



ALGORITMOS III

Prof. Ms. Ronan Loschi.

ronan.loschi@unifasar.edu.br

31-98759-9555

LISTAS EM

C++

DEFINIÇÃO:

Uma **Lista Encadeada** é uma estrutura de dados linear composta por nós (elementos) que armazenam um valor e um ponteiro para o próximo nó. Diferente dos arrays, as listas encadeadas **não ocupam posições contíguas na memória**.

Ela é útil quando:

- Precisa de inserções/remoções frequentes em qualquer posição.
- O tamanho da coleção varia com frequência.
- Você quer economizar espaço, já que não há necessidade de definir tamanho fixo.

DEFINIÇÃO:

Em uma **lista encadeada** pode-se acessar elementos da lista de forma aleatória, ou seja, inserir e remover elementos em qualquer parte da lista.

Cada elemento é **um nó** que contém informação sobre o elo de ligação entre cada um da lista.

A característica mais importante deste tipo de estrutura é sua forma dinâmica. Os nós são inseridos ou removidos de acordo com a necessidade, em tempo de execução.

OBJETIVO:

Onde as LISTAS são mais utilizadas?

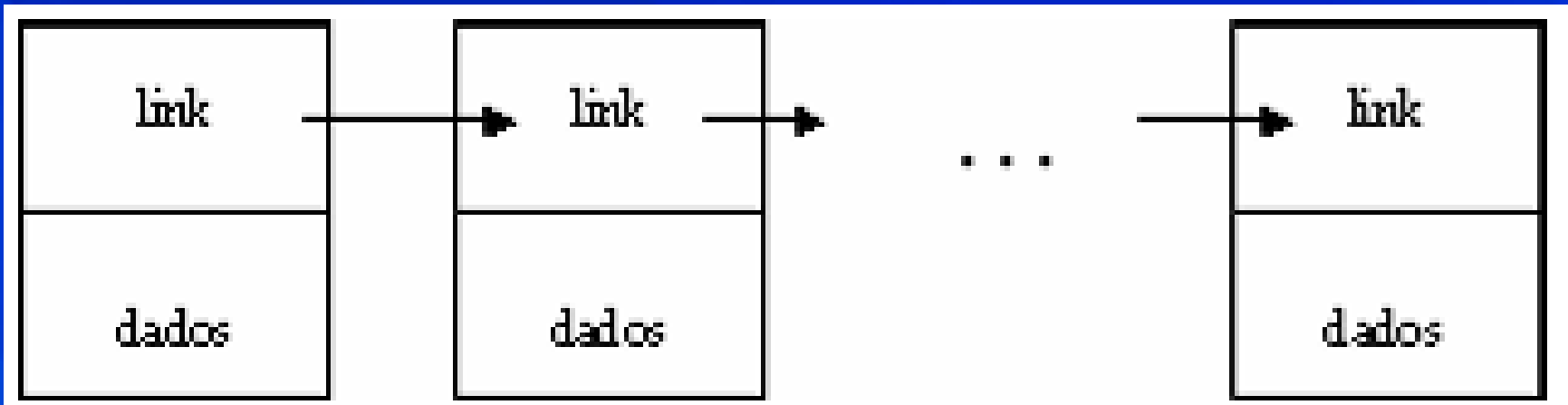
- Implementação de **filas**, **pilhas** e grafos
- Sistemas operacionais (controle de processos, buffers)
- Navegadores (histórico de navegação)
- Manipulação dinâmica de dados
- Aplicações que exigem acesso sequencial e modificações constantes

A lista não impõe nenhuma ordem de uso como LIFO ou FIFO por si só. Isso depende de como você usa a lista.

EXEMPLO:

Lista encadeada simples:

- Os nós conectados através de ponteiros formam **uma lista** encadeada. Cada nó é composto basicamente por duas informações: um campo de dados e um campo de ponteiros, onde se armazenam os **endereços das conexões** entre os nós.



[10 | *] → [20 | *] → [30 | NULL]

SINTAXE COM C++:

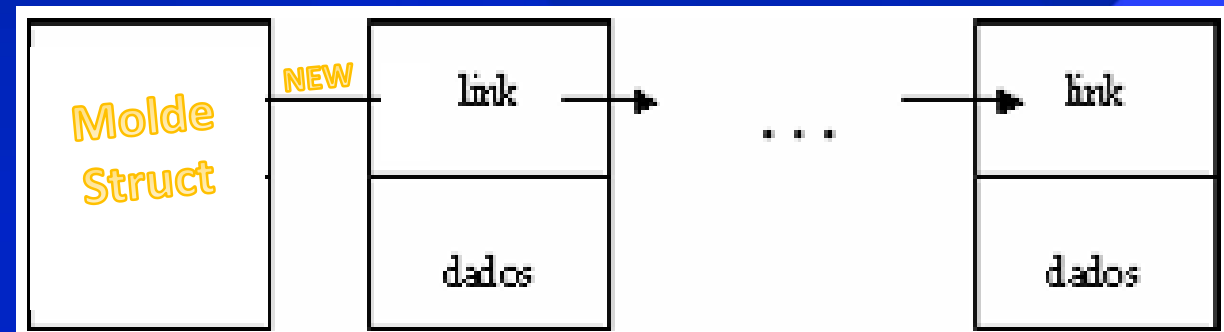
Em C++, usamos **struct** para representar um nó de uma estrutura encadeada (como listas, árvores, grafos) porque ela nos permite **agrupar dados diferentes em uma só unidade**. Mas tecnicamente, qualquer tipo que agrupe dados e ponteiros pode ser usado como nó.

Linguagem	Representação comum de nó
C	<code>struct</code>
C++	<code>struct</code> ou <code>class</code>
Java	<code>class</code>
Python	<code>class</code>
JavaScript	<code>object</code> (literal ou class)

SINTAXE COM C++:

```
struct Node {  
    int valor;  
    Node* proximo;  
};
```

```
Node* novo = new Node;  
novo->valor = 42;  
novo->proximo = nullptr;
```



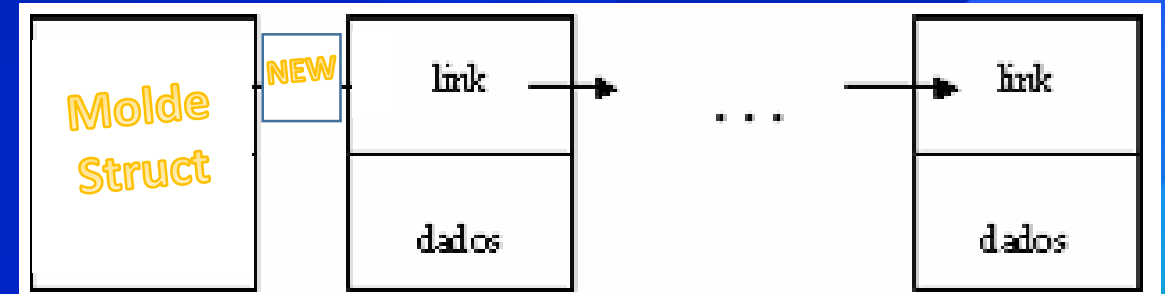

SINTAXE COM C++:

O **novo** aponta para o **próximo**, e o novo passa a ser a cabeça. Ou seja: O novo nó é inserido antes, tornando-se o primeiro da lista.

```
#include <iostream>
using namespace std;

struct Node {
    int valor;
    Node* proximo;
};

void inserirInicio(Node*& cabeca, int valor1) {
    Node* novo = new Node;
    novo->valor = valor1;
    novo->proximo = cabeca;
    cabeca = novo;
}
```



SINTAXE COM C++:

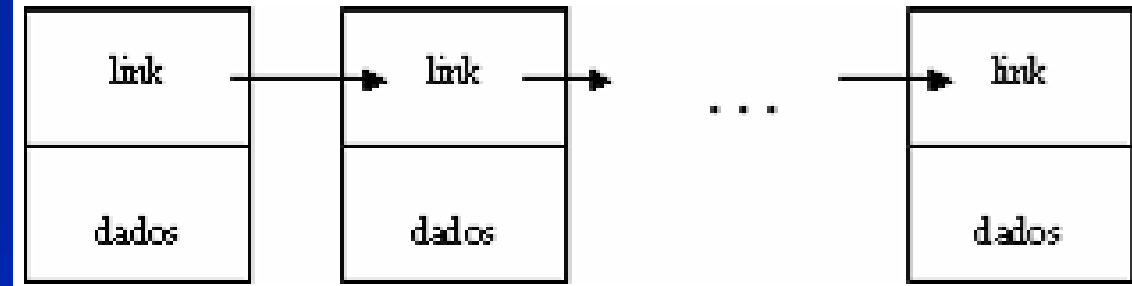
```
#include <iostream>
using namespace std;
```

```
struct Node {
    int valor;
    Node* proximo;
};
```

Molde

```
void inserirInicio(Node*& cabeca, int valor1) {
    Node* novo = new Node;
    novo->valor = valor1;
    novo->proximo = cabeca;
    cabeca = novo;
}
```

```
void exibirLista(Node* cabeca) {
    while (cabeca != nullptr) {
        cout << cabeca->valor << " ? ";
        cabeca = cabeca->proximo;
    }
    cout << "NULL" << endl;
}
```



Novo
(new)

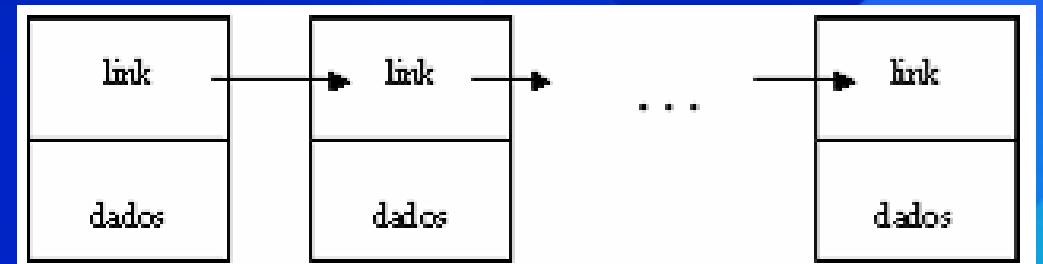
próximo

A **struct** em C++ é como um **molde ou forma** (template) para criar objetos compostos — neste caso, nós (nodes) de uma lista encadeada.

SINTAXE COM C++:

```
// Definição da estrutura de um nó da lista  
struct Node {  
    int valor;           // Valor armazenado no nó  
    Node* proximo;      // Ponteiro para o próximo nó da lista  
};
```

Em uma **lista encadeada**, os elementos são organizados em **nós** (ou "nodes") conectados entre si por meio de **ponteiros**.



Cada nó armazena dois tipos de informação:

- **Um dado (valor):** Neste exemplo, é uma variável do tipo int chamada valor. Por exemplo, um número, um ID, uma nota etc.
- **Um ponteiro (proximo):** Este é um ponteiro para **outro Node**. Se não houver mais elementos após esse nó, esse ponteiro conterá o valor **nullptr (nulo)**, indicando o fim da lista.

SINTAXE COM C++:

```
// Função para inserir um novo elemento no INÍCIO da lista  
void inserirInicio(Node*& cabeca, int valor1) {
```

void: não retorna nenhum valor. **inserirInicio:** nome da função,

Node*& cabeca: Esse parâmetro é uma referência para um ponteiro do tipo Node. Isso significa que qualquer alteração feita nesse ponteiro afeta diretamente a lista original, pois estamos manipulando o endereço da cabeça da lista (o primeiro nó).

int valor1: o valor que queremos armazenar no novo nó da lista.

SINTAXE COM C++:

```
// Cria um novo nó dinamicamente  
Node* novo = new Node;
```

Aqui estamos usando o operador **new** para **alocar dinamicamente** um novo nó na memória.

O **ponteiro novo** agora aponta para esse novo nó recém-criado. Isso garante que o nó continue existindo mesmo após o fim da função (porque está na heap (memória dinâmica), e não na stack (pilha)).

O **Node** criado com **new** fica na **heap**. Ele só vai desaparecer quando você usar **delete p**

SINTAXE COM C++:

Característica	Stack	Heap
Vida útil	Curta (função termina)	Longa (até <code>delete</code>)
Alocação	Automática	Manual (<code>new</code>)
Velocidade	Mais rápida	Mais lenta
Tamanho	Limitado	Maior
Uso na lista	Não serve para nós duradouros	Ideal para listas dinâmicas

heap é o local na memória onde os nós da lista são alocados. Isso permite que os dados permaneçam vivos mesmo quando saem do escopo da função. Ou seja, a lista não é a heap, mas usa a heap para ser dinâmica.

Uma **lista encadeada** é **neutra** em termos de comportamento FIFO ou LIFO.

SINTAXE COM C++:

```
// Atribui o valor passado como parâmetro ao campo 'valor' do novo nó  
novo->valor = valor1;
```

inicializando o campo valor do novo nó com o conteúdo da **variável valor1** passada como argumento da função.

Ou seja, se chamarmos **inserirInicio(cabeca, 42);**, então **novo->valor** vai receber **42**.

SINTAXE COM C++:

```
// O novo nó agora aponta para o antigo primeiro nó da lista  
novo->proximo = cabeca;  
  
// A cabeça da lista agora passa a ser o novo nó  
cabeca = novo;
```

A variável **cabeca** representa o início atual da lista, ou seja, o nó que está no topo (primeiro).

Como estamos inserindo um novo nó no início, esse **novo nó** deve apontar para o nó que anteriormente era o primeiro. Isso é feito ligando o campo **próximo** do novo nó à variável **cabeca**. Agora que o novo nó foi corretamente criado e apontado, precisamos **atualizar a "cabeça" da lista** (a variável **cabeca**) para que ela aponte para o novo primeiro nó.

cabeca → [20 | *] → [30 | NULL]

novo → [10 | *] → [20 | *] → [30 | NULL]

1) Recebe um ponteiro para a cabeça da lista, por referência (**com &**), para que a alteração afete a lista original

2) Recebe um valor inserido no final da lista.

3) Cria um novo nó e o coloca o valor no final da lista

4) IF verifica se a lista está vazia

5) Se sim, o novo nó vira a cabeça da lista, porque é o primeiro e único nó.

6) Criamos um ponteiro auxiliar temp, que começa apontando para a cabeça da lista.

7) Usamos um while para percorrer a lista até o último nó, ou seja, até encontrar um nó que tenha **proximo == nullptr**.

EXEMPLO – função PARA inserir no FINAL da lista.

```
void inserirFinal(Node*& cabeça, int valor) {  
    Node* novo = new Node{valor, nullptr};  
    if (cabeça == nullptr) {  
        cabeça = novo;  
        return;  
    }  
    Node* temp = cabeça;  
    while (temp->proximo != nullptr)  
        temp = temp->proximo;  
    temp->proximo = novo;  
}
```

head → [10] → [20] → [30] → NULL

inserirFinal(head, 40);

head → [10] → [20] → [30] → [40] → NULL

EXEMPLO – função PARA remover ELEMENTO da lista.

```
void removerElemento(Node*& head, int valor) {  
    if (!head) return;  
    if (head->valor == valor) {  
        Node* temp = head;  
        head = head->proximo;  
        delete temp;  
        return;  
    }  
    Node* atual = head;  
    while (atual->proximo && atual->proximo->valor != valor) {  
        atual = atual->proximo;  
    }  
    if (atual->proximo) {  
        Node* temp = atual->proximo;  
        atual->proximo = temp->proximo;  
        delete temp;  
    }  
}
```

head → [10] → [20] → [30] → NULL

removerElemento(head, 20);

head → [10] → [30] → NULL

- 1) Recebe a referência (**Node*& head**) para altara a lista original;
- 2) Recebe o valor que deve ser removido da lista;
- 3) Se head é VAZIA (**nullptr**), não há nada para remover;
- 4) **Head -> valor == valor** verifica se é o valor a ser removido;
- 5) Se sim: **a)** Salva o ponteiro do nó atual em Temp ; **b)** avança a cabeça para o próximo nó; **c)** Libera o nó com delete temp **d)** Retorna imediatamente;
- 6) Começa do primeiro nó (head)
Enquanto o próximo nó existe (**atual->proximo**) e não tem o valor procurado (**atual->proximo->valor != valor**), continua avançando.
- 7) Ao final do laço, atual aponta para o nó antes do que queremos remover.
- 8) Se encontrou o valor desejado: **atual->proximo** existe (ou seja, encontramos o valor). **A)** Armazena o nó a ser removido em **temp**; **B)** Atualiza o ponteiro do nó anterior (**atual->proximo**) para pular o nó a ser removido **C)** Libera da memória com **delete temp**

EXEMPLO

Faça está um código em C++, onde o usuário pode:

1. Inserir números inteiros em uma lista encadeada.
2. Escolher se quer remover um valor da lista.
3. Repetir o processo de remoção.
4. Escolher se quer inserir mais números novamente.
5. A lista é exibida sempre após as ações.

```

#include <iostream>
#include <string>
using namespace std;

// Estrutura de um nó da lista
struct Node {
    int valor;           // Valor armazenado no nó
    Node* proximo;       // Ponteiro para o próximo nó
};

// Função para inserir elemento no final da lista
void inserirFinal(Node*& head, int valor) {
    Node* novo = new Node{valor, nullptr};

    // Se a lista estiver vazia, o novo nó vira a cabeça
    if (!head) {
        head = novo;
        return;
    }

    // Senão, percorre até o fim da lista e insere lá
    Node* temp = head;
    while (temp->proximo)
        temp = temp->proximo;

    temp->proximo = novo;
}

```

```
// Função para exibir os elementos da lista
void exibirLista(Node* head) {
    cout << "Lista atual: ";
    while (head) {
        cout << head->valor << " -> ";
        head = head->proximo;
    }
    cout << "NULL" << endl;
}
```

```

// Função para remover o primeiro nó com um valor específico
void removerElemento(Node*& head, int valor) {
    // Se a lista estiver vazia, não há o que remover
    if (!head) return;

    // Caso o valor esteja na cabeça da lista
    if (head->valor == valor) {
        Node* temp = head;
        head = head->proximo;
        delete temp;
        cout << "Valor " << valor << " removido da lista.\n";
        return;
    }

    // Busca pelo valor nos demais nós
    Node* atual = head;
    while (atual->proximo && atual->proximo->valor != valor) {
        atual = atual->proximo;
    }

    // Se o valor foi encontrado, remove o nó
    if (atual->proximo) {
        Node* temp = atual->proximo;
        atual->proximo = temp->proximo;
        delete temp;
        cout << "Valor " << valor << " removido da lista.\n";
    } else {
        cout << "Valor " << valor << " não encontrado na lista.\n";
    }
}

```



```
int main() {
    Node* lista = nullptr;
    string resposta;
    int valor;

    cout << "=== Lista Encadeada Dinâmica ===\n";

    // Loop principal de inserção/remover/repetir
    do {
        // Inserção de valores na lista
        cout << "\nDigite números inteiros para inserir na lista (digite -1 para parar):\n";
        while (true) {
            cout << "Valor: ";
            cin >> valor;
            if (valor == -1) break;
            inserirFinal(lista, valor);
        }

        // Exibe a lista após inserção
        exibirLista(lista);
    }
```

```
// Remoção de valores
cout << "\nDeseja remover algum valor da lista? (s/n): ";
cin >> resposta;

while (resposta == "s" || resposta == "S") {
    cout << "Informe o valor a ser removido: ";
    cin >> valor;
    removerElemento(lista, valor);
    exibirLista(lista);

    cout << "Deseja remover outro valor? (s/n): ";
    cin >> resposta;
}

// Verifica se o usuário quer inserir mais valores
cout << "\nDeseja inserir mais números na lista? (s/n): ";
cin >> resposta;

} while (resposta == "s" || resposta == "S");

cout << "\nFim do programa. Lista final:\n";
exibirLista(lista);

return 0;
}
```

EXERCÍCIO

Desenvolva um programa em C++ que utilize uma lista encadeada dinâmica para armazenar dados de pessoas, sendo que cada pessoa deve conter **nome (string)**, **idade (int)** e **altura (float)**.

O programa deve permitir que o usuário **cadastre várias pessoas**, uma por vez, inserindo os dados **ao final da lista**. Após cada ciclo de inserção, o programa deve exibir todos os registros cadastrados.

Em seguida, o usuário deve ter a opção de **remover pessoas informando o nome**. Caso o nome exista na lista, o respectivo nó deve ser removido. Caso contrário, deve ser exibida uma mensagem informando que o nome não foi encontrado. O processo de remoção pode ser repetido quantas vezes o usuário desejar.

Ao final de cada etapa de remoção, o programa deve perguntar se o usuário deseja inserir mais pessoas. Caso a resposta seja positiva, o ciclo de inserção e remoção se repete.

Quando o usuário decidir não inserir mais dados, o programa deve exibir a lista final de pessoas e encerrar.

O programa deve utilizar alocação dinâmica de memória (com new e delete), organização com funções separadas para inserção, remoção e exibição, e permitir interação com o usuário por meio de entrada via teclado.

OBRIGADO!