🔗 Listas Simplesmente Encadeadas em C++

🧩 1. O que é uma Lista Encadeada?

Diferente das **estruturas sequenciais (vetores)**, onde os elementos ficam em posições **contíguas** na memória, uma **lista encadeada** guarda os elementos em **nós separados** que estão **ligados por ponteiros**.

Cada **nó (node)** possui:

- **um valor (chave ou dado)**
- **um ponteiro para o próximo nó**

Representação visual:

[10 | *] → [20 | *] → [30 | *] → NULL

- O ponteiro head aponta para o **primeiro nó da lista**.
- O **último nó** tem seu ponteiro next igual a NULL (indicando o fim da lista).

⚙️ 2. Estrutura da Lista Encadeada em C++

Estrutura de um nó:

```
struct No {
    int valor;
    No* proximo;
};
```

Estrutura da lista:

```
class ListaEncadeada {
private:
    No* head; // aponta para o primeiro nó da lista
public:
    ListaEncadeada() {
        head = nullptr; // lista começa vazia
    }
};
```

🔍 3. Operações Fundamentais

As três principais operações são:

Operação	O que faz	Complexidade
Busca (search)	Procura um valor	O(n)
Inserção (insertOrdered)	Insere na posição correta	O(n)
Remoção (removeOrdered)	Remove um valor existente	O(n)

🟢 3.1 Busca (Search)

Percorremos a lista com um ponteiro (current) até encontrar o valor desejado ou chegar ao final (nullptr).

Exemplo:

```
No* buscar(int x) {
    No* atual = head;
    while (atual != nullptr && atual->valor < x) {
        atual = atual->proximo;
    }
    if (atual != nullptr && atual->valor == x)
        return atual;
    else
        return nullptr;
}
```

📘 **Explicação:**

- O laço percorre os nós enquanto o valor atual for **menor que x**.
- Se encontrar um nó com valor == x, retorna esse nó.
- Se chegar no fim (nullptr), significa que não encontrou.

🟡 3.2 Inserção Ordenada (insertOrdered)

A ideia é manter a lista **ordenada em ordem crescente**.

O novo elemento pode ser inserido:

1. No início (antes do primeiro nó),
2. No meio (entre dois nós),
3. No final da lista.

Exemplo:

```
void inserirOrdenado(int x) {
    No* novo = new No;
    novo->valor = x;
    novo->proximo = nullptr;

    // Caso 1: lista vazia ou novo valor menor que o primeiro
    if (head == nullptr || head->valor >= x) {
        novo->proximo = head;
        head = novo;
        return;
    }

    // Caso 2: inserir no meio ou fim
    No* atual = head;
    while (atual->proximo != nullptr && atual->proximo->valor < x) {
        atual = atual->proximo;
    }

    // Inserção entre "atual" e "atual->proximo"
    novo->proximo = atual->proximo;
    atual->proximo = novo;
}
```

🧠 **Resumo da lógica:**

- Se a lista estiver vazia ou o novo valor for o menor, ele vira o novo head.
- Senão, percorre até achar o ponto certo para inserir.
- Ajusta os ponteiros: o novo aponta para o próximo, e o nó anterior aponta para o novo.

🔴 3.3 Remoção Ordenada (removeOrdered)

Para remover, é preciso:

1. Encontrar o nó que contém o valor desejado (x);
2. Atualizar o ponteiro do nó anterior (prev) para “pular” o nó removido;
3. Liberar a memória com delete.

Exemplo:

```
void remover(int x) {
    if (head == nullptr) {
        cout << "Lista vazia!" << endl;
        return;
    }
```

```
    // Caso 1: o valor está no primeiro nó
    if (head->valor == x) {
        No* temp = head;
        head = head->proximo;
        delete temp;
        return;
    }

    // Caso 2: o valor está no meio ou fim
    No* atual = head;
    No* anterior = nullptr;

    while (atual != nullptr && atual->valor < x) {
        anterior = atual;
        atual = atual->proximo;
    }

    if (atual != nullptr && atual->valor == x) {
        anterior->proximo = atual->proximo;
        delete atual;
    } else {
        cout << "Valor não encontrado!" << endl;
    }
}
```

📊 4. Análise de Complexidade

Todas as operações principais têm **complexidade linear — O(n)**, pois é necessário percorrer a lista **do início até o ponto desejado**.

✅ Porém, **operações no início** (como inserir ou remover o primeiro nó) têm **complexidade O(1)**, pois não exigem percorrer a lista.

💬 5. Vantagens e Desvantagens

Vantagens	**Desvantagens**
Uso dinâmico da memória (não precisa tamanho fixo)	Acesso lento (precisa percorrer)
Inserções e remoções fáceis no início	Busca linear O(n)
Evita desperdícios de espaço	Mais ponteiros → mais uso de memória

🔁 6. Visualizando a Inserção

Suponha que temos a lista:

[10] → [20] → [30] → NULL

E queremos inserir o valor 25.

Passos:

1. Começamos em 10, depois 20 (pois 25 > 20).
2. O próximo é 30, que é maior → inserimos **antes** de 30.

Resultado final:

[10] → [20] → [25] → [30] → NULL