

# Listas Simplesmente Encadeadas

## Algoritmos e Estrutura de Dados

### Alocação Dinâmica

prof. Frederico Santos de Oliveira

Universidade Federal de Mato Grosso  
Instituto de Engenharia

# Roteiro da Aula

1 Introdução

2 Exercício 1

# Introdução

## Estrutura Nodo

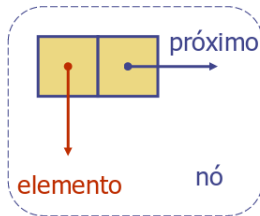
- Assim como fizemos com a Pilha e a Fila, precisamos definir a estrutura de dados **nodo**.
- O nodo é uma entidade elementar das estruturas de dados **Pilha**, **Fila** e **Lista**.

# Introdução

## Estrutura Nodo

Cada nodo armazena:

- Um elemento.
  - ▶ pode ser um inteiro, uma string, um vetor, um registro, não importa.
  - ▶ Aqui é denominado apenas **item**.
- Uma ligação para o próximo nodo.
  - ▶ Em termos de implementação, trata-se de um ponteiro do tipo nodo, o qual denominaremos **prox**.



# Introdução

## Estrutura Nodo

Primeiramente, vamos implementar o tipo de dados **nodo**. Vamos analisar o pseudo-código apresentado nas aulas anteriores.

# Introdução

## Estrutura Nodo

---

### Algoritmo 1: Nodo

---

```
1 início
2   registro {
3     Inteiro: item;
4     Ponteiro Nodo: prox;
5   } Nodo;
```

---

# Introdução

## Estrutura Nodo

Na linguagem C, a implementação fica conforme o código a seguir:

```
1 typedef struct nodo_t {  
2     int item;  
3     struct nodo_t *prox;  
4 } nodo;
```

Vamos chamar o arquivo que contém esse código de `nodo.h`.

# Roteiro da Aula

1 Introdução

2 Exercício 1



# Exercício 1

## Lista Simples

Neste exercício, vamos implementar a estrutura de dados **Lista Simplesmente Encadeada**:

- Nessa estrutura existem dois ponteiros:
  - ▶ um aponta para o **primeiro** nodo da Lista.
  - ▶ o outro que aponta para o **último** nodo da Lista.
- Diferentemente da Pilha e da Fila, não existe o nodo **cabeça**.
- Quando a Lista está vazia, os ponteiros primeiro e último apontam para NULL.

# Exercício 1

## Lista Simples

---

### Algoritmo 2: Lista

---

```
1 início
2   registro {
3     | Ponteiro Nodo: primeiro, último;
4   } Lista;
```

---

# Exercício 1

## Lista Simples

Na linguagem C, a implementação fica conforme o código a seguir:

```
1 #include "nodo.h"
2 typedef struct {
3     nodo *primeiro;
4     nodo *ultimo;
5 } lista;
```

Vamos chamar o arquivo que contém esse código de `lista.h`.

# Exercício 1

## Lista Simples

A seguir, vamos implementar a função `CriarListaVazia( $L$ )`.

## Exercício 2

### Lista Simples

Vamos adicionar o protótipo dessa função no arquivo `lista.h`.

```
1 void criarListaVazia(lista *l);
```

Agora, vamos implementar essa função.

# Exercício 1

## Lista Simples

- Primeiramente, vamos criar um arquivo chamado `lista.c` que deve conter a implementação dessas funções.
- Para evitar que o compilador não encontre determinada função, vamos incluir o protótipo das funções, inserindo a linha de código a seguir no início do arquivo `lista.c`:

```
1 #include "lista.h"
```

- A seguir, vamos implementar a função `CriarListaVazia(L)`, que recebe uma lista `L` e aponta devidamente os ponteiros `primeiro` e `ultimo`.

# Exercício 1

## Lista Simples

---

### Algoritmo 3: CriarListaVazia

---

**Entrada:** Lista L.

```
1 início
2   L.primeiro  $\leftarrow$  NULL
3   L.último  $\leftarrow$  NULL
```

---

# Exercício 1

## Lista Simples

Na linguagem C, a implementação fica conforme o código a seguir:

```
1 void criarListaVazia(lista *l) {  
2     l->primeiro = NULL ;  
3     l->ultimo = NULL ;  
4 }
```



# Exercício 1

## Lista Simples

A seguir, vamos implementar a função  $\text{ListaVazia}(L)$ , que verifica se uma lista está vazia.

# Exercício 1

## Lista Simples

---

### Algoritmo 4: ListaVazia

---

**Entrada:** Lista L.

**Saída:** Booleano (V ou F) indicando se L está vazia.

```
1 início
2   se (L.primeiro = NULL) então
3     retorna Verdadeiro
4   senão
5     retorna Falso
```

---

# Exercício 1

## Lista Simples

Na linguagem C, a implementação fica conforme o código a seguir:

```
1 int listaVazia(lista * l ) {  
2     return l->primeiro == NULL ;  
3 }
```

# Exercício 1

## Lista Simples

A seguir, vamos implementar a função  $\text{InserirInicio}(L, x)$ , que insere o elemento  $x$  no início de uma lista.

# Exercício 1

## Lista Simples

---

### Algoritmo 5: InserirInício

---

**Entrada:** Lista L, item x.

```
1 início
2   novo ← ALOCA_NODO()
3   novo.item ← x
4   novo.prox ← L.primeiro
5   L.primeiro ← novo
6   // Verifica se a lista está vazia.
7   se (L.último = NULL) então
8     L.último ← L.primeiro
```

---

# Exercício 1

## Lista Simples

Na linguagem C, a implementação fica conforme o código a seguir:

```
1 void inserirInicio(lista *l, int x ) {  
2     nodo *novo = malloc(sizeof(nodo));  
3     novo->item = x;  
4     novo->prox = l->primeiro;  
5     if (l->primeiro == NULL)  
6         l->ultimo = novo ;  
7     l->primeiro = novo ;  
8 }
```

# Exercício 1

## Lista Simples

A seguir, vamos implementar a função `InserirFinal( $L, x$ )`, que insere o elemento  $x$  no final de uma lista.

# Exercício 1

## Lista Simples

---

### Algoritmo 6: InserirFinal

---

**Entrada:** Lista L, item x.

```
1 início
2   novo ← ALOCA_NODO()
3   novo.item ← x
4   novo.prox ← NULL
5   // Verifica se a lista está vazia.
6   se (L.primeiro = NULL) então
7       L.primeiro ← novo
8       L.último ← L.primeiro
9   senão
10      L.último.prox ← novo
11      L.último ← novo
```



# Exercício 1

## Lista Simples

Na linguagem C, a implementação fica conforme o código a seguir:

```
1 void inserirFinal (lista *l, int x ) {  
2     nodo *novo = malloc(sizeof(nodo)) ;  
3     novo->item = x ;  
4     novo->prox = NULL ;  
5     if (l->primeiro == NULL )  
6         l->primeiro = novo ;  
7     else  
8         l->ultimo->prox = novo ;  
9     l->ultimo = novo ;  
10 }
```

# Exercício 1

## Lista Simples

A seguir, vamos implementar a função  $\text{BuscarPosicao}(L, p)$ , que retorna um ponteiro para o nodo na posição  $p$  de uma lista.

# Exercício 1

## Lista Simples

---

### Algoritmo 7: BuscarPosicao

---

**Entrada:** Lista  $L$ , item  $p$ .

**Saída:** Nodo na posição  $p$  ou NULL caso não encontrado.

```
1 início
2   aux ← L.primeiro
3   c ← 1
4   enquanto ( $aux \neq NULL$ ) AND ( $c < p$ ) faça
5       aux ← aux.prox
6       c ← c + 1
7   retorna aux
```

---

# Exercício 1

## Lista Simples

Na linguagem C, a implementação fica conforme o código a seguir:

```
1 nodo *buscarPosicao(lista *l, int p ) {  
2     nodo *aux = l->primeiro;  
3     int c = 1;  
4     while ( aux != NULL && c < p ) {  
5         aux = aux->prox;  
6         c ++;  
7     }  
8     return aux;  
9 }
```

# Exercício 1

## Lista Simples

A seguir, vamos implementar a função `inserirPosicao( $L$ ,  $x$ ,  $p$ )`, que insere um elemento  $x$  na posição  $p$  de uma lista.

# Exercício 1

## Lista Simples

---

### Algoritmo 8: InserirPosição

---

**Entrada:** Lista  $L$ , item  $x$ , posição  $p$ .

```
1 início
2   // Verifica se o nodo deve ser inserido no início.
3   se ( $p = 1$ ) então
4     ListInsereInício( $L, x$ )
5   senão
6     novo  $\leftarrow$  ALOCA_NODO()
7     novo.item  $\leftarrow x$ 
8     // Busca o nodo na posição anterior a  $p$ .
9     anterior  $\leftarrow$  BuscarPosição( $L, p - 1$ )
10    // Insere o novo nodo entre os nodos
11    // anterior e posterior.
12    posterior  $\leftarrow$  anterior.prox
13    anterior.prox  $\leftarrow$  novo
14    novo.prox  $\leftarrow$  posterior
15    // Verifica se o nodo na posição  $p - 1$  é o último,
16    // ou seja, seu posterior é NULL.
17    se ( $posterior = \text{NULL}$ ) então
18      L.último  $\leftarrow$  novo
```

---

# Exercício 1

## Lista Simples

Na linguagem C, a implementação fica conforme o código a seguir:

```
10 void inserirPosicao (lista *l, int x, int p ) {  
11     if (p ==1)  
12         inserirInicio(l, x);  
13     else {  
14         nodo *anterior = l->primeiro, *posterior;  
15         nodo *novo = malloc(sizeof(nodo));  
16         novo->item = x ;  
17         anterior = buscarPosicao(l ,p -1);  
18         posterior = anterior->prox;  
19         anterior->prox = novo;  
20         novo-> prox = posterior;  
21         if (posterior == NULL )  
22             l->ultimo = novo;  
23     }  
24 }
```

# Exercício 1

## Lista Simples

A seguir, vamos implementar a função `RemoverInicio( $L$ )`, que remove um elemento que se encontra no início de uma lista.



# Exercício 1

## Lista Simples

---

### Algoritmo 9: RemoverInício

---

**Entrada:** Lista L, item x.

```
1 início
2   // Verifica se a lista está vazia.
3   se (ListaVazia(L)) então
4     | Imprima "Erro underflow: lista vazia"
5   senão
6     aux ← L.primeiro
7     x ← aux.item
8     L.primeiro ← L.primeiro.prox
9     // Verifica se a lista possui apenas um nodo.
10    se (L.primeiro = NULL) então
11      | // Primeiro e Último apontam para NULL.
12      | L.último ← NULL
13    aux.prox ← NULL
14    DESALOCA_NODO(aux)
15    retorna x
```

# Exercício 1

## Lista Simples

```
1 int removerInicio(lista * l) {
2     nodo * aux;
3     int x = -1;
4     if (listaVazia(l) )
5         printf("Lista Vazia\n");
6     else {
7         aux = l->primeiro;
8         x = aux->item;
9         l->primeiro = l->primeiro->prox;
10        if (l->primeiro == NULL)
11            l->ultimo = NULL;
12        aux->prox = NULL;
13        free(aux);
14    }
15    return x;
16 }
```

# Exercício 1

## Lista Simples

A seguir, vamos implementar a função `RemoveFinal( $L$ )`, que remove um elemento que se encontra no final de uma lista.

# Exercício 1

## Lista Simples

---

**Algoritmo 10:** RemoverFinal

---

**Entrada:** Lista L

**Saída:** Item removido

```
1 início
2   // Verifica se a lista está vazia.
3   se (ListaVazia(L)) então
4     | Imprima "Erro underflow: lista vazia"
5   senão
6     // Verifica se a lista possui um único elemento.
7     se (L.primeiro = L.último) AND (NOT(ListaVazia(L))) então
8       | aux ← L.primeiro
9       | L.primeiro ← NULL
10      | L.último ← NULL
11    // Caso contrário, a lista possui
12    // pelo menos dois elementos.
13    senão
14      // Busca o penúltimo elemento.
15      anterior ← L.primeiro
16      enquanto (anterior.prox ≠ L.último) faça
17        | anterior ← anterior.prox
18      aux ← L.último
19      anterior.prox ← NULL
20      L.último ← anterior
21    x ← aux.item
22    aux.prox ← NULL
23    DESALOCA_NODO(aux)
24    retorna x
```

---

# Exercício 1

## Lista Simples

```
25 int removerFinal(lista *l) {
26     int x = -1;
27     nodo *aux, *anterior;
28     if (listaVazia(l))
29         printf("Lista Vazia\n");
30     else if (l->primeiro == l->ultimo) {
31         aux = l->primeiro;
32         l->primeiro = NULL;
33         l->ultimo = NULL;
34     }
35     else {
36         anterior = l->primeiro;
37         while (anterior->prox != l->ultimo)
38             anterior = anterior->prox;
39         aux = l->ultimo;
40         anterior->prox = NULL;
41         l->ultimo = anterior;
42     }
43     x = aux->item;
44     aux->prox = NULL;
45     free(aux);
46     return x;
47 }
```

# Exercício 1

## Lista Simples

A seguir, vamos implementar a função `removePosicao( $L, p$ )`, que remove um elemento que se encontra na posição  $p$  de uma lista.

# Exercício 1

## Lista Simples

**Algoritmo 11:** RemoverPosição

**Entrada:** Lista  $L$ , posição  $p$

**Saída:** Item removido

1 **início**

2   // Verifica se a lista está vazia.

3   **se** ( $ListaVazia(L)$ ) **então**

4     | Imprima "Erro underflow: lista vazia"

5   **senão**

6     // Verifica se o nodo a ser removido é o primeiro.

7     **se** ( $p = 1$ ) **então**

8       | **retorna** RemoverInicio( $L$ )

9     **senão**

10       // Busca o nodo na posição anterior a  $p$ .

11       anterior  $\leftarrow$  ListaBuscarPosição( $L, p - 1$ )

12       // Remove o nodo na posição  $p$ , apontado por aux.

13       aux  $\leftarrow$  anterior.prox

14       posterior  $\leftarrow$  aux.prox

15       anterior.prox  $\leftarrow$  posterior

16       // Verifica se o nodo a ser removido é o último.

17       **se** ( $aux = L.último$ ) **então**

18         |  $L.último \leftarrow$  anterior

19        $x \leftarrow$  aux.item

20       aux.prox  $\leftarrow$  NULL

21       DESALOCA\_NODO(aux)

22       **retorna**  $x$

# Exercício 1

## Lista Simples

```
48 int removerPosicao(lista *l, int p) {
49     int x = -1;
50     nodo *anterior, *posterior, *aux;
51     if (p==1)
52         x = removerInicio(l);
53     else {
54         anterior = buscarPosicao(l, p-1);
55         aux = anterior->prox;
56         posterior = aux->prox;
57         anterior->prox = posterior;
58         if (aux == l->ultimo)
59             l->ultimo = anterior;
60         x = aux->item;
61         aux->prox = NULL;
62         free(aux);
63     }
64     return x;
65 }
```



# Exercício 1

## Lista Simples

- Ainda falta uma função: aquela que apaga todos os nodos de uma lista.
- Vamos implementá-la a seguir.

# Exercício 1

## Lista Simples

---

### Algoritmo 12: ApagarLista

---

**Entrada:** Lista L.

```
1 início
2   aux ← L.primeiro
3   enquanto (L.primeiro ≠ NULL) faça
4     L.primeiro ← L.primeiro.prox
5     DESALOCA_NODO(aux)
6     aux = L.primeiro
```

---

# Exercício 1

## Lista Simples

Na linguagem C, a implementação fica conforme o código a seguir:

```
1 void apagarLista(lista *l) {  
2     nodo *aux = l->primeiro;  
3     while (aux != NULL) {  
4         l->primeiro = l->primeiro->prox;  
5         free(aux);  
6         aux = l->primeiro;  
7     }  
8 }
```

# Exercício 1

## Lista Simples

Vamos implementar a função que imprime os valores de uma lista.

# Exercício 1

## Lista Simples

---

### Algoritmo 13: ImprimirLista

---

**Entrada:** Lista L.

```
1 início  
2   aux ← L.primeiro  
3   enquanto (aux ≠ NULL) faça  
4     Imprima(aux.item)  
5     aux = aux.prox
```

---

# Exercício 1

## Lista Simples

Na linguagem C, a implementação fica conforme o código a seguir:

```
1 void imprimirLista(lista *l) {  
2     nodo *aux = l->primeiro;  
3     while (aux != NULL) {  
4         printf("%d ", aux->item);  
5         aux = aux->prox;  
6     }  
7     printf("\n");  
8 }
```

# Exercício 1

## Lista Simples

Após implementar essas funções, faça o teste, inserindo e removendo alguns valores.

# Exercício 1

## Lista Simples

Em um arquivo chamado `executa_lista.c`, insira o código a seguir:

```
1 #include "lista.h"
2 int main() {
3     int i;
4     lista *l = malloc(sizeof(lista));
5     criarListaVazia(l);
6     for (i=0; i<10; i++) {
7         inserirInicio(l,i);
8         printf("Inseriu Inicio %d\n",i);
9     }
10    for (i=-1;i>-3;i--) {
11        inserirFinal(l,i);
12        printf("Inseriu final %d\n",i);
13    }
14    printf("Inseriu %d na posicao 4\n",10);
15    inserirPosicao(l,10,4);
16    imprimirLista(l);
17    /** Continua no proximo slide **/
```



# Exercício 1

## Lista Simples

```
1 #include "lista.h"
2 int main() {
3     /**Codigo anterior **/
4     i = removerInicio(l);
5     printf("Removeu Inicio %d\n",i);
6     imprimirLista(l);
7     i = removerFinal(l);
8     printf("Removeu Final %d\n",i);
9     imprimirLista(l);
10    i = removerInicio(l);
11    printf("Removeu Inicio %d\n",i);
12    imprimirLista(l);
13    i = removerPosicao(l,7);
14    printf("Removeu %d na posicao %d\n",i,7);
15    apagarLista(l);
16    printf("Apagou a lista\n");
17    free(l);
18 }
```

# Listas Simplesmente Encadeadas

## Algoritmos e Estrutura de Dados

### Alocação Dinâmica

prof. Frederico Santos de Oliveira

Universidade Federal de Mato Grosso  
Instituto de Engenharia