



## Aula 13 - Técnicas de Encadeamento (Encadeamento Duplo)

**Prof. Me. Claudiney R. Tinoco**  
[profclaudineytinoco@gmail.com](mailto:profclaudineytinoco@gmail.com)

Faculdade de Computação (FACOM)  
Bacharelado em Ciência da Computação (BCC)  
Bacharelado em Sistemas de Informação (BSI)

Algoritmos e Estruturas de Dados 1 (AED1)  
GBC024 - GSI006



# 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 =

INFO	PROX
------	------

# 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**





# Listas Duplamente Encadeadas

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

$L \rightarrow \text{NULL}$

# Listas Duplamente Encadeadas

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

$L \rightarrow \text{NULL}$

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



# Listas Duplamente Encadeadas

- 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

```
List criando() {  
    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:**
  - **Insere 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:**
  - **Insere elemento**
    - Melhor local de inserção é no início da lista
    - Evita percorrimento da lista
  - **Remove elemento**
    - Encontrar o elemento envolve percorrimento
    - Deve percorrer até o final da lista para determinar que o elemento não existe



# Insere Elemento

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



# Insere Elemento

- Lista vazia:

**Ex:** inserir 5

**L → NULL**

# Insere Elemento

- **Lista vazia:**
  - Aloca o novo nó

**Ex:** inserir 5



# Insere Elemento

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

**Ex:** inserir 5





# Insere 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



# Insere 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



# Insere 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



# Insere 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



# Insere Elemento

- Lista com elementos:

**Ex:** inserir 2

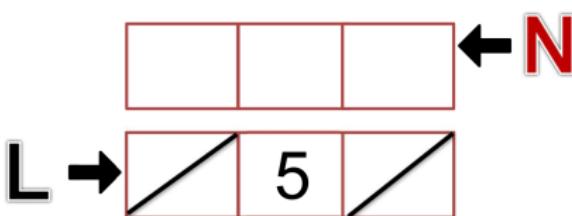




# Insere Elemento

- Lista com elementos:
  - Aloca o novo nó

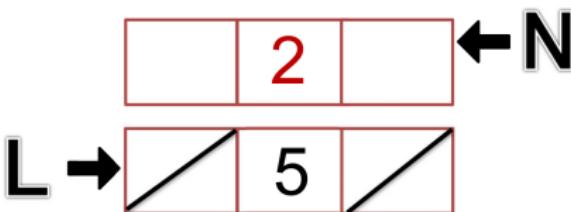
Ex: inserir 2



# Insere Elemento

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

Ex: inserir 2

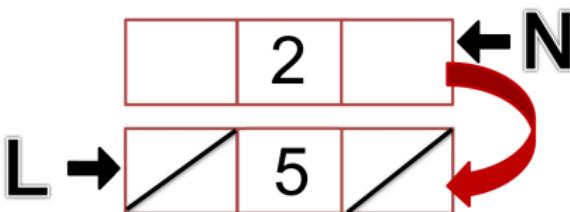




# Insere Elemento

- 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 ( $N->prox=L$ )

Ex: inserir 2

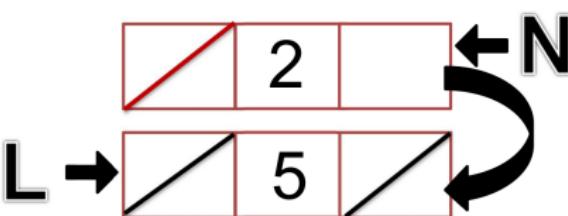




# Insere Elemento

- **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 ( $N->prox=L$ )
    - Campo **ant** com **NULL** ( $N->ant = L->ant$ )

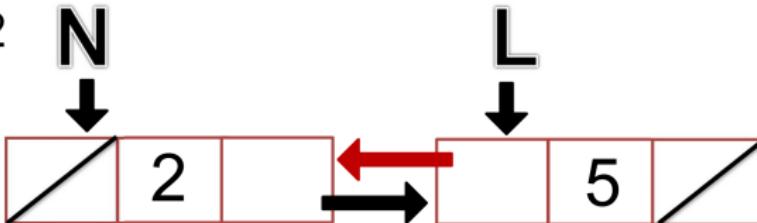
Ex: inserir 2



# Insere Elemento

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

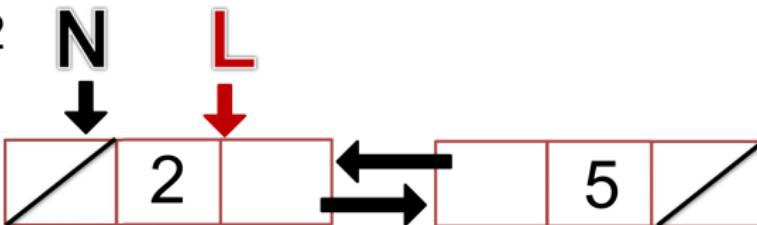
Ex: inserir 2



# Insere Elemento

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

Ex: inserir 2

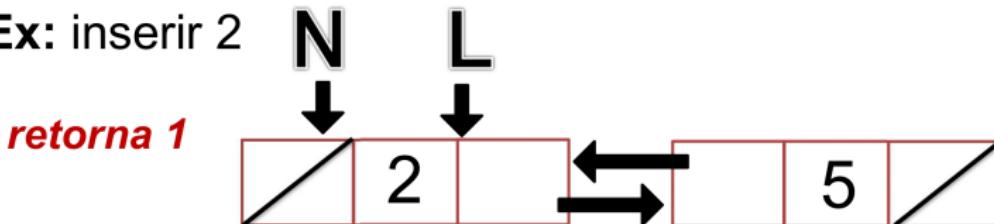




# Insere Elemento

- 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





# Insere 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
    - $1^{\circ} \text{ nó} < \text{Elemento} < \text{último nó}$



# 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:

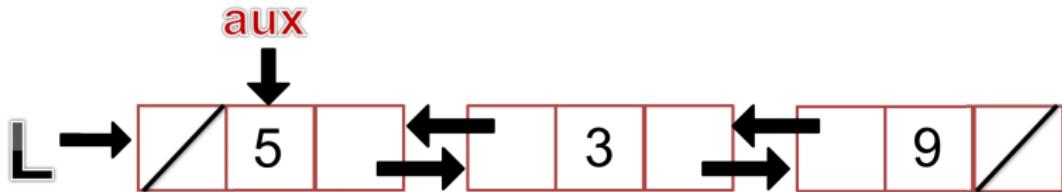
Ex: remover 7



## Remove Elemento

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

Ex: remover 7



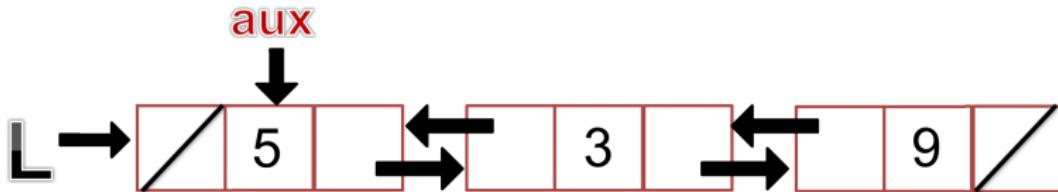
## Remove Elemento

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

**$5 \neq 7$**

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

Ex: remover 7



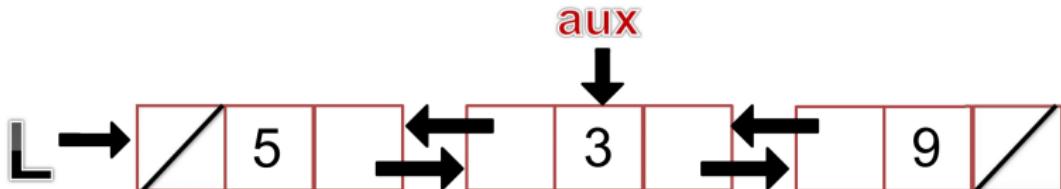


## Remove Elemento

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

**3 ≠ 7**  
*aux->prox ≠ NULL E aux->info ≠ elem?  
Sim (avançar)*

Ex: remover 7

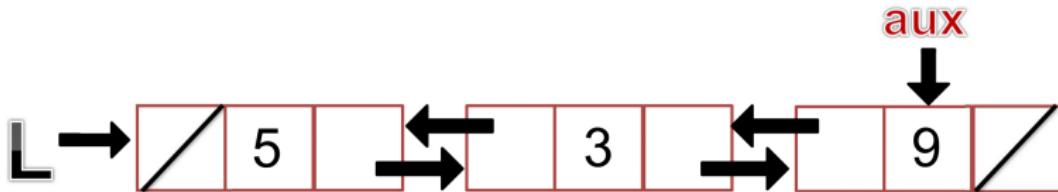


## Remove Elemento

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

Ex: remover 7

= **9 ≠ 7**  
*aux->prox ≠ NULL E aux->info ≠ elem?*  
*Não (parar)*

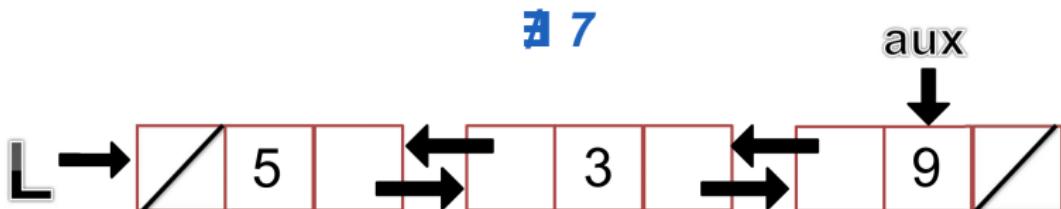




## Remove Elemento

- 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

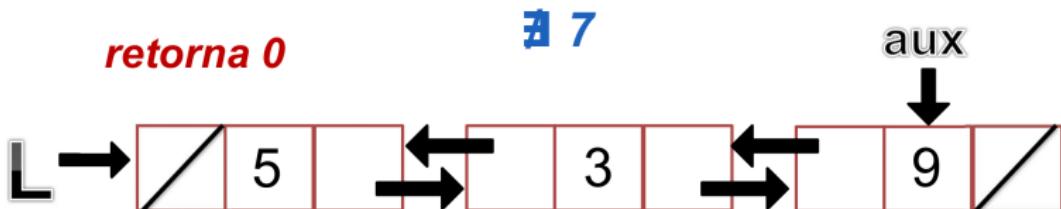




## Remove Elemento

- 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





# Remove Elemento

- Lista com um único nó:

Ex: remover 5

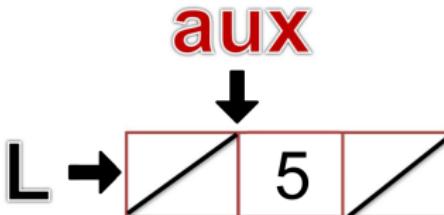




# Remove Elemento

- 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)**)





# 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



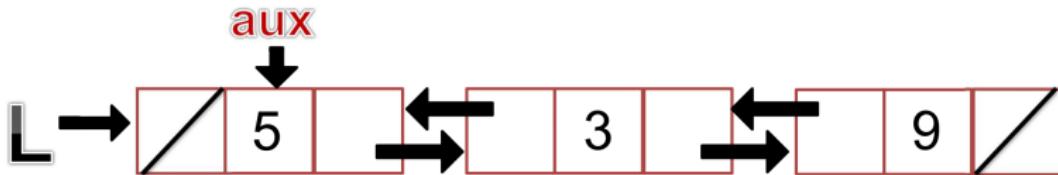
# Remove Elemento

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

**5 = 5**

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

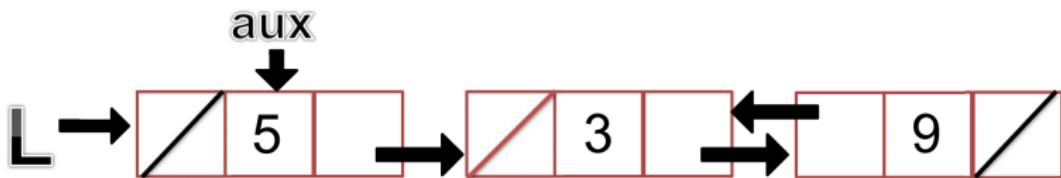
**Ex:** remover 5



# Remove Elemento

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

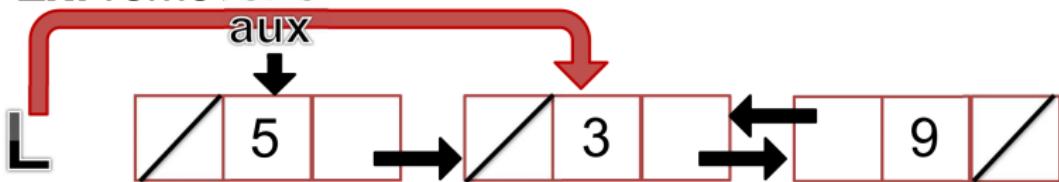
Ex: remover 5



## Remove Elemento

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

Ex: remover 5

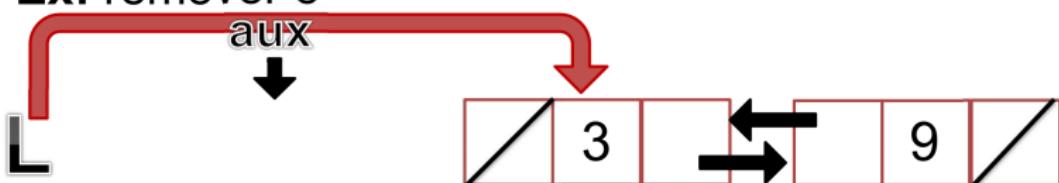




# Remove Elemento

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

Ex: remover 5

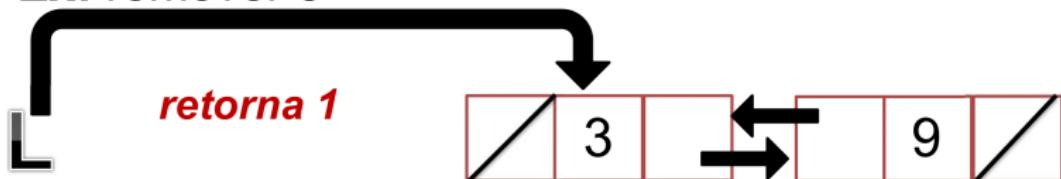




# Remove Elemento

- 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:

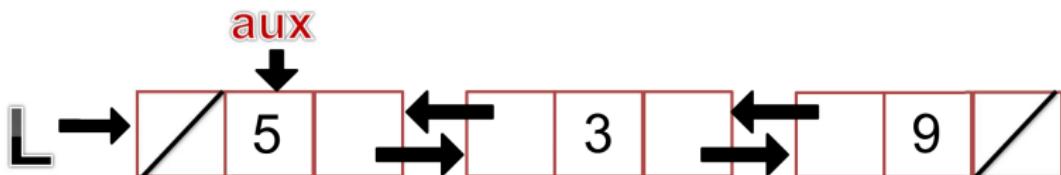
Ex: remover 9



## Remove Elemento

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

Ex: remover 9



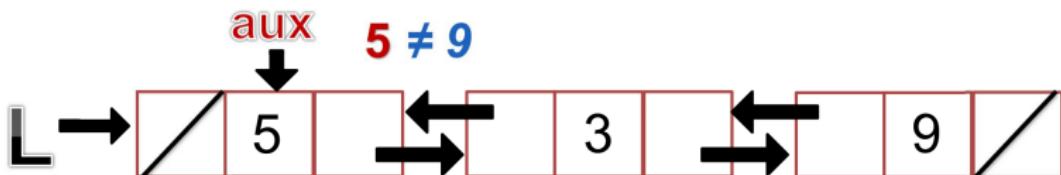


## Remove Elemento

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

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

Ex: remover 9

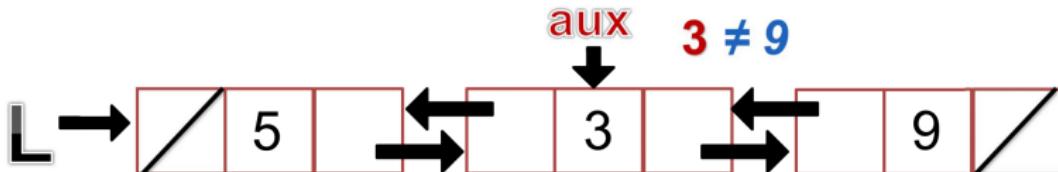


## Remove Elemento

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

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

Ex: remover 9

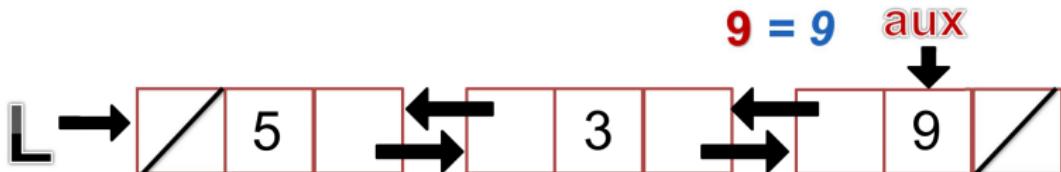


## Remove Elemento

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

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

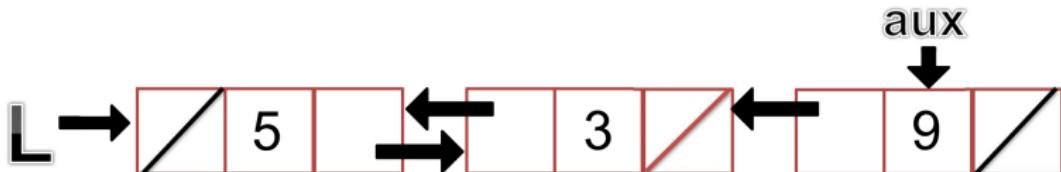
Ex: remover 9



## 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** ( $aux->ant->prox = NULL$ )

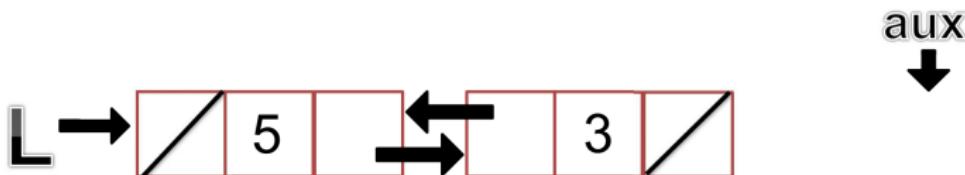
Ex: remover 9



## Remove Elemento

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



## Remove Elemento

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

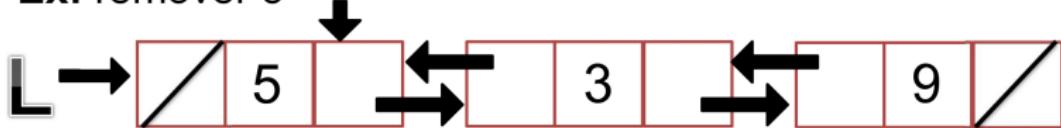
Ex: remover 3



## Remove Elemento

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

Ex: remover 3 **aux**

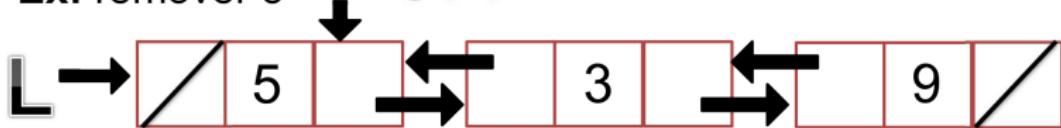


## Remove Elemento

- 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**

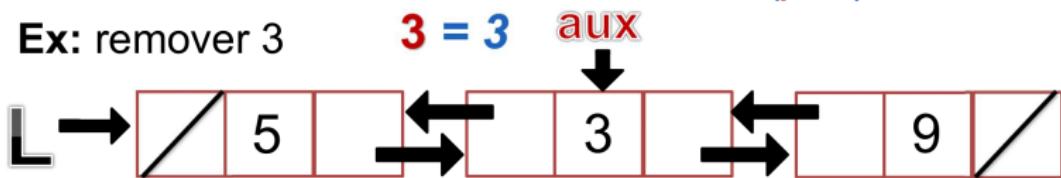


## Remove Elemento

- 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

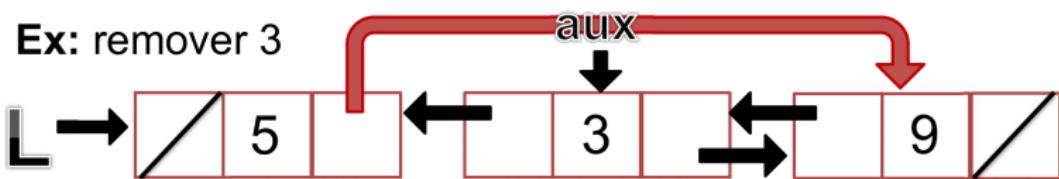




## Remove Elemento

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

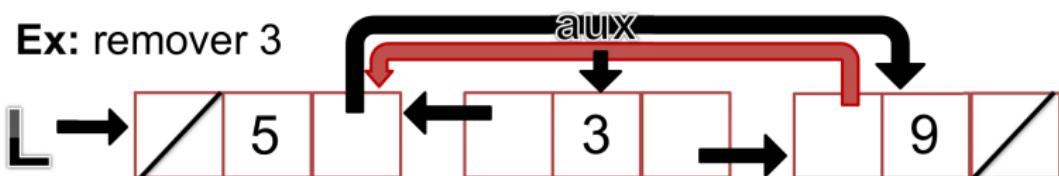
Ex: remover 3



# Remove Elemento

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

Ex: remover 3

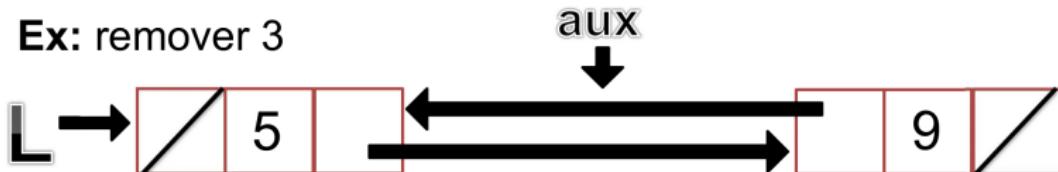




## Remove Elemento

- 1º nó < Elemento < último nó da lista:
  - Faz ponteiro **auxiliar** apontar para o 1º nó
  - Percorre a lista **até encontrar o elemento** ( $\text{aux} \rightarrow \text{info} = \text{elem}$ )
  - Faz o **antecessor do nó** apontar para o **sucessor** ( $\text{aux} \rightarrow \text{ant} \rightarrow \text{prox} = \text{aux} \rightarrow \text{prox}$ )
  - Faz o **sucessor do nó** apontar para o **antecessor** ( $\text{aux} \rightarrow \text{prox} \rightarrow \text{ant} = \text{aux} \rightarrow \text{ant}$ )
  - 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:**
  - **Insere 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 percorrimento da lista
  - **Remove ordenado**
    - Encontrar o elemento envolve percorrimento
    - Inexistência do elemento é determinada por final de lista ou encontrar elemento maior



# Encadeamento Duplo

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

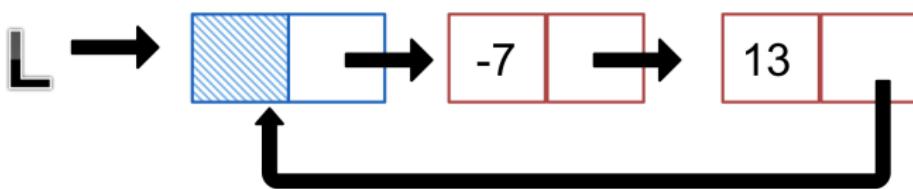


# Combinação das Técnicas de Encadeamento

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

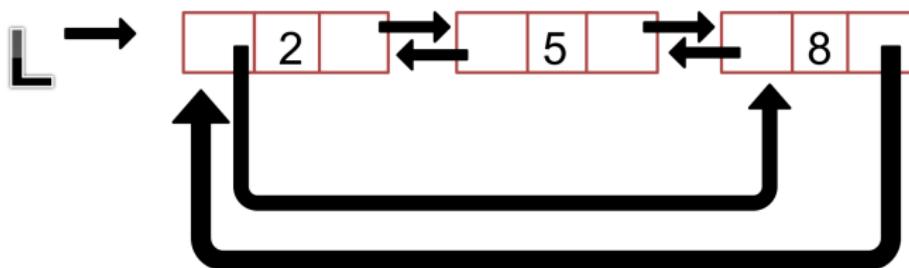
# Exemplos

- **Lista circular com nó cabeçalho:**



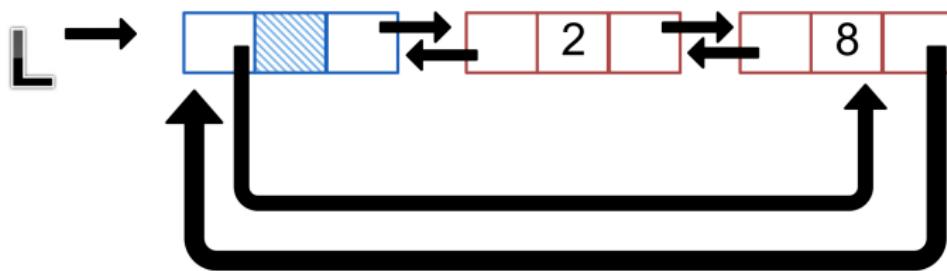
# Exemplos

- **Lista circular duplamente encadeada:**



# Exemplos

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





# Referências

## ✓ Básica

- CELES, W., CERQUEIRA, R. e RANGEL, J. L. “Introdução a estruturas de dados”. Campus Elsevier, 2004.
- TENENBAUM, A. M., LANGSAM, Y. e AUGENSTEIN, M.J. “Estrutura de Dados Usando C”. Makron Books.

## ✓ Extra

- BACKES, André. “Programação Descomplicada Linguagem C”. Projeto de extensão que disponibiliza vídeo-aulas de C e Estruturas de Dados. Disponível em: <https://www.youtube.com/user/progdescomplicada>. Acessado em: 25/04/2022.

## ✓ Baseado nos materiais dos seguintes professores:

- Prof. André Backes (UFU)
- Prof. Bruno Travençolo (UFU)
- Prof. Luiz Gustavo de Almeida Martins (UFU)



# Dúvidas?

**Prof. Me. Claudiney R. Tinoco**  
[profclaudineytinoco@gmail.com](mailto:profclaudineytinoco@gmail.com)

Faculdade de Computação (FACOM)  
Universidade Federal de Uberlândia (UFU)