

# Listas

**Prof. Roberto Hugo Wanderley Pinheiro**

roberto.hugo@ufca.edu.br



**UNIVERSIDADE  
FEDERAL DO CARIRI**

# Roteiro

- Definição
- Operações Básicas
- Tipos
- Lista Sequencial Estática
- Lista Dinâmica Simplesmente Encadeada
- Lista Dinâmica Duplamente Encadeada
- Lista Dinâmica Circular

# Definição

- Lista é uma Estrutura de Dados Simples que possui um ou mais elementos do mesmo tipo organizados linearmente, isto é, um existe após o outro, sem ramificações
- Caso a lista tenha zero elementos, ela é dita vazia
- Aplicações
  - Cadastro de funcionários
  - Itens em estoque
  - Baralho de cartas

# Operações Básicas

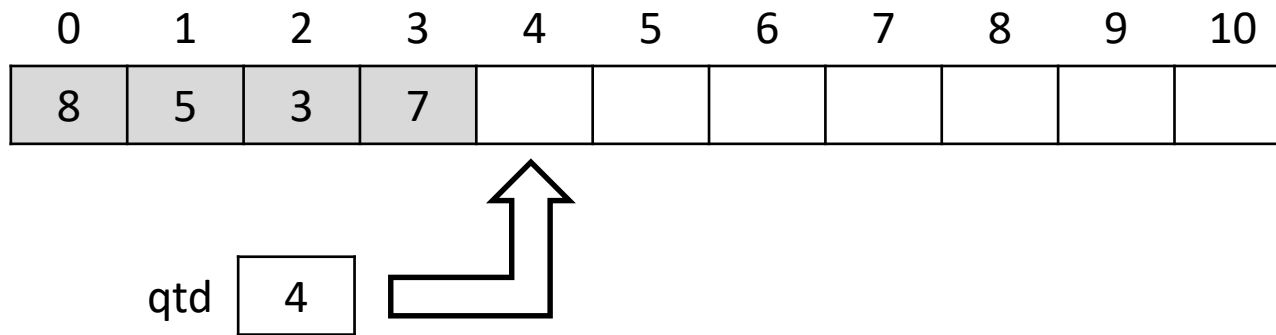
- Criar a lista
- Inserir um elemento
- Excluir um elemento
- Acessar um elemento
- Destruir a lista

# Tipos

- Uma lista pode ser
  - **Estática:** se for implementada com vetores
  - **Dinâmica:** se for implementada usando ponteiro para o próximo elemento
- E também pode ser
  - **Homogênea:** se armazenar apenas um tipo de dados primitivo
  - **Heterogênea:** caso contrário

# Tipos

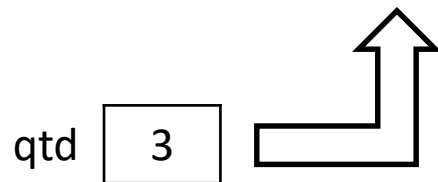
## Estática Homogênea



# Tipos

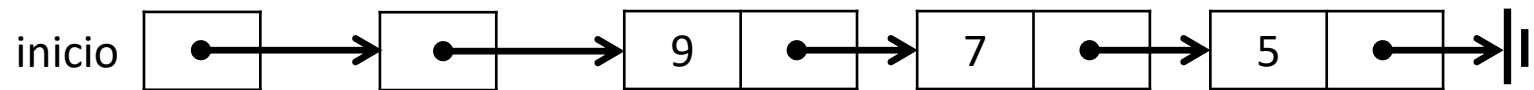
## Estática Heterogênea

0	1	2	3	4	5	6	7	8	9	10
João M 20	Ana F 37	Caio M 15								



# Tipos

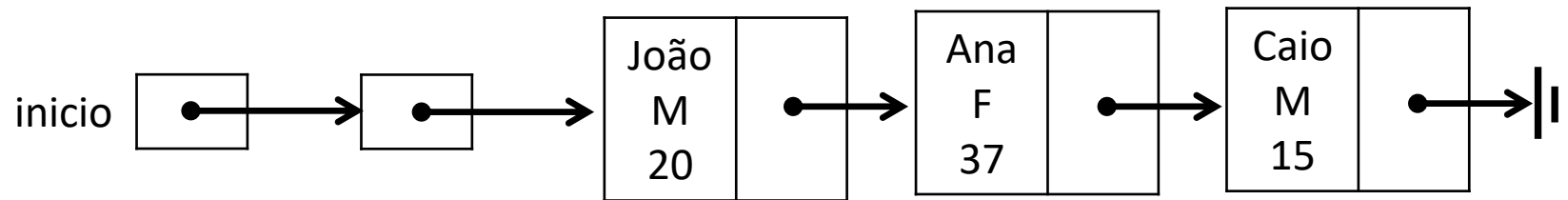
## Dinâmica Homogênea





# Tipos

## Dinâmica Heterogênea



# Lista Sequencial Estática

- Lista no qual o sucessor de um elemento ocupa a próxima posição de memória, pois é implementada com um vetor
- Vantagens
  - Acesso rápido e direto aos elementos pelo índice
  - Tempo constante para acessar um elemento
  - Facilidade em modificar informações
- Desvantagens
  - Definição prévia do tamanho
    - Memória alocada em compilação
  - Limite de tamanho (MAX)
  - Dificuldade para inserir ou remover elementos no começo da lista ou entre dois já existentes
    - Você precisa deslocar a lista inteira

# Lista Sequencial Estática

- Faremos a implementação passo-a-passo de acordo com os arquivos usando TAD e as funções básicas necessárias
- Arquivos
  - `main.c`
  - `listaSequencialEstatica.c`
  - `listaSequencialEstatica.h`
- Funções Básicas
  - Criar a lista
  - Inserir um elemento
  - Excluir um elemento
  - Acessar um elemento
  - Destruir a lista

# Lista Sequencial Estática

## ● listaSequencialEstatica.h

`#define MAX 50`  Definição de uma constante

```
struct aluno {  
    int matricula;  
    char nome[50];  
    float av1;  
    float av2;  
    float pr;  
};
```

Nesse exemplo, cada elemento da lista será um struct com vários dados

`typedef struct lista Lista;`  Definição do struct lista está no arquivo .c

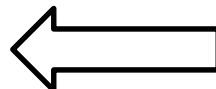
`Lista* criar();`  Assinatura/protótipo da função que está no arquivo .c

# Lista Sequencial Estática

## ● listaSequencialEstatica.c

```
#include <stdlib.h>
```

```
#include "listaSequencialEstatica.h"
```



Precisa incluir .h por conta de:

- Tamanho máximo
- Estrutura do Aluno
- Novo tipo nomeado Lista

```
struct lista {  
    int qtd;  
    struct aluno dados[MAX];  
};
```

```
Lista* criar() {
```

```
    Lista *lse;
```

```
    lse = (Lista*)malloc(sizeof(Lista));
```

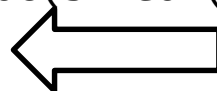
```
    if (lse != NULL) {
```

```
        lse->qtd = 0;
```

```
    }
```

```
    return lse;
```

```
}
```



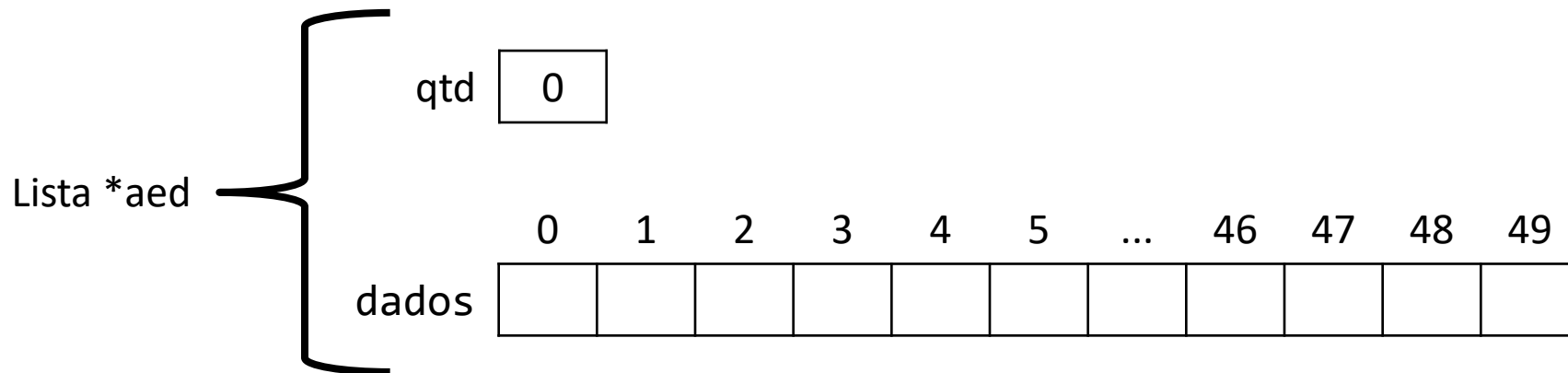
Para evitar acessar lse->qtd que não existe por memória não ter sido alocada corretamente

# Lista Sequencial Estática

## ● main.c

```
#include <stdio.h>
#include <stdlib.h>
#include "listaSequencialEstatica.h"
```

```
int main() {
    Lista *aed;
    aed = NULL;
    aed = criar(); ← Criação de uma lista vazia!
    return 0;
}
```



# Lista Sequencial Estática

- Assim, temos
  - listaSequencialEstatica.h
    - Tamanho máximo da Lista
    - Estrutura que será usada na Lista
    - Definição do tipo de dado
    - Protótipos das funções
  - listaSequencialEstatica.c
    - Estrutura da Lista em si
    - Implementação das funções
  - main.c
    - Uso prático da Lista

# Lista Sequencial Estática

- Sendo as Funções Básicas
  - Criar a lista ✓
  - Inserir um elemento
  - Excluir um elemento
  - Acessar um elemento
  - Destruir a lista
- Vamos fazer as outras...



# Lista Sequencial Estática

## ● Finalizar logo o listaSequencialEstatica.h

```
#define MAX 50
```

```
struct aluno {  
    int matricula;  
    char nome[50];  
    float av1;  
    float av2;  
    float pr;  
};
```

```
typedef struct lista Lista;
```

```
Lista* criar();  
void destruir(Lista *);  
int tamanho(Lista *);  
int cheia(Lista *);  
int vazia(Lista *);  
int inserirFim(Lista *, struct aluno);  
int inserirInicio(Lista *lse, struct aluno);  
int inserirOrdenado(Lista *, struct aluno);  
int removerFim(Lista *);  
int removerInicio(Lista *);  
int removerValor(Lista *, int);  
int acessarIndice(Lista *, int, struct aluno *);  
int acessarValor(Lista *, int, struct aluno *);
```

Sim, a implementação completa de uma Lista Sequencial Estática precisa de todas essas funções

# Lista Sequencial Estática

## ● listaSequencialEstatica.c

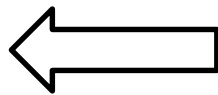
```
#include <stdlib.h>
```

```
#include "listaSequencialEstatica.h"
```

```
struct lista {  
    int qtd;  
    struct aluno dados[MAX];  
};
```

```
Lista* criar() {  
    ...  
}
```

```
void destruir(Lista *lse) {  
    free(lse);  
}
```



**Só isso a função? Melhor fazer no main.c**

Não. Usamos TAD justamente para deixar tudo mais organizado, colocar essa função do main.c iria contra esse princípio

# Lista Sequencial Estática

- Funções Básicas
  - Criar a lista ✓
  - Inserir um elemento
  - Excluir um elemento
  - Acessar um elemento
  - Destruir a lista ✓
- Antes de continuar com as demais Funções Básicas, vamos elaborar algumas auxiliares...

# Lista Sequencial Estática

## ● listaSequencialEstatica.c

```
#include <stdlib.h>
#include "listaSequencialEstatica.h"
```

```
struct lista {};
Lista* criar() {}
void destruir(Lista *lse) {}
```

```
int tamanho(Lista *lse) {
    if (lse == NULL)
        return -1;
    else
        return lse->qtd;
}
```

```
int cheia(Lista *lse) {
    if (lse == NULL)
        return -1;
    else if (lse->qtd == MAX)
        return 1;
    else
        return 0;
}
```

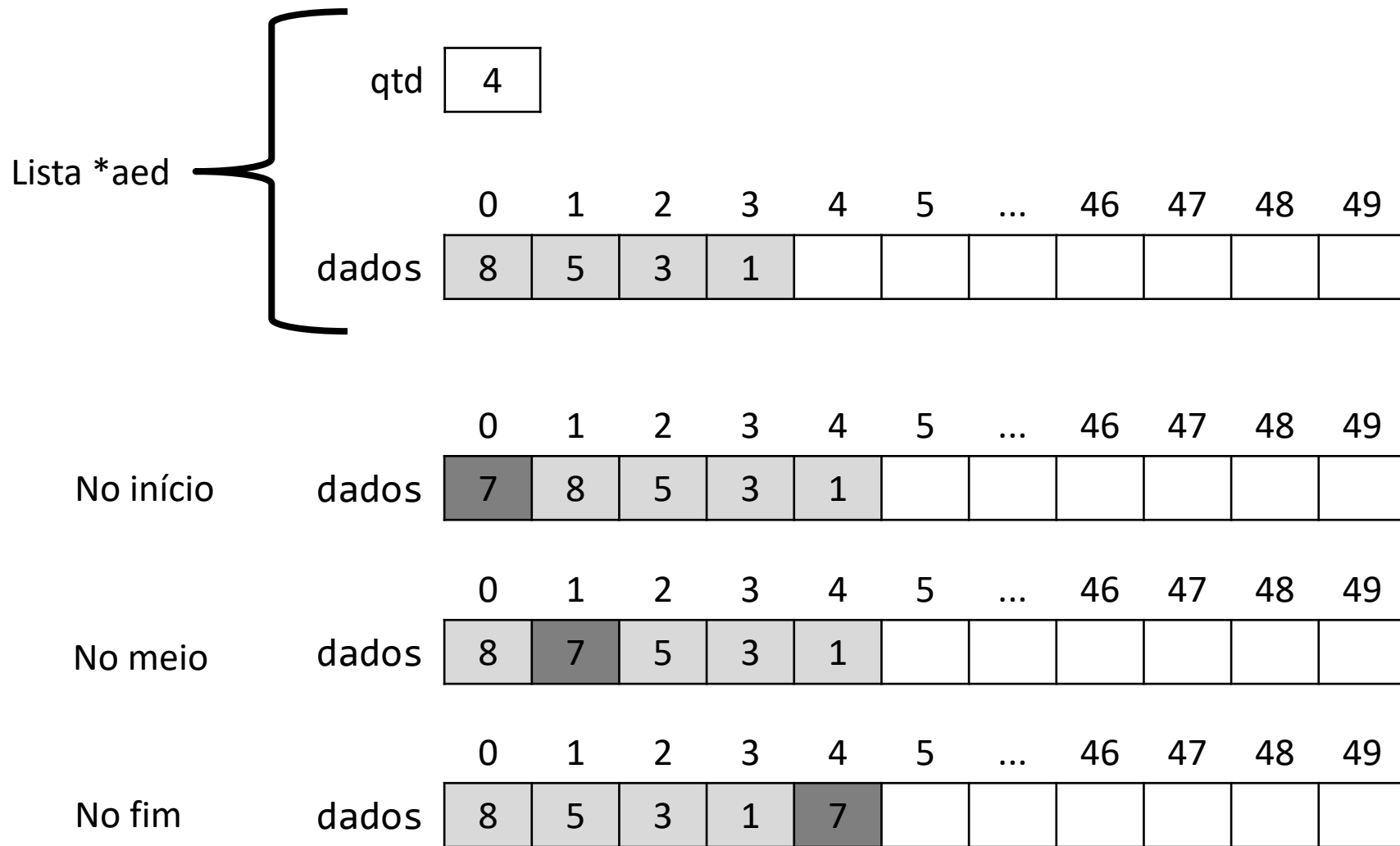
```
int vazia(Lista *lse) {
    if (lse == NULL)
        return -1;
    else if (lse->qtd == 0)
        return 1;
    else
        return 0;
}
```

# Lista Sequencial Estática

- Funções Básicas
  - Criar a lista ✓
  - **Inserir um elemento**
  - Excluir um elemento
  - Acessar um elemento
  - Destruir a lista ✓
- Para inserir um elemento, iremos elaborar três funções, pois o elemento pode ser inserido em várias posições na lista

# Lista Sequencial Estática

- Inserir um elemento



# Lista Sequencial Estática

- listaSequencialEstatica.c

```
int inserirFim(Lista *lse, struct aluno novo) {  
    if (lse == NULL)  
        return 0;  
    else if (cheia(lse))  
        return 0;  
    else {  
        lse->dados[lse->qtd] = novo;  
        lse->qtd++;  
        return 1;  
    }  
}
```

# Lista Sequencial Estática

- listaSequencialEstatica.c

```
int inserirInicio(Lista *lse, struct aluno novo) {
    if (lse == NULL)
        return 0;
    else if (cheia(lse))
        return 0;
    else {
        int i;
        for (i = (lse->qtd)-1 ; i >= 0 ; i--) {
            lse->dados[i+1] = lse->dados[i];
        }
        lse->dados[0] = novo;
        lse->qtd++;
        return 1;
    }
}
```



# Lista Sequencial Estática

- Exercício em Sala: Inserir Ordenado
  - Uma maneira de criar uma **inserção ordenada**

	0	1	2	3	4	5	...	46	47	48	49
dados	1	2	4	7							

Inserir número 5!

**Passo 1:** Encontrar a posição

**Passo 2:** Após encontrar, mover elementos para abrir espaço

**Passo 3:** Inserir novo número no local “vago”

	0	1	2	3	4	5	...	46	47	48	49
dados	1	2	4	5	7						

# Lista Sequencial Estática

## ● listaSequencialEstatica.c

```
int inserirOrdenado(Lista *lse, struct aluno novo) {
    if (lse == NULL)
        return 0;
    else if (cheia(lse))
        return 0;
    else {
        int i, pos = 0;
        while (lse->dados[pos].matricula < novo.matricula && pos < lse->qtd) {
            pos++;
        }
        for (i = (lse->qtd)-1 ; i >= pos ; i--) {
            lse->dados[i+1] = lse->dados[i];
        }
        lse->dados[pos] = novo;
        lse->qtd++;
        return 1;
    }
}
```

# Lista Sequencial Estática

- Funções Básicas
  - Criar a lista ✓
  - Inserir um elemento ✓
  - **Excluir um elemento**
  - Acessar um elemento
  - Destruir a lista ✓
- Similarmente a inserir, a função excluir também pode ocorrer no início, meio e fim

# Lista Sequencial Estática

- listaSequencialEstatica.c

```
int removerFim(Lista *lse) {  
    if (lse == NULL)  
        return 0;  
    else if (vazia(lse))  
        return 0;  
    else {  
        lse->qtd--;  
        return 1;  
    }  
}
```

# Lista Sequencial Estática

- listaSequencialEstatica.c

```
int removerInicio(Lista *lse) {
    if (lse == NULL)
        return 0;
    else if (vazia(lse))
        return 0;
    else {
        int i;
        for (i = 0 ; i < (lse->qtd)-1 ; i++) {
            lse->dados[i] = lse->dados[i+1];
        }
        lse->qtd--;
        return 1;
    }
}
```

# Lista Sequencial Estática

- Exercício em Sala: Remover Valor
  - Serve para remover um elemento específico que pode estar no início, meio ou fim da Lista

	0	1	2	3	4	5	...	46	47	48	49
dados	1	2	5	7							

Remover número 5!

**Passo 1:** Encontrar o elemento especificado

**Passe 2:** Caso não encontre o elemento, não é possível remover

**Passo 3:** Após encontrar, mover elementos da frente para ocupar seu espaço

	0	1	2	3	4	5	...	46	47	48	49
dados	1	2	7								

# Lista Sequencial Estática

## ● listaSequencialEstatica.c

```
int removerValor(Lista *lse, int x) {
    if (lse == NULL)
        return 0;
    else if (vazia(lse))
        return 0;
    else {
        int i, pos = 0;
        while (lse->dados[pos].matricula != x && pos < lse->qtd)
            pos++;
        if (pos == lse->qtd)
            return 0;

        for (i = pos ; i < (lse->qtd)-1 ; i++) {
            lse->dados[i] = lse->dados[i+1];
        }
        lse->qtd--;
        return 1;
    }
}
```

# Lista Sequencial Estática

- Funções Básicas
  - Criar a lista ✓
  - Inserir um elemento ✓
  - Excluir um elemento ✓
  - **Acessar um elemento**
  - Destruir a lista ✓
- Podemos acessar um elemento pela sua posição (acesso direto) ou pelo seu valor (requer busca)



# Lista Sequencial Estática

- listaSequencialEstatica.c

```
int acessarIndice(Lista *lse, int pos, struct aluno *a) {  
    if (lse == NULL)  
        return 0;  
    else if (pos < 0 || pos >= lse->qtd)  
        return 0;  
    else {  
        *a = lse->dados[pos];  
        return 1;  
    }  
}
```

# Lista Sequencial Estática

- listaSequencialEstatica.c

```
int acessarValor(Lista *lse, int x, struct aluno *a) {
    if (lse == NULL)
        return 0;
    else {
        int pos = 0;
        while (lse->dados[pos].matricula != x && pos < lse->qtd)
            pos++;
        if (pos == lse->qtd)
            return 0;
        else {
            *a = lse->dados[pos];
            return 1;
        }
    }
}
```

# Lista Sequencial Estática

- Funções Básicas
  - Criar a lista ✓
  - Inserir um elemento ✓
  - Excluir um elemento ✓
  - Acessar um elemento ✓
  - Destruir a lista ✓
- Encerramos... Mais ou menos
- Falta elaborar o programa principal

# Lista Sequencial Estática

- Exercício em Sala: Faça um programa main.c que contenha as seguintes funcionalidades
  - Duas listas, uma comum e outra ordenada
  - Um menu retornável que lhe permita realizar todas as Operações Básicas com as duas listas de modo que elas funcionem corretamente, isto é
    - Que você cadastre alunos
    - Que você consulte alunos
    - Que você remova alunos

# Lista Sequencial Estática

## ● main.c

```
#include <stdio.h>
#include <stdlib.h>
#include "listaSequencialEstatica.h"
```

```
int main() {
    Lista *aed;
    aed = criar();
    destruir(aed);
    return 0;
}
```

# Lista Sequencial Estática

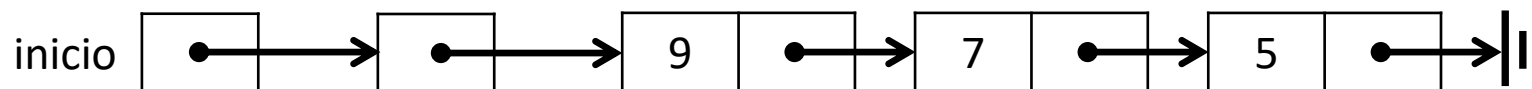
- Concluindo... Quando devo usar?
  - Listas pequenas
  - Aplicação que apenas insira e remova no final
  - Tamanho máximo bem definido
  - Operação mais frequente for acesso direto
- Quantos mais itens desses existirem na sua aplicação, melhor

# Lista Dinâmica Simplesmente Encadeada

- Lista no qual o sucessor de um elemento ocupa uma posição de memória acessada por um ponteiro no elemento atual
- Vantagens
  - Melhor utilização da memória
    - Não há alocação de memória para vários elementos de uma só vez
  - Sem definição de tamanho, logo sem limite
    - Memória alocada durante a execução do programa
  - Insere ou remove elementos sem precisar deslocar outros
- Desvantagens
  - Tempo de acesso variável
    - Depende do quão “fundo” na lista está o elemento
  - Sem acesso aos elementos por índice
    - É necessário percorrer todos os seus antecessores na lista

# Lista Dinâmica Simplesmente Encadeada

- Faremos a implementação passo-a-passo de acordo com as arquivos usando TAD e as funções básicas necessárias
- Arquivos
  - `main.c`
  - `listaSimplesmenteEncadeada.c`
  - `listaSimplesmenteEncadeada.h`
- Funções Básicas
  - Criar a lista
  - Inserir um elemento
  - Excluir um elemento
  - Acessar um elemento
  - Destruir a lista



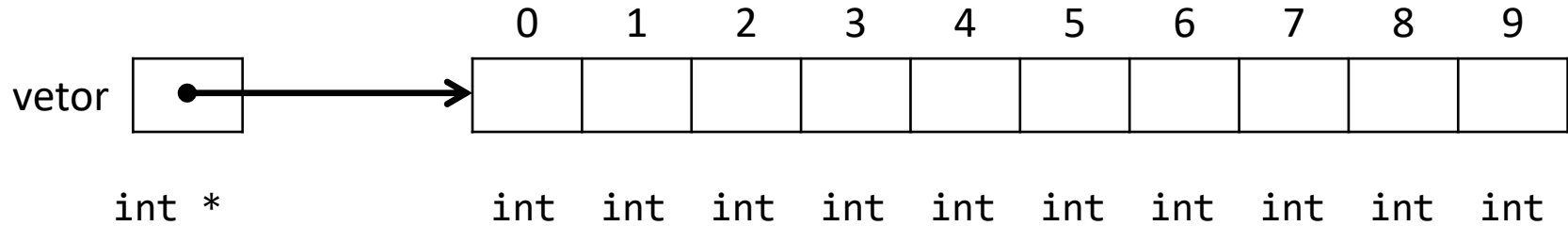


# Lista Dinâmica Simplesmente Encadeada

- Antes de começar é importante salientar que usaremos ponteiro de ponteiro aqui
- Isto é, em vez de `struct algo *` usaremos `struct algo **`
- Motivo?
  - Para poder alterar o início da lista nas funções
- Com apenas um nível de ponteiro eu não consigo alterar o ponteiro que desejo
- Não faz sentido?
  - Vamos para um exemplo

# Lista Dinâmica Simplesmente Encadeada

```
int *vetor = (int *)malloc(10*sizeof(int));
```



- Quando eu passo `vetor` por uma função, eu consigo alterar os valores dentro do vetor, mas não para quem o vetor aponta
  - Não é possível alterar o `int *` apenas os `int`
- Se eu quiser alterar o `int *` dentro de uma função eu preciso de um `int **`
- É como faremos com as Listas Dinâmicas

# Lista Dinâmica Simplesmente Encadeada

## ● listaSimplesmenteEncadeada.h

```
struct aluno {  
    int matricula;  
    char nome[50];  
    float av1;  
    float av2;  
    float pr;  
};
```

Manteremos a estrutura de alunos.  
Assim, cada elemento da lista será  
um struct com vários dados

```
typedef struct elemento* Lista;
```

Definição do struct elemento está no arquivo .c  
Perceba que dessa vez usamos ponteiro no  
typedef, isso serve para facilitar a criação e  
leitura do ponteiro de ponteiro no main()

```
Lista* criar();
```

Assinatura/protótipo da função que está no arquivo .c

# Lista Dinâmica Simplesmente Encadeada

## ● listaSimplesmenteEncadeada.c

```
#include <stdlib.h>
```

```
#include "listaSimplesmenteEncadeada.h"
```

Precisa incluir .h por conta de:

- Estrutura do Aluno
- Novo tipo nomeado Lista

```
struct elemento {
```

```
    struct aluno dados;
```

```
    struct elemento *prox;
```

Temos os dados e uma definição Recursiva que aponta para o próximo elemento da Lista

```
};
```

```
typedef struct elemento Elemento;
```

Pra facilitar as operações, defini um novo tipo de dado para o Elemento em si

```
Lista* criar() {
```

```
    Lista *ldse;
```

```
    ldse = (Lista*)malloc(sizeof(Lista));
```

```
    if (ldse != NULL) {
```

```
        *ldse = NULL;
```

Se alocou memória com sucesso, definir o ponteiro como nulo, pois não existem elementos na Lista!

```
    }
```

```
    return ldse;
```

```
}
```

# Lista Dinâmica Simplesmente Encadeada

## ● main.c

```
#include <stdio.h>
#include <stdlib.h>
#include "listaSimplesmenteEncadeada.h"

int main() {
    Lista *aed;
    aed = NULL;
    aed = criar(); ← Criação de uma lista vazia!
    return 0;
}
```



struct elemento \*\*  
(Lista \*)

struct elemento \*  
(Lista)

# Lista Dinâmica Simplesmente Encadeada

- Assim, temos
  - listaSimplesmenteEncadeada.h
    - Estrutura que será usada na Lista
    - Definição do tipo de dado
    - Protótipos das funções
  - listaSimplesmenteEncadeada.c
    - Estrutura da Lista em si
    - Definição de um tipo auxiliar Elemento
    - Implementação das funções
  - main.c
    - Uso prático da Lista

# Lista Dinâmica Simplesmente Encadeada

- Sendo as Funções Básicas
  - Criar a lista ✓
  - Inserir um elemento
  - Excluir um elemento
  - Acessar um elemento
  - Destruir a lista
- Vamos fazer as outras...

# Lista Dinâmica Simplesmente Encadeada

## ● Finalizar logo o listaSimplesmenteEncadeada.h

```
struct aluno {
    int matricula;
    char nome[50];
    float av1;
    float av2;
    float pr;
};

typedef struct elemento *Lista;

Lista* criar();
void destruir(Lista *);
int tamanho(Lista *);
int cheia(Lista *);
int vazia(Lista *);
int inserirFim(Lista *, struct aluno);
int inserirInicio(Lista *, struct aluno);
int inserirOrdenado(Lista *, struct aluno);
int removerFim(Lista *);
int removerInicio(Lista *);
int removerValor(Lista *, int);
int acessarIndice(Lista *, int, struct aluno *);
int acessarValor(Lista *, int, struct aluno *);
```



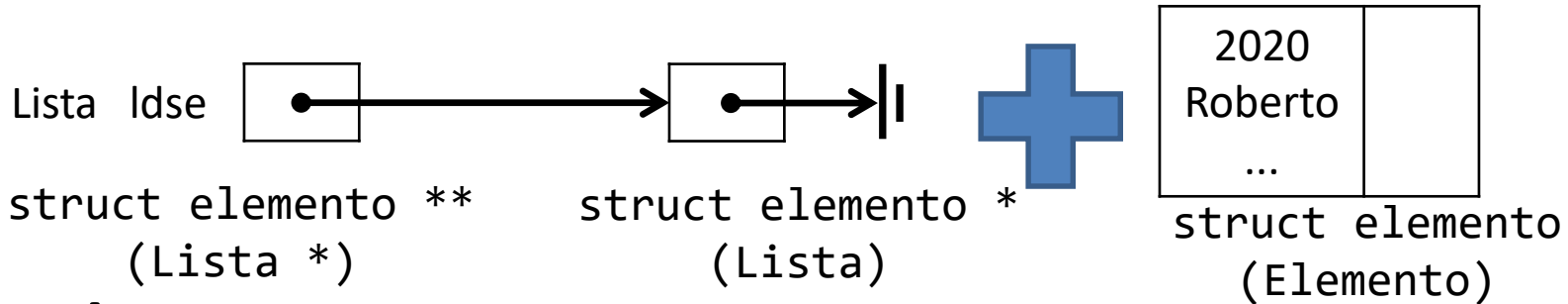
# Lista Dinâmica Simplesmente Encadeada

- listaSimplesmenteEncadeada.c
  - Vamos ver primeiro a função inserirInicio para compreender melhor a estrutura dos ponteiros

```
int inserirInicio(Lista *ldse, struct aluno novosdados) {
    if (ldse == NULL) {
        return 0;
    }
    else {
        Elemento *novo = (Elemento*)malloc(sizeof(Elemento));
        if (novo == NULL) return 0;
        novo->dados = novosdados;
        novo->prox = *ldse;
        *ldse = novo;
        return 1;
    }
}
```

# Lista Dinâmica Simplesmente Encadeada

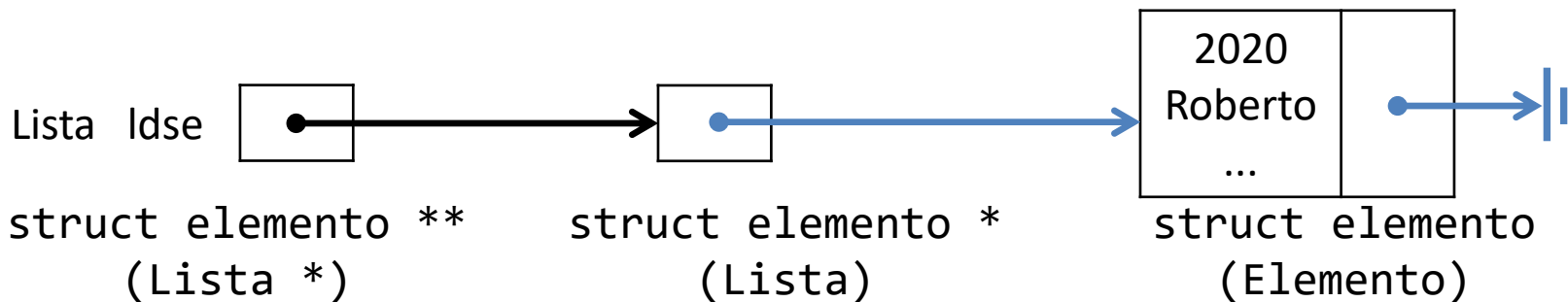
- Vamos inserir um aluno na Lista vazia



- Ao executar...

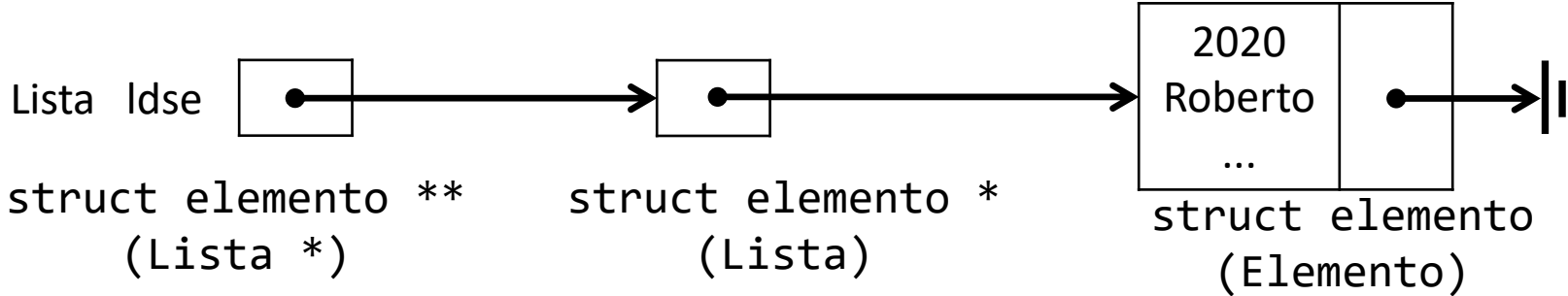
```
novo->prox = *ldse;  
*ldse = novo;
```

- Temos



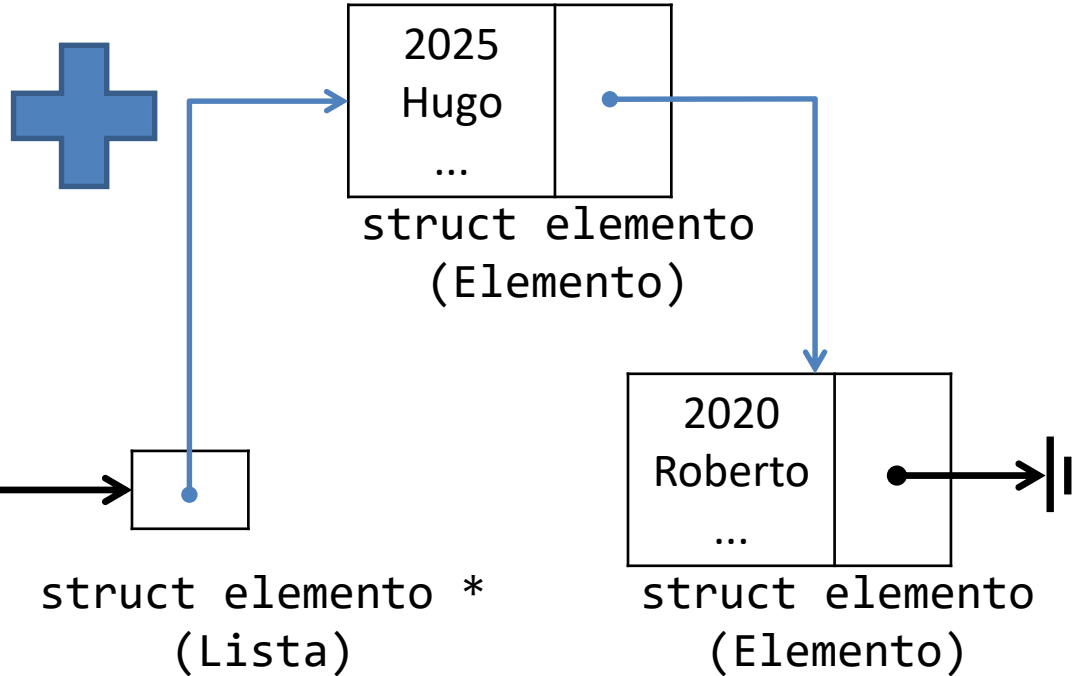
# Lista Dinâmica Simplesmente Encadeada

- Vamos inserir outro aluno na Lista

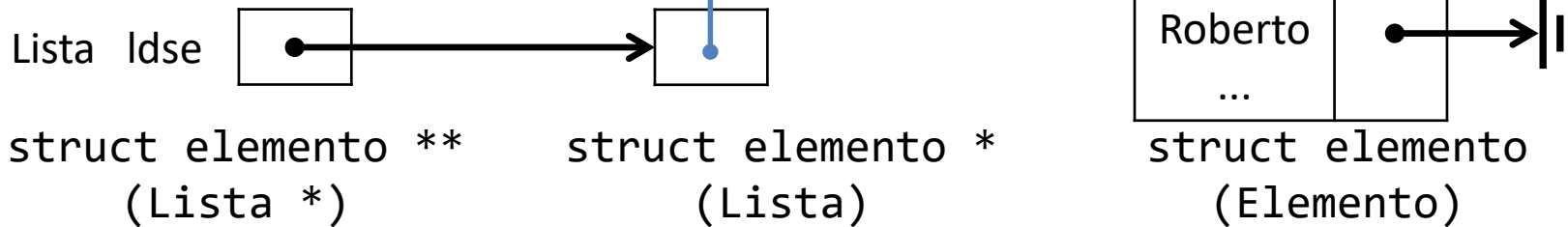


- Ao executar...

```
novo->prox = *ldse;  
*ldse = novo;
```

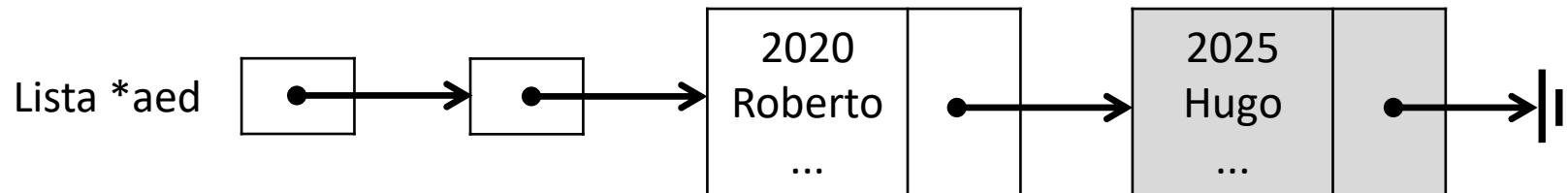
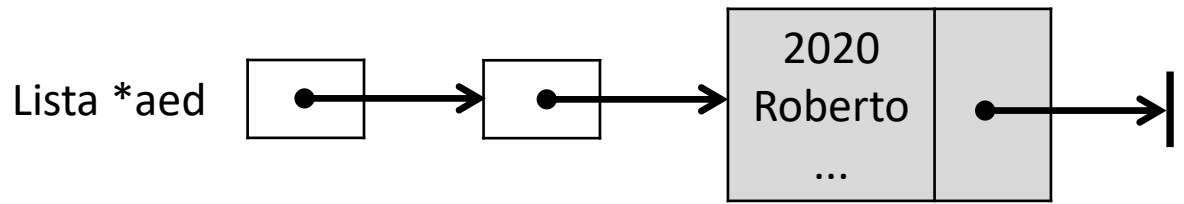
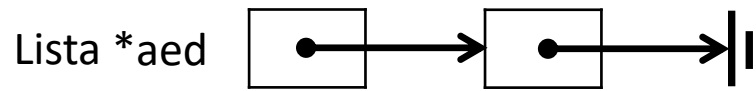


- Temos



# Lista Dinâmica Simplesmente Encadeada

- Exercício em Sala: Inserir no Fim



# Lista Dinâmica Simplesmente Encadeada

## ● listaSimplesmenteEncadeada.c

```
int inserirFim(Lista *ldse, struct aluno novosdados) {
    if (ldse == NULL) {
        return 0;
    }
    else {
        Elemento *novo = (Elemento*)malloc(sizeof(Elemento));
        if (novo == NULL) return 0;
        novo->dados = novosdados;
        novo->prox = NULL;
        if (vazia(ldse)) {
            *ldse = novo;
        }
        else {
            Elemento *aux = *ldse;
            while (aux->prox != NULL) aux = aux->prox;
            aux->prox = novo;
        }
        return 1;
    }
}
```

# Lista Dinâmica Simplesmente Encadeada

## ● listaSimplesmenteEncadeada.c

```
int inserirOrdenado(Lista *ldse, struct aluno novosdados) {
    if (ldse == NULL) {
        return 0;
    }
    else {
        Elemento *novo = (Elemento*)malloc(sizeof(Elemento));
        if (novo == NULL) return 0;
        novo->dados = novosdados;
        if (vazia(ldse) || (*ldse)->dados.matricula > novo->dados.matricula) {
            novo->prox = *ldse;
            *ldse = novo;
        }
        else {
            // Busca pelo local correto
        }
        return 1;
    }
}
```

# Lista Dinâmica Simplesmente Encadeada

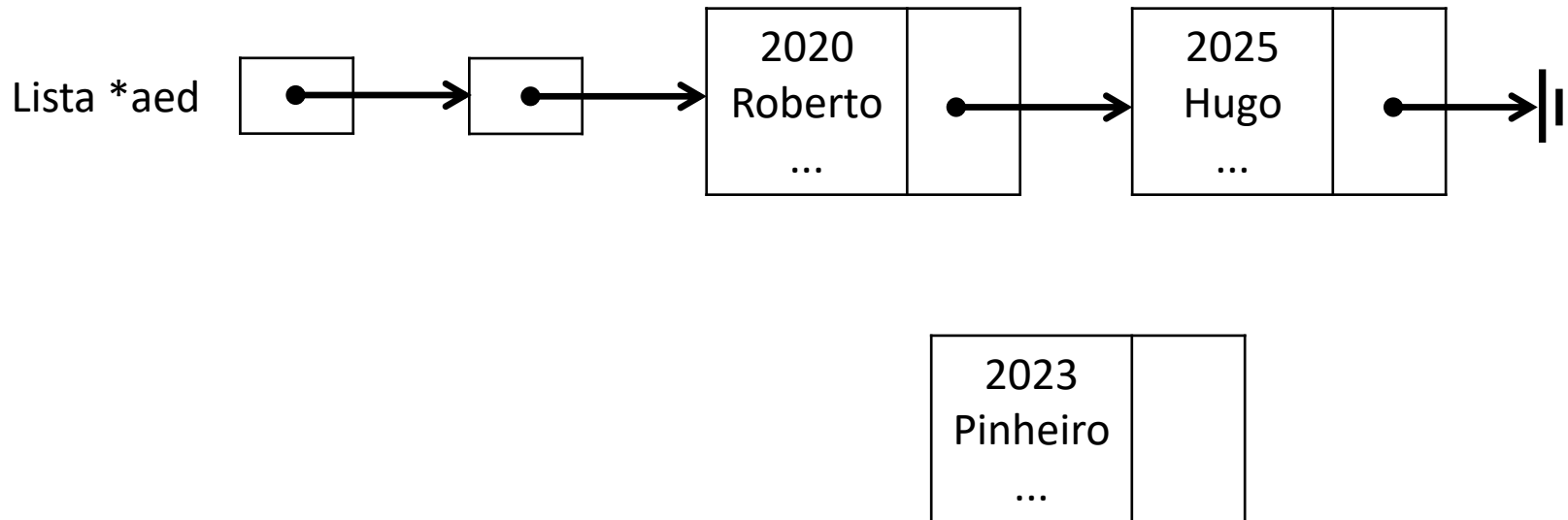
## ● listaSimplesmenteEncadeada.c

```
int inserirOrdenado(Lista *ldse, struct aluno novosdados) {  
  
    (...)  
  
    else {  
        Elemento *ant = *ldse;  
        Elemento *aux = ant->prox;  
        while (aux != NULL && aux->dados.matricula < novo->dados.matricula) {  
            ant = aux;  
            aux = aux->prox;  
        }  
        ant->prox = novo;  
        novo->prox = aux;  
    }  
    return 1;  
}  
}
```

# Lista Dinâmica Simplesmente Encadeada

## ● Exemplo de Inserir ordenado

```
Elemento *ant = *ldse;  
Elemento *aux = ant->prox;  
while (aux != NULL && aux->dados.matricula < novo->dados.matricula) {  
    ant = aux;  
    aux = aux->prox;  
}  
ant->prox = novo;  
novo->prox = aux;
```





# Lista Dinâmica Simplesmente Encadeada

- Funções Básicas
  - Criar a lista ✓
  - Inserir um elemento ✓
  - Excluir um elemento
  - Acessar um elemento
  - Destruir a lista
- Antes de continuar com as demais Funções Básicas, vamos elaborar algumas auxiliares...

# Lista Dinâmica Simplesmente Encadeada

## ● listaSimplesmenteEncadeada.c

```
int tamanho(Lista *ldse) {
    if (vazia(ldse)) {
        return 0;
    }
    int cont = 0;
    Elemento *aux = *ldse;
    while (aux != NULL) {
        cont++;
        aux = aux->prox;
    }
    return cont;
}
```

```
int cheia(Lista *ldse) {
    return 0;
}
```

```
int vazia(Lista *ldse) {
    if (ldse == NULL) {
        return 1;
    }
    else if (*ldse == NULL) {
        return 1;
    }
    else {
        return 0;
    }
}
```

# Lista Dinâmica Simplesmente Encadeada

- Funções Básicas
  - Criar a lista ✓
  - Inserir um elemento ✓
  - Excluir um elemento
  - Acessar um elemento
  - **Destruir a lista**
- Agora vamos destruir...

# Lista Dinâmica Simplesmente Encadeada

- listaSimplesmenteEncadeada.c

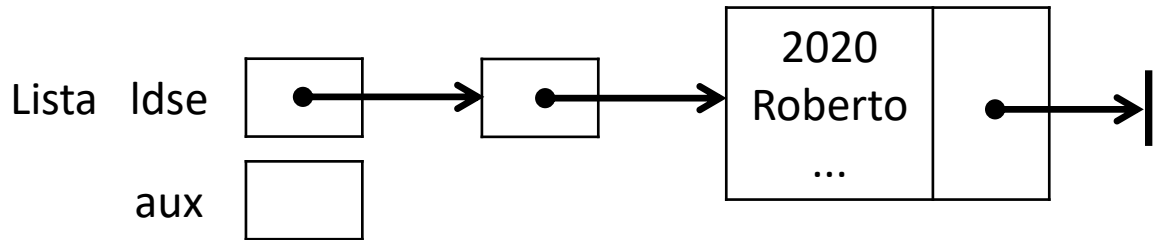
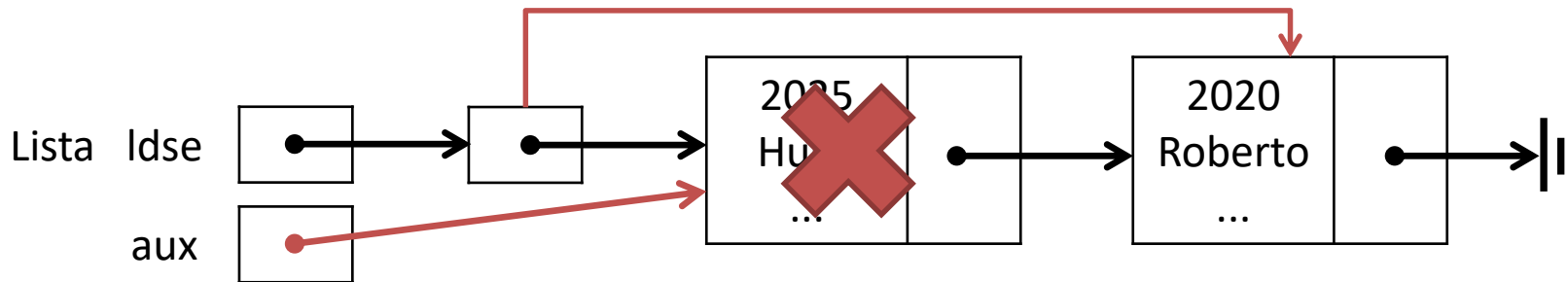
```
void destruir(Lista *ldse) {  
    if (ldse != NULL) {  
        Elemento *aux;  
        while (*ldse != NULL) {  
            aux = *ldse;  
            *ldse = (*ldse)->prox;  
            free(aux);  
        }  
        //free(ldse);  
    }  
}
```

Para de fato limpar toda a memória precisaria desse free, mas é melhor não fazer ele, para não perder o NULL que indica que a lista está vazia

# Lista Dinâmica Simplesmente Encadeada

- Vamos executar a função `destruir()`!

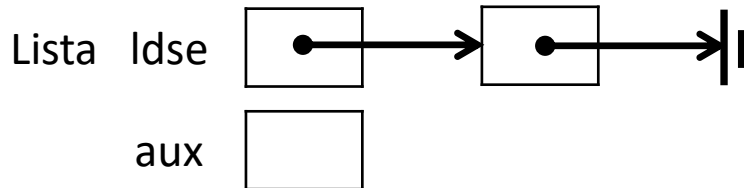
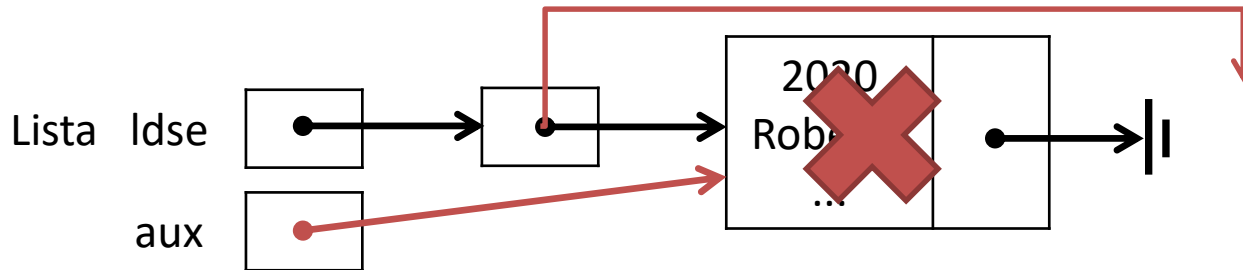
```
aux = *ldse;  
*ldse = (*ldse)->prox;  
free(aux);
```



# Lista Dinâmica Simplesmente Encadeada

- Vamos executar a função `destruir()`!

```
aux = *ldse;  
*ldse = (*ldse)->prox;  
free(aux);
```



# Lista Dinâmica Simplesmente Encadeada

- Funções Básicas
  - Criar a lista ✓
  - Inserir um elemento ✓
  - **Excluir um elemento**
  - Acessar um elemento
  - Destruir a lista ✓
- Similarmente a inserir, a função excluir também pode ocorrer no início, meio e fim

# Lista Dinâmica Simplesmente Encadeada

## ● listaSimplesmenteEncadeada.c

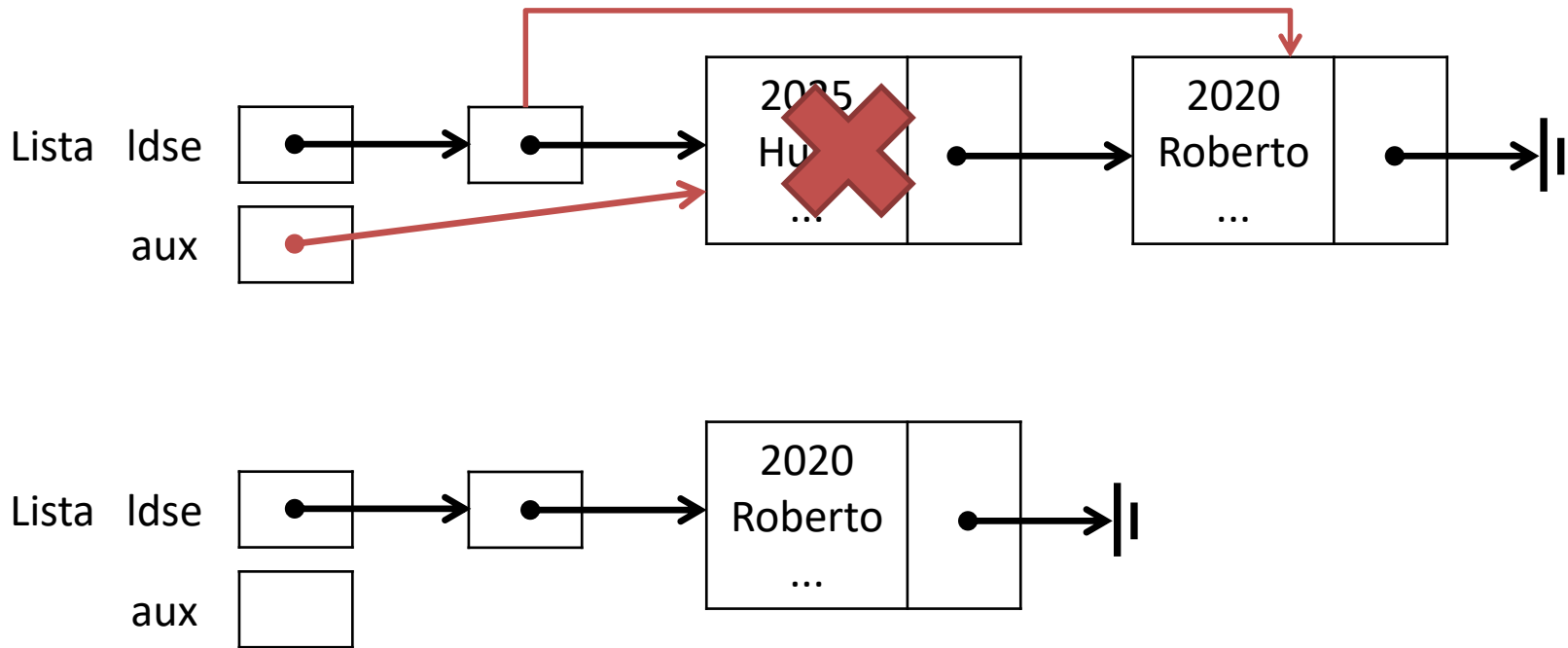
```
int removerInicio(Lista *ldse) {  
    if (vazia(ldse)) {  
        return 0;  
    }  
    else {  
        Elemento *aux = *ldse;  
        *ldse = aux->prox;  
        free(aux);  
        return 1;  
    }  
}
```



# Lista Dinâmica Simplesmente Encadeada

## ● Exemplo de Remoção no Início

```
Elemento *aux = *ldse;  
*ldse = aux->prox;  
free(aux);
```



# Lista Dinâmica Simplesmente Encadeada

## ● listaSimplesmenteEncadeada.c

```
int removerFim(Lista *ldse) {
    if (vazia(ldse)) {
        return 0;
    }
    else if ((*ldse)->prox == NULL){
        Elemento *aux = *ldse;
        *ldse = aux->prox;
        free(aux);
        return 1;
    }
    else {
        // Caminhando até o final da lista
    }
}
```

# Lista Dinâmica Simplesmente Encadeada

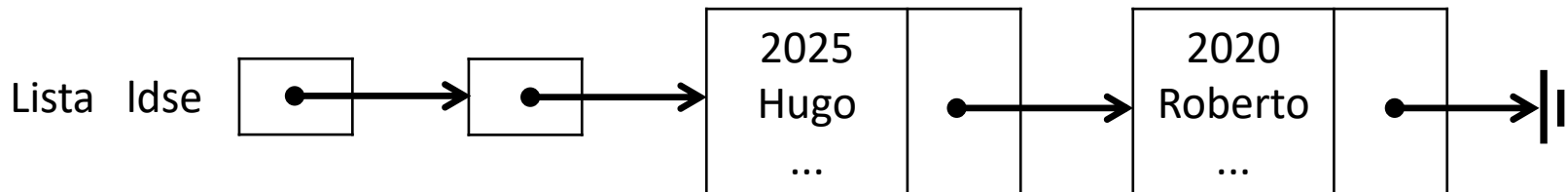
## ● listaSimplesmenteEncadeada.c

```
int removerFim(Lista *ldse) {  
  
    (...)  
  
    else {  
        Elemento *ant = *ldse;  
        Elemento *aux = ant->prox;  
        while (aux->prox != NULL) {  
            ant = aux;  
            aux = aux->prox;  
        }  
        ant->prox = aux->prox;  
        free(aux);  
        return 1;  
    }  
}
```

# Lista Dinâmica Simplesmente Encadeada

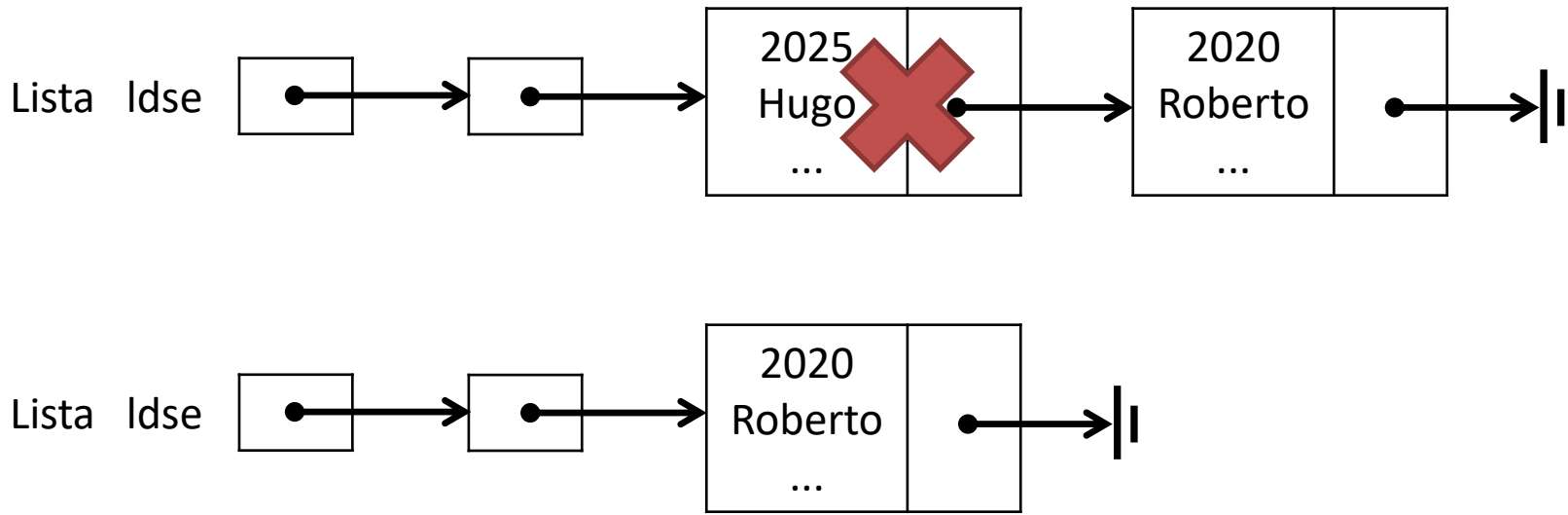
## ● Exemplo de Remoção no Final

```
Elemento *ant = *ldse;  
Elemento *aux = ant->prox;  
while (aux->prox != NULL) {  
    ant = aux;  
    aux = aux->prox;  
}  
ant->prox = aux->prox;  
free(aux);
```



# Lista Dinâmica Simplesmente Encadeada

- Exercício em Sala: Remover Valor
  - Remover Valor é remover um elemento específico
  - Exemplo: remover matrícula 2025



# Lista Dinâmica Simplesmente Encadeada

## ● listaSimplesmenteEncadeada.c

```
int removerValor(Lista *ldse, int x) {
    if (vazia(ldse)) {
        return 0;
    }
    else if ((*ldse)->dados.matricula == x){
        Elemento *aux = *ldse;
        *ldse = aux->prox;
        free(aux);
        return 1;
    }
    else {
        // Buscando o elemento
    }
}
```

# Lista Dinâmica Simplesmente Encadeada

## ● listaSimplesmenteEncadeada.c

```
int removerValor(Lista *ldse, int x) {  
    (...)  
  
    else {  
        Elemento *ant = *ldse;  
        Elemento *aux = ant->prox;  
        while (aux != NULL && aux->dados.matricula != x) {  
            ant = aux;  
            aux = aux->prox;  
        }  
        if (aux == NULL) return 0;  
        ant->prox = aux->prox;  
        free(aux);  
        return 1;  
    }  
}
```

# Lista Dinâmica Simplesmente Encadeada

- Funções Básicas
  - Criar a lista ✓
  - Inserir um elemento ✓
  - Excluir um elemento ✓
  - **Acessar um elemento**
  - Destruir a lista ✓
- Podemos acessar um elemento pela sua posição ou pelo seu valor
  - Ambos os casos requer uma busca na Lista



# Lista Dinâmica Simplesmente Encadeada

- listaSimplesmenteEncadeada.c

```
int acessarIndice(Lista *ldse, int pos, struct aluno *a) {  
    if (vazia(ldse))  
        return 0;  
    else if (pos < 0)  
        return 0;  
    else {  
        int cont = 0;  
        Elemento *aux = *ldse;  
        while (aux != NULL && pos != cont) {  
            aux = aux->prox;  
            cont++;  
        }  
        if (aux == NULL) return 0;  
        *a = aux->dados;  
        return 1;  
    }  
}
```

# Lista Dinâmica Simplesmente Encadeada

## ● listaSimplesmenteEncadeada.c

```
int acessarValor(Lista *ldse, int x, struct aluno *a) {
    if (vazia(ldse))
        return 0;
    else {
        Elemento *aux = *ldse;
        while (aux != NULL && aux->dados.matricula != x) {
            aux = aux->prox;
        }
        if (aux == NULL) return 0;
        *a = aux->dados;
        return 1;
    }
}
```

# Lista Dinâmica Simplesmente Encadeada

- Funções Básicas
  - Criar a lista ✓
  - Inserir um elemento ✓
  - Excluir um elemento ✓
  - Acessar um elemento ✓
  - Destruir a lista ✓
- Encerramos... Mais ou menos
- Falta elaborar o programa principal
- Eis a “magia” TAD...

# Lista Dinâmica Simplesmente Encadeada

- Exercício em Sala: Pegue o mesmo programa que você fez para Lista Estática e use-o para a Lista Dinâmica e veja-o funcionar normalmente!
  - Obs: se você criou funções que não vimos aqui, como `imprimirLista()` você vai precisar fazer a versão da Lista Dinâmica para que o `main()` antigo funcione!

# Lista Dinâmica Simplesmente Encadeada

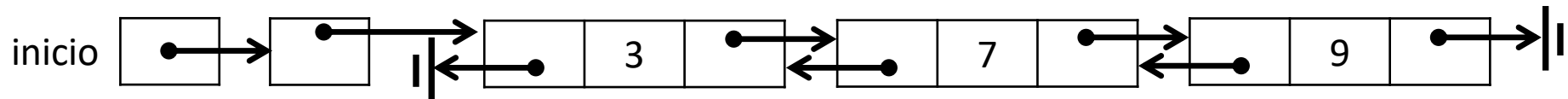
- Concluindo... Quando devo usar?
  - Listas ordenadas
  - Tamanho indefinido a priori
  - Operação mais frequente for acesso inserção e remoção (ordenada)
- Quantos mais itens desses existirem na sua aplicação, melhor

# Lista Dinâmica Duplamente Encadeada

- Lista no qual todo elemento possui um ponteiro para o elemento antecessor e sucessor
- Possui as mesmas Vantagens e Desvantagens da Lista Dinâmica Simplesmente Encadeada, comparativamente temos
- Vantagens
  - Capacidade de acessar elemento antecessor
- Desvantagens
  - Mais difícil de implementar
  - Um pouco mais de memória

# Lista Dinâmica Duplamente Encadeada

- Faremos a implementação passo-a-passo de acordo com as arquivos usando TAD e as funções básicas necessárias
- Arquivos
  - `main.c`
  - `listaDuplamenteEncadeada.c`
  - `listaDuplamenteEncadeada.h`
- Funções Básicas
  - Criar a lista
  - Inserir um elemento
  - Excluir um elemento
  - Acessar um elemento
  - Destruir a lista



# Lista Dinâmica Duplamente Encadeada

## ● listaDuplamenteEncadeada.h

```
struct aluno {
    int matricula;
    char nome[50];
    float av1;
    float av2;
    float pr;
};

typedef struct elemento *Lista;

Lista* criar();
void destruir(Lista *);
int tamanho(Lista *);
int vazia(Lista *);
int inserirFim(Lista *, struct aluno);
int inserirInicio(Lista *, struct aluno);
int inserirOrdenado(Lista *, struct aluno);
int removerFim(Lista *);
int removerInicio(Lista *);
int removerValor(Lista *, int);
int acessarIndice(Lista *, int, struct aluno *);
int acessarValor(Lista *, int, struct aluno *);
```

Funções em azul são idênticas às implementadas na Lista Dinâmica Simplesmente Encadeada! Afinal, em nenhuma delas eu preciso modificar o ponteiro do elemento anterior.



# Lista Dinâmica Duplamente Encadeada

## ● listaDuplamenteEncadeada.c

```
#include <stdlib.h>
```

```
#include "listaDuplamenteEncadeada.h"
```

```
struct elemento {
```

```
    struct elemento *ant;
```

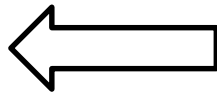
```
    struct aluno;
```

```
    struct elemento *prox;
```

```
};
```

```
typedef struct elemento Elemento;
```

```
// Funções...
```



Temos os dados e uma definição Recursiva que aponta para o elemento anterior e o próximo

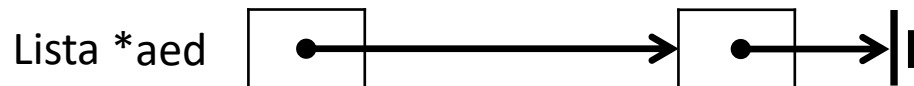
Iremos ver apenas as funções que diferem da Lista Simplesmente Encadeada

# Lista Dinâmica Duplamente Encadeada

## ● main.c

```
#include <stdio.h>
#include <stdlib.h>
#include "listaDuplamenteEncadeada.h"

int main() {
    Lista *aed;
    aed = NULL;
    aed = criar(); ← Criação de uma lista vazia!
    return 0;
}
```



struct elemento \*\*  
(Lista \*)

struct elemento \*  
(Lista)

# Lista Dinâmica Duplamente Encadeada

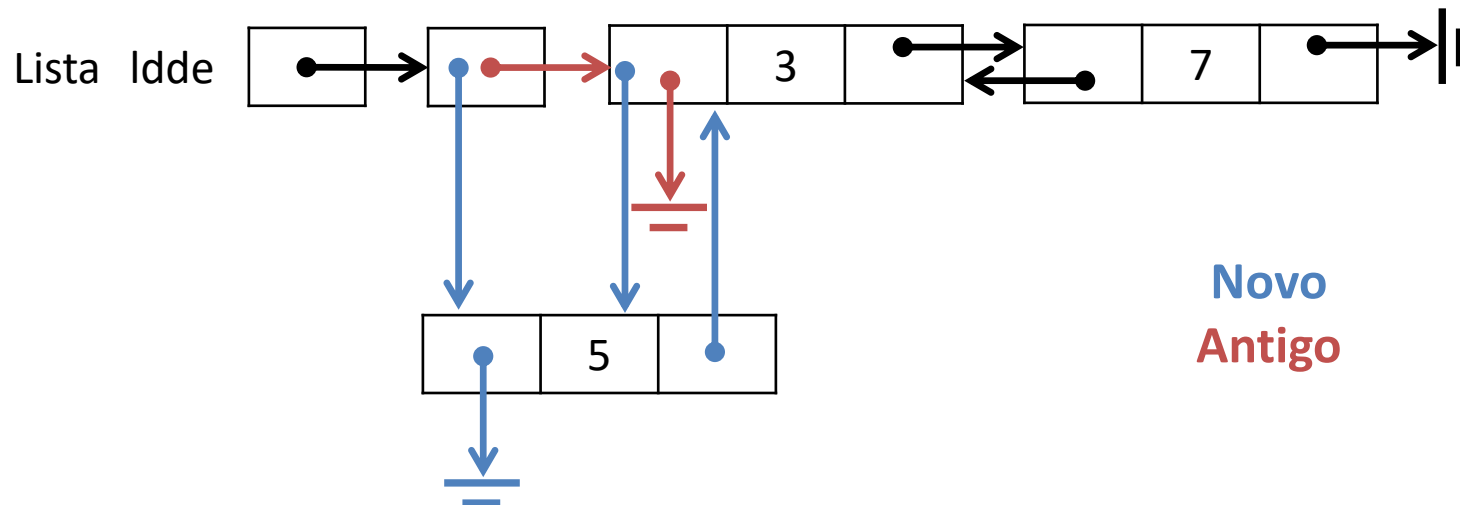
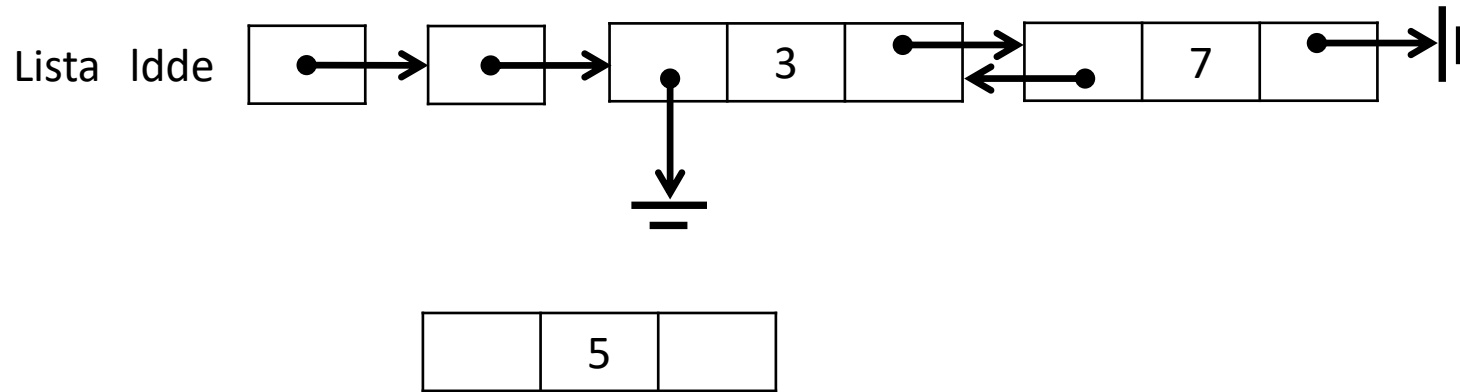
- Assim, temos
  - listaDuplamenteEncadeada.h
    - Estrutura que será usada na Lista
    - Definição do tipo de dado
    - Protótipos das funções
  - listaDuplamenteEncadeada.c
    - Estrutura da Lista em si
    - Definição de um tipo auxiliar Elemento
    - Implementação das funções
  - main.c
    - Uso prático da Lista

# Lista Dinâmica Duplamente Encadeada

- Sendo as Funções Básicas
  - Criar a lista ✓
  - **Inserir um elemento**
  - **Excluir um elemento**
  - Acessar um elemento ✓
  - Destruir a lista ✓
- Vamos ver apenas Inserção e Remoção, pois diferente da Lista Simplesmente Encadeada

# Lista Dinâmica Duplamente Encadeada

- Exemplo de Inserção no Início



# Lista Dinâmica Duplamente Encadeada

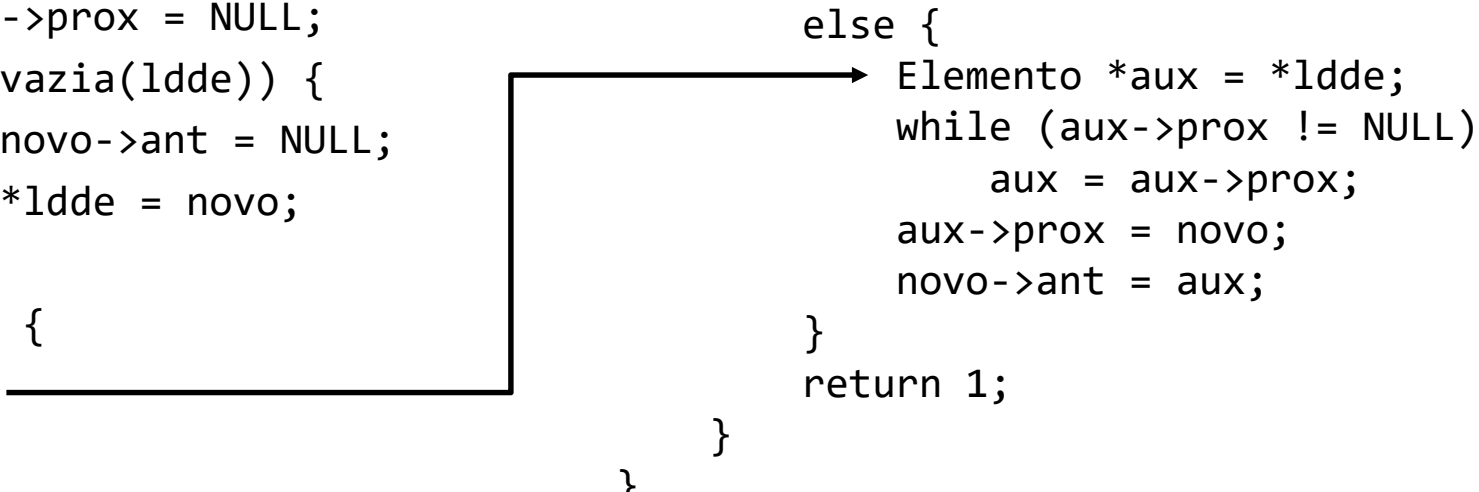
## ● listaDuplamenteEncadeada.c

```
int inserirInicio(Lista *ldde, struct aluno novosdados) {
    if (ldde == NULL) {
        return 0;
    }
    else {
        Elemento *novo = (Elemento*)malloc(sizeof(Elemento));
        if (novo == NULL) return 0;
        novo->ant = NULL;
        novo->dados = novosdados;
        novo->prox = *ldde;
        if (*ldde != NULL) {
            (*ldde)->ant = novo;
        }
        *ldde = novo;
        return 1;
    }
}
```

# Lista Dinâmica Duplamente Encadeada

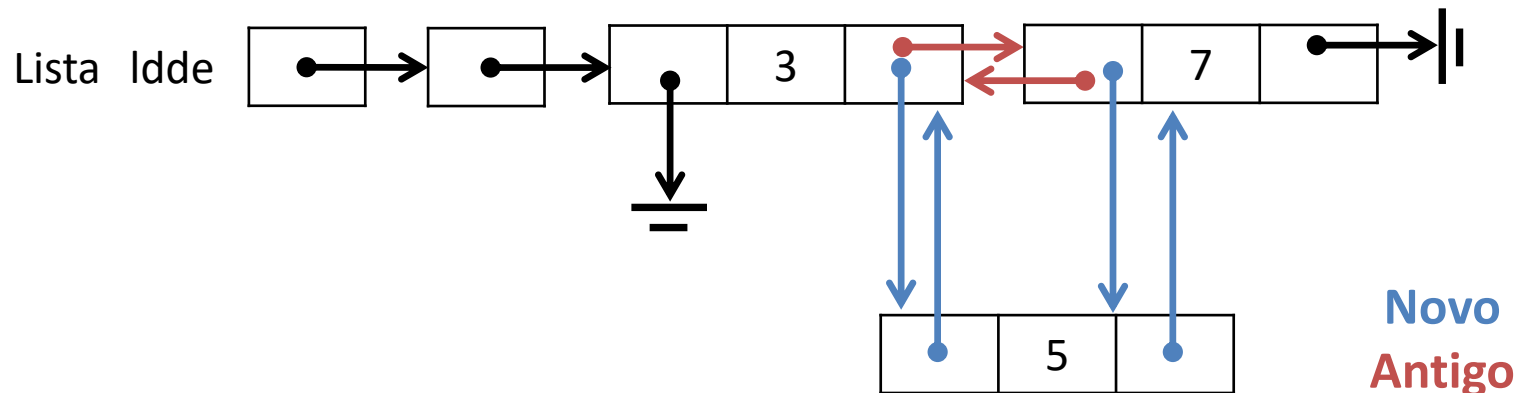
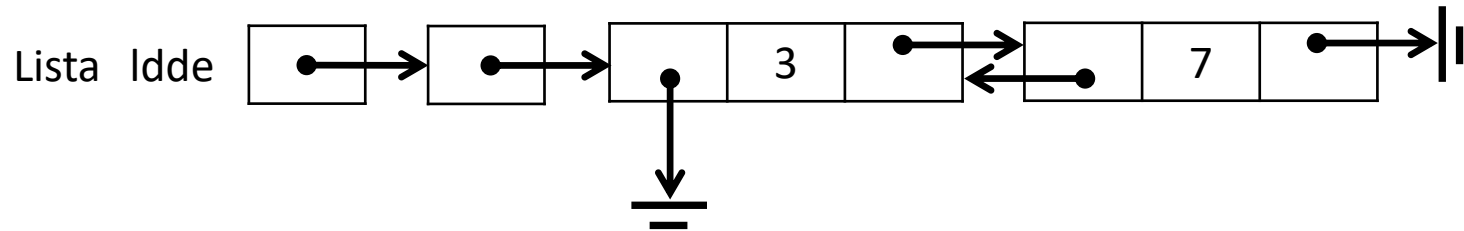
## ● listaDuplamenteEncadeada.c

```
int inserirFim(Lista *ldde, struct aluno novosdados) {
    if (ldde == NULL) {
        return 0;
    }
    else {
        Elemento *novo = (Elemento*)malloc(sizeof(Elemento));
        if (novo == NULL) return 0;
        novo->dados = novosdados;
        novo->prox = NULL;
        if (vazia(ldde)) {
            novo->ant = NULL;
            *ldde = novo;
        }
        else {
            Elemento *aux = *ldde;
            while (aux->prox != NULL)
                aux = aux->prox;
            aux->prox = novo;
            novo->ant = aux;
        }
        return 1;
    }
}
```



# Lista Dinâmica Duplamente Encadeada

- Exemplo de Inserção no Meio (ordenada)





# Lista Dinâmica Duplamente Encadeada

## ● listaDuplamenteEncadeada.c

```
int inserirOrdenado(Lista *ldde, struct aluno novosdados) {
    if (ldde == NULL) {
        return 0;
    }
    Elemento *novo = (Elemento*)malloc(sizeof(Elemento));
    if (novo == NULL) return 0;
    novo->dados = novosdados;
    if (vazia(ldde) || (*ldde)->dados.matricula > novosdados.matricula) {
        novo->ant = NULL;
        novo->prox = *ldde;
        if (*ldde != NULL) {
            (*ldde)->ant = novo;
        }
        *ldde = novo;
        return 1;
    }
}
```

# Lista Dinâmica Duplamente Encadeada

- listaDuplamenteEncadeada.c

```
else {
    Elemento *ant = *ldde;
    Elemento *aux = ant->prox;
    while (aux!=NULL && aux->dados.matricula < novosdados.matricula){
        ant = aux;
        aux = aux->prox;
    }
    ant->prox = novo;
    if (aux != NULL) {
        aux->ant = novo;
    }
    novo->ant = ant;
    novo->prox = aux;
    return 1;
}
```

# Lista Dinâmica Duplamente Encadeada

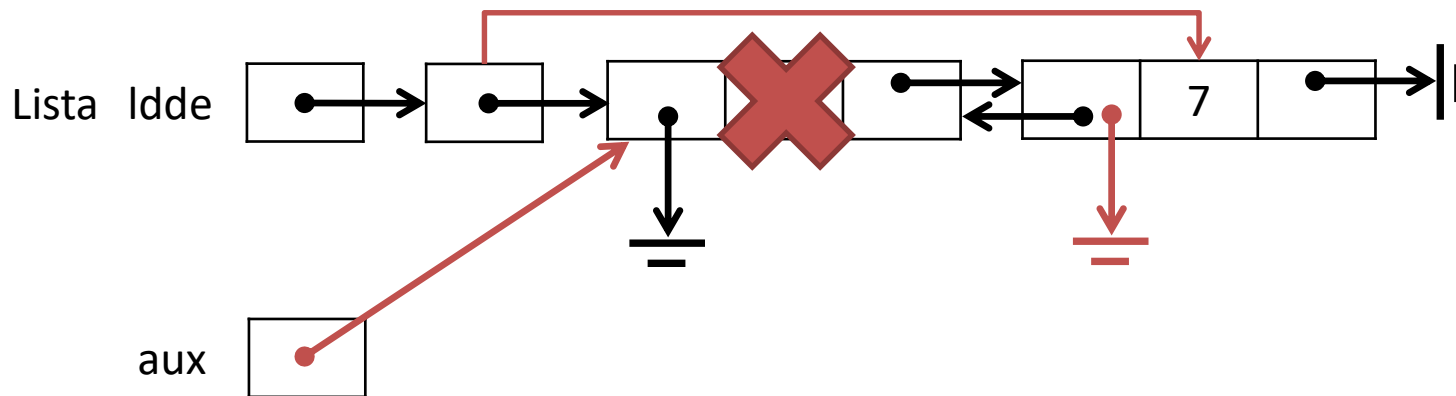
## ● listaDuplamenteEncadeada.c

```
int removerInicio(Lista *ldde) {  
    if (vazia(ldde)) {  
        return 0;  
    }  
    else {  
        Elemento *aux = *ldde;  
        *ldde = aux->prox;  
        if (aux->prox != NULL) {  
            aux->prox->ant = NULL;  
        }  
        free(aux);  
        return 1;  
    }  
}
```

# Lista Dinâmica Duplamente Encadeada

## ● Exemplo de Remoção no Início

```
Elemento *aux = *ldde;  
*ldde = aux->prox;  
if (aux->prox != NULL)  
    aux->prox->ant = NULL;  
free(aux);
```



# Lista Dinâmica Duplamente Encadeada

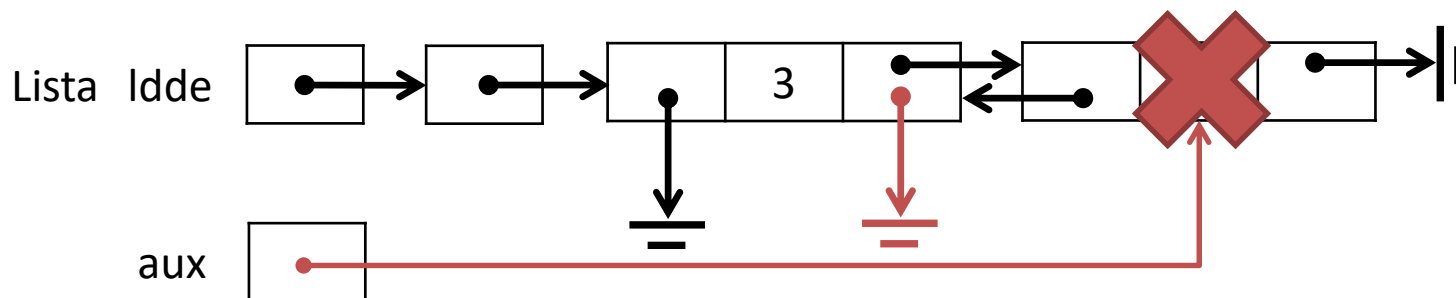
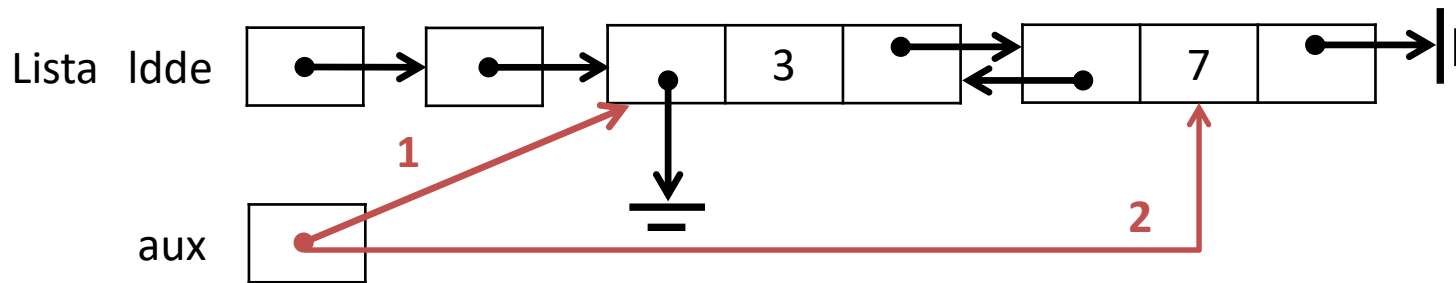
## ● listaDuplamenteEncadeada.c

```
int removerFim(Lista *ldde) {  
    if (vazia(ldde))  
        return 0;  
    else if ((*ldde)->prox == NULL) {  
        Elemento *aux = *ldde;  
        *ldde = NULL;  
        free(aux);  
        return 1;  
    }  
    else {  
        Elemento *aux = *ldde;  
        while (aux->prox != NULL)  
            aux = aux->prox;  
        aux->ant->prox = NULL;  
        free(aux);  
        return 1;  
    }  
}
```

# Lista Dinâmica Duplamente Encadeada

## ● Exemplo de Remoção no Fim

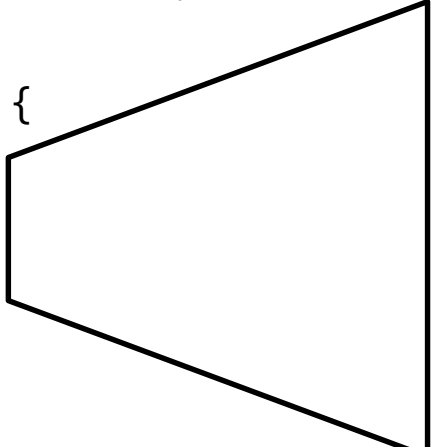
```
Elemento *aux = *ldde;  
while (aux->prox != NULL)  
    aux = aux->prox;  
aux->ant->prox = NULL;  
free(aux);
```



# Lista Dinâmica Duplamente Encadeada

## ● listaDuplamenteEncadeada.c

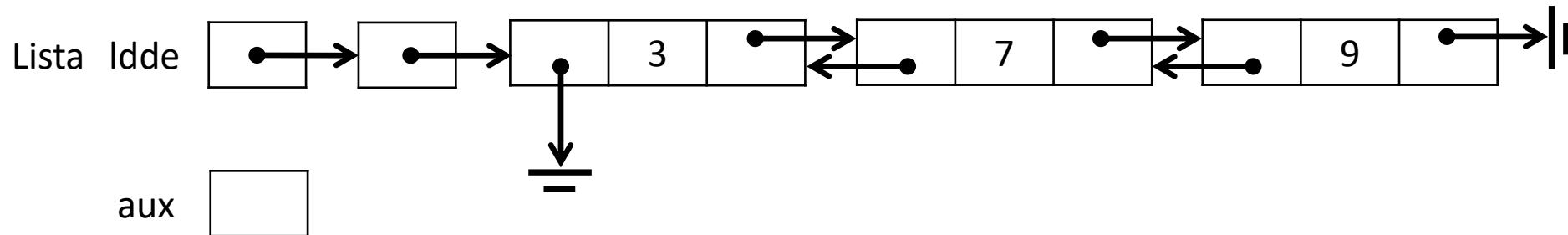
```
int removerValor(Lista *ldde, int x) {  
    if (vazia(ldde)) {  
        return 0;  
    }  
    else if ((*ldde)->dados.matricula == x) {  
        Elemento *aux = *ldde;  
        *ldde = aux->prox;  
        free(aux);  
        return 1;  
    }  
    else {  
        Elemento *aux = *ldde;  
        while (aux != NULL && aux->dados.matricula != x)  
            aux = aux->prox;  
        if (aux == NULL) return 0;  
        aux->ant->prox = aux->prox;  
        if (aux->prox != NULL)  
            aux->prox->ant = aux->ant;  
        free(aux);  
        return 1;  
    }  
}
```



# Lista Dinâmica Duplamente Encadeada

- Exemplo de Remoção de elemento específico
  - Início

```
if ((*ldde)->dados.matricula == x) {  
    Elemento *aux = *ldde;  
    *ldde = aux->prox;  
    free(aux);  
}
```

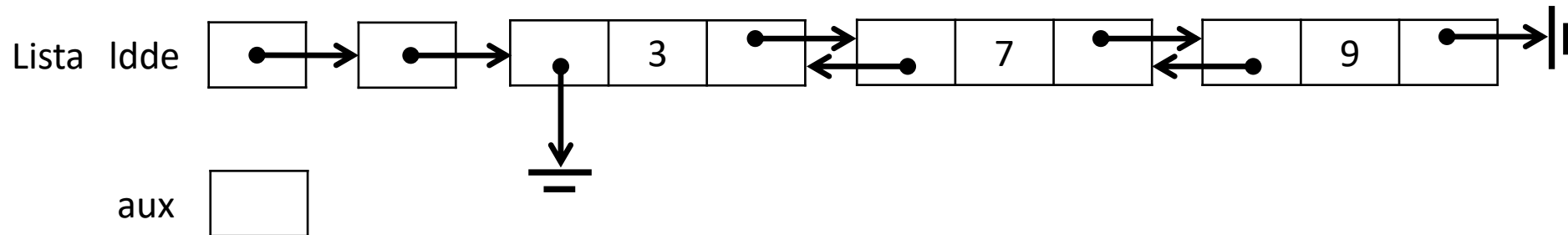




# Lista Dinâmica Duplamente Encadeada

- Exemplo de Remoção de elemento específico
  - Meio

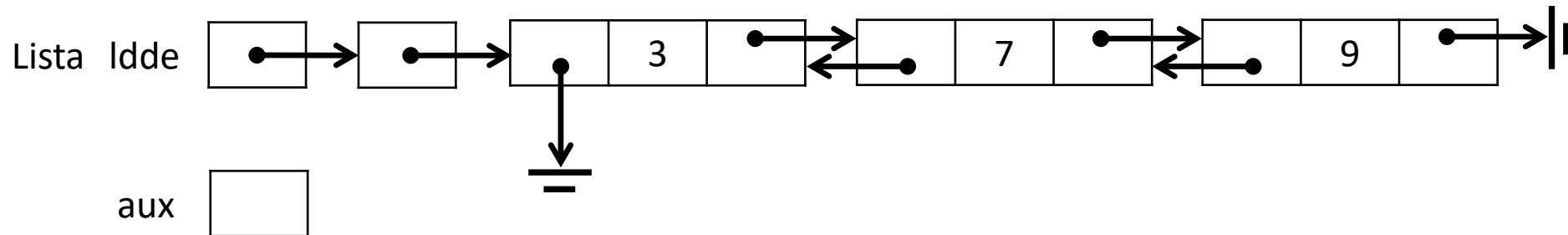
```
Elemento *aux = *ldde;  
while (aux != NULL && aux->dados.matricula != x)  
    aux = aux->prox;  
if (aux == NULL) return 0;  
aux->ant->prox = aux->prox;  
if (aux->prox != NULL)  
    aux->prox->ant = aux->ant;  
free(aux);
```



# Lista Dinâmica Duplamente Encadeada

- Exemplo de Remoção de elemento específico
  - Fim

```
Elemento *aux = *ldde;  
while (aux != NULL && aux->dados.matricula != x)  
    aux = aux->prox;  
if (aux == NULL) return 0;  
aux->ant->prox = aux->prox;  
if (aux->prox != NULL)  
    aux->prox->ant = aux->ant;  
free(aux);
```



# Lista Dinâmica Duplamente Encadeada

- Exercício em Sala: Pegue o mesmo programa que você fez para Lista Estática, que usou também na Lista Dinâmica Simplesmente Encadeada e coloque-o na Lista Dinâmica Duplamente Encadeada para testar seu código!

# Lista Dinâmica Duplamente Encadeada

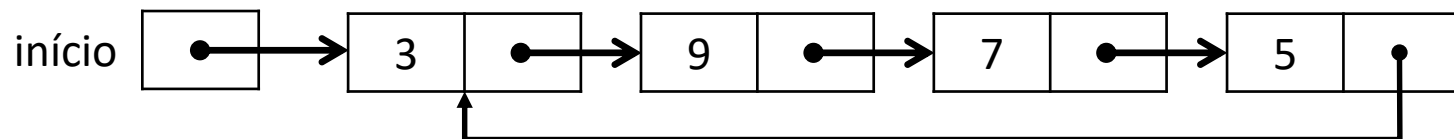
- Concluindo... Quando devo usar?
  - Listas ordenadas
  - Tamanho indefinido a priori
  - Operação mais frequente for acesso inserção e remoção (ordenada)
  - **Necessidade de acessar informação de um elemento antecessor**
- Quantos mais itens desses existirem na sua aplicação, melhor

# Lista Dinâmica Circular

- Lista no qual o sucessor de um elemento ocupa uma posição de memória acessada por um ponteiro no elemento atual e que o último elemento aponta para o primeiro elemento da lista
- Possui as mesmas Vantagens e Desvantagens da Lista Dinâmica Simplesmente Encadeada, comparativamente temos
- Vantagens
  - Capacidade percorrer a lista repetidas vezes
  - Não precisa considerar casos especiais de inserção e remoção no final
    - Já que todo elemento aponta pro próximo, mesmo o último
- Desvantagens
  - Mais difícil de implementar
  - Lista não possui um final definido

# Lista Dinâmica Circular

- Faremos a implementação passo-a-passo de acordo com as arquivos usando TAD e as funções básicas necessárias
- Arquivos
  - `main.c`
  - `listaCircular.c`
  - `listaCircular.h`
- Funções Básicas
  - Criar a lista
  - Inserir um elemento
  - Excluir um elemento
  - Acessar um elemento
  - Destruir a lista



# Lista Dinâmica Circular

## ● listaCircular.h

```
struct aluno {
    int matricula;
    char nome[50];
    float av1;
    float av2;
    float pr;
};

typedef struct elemento *Lista;

Lista* criar();
void destruir(Lista *);
int tamanho(Lista *);
int cheia(Lista *);
int vazia(Lista *);
int inserirFim(Lista *, struct aluno);
int inserirInicio(Lista *, struct aluno);
int inserirOrdenado(Lista *, struct aluno);
int removerFim(Lista *);
int removerInicio(Lista *);
int removerValor(Lista *, int);
int acessarIndice(Lista *, int, struct aluno *);
int acessarValor(Lista *, int, struct aluno *);
```

Apenas algumas funções (em azul) são idênticas às implementadas na Lista Dinâmica Simplesmente Encadeada! Como agora o último elemento da Lista não é mais definido pelo NULL, precisamos alterar todas as outras...

# Lista Dinâmica Circular

## ● listaCircular.c

```
#include <stdlib.h>
#include "listaCircular.h"

struct elemento {
    struct aluno dados;
    struct elemento *prox;
};

typedef struct elemento Elemento;

...
```

Igual a Lista Simplesmente  
Encadeada, tirando a  
implementação das funções  
citadas anteriormente

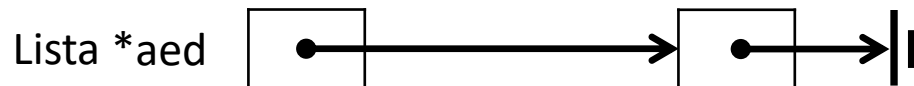


# Lista Dinâmica Circular

## ● main.c

```
#include <stdio.h>
#include <stdlib.h>
#include "listaCircular.h"

int main() {
    Lista *aed;
    aed = NULL;
    aed = criar(); ← Criação de uma lista vazia!
    return 0;
}
```



struct elemento \*\*  
(Lista \*)

struct elemento \*  
(Lista)

# Lista Dinâmica Circular

- Assim, temos
  - listaCircular.h
    - Estrutura que será usada na Lista
    - Definição do tipo de dado
    - Protótipos das funções
  - listaCircular.c
    - Estrutura da Lista em si
    - Definição de um tipo auxiliar Elemento
    - Implementação das funções
  - main.c
    - Uso prático da Lista

# Lista Dinâmica Circular

- Sendo as Funções Básicas
  - Criar a lista ✓
  - Inserir um elemento
  - Excluir um elemento
  - Acessar um elemento
  - Destruir a lista
- Criar é igual à Lista Simplesmente Encadeada
- Vamos fazer as outras...

# Lista Dinâmica Circular

- listaCircular.c

```
void destruir(Lista *lc) {
```

```
    if (lc != NULL && *lc != NULL) {
```

```
        Elemento *aux = *lc;
```

```
        Elemento *backup = *lc;
```

```
        while ((*lc)->prox != backup) {
```

```
            aux = *lc;
```

```
            *lc = (*lc)->prox;
```

```
            free(aux);
```

```
        }
```

```
        free(aux);
```

```
        *lc = NULL;
```

```
        //free(lc);
```

```
    }
```

Se eu não testar, dá erro. Pois, no *while*, eu já tento acessar o *prox* do primeiro elemento!

Não posso perder o começo da Lista

Eu verifico se cheguei no final da Lista quando o próximo aponta para o primeiro elemento da Lista!

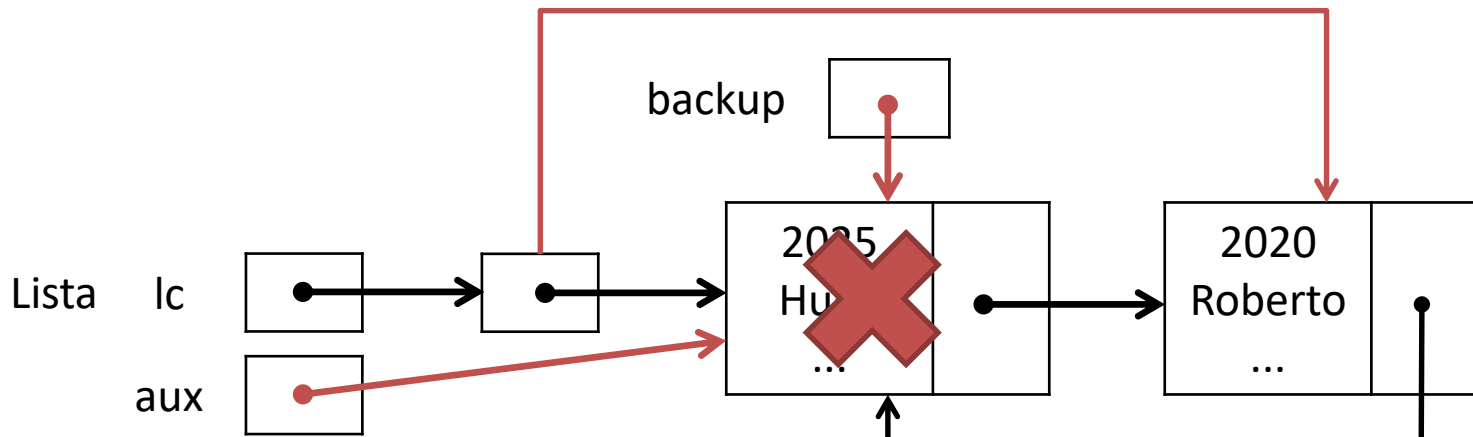
Para remover o último elemento que sobrou

Trechos modificados com relação à Lista Simplesmente Encadeada

# Lista Dinâmica Circular

- Vamos executar a função destruir()!

```
while ((*lc)->prox != backup) {  
    aux = *lc;  
    *lc = (*lc)->prox;  
    free(aux);  
}  
free(aux);
```



# Lista Dinâmica Circular

- Sendo as Funções Básicas
  - Criar a lista ✓
  - Inserir um elemento
  - Excluir um elemento
  - Acessar um elemento
  - Destruir a lista ✓
- Uma funções extras antes de seguir com as funções básicas...

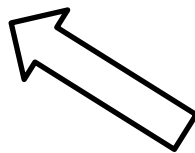
# Lista Dinâmica Circular

## ● listaCircular.c

```
int tamanho(Lista *lc) {  
    if (vazia(lc)) {  
        return 0;  
    }  
    int cont = 0;  
    Elemento *aux = *lc;  
    do {  
        cont++;  
        aux = aux->prox;  
    } while (aux != *lc);  
    return cont;  
}
```

```
int cheia(Lista *lc) {  
    return 0;  
}
```

```
int vazia(Lista *lc) {  
    if (lc == NULL) {  
        return 1;  
    }  
    else if (*lc == NULL) {  
        return 1;  
    }  
    else {  
        return 0;  
    }  
}
```



*do-while* no lugar do *while* para conseguir entrar de certeza a primeira vez, mesmo que a Lista tenha apenas um elemento!

# Lista Dinâmica Circular

- Sendo as Funções Básicas
  - Criar a lista ✓
  - **Inserir um elemento**
  - Excluir um elemento
  - Acessar um elemento
  - Destruir a lista ✓
- Vamos inserir...



# Lista Dinâmica Circular

## ● listaCircular.c

```
int inserirInicio(Lista *lc, struct aluno novosdados) {  
    if (lc == NULL) {  
        return 0;  
    }  
    else {  
        Elemento *novo = (Elemento*)malloc(sizeof(Elemento));  
        if (novo == NULL) return 0;  
        novo->dados = novosdados;  
        if (*lc == NULL) {  
            novo->prox = novo;  
        }  
        else {  
            novo->prox = *lc;  
            Elemento *aux = *lc;  
            while (aux->prox != *lc)  
                aux = aux->prox;  
            aux->prox = novo;  
        }  
        *lc = novo;  
        return 1;  
    }  
}
```

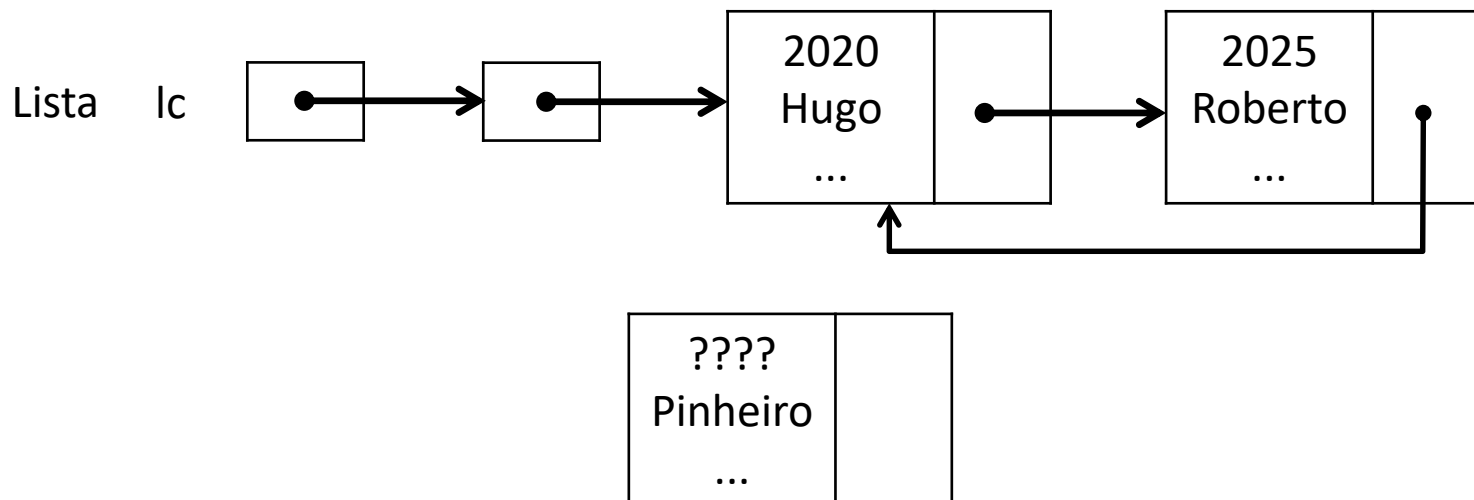
# Lista Dinâmica Circular

## ● listaCircular.c

```
int inserirFim(Lista *lc, struct aluno novosdados) {  
    if (lc == NULL) {  
        return 0;  
    }  
    else {  
        Elemento *novo = (Elemento*)malloc(sizeof(Elemento));  
        if (novo == NULL) return 0;  
        novo->dados = novosdados;  
        if (vazia(lc)) {  
            *lc = novo;  
            novo->prox = novo;  
        }  
        else {  
            novo->prox = *lc;  
            Elemento *aux = *lc;  
            while (aux->prox != *lc) aux = aux->prox;  
            aux->prox = novo;  
        }  
        return 1;  
    }  
}
```

# Lista Dinâmica Circular

- Exercício em Sala: Inserir ordenado
  - Temos três casos
    - Se a Lista estiver vazia
    - Se for entrar no começo da Lista
    - Se for entrar em qualquer outro local
      - No meio ou no fim é a mesma ideia, pois a Lista é circular!



# Lista Dinâmica Circular

## ● listaCircular.c

```
int inserirOrdenado(Lista *lc, struct aluno novosdados) {
    if (lc == NULL) {
        return 0;
    }
    else {
        Elemento *novo = (Elemento*)malloc(sizeof(Elemento));
        if (novo == NULL) return 0;
        novo->dados = novosdados;
        if (vazia(lc)) {
            novo->prox = novo;
            *lc = novo;
        }
        else if ((*lc)->dados.matricula > novo->dados.matricula) {
            novo->prox = *lc;
            Elemento *aux = *lc;
            while (aux->prox != *lc)
                aux = aux->prox;
            aux->prox = novo;
            *lc = novo;
        }
    }
}
```

# Lista Dinâmica Circular

## ● listaCircular.c

```
    else {
        Elemento *ant = *lc;
        Elemento *aux = ant->prox;
        while (aux != *lc && aux->dados.matricula < novo->dados.matricula) {
            ant = aux;
            aux = aux->prox;
        }
        ant->prox = novo;
        novo->prox = aux;
    }
    return 1;
}
```

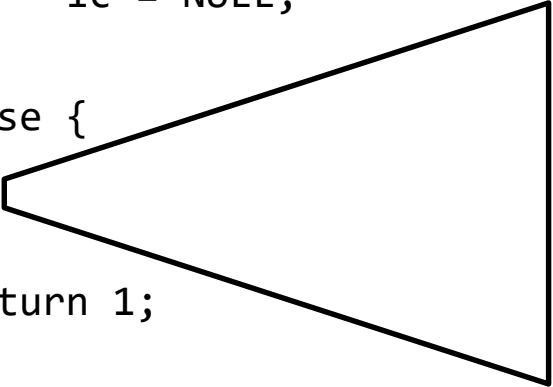
# Lista Dinâmica Circular

- Funções Básicas
  - Criar a lista ✓
  - Inserir um elemento ✓
  - **Excluir um elemento**
  - Acessar um elemento
  - Destruir a lista ✓

# Lista Dinâmica Circular

## ● listaCircular.c

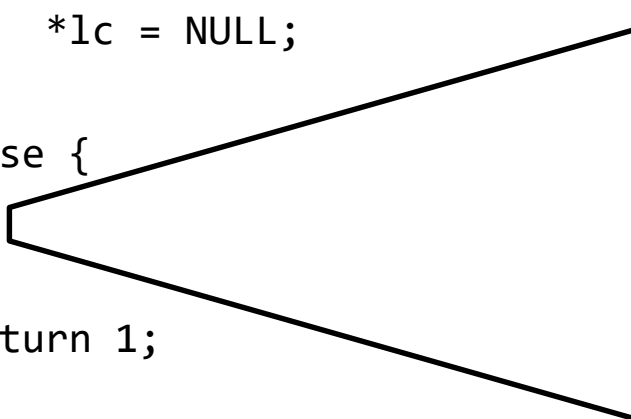
```
int removerInicio(Lista *lc) {  
    if (vazia(lc))  
        return 0;  
    else {  
        if (*lc == (*lc)->prox) {  
            free(*lc);  
            *lc = NULL;  
        }  
        else {  
            return 1;  
        }  
    }  
}
```



```
Elemento *aux = *lc, *backup = *lc;  
*lc = aux->prox;  
free(aux);  
aux = *lc;  
while (aux->prox != backup)  
    aux = aux->prox;  
aux->prox = *lc;
```

# Lista Dinâmica Circular

## ● listaCircular.c

```
int removerFim(Lista *lc) {  
    if (vazia(lc)) {  
        return 0;  
    }  
    else if (*lc == (*lc)->prox){  
        free(*lc);  
        *lc = NULL;  
    }  
    else {  
          
    }  
    return 1;  
}
```

```
Elemento *ant = *lc;  
Elemento *aux = ant->prox;  
while (aux->prox != *lc) {  
    ant = aux;  
    aux = aux->prox;  
}  
ant->prox = aux->prox;  
free(aux);
```



# Lista Dinâmica Circular

- listaCircular.c

```
int removerValor(Lista *lc, int x) {
    if (vazia(lc)) {
        return 0;
    }
    else if ((*lc)->dados.matricula == x){
        ●—————→ if (*lc == (*lc)->prox) {
                        free(*lc);
                        *lc = NULL;
                    }
        else {
            Elemento *aux = *lc
            Elemento *backup = *lc;
            *lc = aux->prox;
            free(aux);
            aux = *lc;
            while (aux->prox != backup)
                aux = aux->prox;
            aux->prox = *lc;
        }
    }
    return 1;
}
```

# Lista Dinâmica Circular

## ● listaCircular.c

```
int removerValor(Lista *lc, int x) {  
    if (vazia(lc)) {  
        return 0;  
    }  
    else if ((*lc)->dados.matricula == x){  
  
        }  
    else {  
        ●————→ Elemento *ant = *lc;  
        Elemento *aux = ant->prox;  
        while (aux != *lc && aux->dados.matricula != x) {  
            ant = aux;  
            aux = aux->prox;  
        }  
        if (aux == *lc) return 0;  
        ant->prox = aux->prox;  
        free(aux);  
    }  
    return 1;  
}
```

# Lista Dinâmica Circular

- Funções Básicas
  - Criar a lista ✓
  - Inserir um elemento ✓
  - Excluir um elemento ✓
  - **Acessar um elemento**
  - Destruir a lista ✓
- Podemos acessar um elemento pela sua posição ou pelo seu valor
  - Ambos os casos requer uma busca na Lista

# Lista Dinâmica Circular

## ● listaCircular.c

```
int acessarIndice(Lista *lc, int pos, struct aluno *a) {
    if (vazia(lc) || pos < 0)
        return 0;
    else if (pos == 0)
        *a = (*lc)->dados;
    else {
        int cont = 0;
        Elemento *aux = *lc;
        do {
            aux = aux->prox;
            cont++;
        } while (aux != *lc && pos != cont);
        if (aux == *lc) return 0;
        *a = aux->dados;
    }
    return 1;
}
```

# Lista Dinâmica Circular

## ● listaCircular.c

```
int acessarValor(Lista *lc, int x, struct aluno *a) {
    if (vazia(lc))
        return 0;
    else if ((*lc)->dados.matricula == x)
        *a = (*lc)->dados;
    else {
        Elemento *aux = *lc;
        do {
            aux = aux->prox;
        } while (aux != *lc && aux->dados.matricula != x);
        if (aux == *lc) return 0;
        *a = aux->dados;
    }
    return 1;
}
```

# Lista Dinâmica Circular

- Funções Básicas
  - Criar a lista ✓
  - Inserir um elemento ✓
  - Excluir um elemento ✓
  - Acessar um elemento ✓
  - Destruir a lista ✓
- Novamente... Basta usar o mesmo `main()` para testar!

# Lista Dinâmica Circular

- Concluindo... Quando devo usar?
  - Listas ordenadas
  - Tamanho indefinido a priori
  - Operação mais frequente for acesso inserção e remoção (ordenada)
  - **Quando deseja-se voltar ao primeiro item da lista depois de varrer toda a lista**
- Quanto mais itens desses existirem na sua aplicação, melhor