

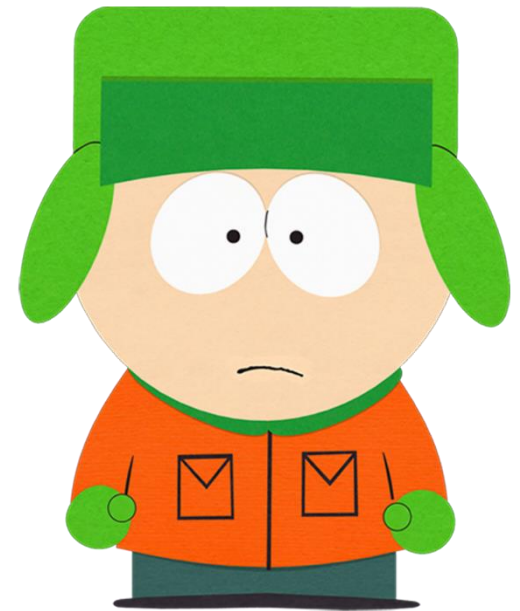
# Estruturas de Dados 3

## Alocação dinâmica e Listas

Mailson de Queiroz Proença

# Alocação de Memória

- Existem dois tipos de alocação de memória:
  - Estática
  - Dinâmica



# Alocação de Memória

- Variáveis são alocadas na memória do computador;
- As variáveis declaradas pelos nossos programas, são alocadas de acordo com seus tamanhos e tipos;
- Obviamente, cada variável tem um tamanho definido. Conforme a tabela abaixo:

Tipo	Tamanho em Bytes	Faixa Mínima
char	1	-127 a 127
int	4	-2.147.483.648 a 2.147.483.647
float	4	6 dígitos de precisão
double	8	15 dígitos de precisão

# Alocação de Memória

## Estática:

- Alocação estática é controlada pelo compilador;
- Não é possível alterar o espaço da alocação de memória;
- Na alocação estática, a área de memória ocupada por ela se mantém constante durante toda a execução.

# Alocação de Memória Estática

- Para o nosso exemplo, vamos efetuar a alocação estática, simplesmente declarando uma variável.

```
int valor;
```

# Alocação de Memória

- **Dinâmica:**

- Alocação dinâmica é controlada pelo desenvolvedor;
- A memória pode ser desalocada durante o decorrer do programa;
- O tamanho da alocação é sob demanda, ou seja, em tempo de execução;
- Vamos utilizar alocação dinâmica de memória principalmente com struct;
- Não se esqueça: um computador tem memória limitada;
- **Não se esqueça de desalocar a memória da variável quando não for mais necessário o uso.**

# Alocação de Memória Dinâmica

## Alocando memória

- Para o nosso exemplo, vamos efetuar a alocação dinâmica de memória utilizando a palavra reservada **new**.

```
alunos *novo_aluno = new alunos;
```



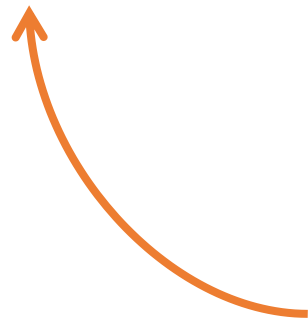
*Efetutando a alocação de memória*

# Alocação de Memória Dinâmica

## Desacolando memória

- Para o nosso exemplo, vamos efetuar a desalocação dinâmica de memória utilizando a palavra reservada delete.

```
delete lista_alunos;
```



*Efetuando a desalocação de memória*



**LISTAS**

# Listas

Uma Lista é uma estrutura que permite representar um conjunto de dados de forma a preservar a relação de ordem existente entre eles.

- Exemplo de aplicações de listas:
  - Lista de Compra
  - Notas de alunos
  - Cadastro de funcionários de uma empresa
  - Itens em estoque em uma empresa
  - Letras de uma palavra
  - Cartas de baralho

# Tipos de Listas

As listas podem ser classificadas da seguinte maneira:

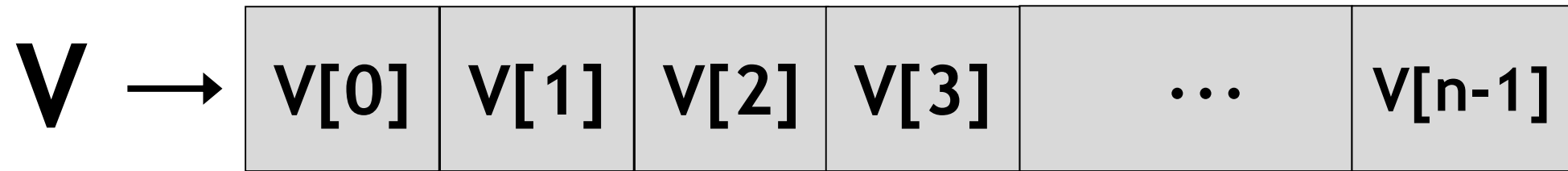
- Listas Estáticas (Vetores)
- Listas Dinâmicas (Listas Encadeadas)

# Listas Estáticas: Vetores

- Estruturas que armazenam uma quantidade fixa de elementos do mesmo tipo;
- Alocação estática de memória;
- Também chamada de Lista Sequencial;
- Nós em posições contíguas de memória;
- Geralmente representado por **vetores**.

# Listas Estáticas: Vetores

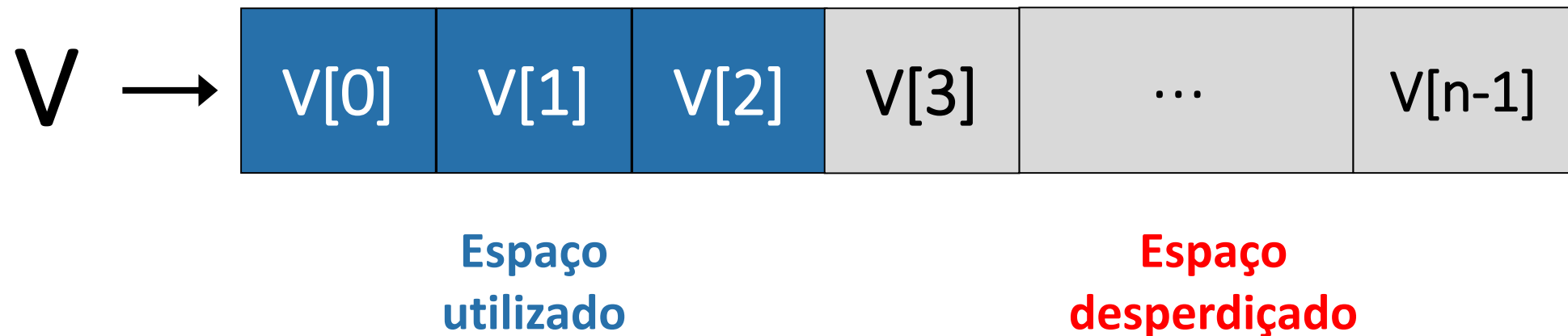
O acesso a um elemento é feito a partir do índice do elemento desejado.



Vetores não podem armazenar mais elementos do que o seu tamanho, logo, o tamanho deve ser o máximo necessário.

# Listas Estáticas: Vetores

Quando a quantidade de elementos é variável, o uso de vetores pode desperdiçar memória, já que nem todas as suas posições são necessariamente ocupadas.

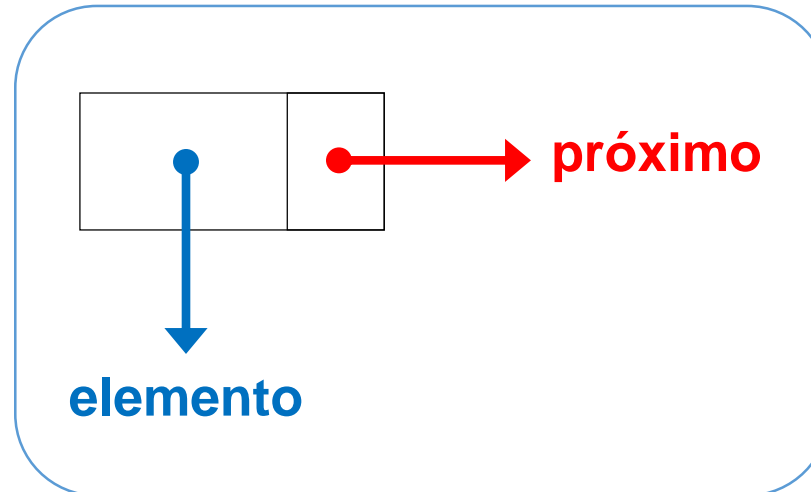


# Listas Dinâmicas

- O tamanho e a capacidade **variam** de acordo com a demanda, a medida que o programa vai sendo executado.
- As posições de memória são alocadas a medida que são necessárias.
- As células encontram-se **aleatoriamente** dispostas na memória e são interligadas por ponteiros, que indicam onde encontra-se a próxima célula.
- Também chamadas de **Listas Encadeadas**.

# Listas Encadeadas

- Uma lista encadeada é uma estrutura de dados dinâmica formada por um conjunto de **Células**.
- Cada célula armazena:
  - Um ou mais elementos (int, float, char, string, ...);
  - Uma ligação para a próxima célula.



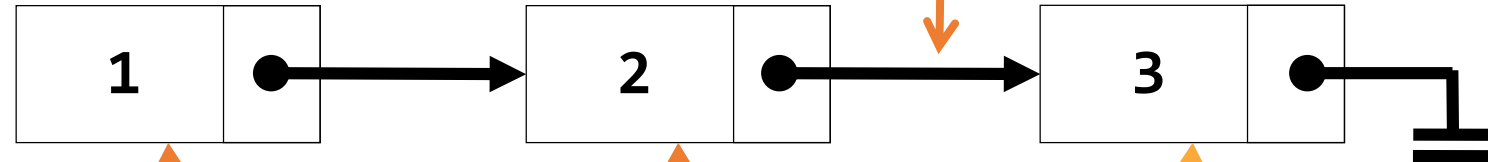
Estrutura da célula



# Listas Encadeadas

Um elemento de uma lista, possui o endereço do próximo. O último elemento da lista tem o valor **NULL**.

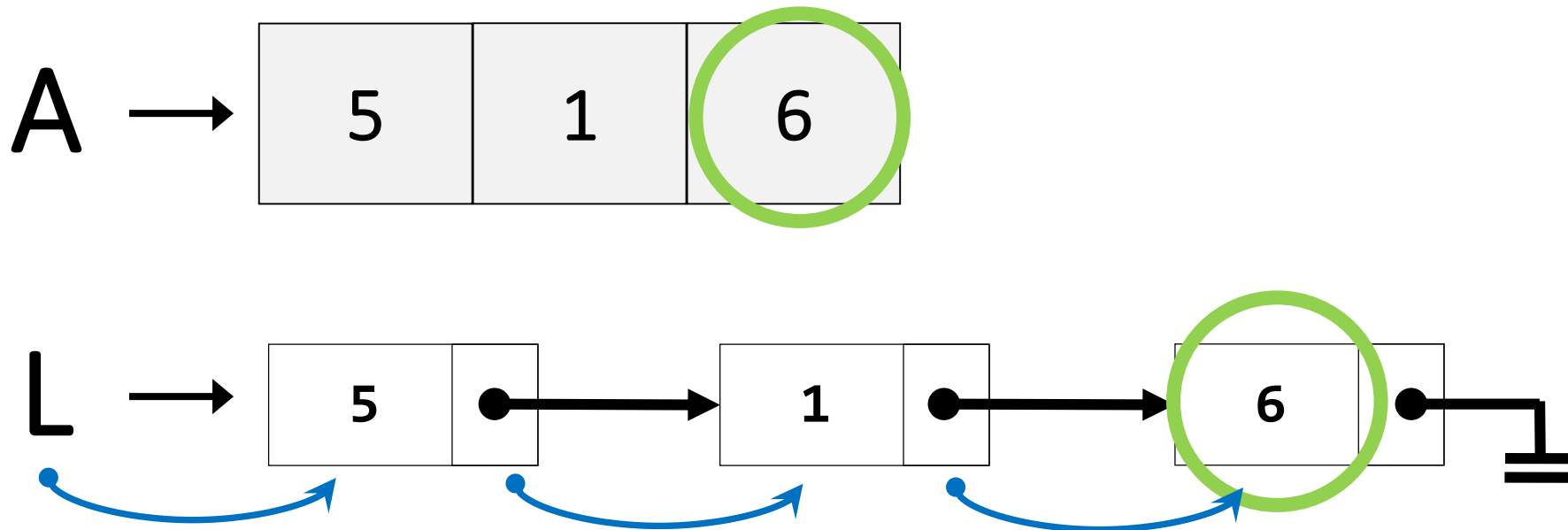
*Endereço de memória para o próximo elemento*



*Elementos da lista ligada simples*

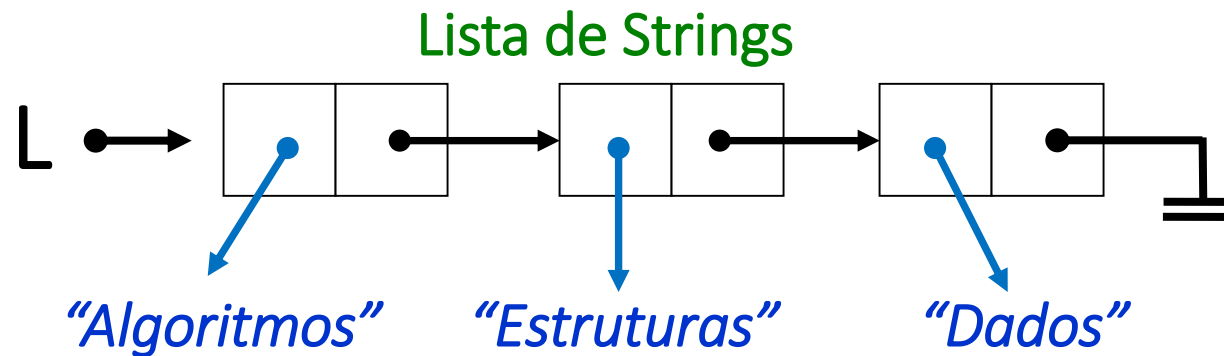
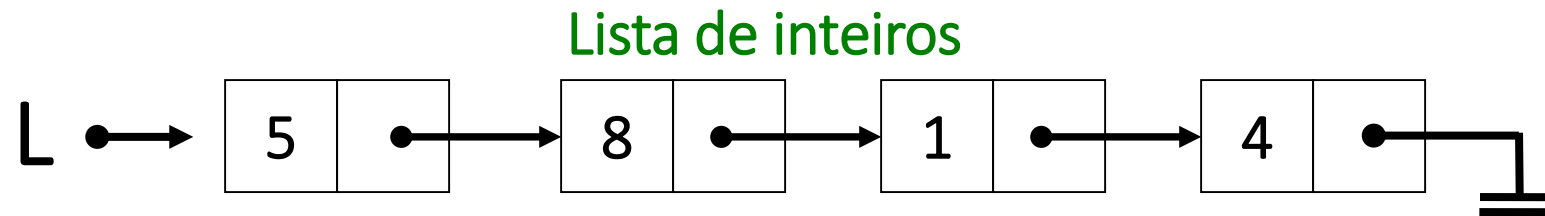
# Vetor x Listas Encadeadas

Ao contrário de um vetor, uma lista não pode acessar seus elementos de modo direto, e sim, de modo sequencial, ou seja, um por vez.



# Listas Simplesmente Encadeadas

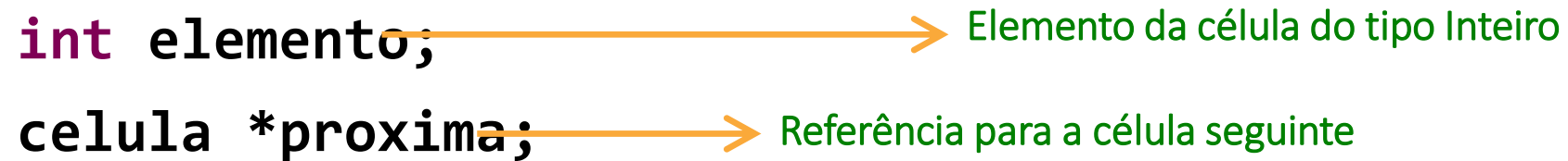
São Listas formadas por células que possuem **uma referência** para outra estrutura do mesmo tipo.



# Listas Simplesmente Encadeadas

Ex.: Definição de um célula para uma Lista de inteiros

```
struct celula
{
    int elemento;
    celula *proxima;
};
```



Elemento da célula do tipo Inteiro

Referência para a célula seguinte

```
celula *lista;
```



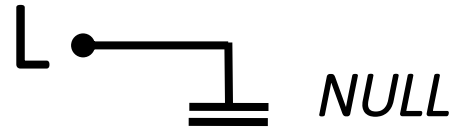
Ponteiro que indica o endereço de Memória da Lista

# Listas Simplesmente Encadeadas

## Principais operações:

- Criar uma estrutura de lista;
- Verificar se a lista está vazia;
- Listar todas as Células;
- Inserir uma Célula no início da lista;
- Inserir uma Célula no final da lista;
- Remover uma Célula;
- Inserir ordenado na lista;
- Esvaziar a lista.

# Criar uma Estrutura de Lista



```
celula *CriarLista(){  
    return NULL; // Lista Inicia Vazia  
}
```

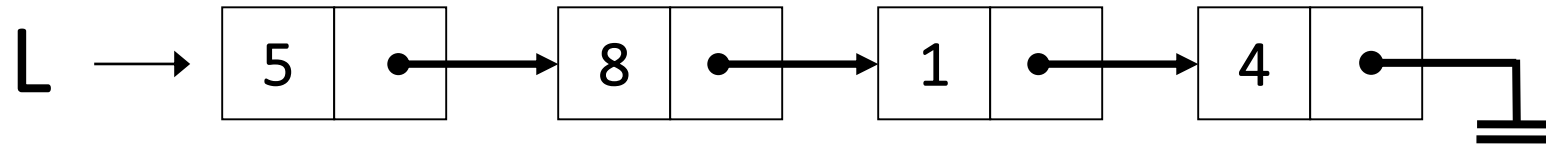
Obs.: O fim de uma lista é indicada por uma referência nula, ou seja, a última célula de uma lista aponta para nulo (elemento **NULL**).

# Verificar se a lista está vazia

```
bool ListaVazia(celula *lista)
{
    if(lista == NULL)
        return true;
    else return false;
}
```

Se a Lista (variável “lista”) for nula então a lista está vazia, ou seja, a lista não possui nenhuma célula.

# Imprimir todos os elementos de uma Lista

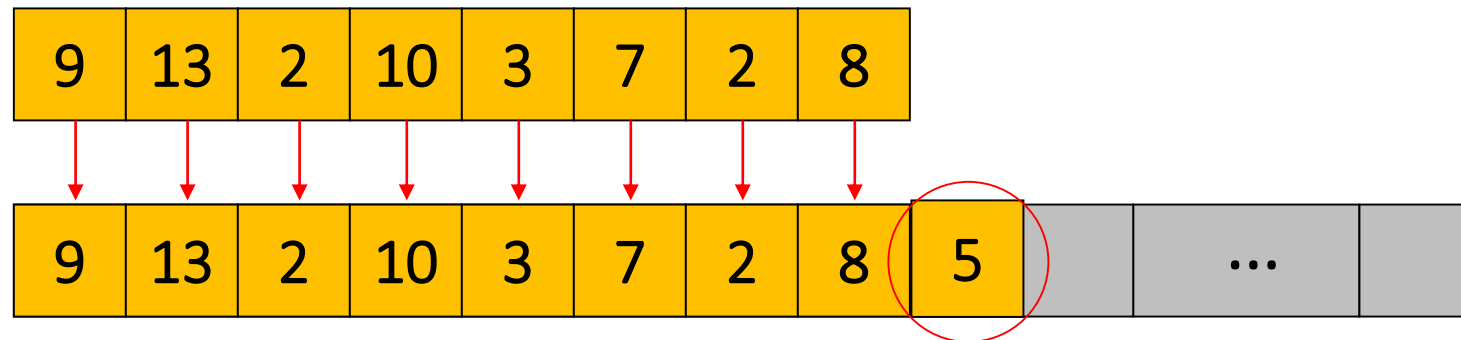


```
void ImprimirLista(celula *lista)
{
    celula *aux = lista;
    if(ListaVazia(lista))
        cout << "Lista vazia";
    else
    {
        while(aux != NULL)
        {
            cout << aux->elemento << endl;
            aux = aux->proximo;
        }
    }
}
```

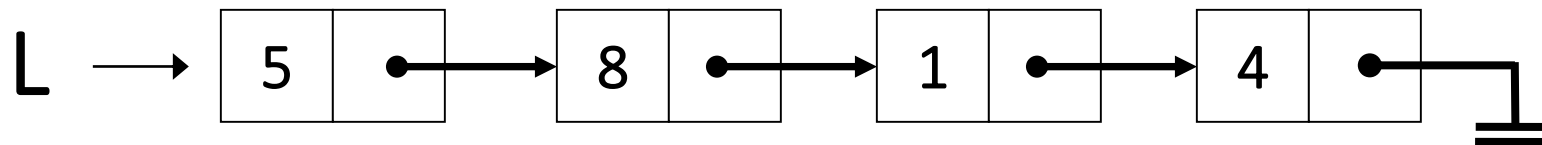


# Inserir uma Célula em uma Lista

Para inserir um elemento em um array, pode ser necessário expandi-lo e copiar os elementos um a um:

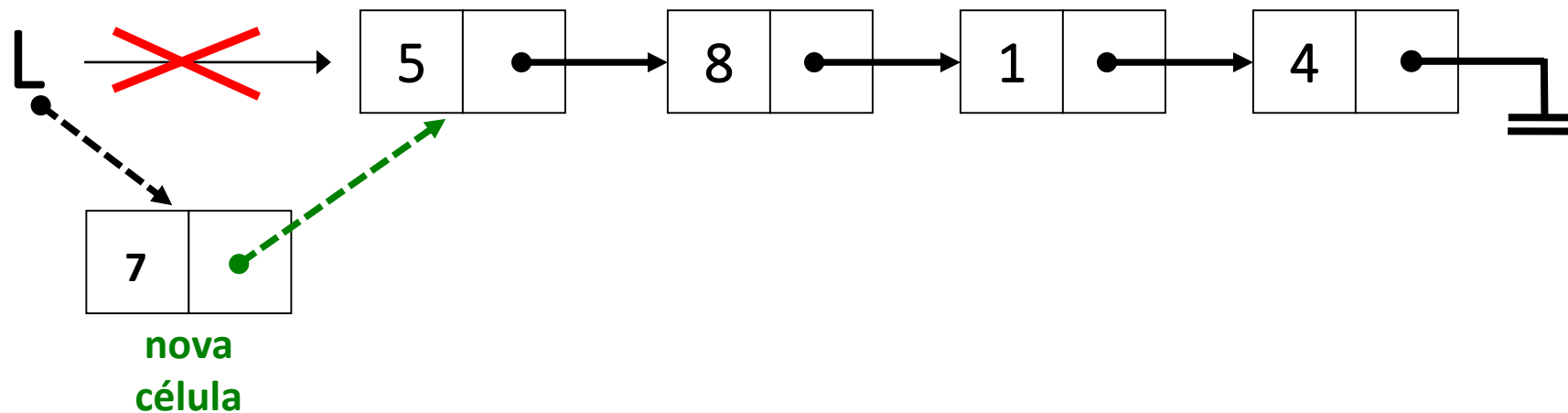


Em uma lista, basta criar um nó, encontrar a posição desejada e inseri-lo.

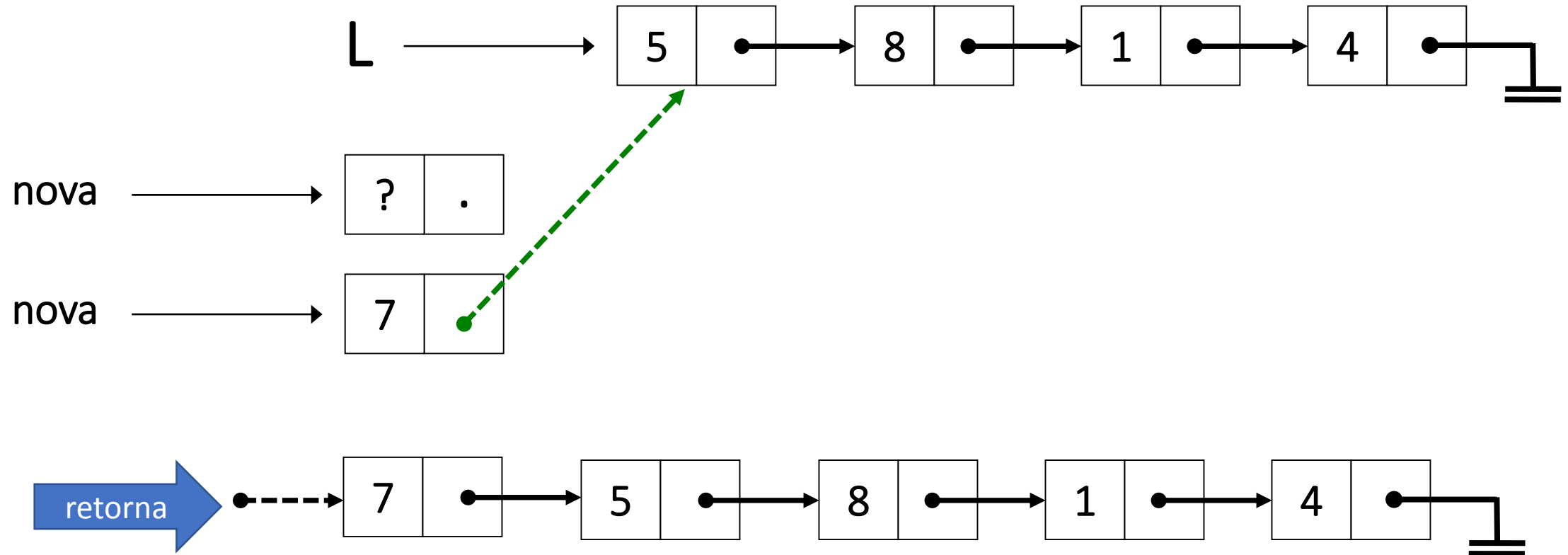


# Inserir uma Célula no início de uma Lista

- Aloca espaço para a nova Célula;
- Define os valores dos elementos da Célula;
- Nova Célula aponta para a primeira Célula da Lista;
- Retorna o ponteiro para a nova primeira Célula da Lista.



# Inserir um Nó no Início de uma Lista



# Inserir um Nó no Início de uma Lista

```
celula *InserirCelula(celula *lista, int elemento)
{
    celula *nova = new celula;
    nova->elemento = elemento;
    nova->proximo = lista;
    return nova;
}
```

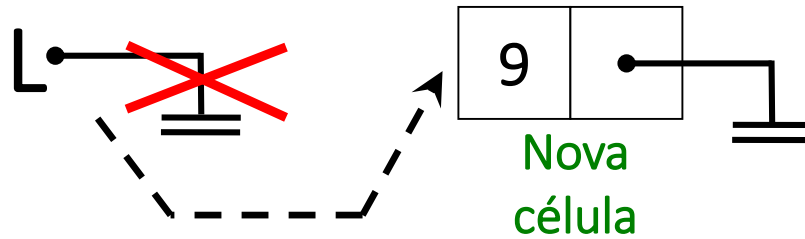
1. Aloca espaço para o novo Nó
2. Define os valores dos elementos do Nó
3. Aponta para o primeiro Nó da Lista
4. Retorna o ponteiro para o novo primeiro Nó da Lista

# Testando a Lista

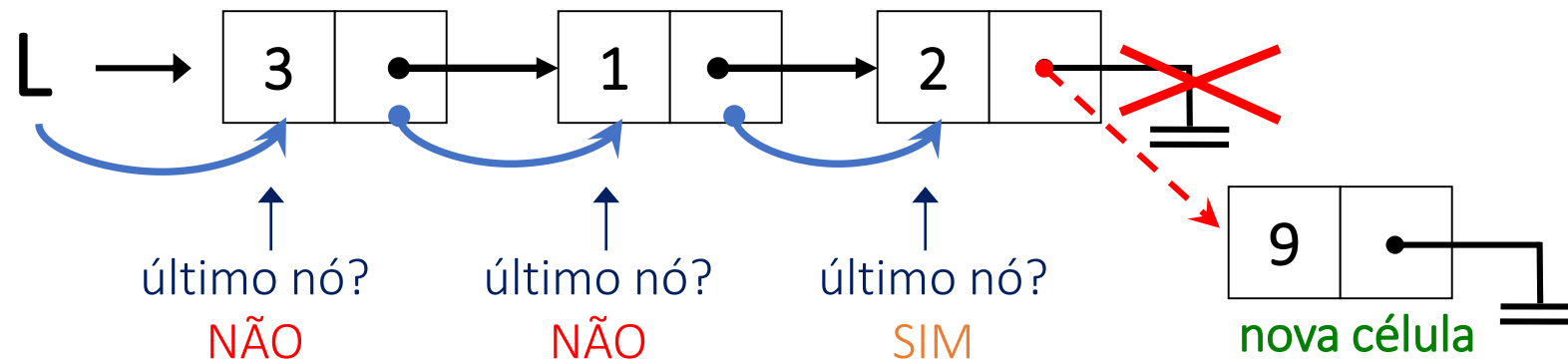
```
int main()
{
    celula *lista; //declara lista não inicializada
    lista = CriarLista(); //cria e inicia lista vazia
    lista = InserirCelula(lista, 20); //insere o nro 20
    lista = InserirCelula(lista, 15); //insere o nro 15
    ImprimirLista(lista);
    return 0;
}
```

# Inserir um Nó no Final de uma Lista

Se a lista estiver vazia:



Caso contrário, inserindo no fim da lista teremos:



# Inserir um Nó no Final de uma Lista

```
celula *InserirNoFim(celula *lista, int valor)
{
    celula *nova = new celula;
    nova->elemento = valor;
    if (ListaVazia(lista)) //lista == NULL
    {
        nova->proxima = lista;
        return nova;
    }
    else
    {
        celula *aux = lista; //procura pelo fim da lista
        while (aux->proxima != NULL)
        {
            aux = aux->proxima;
        }
        aux->proxima = nova;
        nova->proxima = NULL;
        return lista;
    }
}
```

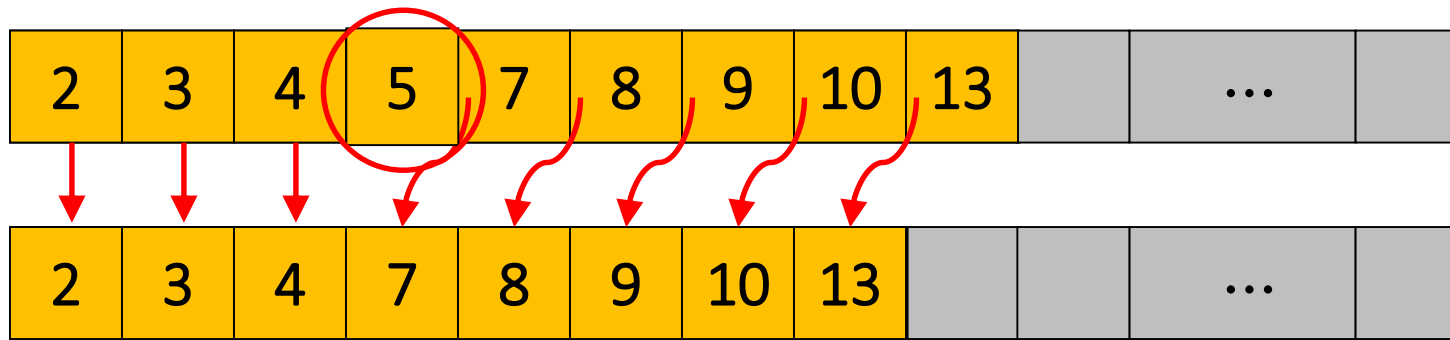
# Testando a Lista

```
int main()
{
    celula *lista; //declara lista não inicializada
    lista = CriarLista(); //cria e inicia lista vazia
    lista = InserirCelula(lista, 20); //insere o nro 20
    lista = InserirCelula(lista, 15); //insere o nro 15
    lista = InserirNoFim(lista, 10); //insere o nro 10
    ImprimirLista(lista);
    return 0;
}
```

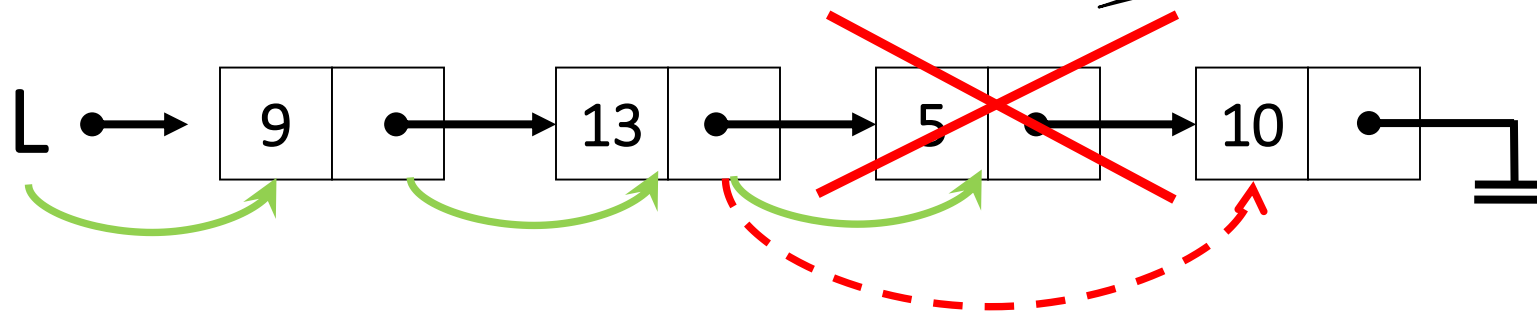


# Remover um Nó de uma Lista

Para remover um elemento de uma posição qualquer do *vetor*, pode ser necessário mover vários elementos:

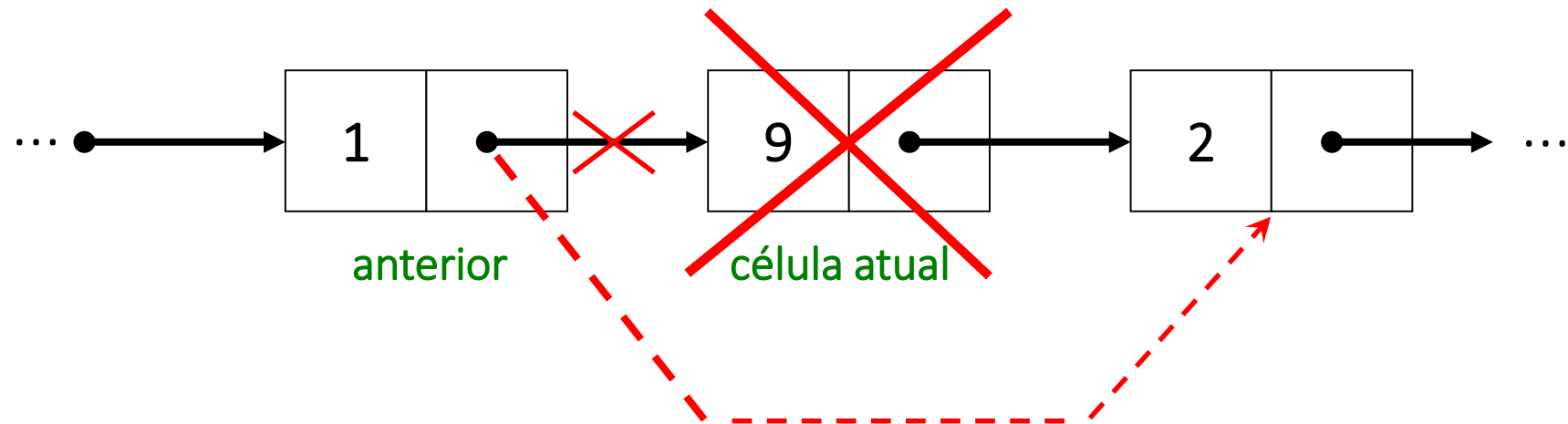


Para remover um elemento de uma lista, basta encontrar o elemento correspondente e alterar os ponteiros



# Remover um Nó de uma Lista

Para excluir uma célula entre duas outras células:



# Remover um Nó de uma Lista

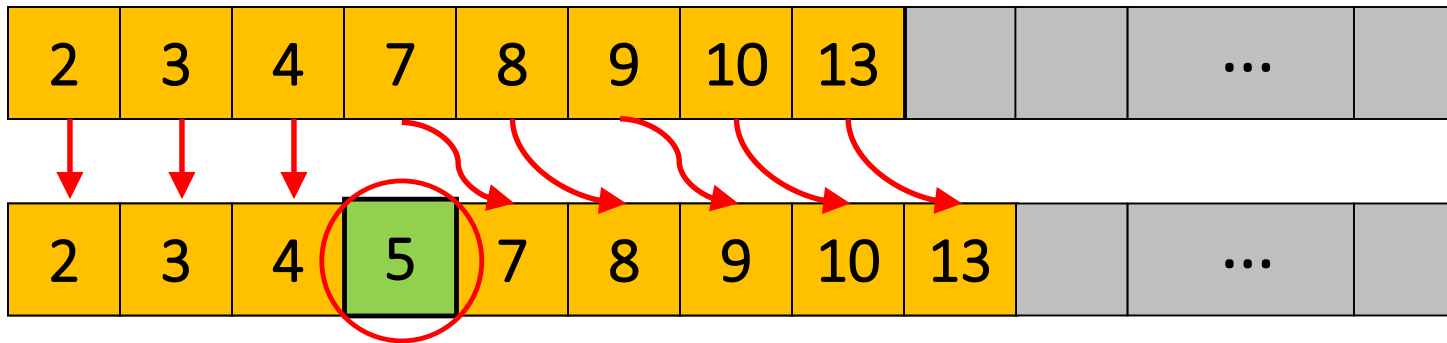
```
celula *RemoverCelula(celula *lista, int valor)
{
    celula *aux = lista; //ponteiro para percorrer a lista
    celula *ant = NULL; // ponteiro para elem anterior
    while (aux != NULL && aux->elemento != valor)
    {
        ant = aux;
        aux = aux->proxima;
    }
    if (aux == NULL)
    {
        cout << "\nElemento nao encontrado.";
        return lista; // retorna lista original
    }
    else if (ant == NULL) //retira elemento do inicio
    {
        lista = aux->proxima;
    }
    else //retira elem do meio ou do final da lista
    {
        ant->proxima = aux->proxima;
    }
    delete aux;
    return lista;
}
```

# Testando a Lista

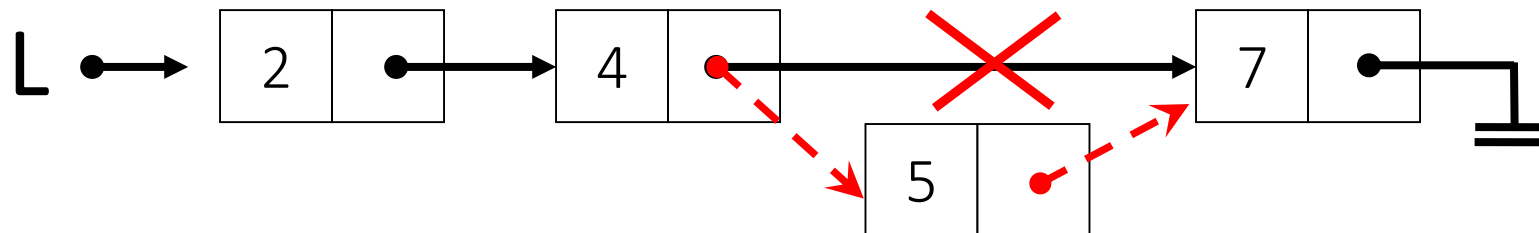
```
int main()
{
    celula *lista; //declara lista não inicializada
    lista = CriarLista(); //cria e inicia lista vazia
    lista = InserirCelula(lista, 20); //insere o nro 20
    lista = InserirCelula(lista, 15); //insere o nro 15
    lista = InserirNoFim(lista, 10); //insere o nro 10
    lista = RemoverCelula(lista, 15); //remove o nro 15
    ImprimirLista(lista);
    return 0;
}
```

# Inserção ordenada em uma Lista (1/4)

Para inserir um elemento num vetor em uma posição qualquer, pode ser necessário mover vários elementos:

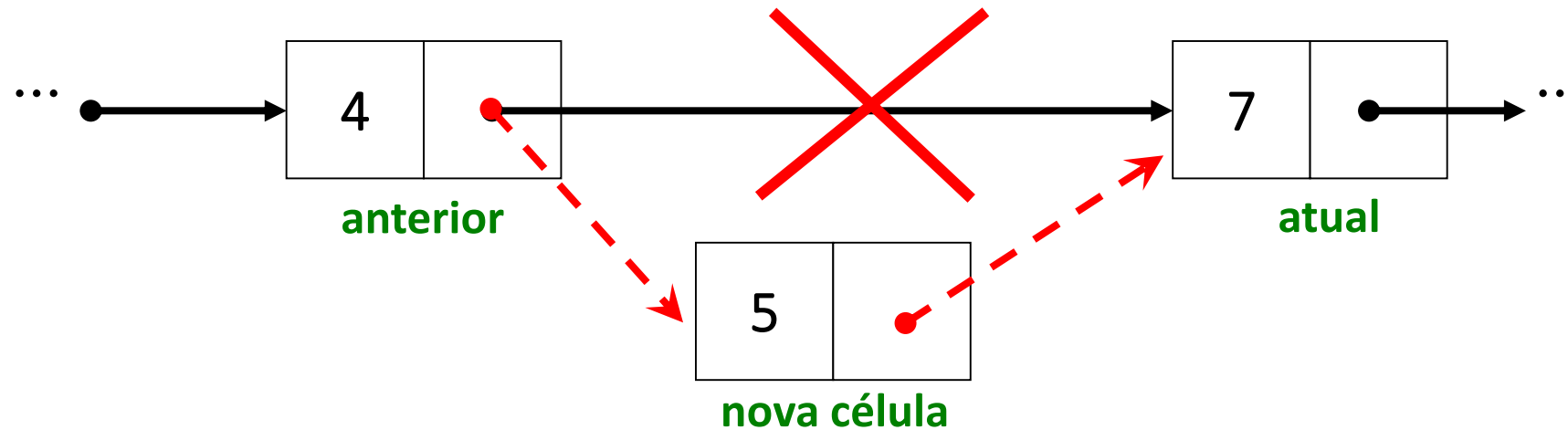


Da mesma maneira, em uma lista, basta criar um nó, encontrar a posição desejada e inseri-lo.



# Inserção ordenada em uma Lista (2/4)

Para inserir uma célula entre duas células:



# Inserção ordenada em uma Lista (3/4)

```
celula *InserirOrdenado(celula *lista, int valor)
{
    celula *nova = new celula;
    nova->elemento = valor;
    celula *aux = lista; //Ponteiro que percorre a lista
    celula *ant = NULL; //Ponteiro para o Nó anterior
    if (aux == NULL) // A lista está vazia
    {
        nova->proxima = NULL;
        lista = nova;
    }
    else // Existe(m) Nó(s) na Lista
    {
        //Percorre a Lista
        while (aux != NULL && nova->elemento > aux->elemento)
        {
            ant = aux;
            aux = aux->proxima;
        }
    }
}
```

# Inserção ordenada em uma Lista (4/4)

```
// O novo nro é menor que todos os nros da Lista
if (ant == NULL) // Insere celula Inicio da Lista
{
    nova->proxima = lista;
    lista = nova;
}
else //Insere célula no Meio ou no Fim da Lista
{
    ant->proxima = nova;
    nova->proxima = aux;
}
}
return lista;
}
```



```

celula *InserirOrdenado(celula *lista, int valor)
{
    celula *nova = new celula;
    nova->elemento = valor;
    celula *aux = lista; //Ponteiro que percorre a lista
    celula *ant = NULL; //Ponteiro para o Nó anterior
    if (aux == NULL) // A lista está vazia
    {
        nova->proxima = NULL;
        lista = nova;
    }
    else // Existe(m) Nó(s) na Lista
    {
        //Percorre a Lista
        while (aux != NULL && nova->elemento > aux->elemento)
        {
            ant = aux;
            aux = aux->proxima;
        }
        // O novo nro é menor que todos os nros da Lista
        if (ant == NULL) // Insere celula Inicio da Lista
        {
            nova->proxima = lista;
            lista = nova;
        }
        else //Insere célula no Meio ou no Fim da Lista
        {
            ant->proxima = nova;
            nova->proxima = aux;
        }
    }
    return lista;
}

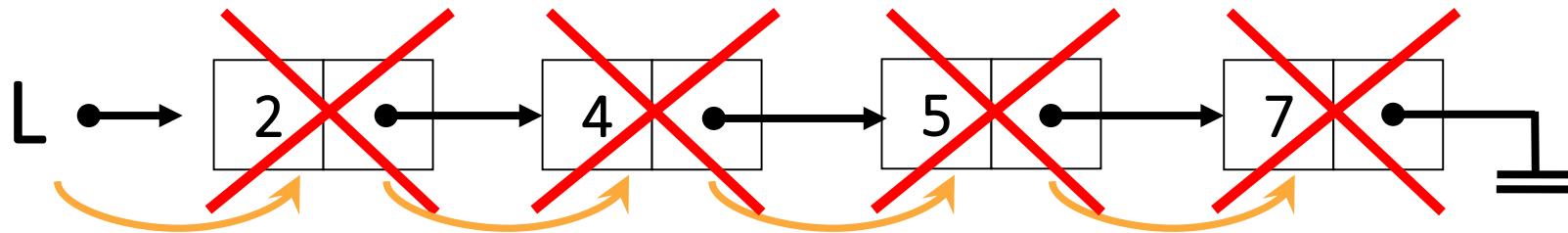
```

# Testando a Lista

```
int main()
{
    celula *lista; //declara lista não inicializada
    lista = CriarLista(); //cria e inicia lista vazia
    lista = InserirOrdenado(lista, 1);
    ImprimirLista(lista);
    lista = InserirOrdenado(lista, 45);
    ImprimirLista(lista);
    lista = InserirOrdenado(lista, 18);
    ImprimirLista(lista);
    return 0;
}
```

# Esvaziar uma Lista

Para esvaziar uma lista em C++ é necessário eliminar cada célula da lista:



# Esvaziar uma Lista

```
celula *EsvaziarLista(celula *lista)
{
    celula *aux = lista; //Ponteiro que percorre a lista
    celula *atual = NULL; //Ponteiro para o Nó que será
removido
    while (aux != NULL) // Percorre a Lista
    {
        atual = aux;
        aux = aux->proxima;
        delete atual;
    }
    return NULL;
}
```

# Testando a Lista

```
int main()
{
    celula *lista; //declara lista não inicializada
    lista = CriarLista(); //cria e inicia lista vazia
    lista = InserirOrdenado(lista, 1);
    ImprimirLista(lista);
    lista = InserirOrdenado(lista, 45);
    ImprimirLista(lista);
    lista = InserirOrdenado(lista, 18);
    ImprimirLista(lista);
    lista = EsvaziarLista(lista);
    ImprimirLista(lista);
    return 0;
}
```

# Exercícios (1/2)

- Crie um menu de opções para acessar cada operação do programa;
- Altere a Função “RemoverCelula” verificando se a Lista não é vazia. Peça para o usuário digitar um valor para remover da Lista;
- Crie uma Função que retorne o maior valor da Lista de Inteiros;
- Crie uma Função que imprima na tela todos os números maiores que o número passado como parâmetro;

## Exercícios (2/2)

- Crie uma Função que receba dois números e faça as seguintes operações:
  - Caso o 1º número existir na lista, troque esse número pelo 2º número
  - Caso o 1º número não existir na lista, informe ao usuário que o número não foi encontrado na lista
- Crie uma Função que insira os números ordenados (em ordem decrescente) na lista.