



Estrutura de Dados 1

Estruturas lineares - Lista Encadeada

Prof. Lucas Boaventura
lucas.boaventura@unb.br





Listas Encadeadas Simples

- **O que é uma lista encadeada?**
 - Estrutura formada por uma sequência de células
 - Cada célula possui:
 - Um dado
 - Um ponteiro para a próxima célula





Listas Encadeadas Simples

- Todos os objetos armazenados são do mesmo tipo
- A ligação entre as células é feita por endereços de memória
- Inicialmente, vamos considerar que o tipo de dado armazenado será **int**





Listas Encadeadas Simples

- Cada célula é representada por uma **struct**
- **Exemplo em C:**

```
1 struct registro {  
2     int conteudo;  
3     struct registro * prox;  
4 };
```





Listas Encadeadas Simples

- **conteudo** = É o dado armazenado
- **prox** = É o endereço da próxima célula

```
1 struct registro {  
2     int conteudo;  
3     struct registro * prox;  
4 };
```





Listas Encadeadas Simples

- É conveniente tratar a célula como um novo tipo
- Uso de typedef para simplificar o código:

```
1 typedef struct registro celula;
```





Listas Encadeadas Simples

- Exemplos de declarações de uma célula:

```
1 celula c;  
2 celula * p;
```

- **c = uma célula**
- **p = um ponteiro para uma célula**





Listas Encadeadas Simples

- Quando temos a **célula diretamente**:
 - c.conteudo
 - c.proximo
- Quando temos um **ponteiro para a célula**:
 - **p->conteudo**
 - **p->prox**





Listas Encadeadas Simples

- A última célula da lista:
 - Possui **prox == NULL**
- NULL indica:
 - Não existe próxima célula
 - Fim da lista encadeada





```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 typedef struct registro {
5     int dado;
6     struct registro * prox;
7 } Celula;
8
9 int main() {
10     Celula p1;
11     Celula p2;
12
13     p1.dado = 5;
14     p1.prox = &p2;
15
16     p2.dado = 10;
17     p2.prox = NULL;
18
19     Celula * p = &p1;
20     while(p) {
21         printf("Dado = %d\n", p->dado);
22         p = p->prox;
23     }
24
25     return 0;
26 }
```





Listas Encadeadas Simples

- Como fica a Organização na Memória?
 - As células não ocupam posições consecutivas na memória
- Elas são alocadas:
 - De forma dinâmica
 - Em locais imprevisíveis da memória
- A ligação é feita exclusivamente pelos ponteiros





Listas Encadeadas Simples

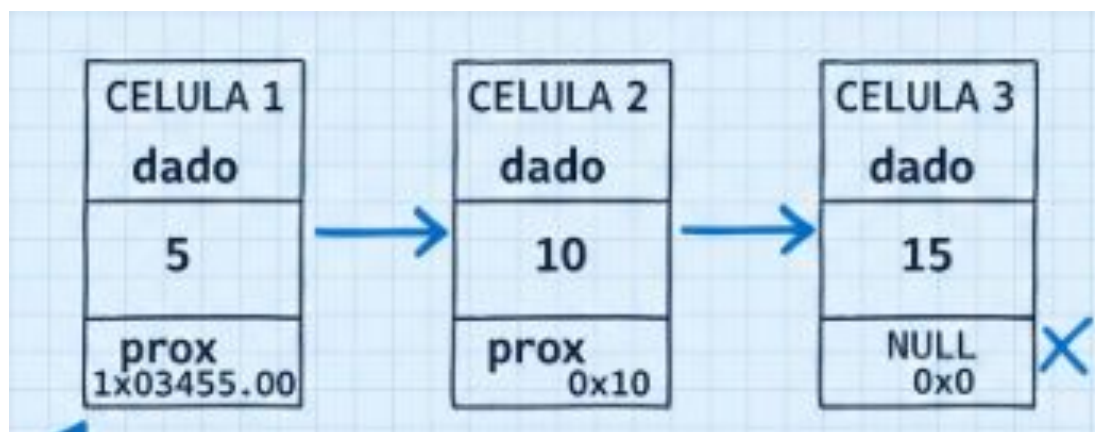
- O endereço de uma lista encadeada é:
 - **O endereço da primeira célula**
- Se '**le**' é esse endereço:
 - Dizemos simplesmente que '**le**' é a lista
- Vamos ver a representação do que é o '**le**'





Listas Encadeadas Simples

- 'le' é um ponteiro para célula
- Ele representa **Toda a lista encadeada**
- 'le' significa **Lista Encadeada**





Listas Encadeadas Simples

- **Uma lista está vazia quando:**
 - **`le == NULL`**
- **Significa:**
 - Não existe nenhuma célula
 - A lista não possui elementos





Listas Encadeadas Simples

- Listas encadeadas são estruturas naturalmente **recursivas**, mas podemos utilizar **iteração** também, sem problemas
- **Observação fundamental:**
 - Se **le** é uma lista não vazia
 - Então **le->prox** também é uma lista





Listas Encadeadas Simples

- **Exemplo: Impressão Recursiva**
 - Imprime o primeiro elemento
 - Chama a função para o restante da lista

```
1 void imprime (Celula * le) {  
2     if (le != NULL) {  
3         printf("%d\n", le->dado);  
4         imprime(le->prox);  
5     }  
6 }
```



Listas Encadeadas Simples

- **Exemplo: Impressão Iterativa**
 - Uso explícito de um ponteiro auxiliar
 - Mesma funcionalidade, mas usando laço

```
1 void imprime (Celula *le) {  
2     for (Celula * p = le; p != NULL; p = p->prox)  
3         printf("%d\n", p->dado);  
4 }
```





Listas Encadeadas Simples

- **Recursão vs Iteração**
- **Recursiva:**
 - Código mais próximo da definição da lista
 - Mais elegante conceitualmente
- **Iterativa:**
 - Evita chamadas recursivas
 - Geralmente mais eficiente em memória





Listas Encadeadas Simples

- **Busca em Lista Encadeada**
 - Objetivo é verificar se um valor x pertence à lista
- **Ideia:**
 - Percorrer a lista célula por célula
 - Comparar x com o conteúdo de cada célula





Listas Encadeadas Simples

- **O que a Função de Busca Retorna?**
 - A função devolve o endereço da célula que contém x
- Caso x não exista **retorna NULL**
- A vantagem é não ser necessário usar variáveis booleanas





Listas Encadeadas Simples

- Implementação da busca Iterativa:

```
1 Celula * busca (int x, Celula *le) {  
2     Celula *p = le;  
3  
4     while (p != NULL && p->dado != x)  
5         p = p->prox;  
6  
7     return p;  
8 }
```



Listas Encadeadas Simples

- **Condição do while:**

- $p \neq \text{NULL} \rightarrow$ ainda há células
- $p \rightarrow \text{conteudo} \neq x \rightarrow$ valor ainda não encontrado

- **O laço termina quando:**

- x é encontrado **OU**
- O fim da lista é atingido

```
1 Celula * busca (int x, Celula *le) {  
2     Celula *p = le;  
3  
4     while (p != NULL && p->dado != x)  
5         p = p->prox;  
6  
7     return p;  
8 }
```





Listas Encadeadas Simples

- **Se a lista estiver vazia:**
 - **le == NULL**
 - O laço não executa
 - A função retorna NULL
 - Comportamento correto e seguro

```
1 Celula * busca (int x, Celula *le) {  
2     Celula *p = le;  
3  
4     while (p != NULL && p->dado != x)  
5         p = p->prox;  
6  
7     return p;  
8 }
```





Listas Encadeadas Simples

- **Busca Recursiva:**

```
1 Celula * busca_rec(int x, Celula *le) {  
2     if (le == NULL) return NULL;  
3     if (le->dado == x) return le;  
4     return busca_rec(x, le->prox);  
5 }
```

- **Casos base:**

- Lista vazia → retorna NULL
- Valor encontrado → retorna a célula

- **Passo recursivo:** Busca no restante da lista





Listas Encadeadas Simples

- **Em algumas situações a primeira célula não armazena dados úteis**
- **Essa célula funciona apenas como um marcador de início da lista**
- **Essa célula especial é chamada de cabeça da lista (head cell ou dummy cell)**





Listas Encadeadas Simples

- **Por que Usar Cabeça de Lista?**
 - Simplifica algoritmos de Inserção e Remoção
 - Evita casos especiais para Lista vazia e Inserção no início da lista
 - O ponteiro da lista nunca é NULL
- Uma lista com cabeça está vazia quando:
 - **le->prox == NULL**
 - Diferente da lista sem cabeça, aqui 'le' sempre aponta para uma célula válida





Listas Encadeadas Simples

- **Criação de uma lista encadeada vazia com cabeça:**
 - O campo conteudo/dado da cabeça é ignorado

```
1 Celula * cria_le() {  
2     Celula * le;  
3     le = malloc(sizeof(Celula));  
4     le->prox = NULL;  
5  
6     return le;  
7 }
```





Listas Encadeadas Simples

- Impressão ignora a célula cabeça:

```
1 void imprime (Celula *le) {  
2     Celula *p;  
3     for (p = le->prox; p != NULL; p = p->prox)  
4         printf( "%d\n", p->dado);  
5 }
```





Listas Encadeadas Simples

- **Diferença: Lista com e sem Cabeça**
- **Sem cabeça:**
 - Lista vazia: **`le == NULL`**
 - Mais casos especiais
- **Com cabeça:**
 - Lista vazia: **`le->prox == NULL`**
 - Algoritmos mais simples e uniformes





Listas Encadeadas Simples

- **Inserção em uma lista encadeada**
 - Problema: inserir uma nova célula em uma lista encadeada
- **Situação considerada:**
 - Inserção entre a célula apontada por um ponteiro **p**
 - E a célula seguinte (**p->prox**)
- A operação só faz sentido se $p \neq \text{NULL}$





Listas Encadeadas Simples

- **Ideia Básica de Inserção:**
 - Cria uma nova célula dinamicamente
 - Ajustar os ponteiros para encaixar a nova célula
 - Não é necessário deslocar elementos, como seria com os vetores
 - Apenas alguns vetores são ajustados, não são todos





Listas Encadeadas Simples

- **Protótipo da Função:**

```
1 void insere (int x, celula *p);
```

- **x:** valor a ser inserido na nova célula;
- **p:** ponteiro para a célula anterior ao ponto de inserção
- **Suposição importante:**
 - **p != NULL**

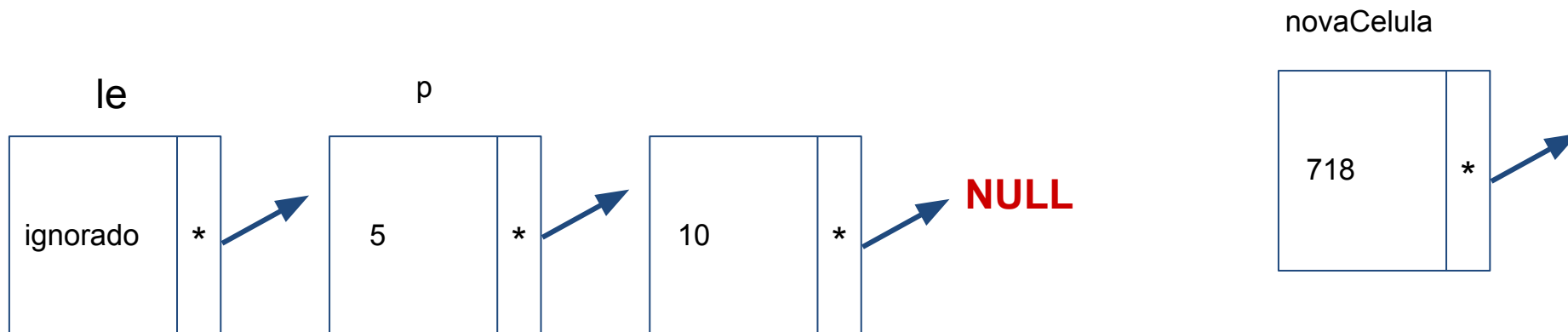




Listas Encadeadas Simples

- **Passo a Passo da Inserção:**

1. **Aloca a memória para a nova célula (malloc)**
2. **Armazena o valor de x no campo 'dado'**
3. **Fazer a nova célula apontar para $p \rightarrow \text{prox}$**
4. **Atualizar $p \rightarrow \text{prox}$ para apontar para a nova célula**

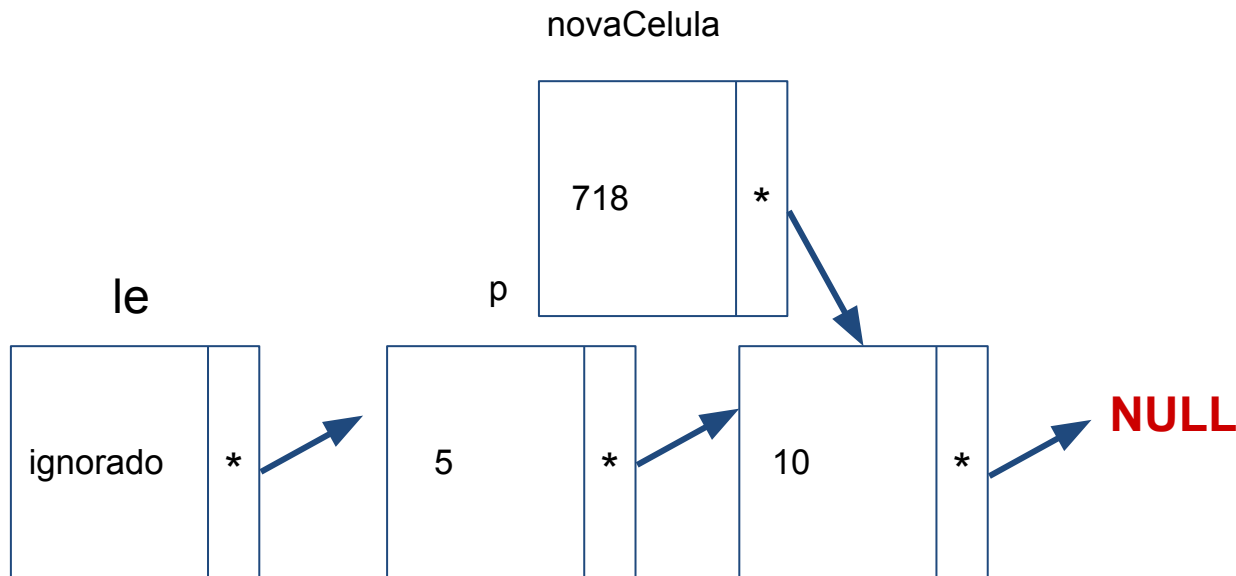




Listas Encadeadas Simples

- **Passo a Passo da Inserção:**

1. Aloca a memória para a nova célula (malloc)
2. Armazena o valor de **x** no campo 'dado'
3. **Fazer a nova célula apontar para p->prox**
4. Atualizar **p->prox** para apontar para a nova célula

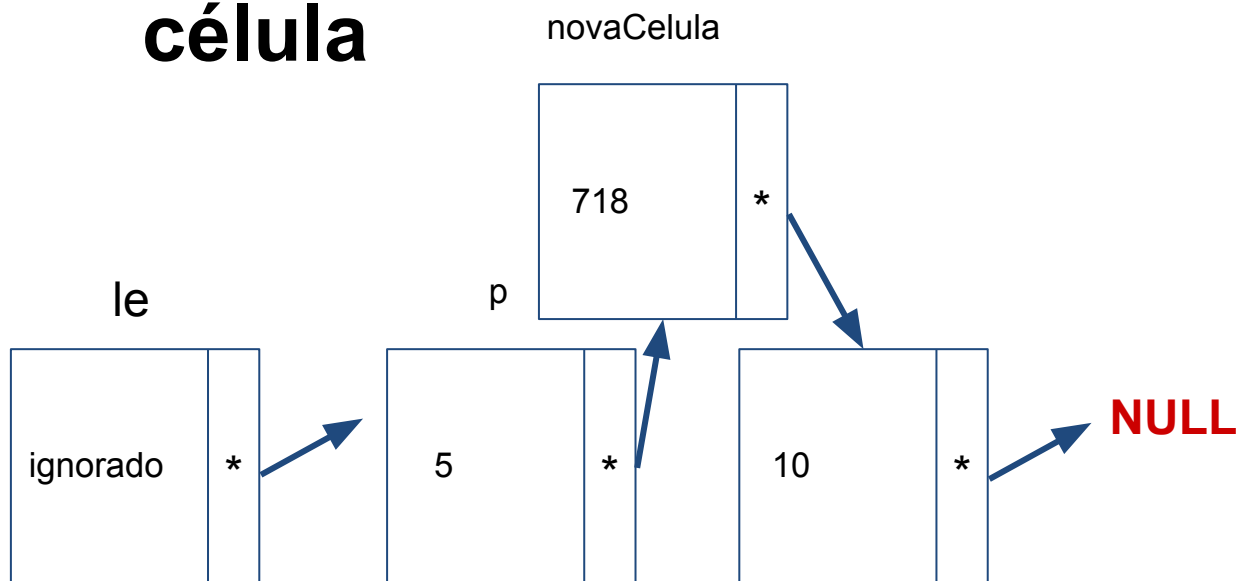




Listas Encadeadas Simples

- **Passo a Passo da Inserção:**

1. Aloca a memória para a nova célula (malloc)
2. Armazena o valor de **x** no campo '**dado**'
3. Fazer a nova célula apontar para $p \rightarrow \text{prox}$
4. **Atualizar $p \rightarrow \text{prox}$ para apontar para a nova célula**

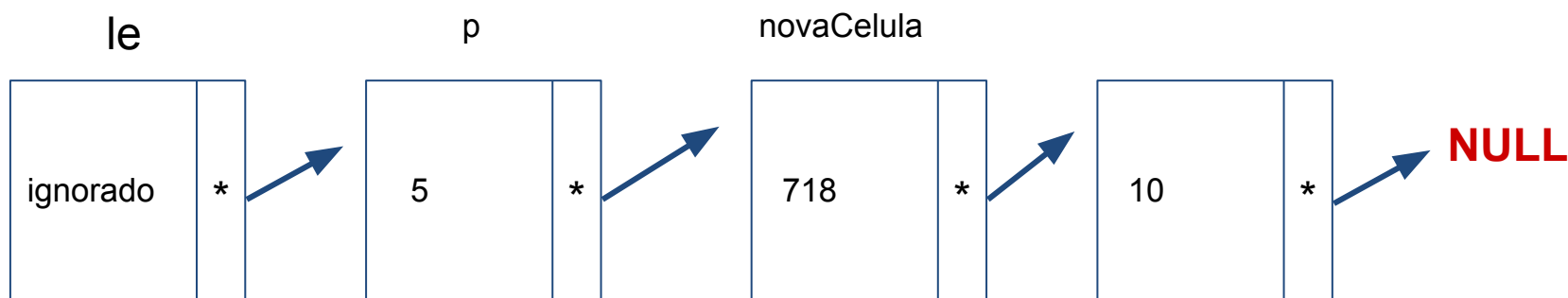




Listas Encadeadas Simples

- **Passo a Passo da Inserção:**

1. Aloca a memória para a nova célula (malloc)
2. Armazena o valor de **x** no campo '**dado**'
3. Fazer a nova célula apontar para $p \rightarrow \text{prox}$
4. **Atualizar $p \rightarrow \text{prox}$ para apontar para a nova célula**





Listas Encadeadas Simples

- Implementação da Função de Inserção:

```
1 void insere(int x, Celula *p) {  
2     Celula * nova;  
3     nova = malloc (sizeof (Celula));  
4     nova->dado = x;  
5     nova->prox = p->prox;  
6     p->prox = nova;  
7 }
```





Listas Encadeadas Simples

- **Inserção no Fim da Lista:**
 - A função funciona corretamente mesmo quando **p->prox == NULL**
- **Nesse caso a nova célula será inserida no final da lista**
- **Não é necessário tratamento especial**





Listas Encadeadas Simples

- **Se a lista tem cabeça:**
 - A função pode ser usada para inserir no início
 - Basta enviar para a função o ponteiro para a célula-cabeça
- **Em listas sem cabeça:**
 - A função não consegue inserir antes da primeira célula
 - Não existe uma célula anterior para apontar





Listas Encadeadas Simples

- **Complexidade da Inserção:**
 - O tempo de execução Não depende da posição onde ocorre a inserção
- **Inserir no início ou no fim custa o mesmo**
- **Complexidade:**
 - $O(1)$ — tempo constante





Listas Encadeadas Simples

- **Comparação com Vetores:**
- **Lista Encadeada:**
 - Inserção rápida
 - Apenas ajuste de ponteiros
- **Vetores:**
 - É necessário deslocar elementos
 - Custo maior, especialmente no início





Listas Encadeadas Simples

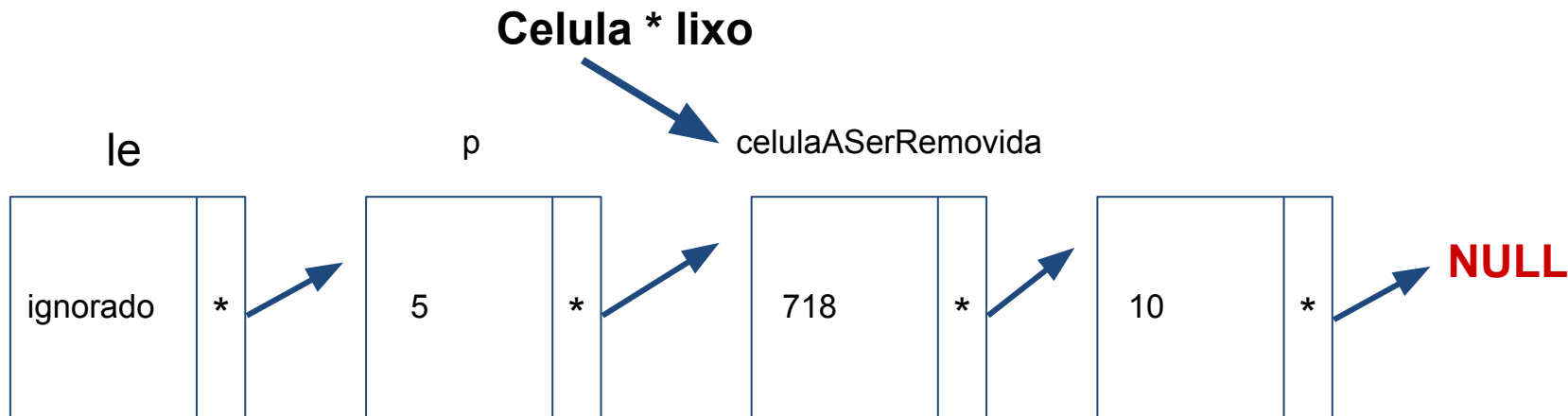
- **Remoção em Lista Encadeada:**
 - A função recebe um **ponteiro 'p'** para uma célula da lista
 - A função remove a célula seguinte (**p->prox**)
- **Considera-se que:**
 - $p \neq \text{NULL}$
 - $p \rightarrow \text{prox} \neq \text{NULL}$





Listas Encadeadas Simples

- **Passo a Passo da Remoção:**
 1. Guardar o endereço da célula a ser removida (lixo = p->prox)
 2. Ajustar o ponteiro:
 - a. p->prox passa a apontar para lixo->prox
 3. Liberar a memória da célula removida (free(lixo))





Listas Encadeadas Simples

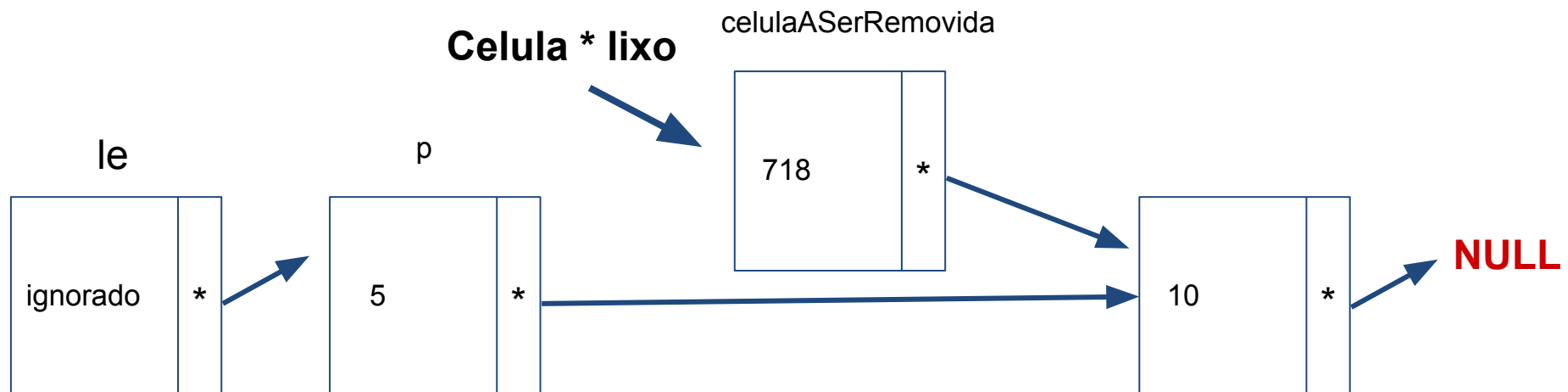
- **Passo a Passo da Remoção:**

1. Guardar o endereço da célula a ser removida
(lixo = p->prox)

2. **Ajustar o ponteiro:**

- a. **p->prox passa a apontar para lixo->prox**

3. Liberar a memória da célula removida (free(lixo))





Listas Encadeadas Simples

- **Passo a Passo da Remoção:**

1. Guardar o endereço da célula a ser removida
(lixo = p->prox)
2. Ajustar o ponteiro:
 - a. p->prox passa a apontar para lixo->prox
3. **Liberar a memória da célula removida 'free(lixo)'**

Celula * lixo

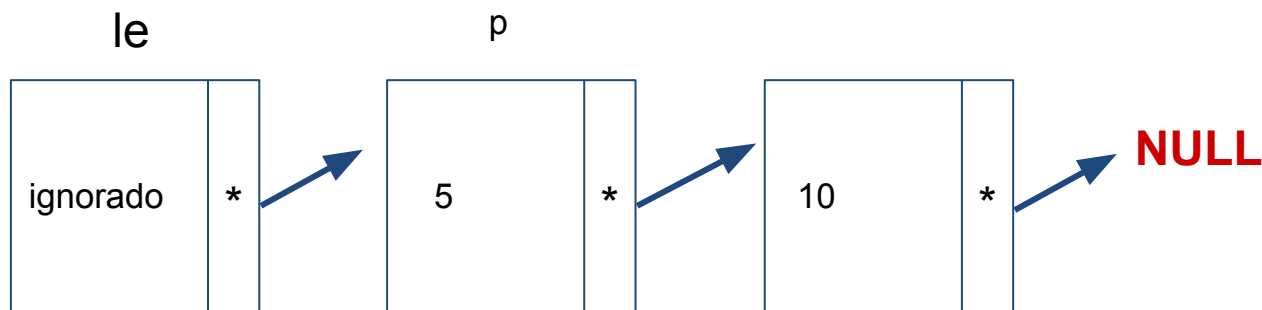




Listas Encadeadas Simples

- **Passo a Passo da Remoção:**

1. Guardar o endereço da célula a ser removida
(lixo = p->prox)
2. Ajustar o ponteiro:
 - a. p->prox passa a apontar para lixo->prox
3. Liberar a memória da célula removida 'free(lixo)'





Listas Encadeadas Simples

- Implementação da função de Remoção:

```
1 void remover(Celula *p) {  
2     Celula *lixo;  
3     lixo = p->prox;  
4     p->prox = lixo->prox;  
5     free (lixo);  
6 }
```





Listas Encadeadas Simples

- **Vantagens da Remoção em Lista Encadeada:**
 - Não é necessário copiar dados e nem deslocar elementos
 - Basta atualizar um ponteiro
 - Código simples e eficiente





Listas Encadeadas Simples

- **Complexidade da Operação de Remoção:**
 - O tempo de execução independe da posição da célula removida
- Remover no início ou no fim custa o mesmo tempo
- **Complexidade:**
 - **$O(1)$** — tempo constante





Listas Encadeadas Simples

- Busca e Remove

```
1 void busca_e_remove (int y, Celula *le) {  
2     Celula *p, *q;  
3     p = le;  
4     q = le->prox;  
5  
6     while (q != NULL && q->dado != y) {  
7         p = q;  
8         q = q->prox;  
9     }  
10  
11     if (q != NULL) {  
12         p->prox = q->prox;  
13         free (q);  
14     }  
15 }
```



Além das Listas Encadeadas Básicas

- **Outros tipos de listas encadeadas:**
 - Após entender listas encadeadas simples
 - É possível criar diversas variações
 - Cada variação resolve problemas diferentes
- **Exemplos:**
 - **Lista Encadeada Circular**
 - **Lista Duplamente Encadeada**





Lista Encadeada Circular

- **Característica principal:**
 - A última célula aponta para a primeira
- **Consequência:**
 - Não existe NULL no encadeamento
- **Endereço da lista:**
 - Pode ser o endereço de qualquer célula





Lista Duplamente Encadeada

- **Cada célula contém:**
 - Ponteiro para a célula anterior
 - Ponteiro para a célula seguinte
- **Vantagem:**
 - Percurso nos dois sentidos
- **Custo:**
 - Maior uso de memória
 - Maior complexidade de manipulação





Outras Discussões

- **Célula-Cabeça e Célula-Rabo:**
 - Convém usar:
 - Uma célula-cabeça (dummy)?
 - Uma célula-rabo?
- **Pode auxiliar em operações de inserção ao final da lista e outras operações.**





Exercícios

1. Por que a seguinte versão da função insere não funciona?

```
1 void insere (int x, celula *p) {  
2     celula nova;  
3     nova.conteudo = x;  
4     nova.prox = p->prox;  
5     p->prox = &nova;  
6 }
```





Exercícios

2. Escreva uma função que copie o conteúdo de um vetor para uma lista encadeada preservando a ordem dos elementos. Faça duas versões: uma iterativa e uma recursiva.

3. Escreva uma função que copie o conteúdo de uma lista encadeada para um vetor preservando a ordem dos elementos. Faça duas versões: uma iterativa e uma recursiva.





Exercícios

- 4. Crie uma lista simplesmente encadeada que armazena dados de um aluno, como: nome, matricula, curso. Inclua 2 células na lista, imprima elas e remova tudo depois. Utilize funções para isso.**

- 5. Aproveitando a ideia de lista encadeada do exercício 4, faça uma busca pelo nome do aluno e imprima o aluno encontrado.**





Dúvidas?

- lucas.boaventura@unb.br

