🔁 Listas Duplamente Encadeadas em C++

🧩 1. O que é uma Lista Duplamente Encadeada?

Uma **lista duplamente encadeada** é uma **estrutura de dados dinâmica** em que **cada nó aponta tanto para o próximo quanto para o anterior**.

Essa característica permite **percorrer a lista nos dois sentidos** (para frente e para trás), diferentemente da **lista simplesmente encadeada**, que só pode ser percorrida em uma direção.

Estrutura geral:

NULL ← [A] ⇄ [B] ⇄ [C] → NULL

- Cada **nó** tem:
    - um **valor (key/dado)**,
    - um **ponteiro para o anterior (previous)**,
    - e um **ponteiro para o próximo (next)**.

A lista tem dois ponteiros principais:

- head → aponta para o **primeiro nó**
- tail → aponta para o **último nó**

⚙️ 2. Estrutura em C++

Estrutura de um nó:

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

Classe da lista:
class ListaDupla {
private:
    No* head; // primeiro nó
    No* tail; // último nó
public:
    ListaDupla() {
        head = nullptr;
        tail = nullptr;
    }
```

};

🔍 3. Operações Principais

Assim como nas listas simples, temos as operações:

- **Busca (search)**
- **Inserção ordenada (insertOrdered)**
- **Remoção (removeOrdered)**

A diferença é que agora precisamos **atualizar dois ponteiros (previous e next)** ao mesmo tempo.

🟢 3.1 Busca (Search)

Percorre a lista **a partir do início (head)** até encontrar o valor desejado.

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;
    return nullptr;
}
```

Também é possível fazer **busca reversa**, começando do tail e usando anterior.

🟡 3.2 Inserção Ordenada

Mantém a lista **ordenada em ordem crescente**.

Há quatro casos possíveis:

1. **Lista vazia**
2. **Inserção no início**
3. **Inserção no final**
4. **Inserção no meio**

Exemplo:

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

    // Caso 1: lista vazia
    if (head == nullptr) {
        head = tail = novo;
        return;
    }

    // Caso 2: inserir no início
    if (x < head->valor) {
        novo->proximo = head;
        head->anterior = novo;
        head = novo;
        return;
    }

    // Percorre até achar a posição
    No* atual = head;
    while (atual != nullptr && atual->valor < x) {
        atual = atual->proximo;
    }

    // Caso 3: inserir no final
    if (atual == nullptr) {
        tail->proximo = novo;
        novo->anterior = tail;
        tail = novo;
        return;
    }

    // Caso 4: inserir no meio
    novo->proximo = atual;
    novo->anterior = atual->anterior;
    atual->anterior->proximo = novo;
    atual->anterior = novo;
}
```

Dica: inserir no início ou final é **O(1)**, mas encontrar o ponto certo é **O(n)**.

🔴 3.3 Remoção Ordenada

Três casos principais:

1. Remover o **primeiro nó**
2. Remover o **último nó**
3. Remover um **nó intermediário**

Exemplo:

```
void remover(int x) {
    No* atual = head;
    while (atual != nullptr && atual->valor < x)
        atual = atual->proximo;

    if (atual == nullptr || atual->valor != x) {
        cout << "Valor não encontrado!" << endl;
        return;
    }

    // Caso 1: primeiro nó
    if (atual == head) {
        head = head->proximo;
        if (head != nullptr)
            head->anterior = nullptr;
        else
            tail = nullptr; // lista ficou vazia
    }

    // Caso 2: último nó
    else if (atual == tail) {
        tail = tail->anterior;
        tail->proximo = nullptr;
    }

    // Caso 3: nó do meio
    else {
        atual->anterior->proximo = atual->proximo;
        atual->proximo->anterior = atual->anterior;
    }

    delete atual; // libera a memória
}
```

⏩ 4. Vantagens da Lista Duplamente Encadeada

Vantagem	**Explicação**
Percorrimento bidirecional	Pode ser percorrida da esquerda para a direita e vice-versa

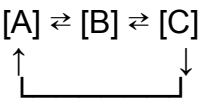
Inserção e remoção rápidas	Basta ajustar ponteiros (next e previous)
Fácil acesso ao início e fim	Ponteiros head e tail dão acesso direto

⚖️ 5. Complexidade

Operação	Complexidade
Busca	O(n)
Inserção	O(n) para achar o local + O(1) para inserir
Remoção	O(n) para achar + O(1) para remover
Inserção no início/fim	O(1)

🔄 6. Lista Duplamente Encadeada Circular

É uma **variação** da lista duplamente encadeada comum, mas com **ligação contínua**:



- O **último nó** (tail) aponta para o **primeiro** (head)
- O **primeiro nó** aponta para o **último**

Características:

✅ **Percorrimento infinito** — não há “fim”

✅ **Sem ponteiros nulos** — reduz erros

✅ **Inserções e remoções mais simples** — não é preciso verificar nullptr

✅ **Navegação bidirecional contínua**

Exemplo Simplificado:

```
class ListaCircular {
private:
   No* head;
public:
   ListaCircular() { head = nullptr; }

   void inserir(int x) {
```

```
        No* novo = new No;
        novo->valor = x;

        if (head == nullptr) {
            head = novo;
            head->proximo = head;
            head->anterior = head;
        } else {
            No* tail = head->anterior;
            tail->proximo = novo;
            novo->anterior = tail;
            novo->proximo = head;
            head->anterior = novo;
        }
    }
};
```

🧠 7. Aplicações Práticas

Estrutura	Baseada em	Observação
Pilha	Inserção/remoção no mesmo extremo	push e pop → O(1)
FIla	Inserção no final, remoção no início	enqueue e dequeue → O(1)
Deque	Inserção e remoção em ambas extremidades	Muito eficiente para cache e buffers circulares

🗒️ Resumo Final para a Prova

Estrutura	Direção	Ponteiros por nó	Percorrimento	Destaque
Simplesmente Encadeada	Única	1 (next)	Do início ao fim	Estrutura básica
Duplamente Encadeada	Duas	2 (next e previous)	Nos dois sentidos	Mais flexível
Duplamente Encadeada Circular	Duas	2 (sem nullptr)	Circular infinito	Inserções mais fáceis