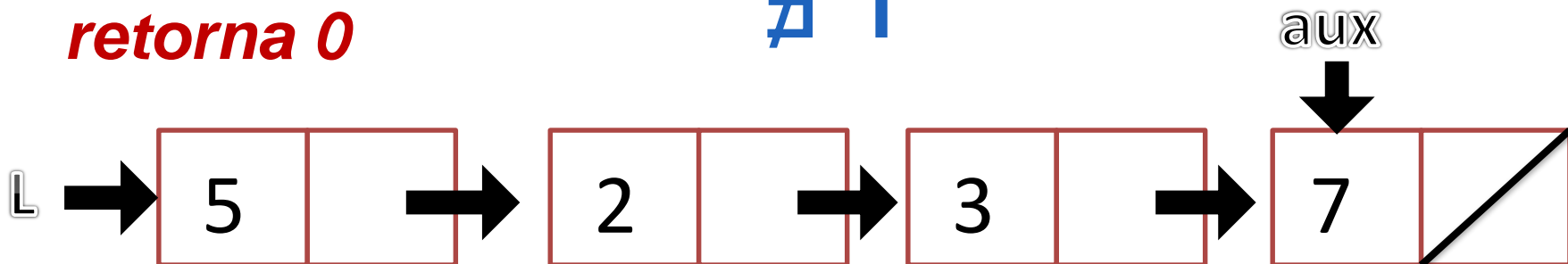
# Operação de Remoção (Lista NÃO Ordenada)

- **Elemento não está na lista:**
  - Colocar um ponteiro auxiliar no 1º nó
  - Percorrer a lista até o seu final
    - Avançar **aux** enquanto  $\exists$  **sucessor de aux**
  - Não existe o elemento
    - Retornar 0 (**operação falha**)

**Ex:** remover 1

**retorna 0**

~~1~~



# Operação de Remoção (Lista NÃO Ordenada)

- Implementação em C:

```
int remove_elem (Lista *lst, int elem) {  
    if (lista_vazia(lst) == 1)  
        return 0; // Falha  
  
    Lista aux = *lst; // Ponteiro auxiliar para o 1º nó  
    // Trata elemento = 1º nó da lista  
    if (elem == (*lst)->info) {  
        *lst = aux->prox; // Lista aponta para o 2º nó  
        free(aux); // Libera memória alocada  
        return 1; }  
}
```

...

# Operação de Remoção (Lista NÃO Ordenada)

- Implementação em C:

```
...  
// Percorrimento até achar o elem ou final de lista  
while (aux->prox != NULL && aux->prox->info != elem)  
    aux = aux->prox;  
if (aux->prox == NULL) // Trata final de lista  
    return 0; // Falha  
// Remove elemento ≠ 1º nó da lista  
Lista aux2 = aux->prox; // Aponta nó a ser removido  
aux->prox = aux2->prox; // Retira nó da lista  
free(aux2); // Libera memória alocada  
return 1;  
}
```

# Operação de Remoção (Lista Ordenada)

- Existem 6 cenários possíveis de remoção:
  - Lista está vazia
  - Elemento é o último nó da lista
  - Elemento está entre o 1º e o penúltimo nó da lista
  - Elemento não está na lista

# Operação de Remoção (Lista Ordenada)

- Existem **6 cenários** possíveis de remoção:

- Lista está vazia
- Elemento é o último nó da lista
- Elemento está entre o 1º e o penúltimo nó da lista
- Elemento não está na lista

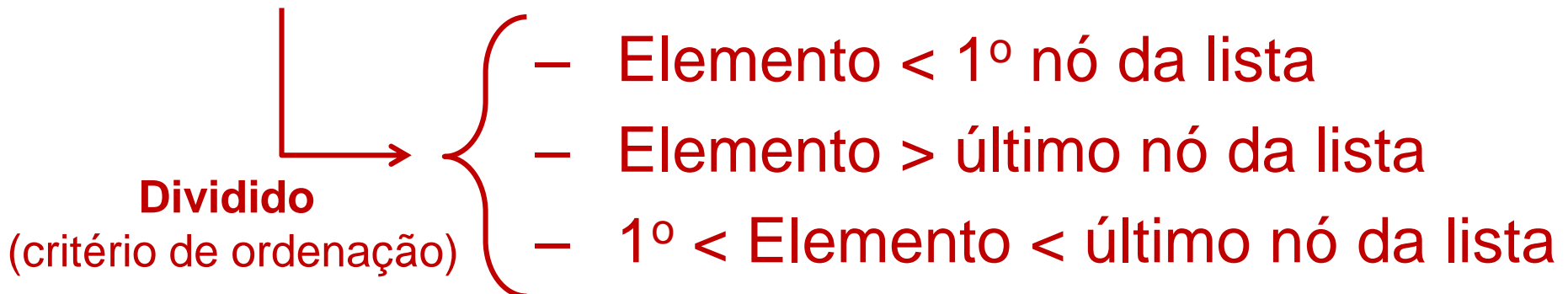


similar ao TAD  
Lista NÃO Ordenada

# Operação de Remoção (Lista Ordenada)

- Existem 6 cenários possíveis de remoção:

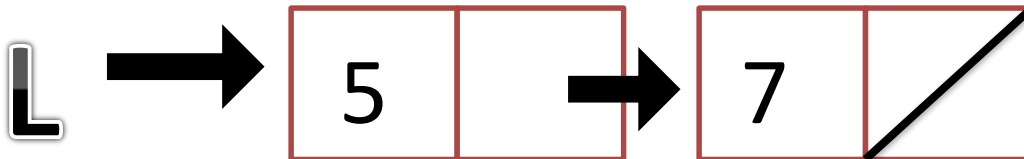
- Lista está vazia
- Elemento é o último nó da lista
- Elemento está entre o 1º e o penúltimo nó da lista
- ~~– Elemento não está na lista~~



# Operação de Remoção (Lista Ordenada)

- Elemento  $< 1^{\circ}$  nó da lista:

**Ex:** remover 2



# Operação de Remoção (Lista Ordenada)

- Elemento  $< 1^{\circ}$  nó da lista:
  - Não existe o elemento

Ex: remover 2

~~2~~ 2





# Operação de Remoção (Lista Ordenada)

- **Elemento  $< 1^{\circ}$  nó da lista:**
  - Não existe o elemento
  - Retorna 0 (**operação falha**)

**Ex:** remover 2

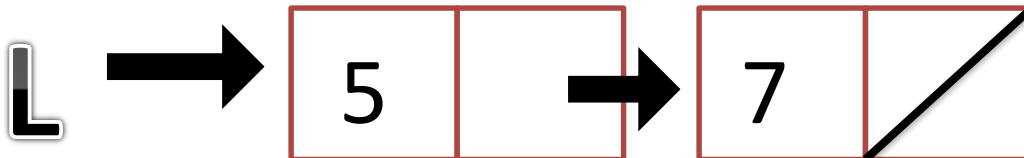


***Retorna 0***

# Operação de Remoção (Lista Ordenada)

- Elemento > último nó da lista:

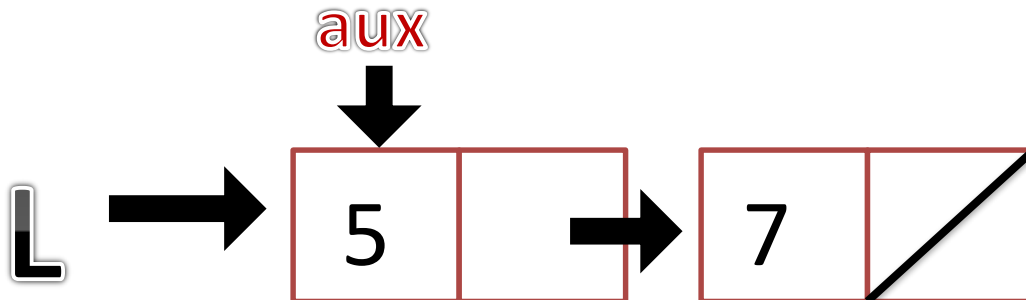
Ex: remover 11



# Operação de Remoção (Lista Ordenada)

- **Elemento > último nó da lista:**
  - Percorre a lista **até o seu final**
    - Faz um ponteiro auxiliar apontar para o 1º nó

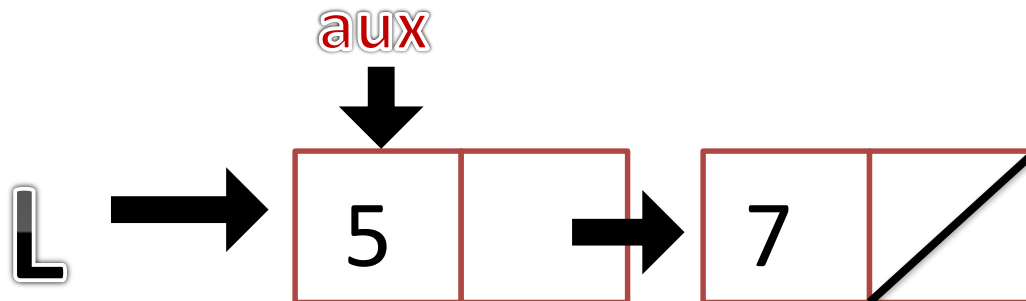
**Ex:** remover 11



# Operação de Remoção (Lista Ordenada)

- **Elemento > último nó da lista:**
  - Percorre a lista **até o seu final**
    - Faz um ponteiro auxiliar apontar para o 1º nó
    - Avançar o ponteiro enquanto  $\exists$  sucessor  
( *$aux \rightarrow prox \neq NULL$* )

**Ex:** remover 11

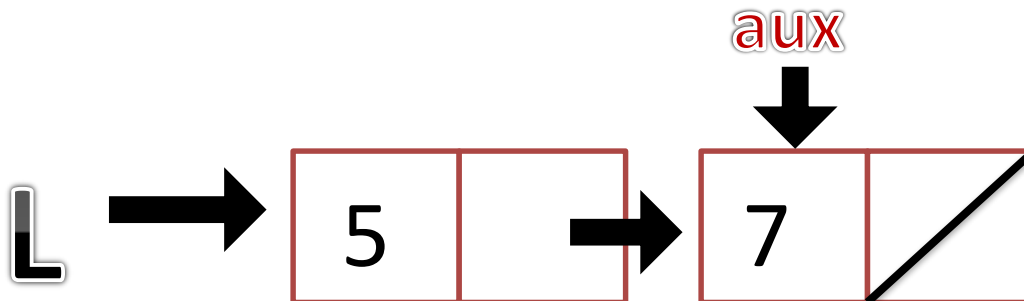


*$\exists$  sucessor E  $7 < 11$ ? Sim  
(avançar)*

# Operação de Remoção (Lista Ordenada)

- **Elemento > último nó da lista:**
  - Percorre a lista **até o seu final**
    - Faz um ponteiro auxiliar apontar para o 1º nó
    - Avançar o ponteiro enquanto  $\exists$  sucessor  
( *$aux \rightarrow prox \neq NULL$* )

**Ex:** remover 11

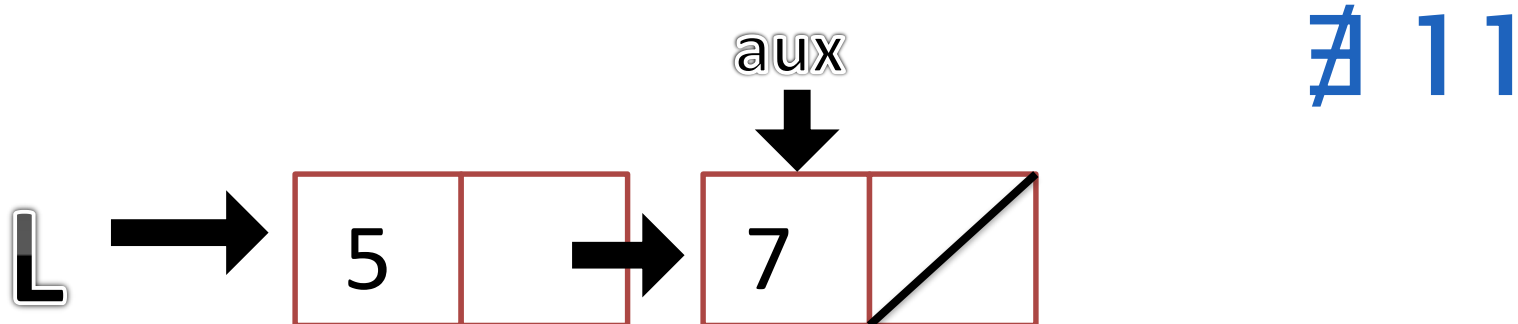


*$\exists$  sucessor? Não  
(parar)*

# Operação de Remoção (Lista Ordenada)

- **Elemento > último nó da lista:**
  - Percorre a lista **até o seu final**
    - Faz um ponteiro auxiliar apontar para o 1º nó
    - Avançar o ponteiro enquanto  $\exists$  sucessor  
(*aux->prox*  $\neq$  NULL)
  - Elemento não está na lista

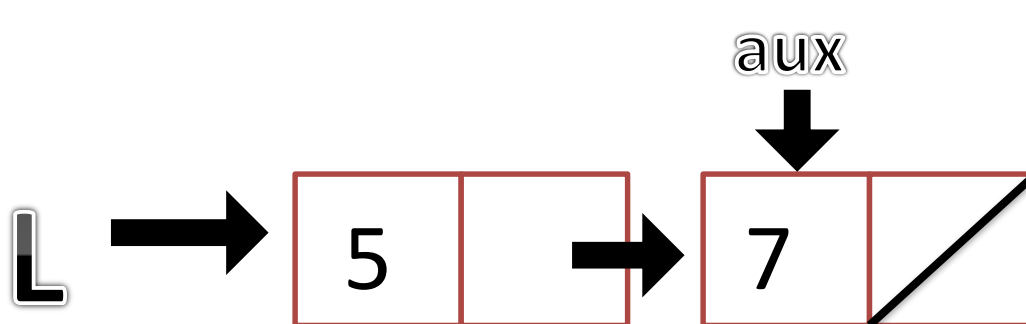
**Ex:** remover 11



# Operação de Remoção (Lista Ordenada)

- **Elemento > último nó da lista:**
  - Percorre a lista **até o seu final**
    - Faz um ponteiro auxiliar apontar para o 1º nó
    - Avançar o ponteiro enquanto  $\exists$  sucessor  
(*aux->prox  $\neq$  NULL*)
  - Elemento não está na lista
    - Retorna 0 (**operação falha**)

**Ex:** remover 11



~~11~~  
**Retorna 0**

# Operação de Remoção (Lista Ordenada)

- 1º < Elemento < último nó da lista:

Ex: remover 7

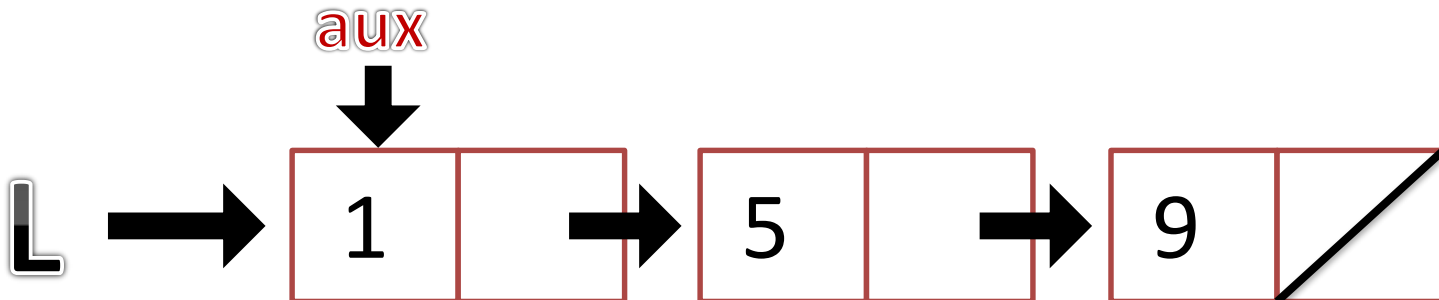




# Operação de Remoção (Lista Ordenada)

- **1º < Elemento < último nó da lista:**
  - Percorre a lista até encontrar nó > elemento
    - Faz um ponteiro auxiliar apontar para o 1º nó

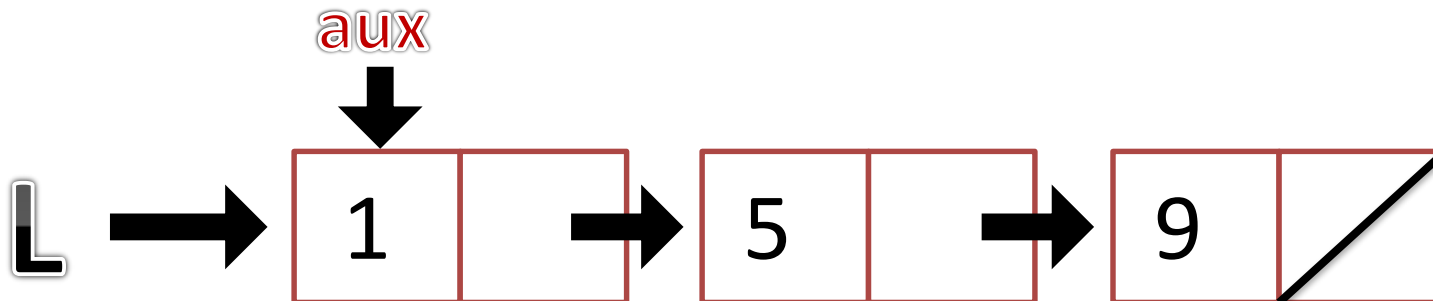
Ex: remover 7



# Operação de Remoção (Lista Ordenada)

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

Ex: remover 7

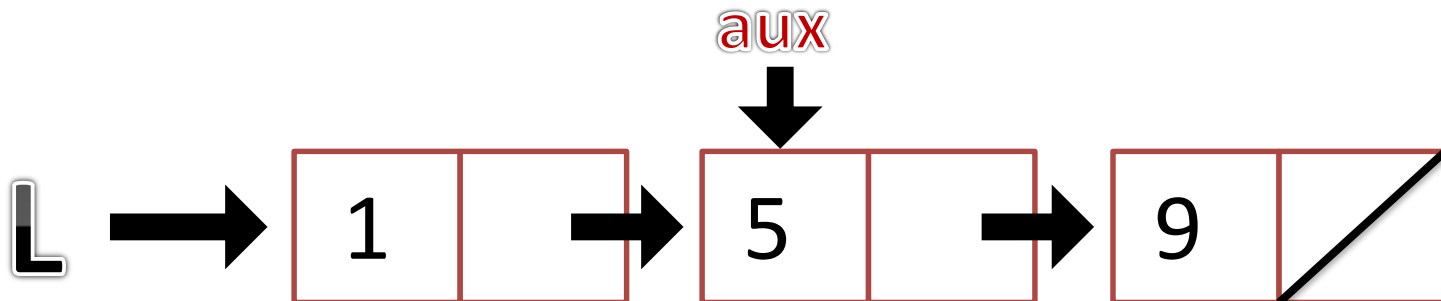


**5 < 7? Sim**  
(*avançar*)

# Operação de Remoção (Lista Ordenada)

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

Ex: remover 7

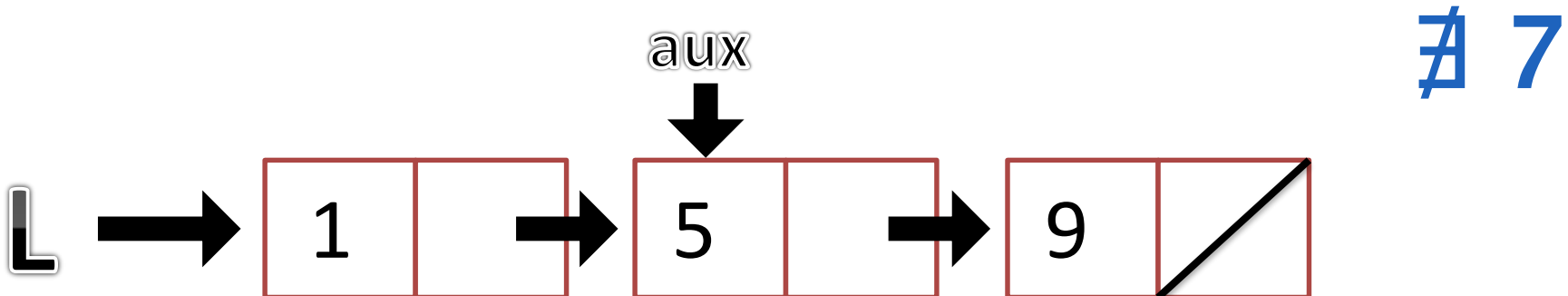


**9 < 7? Não**  
(*parar*)

# Operação de Remoção (Lista Ordenada)

- **1º < Elemento < último nó da lista:**
  - Percorre a lista até encontrar nó > elemento
    - Faz um ponteiro auxiliar apontar para o 1º nó
    - Avançar o ponteiro enquanto **sucessor < elemento**  
(*aux->prox->info < elem*)
  - Elemento não está na lista

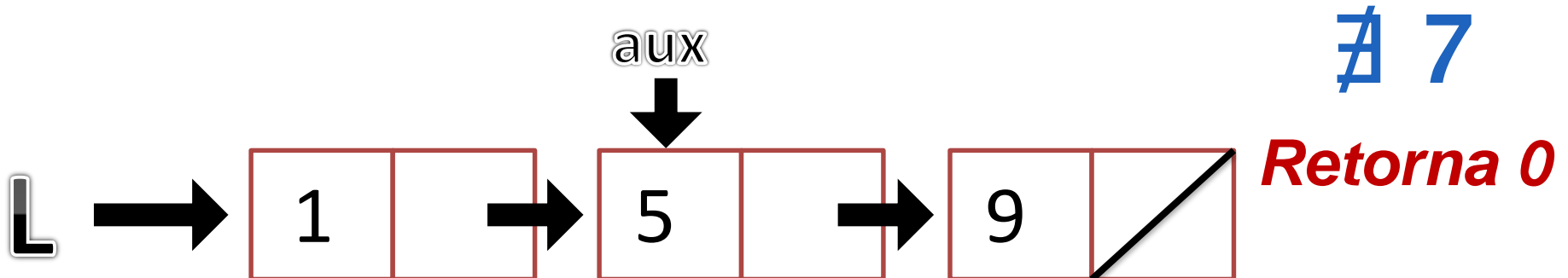
Ex: remover 7



# Operação de Remoção (Lista Ordenada)

- **1º < Elemento < último nó da lista:**
  - Percorre a lista **até encontrar nó > elemento**
    - Faz um ponteiro auxiliar apontar para o 1º nó
    - Avançar o ponteiro enquanto **sucessor < elemento**  
(*aux->prox->info < elem*)
  - Elemento não está na lista
    - Retorna 0 (**operação falha**)

**Ex:** remover 7



# Operação de Remoção (Lista Ordenada)

- Implementação em C:

```
int remove_ord (Lista *lst, int elem) {  
    if (lista_vazia(lst) == 1 || elem < (*lst)->info)  
        return 0; // Falha  
  
    Lista aux = *lst; // Ponteiro auxiliar para o 1º nó  
    // Trata elemento = 1º nó da lista  
    if (elem == (*lst)->info) {  
        *lst = aux->prox; // Lista aponta para o 2º nó  
        free(aux); // Libera memória alocada  
        return 1; }  
}
```

...

# Operação de Remoção (Lista Ordenada)

- Implementação em C:

```
int remove_ord (Lista *lst, int elem) {  
    if (lista_vazia(lst) == 1 || elem < (*lst)->info)  
        return 0; // Falha  
  
    Lista aux = *lst; // Ponteiro auxiliar para o 1º nó  
    // Trata elemento = 1º nó da lista  
    if (elem == (*lst)->info) {  
        *lst = aux->prox; // Lista aponta para o 2º nó  
        free(aux); // Libera memória alocada  
        return 1; }  
}
```

...

# Operação de Remoção (Lista Ordenada)

- Implementação em C:

...

*// Percorrimento até final de lista, achar elem ou nó maior*

*while (aux->prox != NULL && aux->prox->info < elem)*

*aux = aux->prox;*

*if (aux->prox == NULL || aux->prox->info > elem)*

*return 0; // Falha*

*// Remove elemento ≠ 1º nó da lista*

*Lista aux2 = aux->prox; // Aponta nó a ser removido*

*aux->prox = aux2->prox; // Retira nó da lista*

*free(aux2); // Libera memória alocada*

*return 1;*

}



# Operação de Remoção (Lista Ordenada)

- Implementação em C:

```
...  
// Percorrimento até final de lista, achar elem ou nó maior  
while (aux->prox != NULL && aux->prox->info < elem)  
    aux = aux->prox;  
if (aux->prox == NULL || aux->prox->info > elem)  
    return 0; // Falha  
// Remove elemento ≠ 1º nó da lista  
Lista aux2 = aux->prox; // Aponta nó a ser removido  
aux->prox = aux2->prox; // Retira nó da lista  
free(aux2); // Libera memória alocada  
return 1;  
}
```

# Exercícios

1. *Implementar, utilizando a alocação dinâmica e o acesso encadeado, o TAD lista linear não ordenada de números inteiros. Nessa implementação deve contemplar as operações básicas: criar\_lista, lista\_vazia, lista\_cheia, insere\_elem, remove\_elem 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*

2. *Repita a implementação acima para o TAD lista ordenada.*

# Exercícios

3. *Altere a implementação do exercício 1 para contemplar uma lista não ordenada de bebidas, com a seguinte estrutura:*

Nome	Volume (ml)	Preço
char[20]	int	float

*Crie um programa aplicativo similar àquele desenvolvido nos exercícios de alocação dinâmica, ou seja, com as seguintes opções:*

- [1] Inserir registro*
- [2] Apagar último registro*
- [3] Imprimir tabela*
- [4] Sair*

# 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.*