🧱 Estruturas de Dados Sequenciais em C++

As **listas**, **pilhas** e **filas** são estruturas fundamentais usadas para organizar dados na memória.

Quando implementadas **de forma sequencial**, elas utilizam **vetores (arrays)** para armazenar os elementos.

🔹 1. Lista Linear Sequencial

Uma **lista** é uma coleção **ordenada** de elementos do mesmo tipo, armazenados em posições consecutivas de memória.

Exemplo:

L = {a1, a2, a3, ..., an}

onde n é o número de elementos.

📦 Estrutura:

Os dados ficam em um vetor V[0 ... m-1], onde:

- m → capacidade máxima
- n → número de elementos atuais (n ≤ m)

🧩 Operações principais:

Operação	O que faz	Custo
Inserir (k)	Insere elemento na posição k	O(n)
Remover (k)	Remove elemento na posição k	O(n)
Buscar	Percorre até encontrar o elemento	O(n) (sequencial)
Busca binária	Se estiver ordenada	O(log n)
Acessar posição	Por índice direto	O(1)

💡 Exemplo prático:

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

class Lista {
private:
    int* elementos;
```

```
    int tamanho;
    int capacidade;

public:
    Lista(int cap) {
        capacidade = cap;
        tamanho = 0;
        elementos = new int[capacidade];
    }

    void inserir(int valor) {
        if (tamanho < capacidade) {
            elementos[tamanho++] = valor;
        } else {
            cout << "Lista cheia!" << endl;
        }
    }

    void exibir() {
        cout << "Elementos: ";
        for (int i = 0; i < tamanho; i++)
            cout << elementos[i] << " ";
        cout << endl;
    }

    ~Lista() {
        delete[] elementos;
    }
};

int main() {
    Lista lista(5);
    lista.inserir(10);
    lista.inserir(20);
    lista.inserir(30);
    lista.exibir();
    return 0;
}
```

🔹 **2. Pilha Sequencial (Stack)**

A **pilha** segue o princípio **LIFO (Last In, First Out)**:

o último a entrar é o primeiro a sair.

📚 Exemplo do mundo real: uma pilha de pratos — só dá pra pegar o de cima.

🧩 **Estrutura:**

- Armazenada em um vetor V
- Usa um índice topo para indicar o último elemento inserido

⚙️ **Operações:**

Operação	Descrição	Complexidade
push(x)	Empilha um elemento	O(1)
pop( )	Desempilha e retorna o topo	O(1)
top( )	Consulta o elemento do topo	O(1)

💡 **Exemplo prático:**

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

class Pilha {
private:
    int* elementos;
    int topo;
    int capacidade;

public:
    Pilha(int cap) {
        capacidade = cap;
        elementos = new int[capacidade];
        topo = -1;
    }

    void empilhar(int valor) {
        if (topo < capacidade - 1)
            elementos[++topo] = valor;
        else
            cout << "Pilha cheia!" << endl;
    }

    void desempilhar() {
        if (topo >= 0)
            topo--;
        else
            cout << "Pilha vazia!" << endl;
    }

    int verTopo() {
        if (topo >= 0)
            return elementos[topo];
        else {
            cout << "Pilha vazia!" << endl;
```

```
            return -1;
        }
    }

    ~Pilha() {
        delete[] elementos;
    }
};

int main() {
    Pilha p(3);
    p.empilhar(5);
    p.empilhar(10);
    p.empilhar(15);
    cout << "Topo da pilha: " << p.verTopo() << endl;
    p.desempilhar();
    cout << "Topo após desempilhar: " << p.verTopo() << endl;
    return 0;
}
```

🔹 **3. Fila Sequencial (Queue)**

A **fila** segue o princípio **FIFO (First In, First Out)**:

- O primeiro a entrar é o primeiro a sair.

🎟️ Exemplo do mundo real: fila de pessoas em um caixa.

🧩 **Estrutura:**

- Armazenada em um vetor V
- Usa dois índices:
    - inicio → primeiro elemento
    - fim → última posição inserida

⚙️ **Operações:**

Operação	**Descrição**	**Complexidade**
enqueue(x)	Adiciona no final da fila	O(1)
dequeue( )	Remove do início da fila	O(n) (pois desloca os elementos)
front( )	Consulta o primeiro elemento	O(1)

💡 **Exemplo prático:**

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

class Fila {
private:
    int* elementos;
    int tamanho;
    int capacidade;

public:
    Fila(int cap) {
        capacidade = cap;
        tamanho = 0;
        elementos = new int[capacidade];
    }

    void enfileirar(int valor) {
        if (tamanho < capacidade)
            elementos[tamanho++] = valor;
        else
            cout << "Fila cheia!" << endl;
    }

    void desenfileirar() {
        if (tamanho > 0) {
            for (int i = 0; i < tamanho - 1; i++)
                elementos[i] = elementos[i + 1];
            tamanho--;
        } else
            cout << "Fila vazia!" << endl;
    }

    void exibir() {
        cout << "Fila: ";
        for (int i = 0; i < tamanho; i++)
            cout << elementos[i] << " ";
        cout << endl;
    }

    ~Fila() {
        delete[] elementos;
    }
};

int main() {
    Fila f(5);
    f.enfileirar(10);
    f.enfileirar(20);
```

```
    f.enfileirar(30);
    f.exibir();
    f.desenfileirar();
    f.exibir();
    return 0;
}
```

🔹 3.1 Fila Circular

Serve para **evitar o deslocamento** de elementos após cada remoção e **reutilizar o espaço** do vetor.

Usa **aritmética modular** para "voltar ao início" quando chega ao fim.

Fórmulas:

- Inserção: r = (r + 1) % m
- Remoção: f = (f + 1) % m
- Fila vazia: f == r
- Fila cheia: (r + 1) % m == f

🎯 O Problema da Fila Normal (Linear)

Numa **fila sequencial comum**, os elementos são armazenados num **vetor**.

Quando você remove o primeiro elemento (dequeue), precisa **deslocar todos os outros** uma posição para frente.

Isso torna a operação **custosa (O(n))**.

Exemplo:

Fila: [10, 20, 30, 40]

Remover 10 → precisa deslocar tudo:

[20, 30, 40, _]

➡️Por isso, criamos a **fila circular**, que evita deslocamentos.

🔄 A Ideia da Fila Circular

Em vez de deslocar, usamos **dois índices**:

- f → início (front)

- r → fim (rear)

E usamos **aritmética modular** (%) para “voltar ao início do vetor” quando o final é atingido.

É como se o vetor fosse um círculo, onde depois da última posição, vem a primeira novamente.

🧮 As Fórmulas

➕ Inserção (enqueue)

r = (r + 1) % m

➡️ Isso significa:

“Adicione 1 ao índice do final da fila.

Se ele ultrapassar o tamanho m, volte para o início (índice 0).”

Exemplo:

Se m = 5 e r = 4, então:

r = (4 + 1) % 5 = 0

Ou seja, o próximo elemento entra na **posição 0** novamente.

➖ Remoção (dequeue)

f = (f + 1) % m

➡️ Após remover o elemento da frente, o índice f avança para o próximo — e se chegar ao fim, volta ao início do vetor.

⚪ Fila Vazia

f == r

A fila está **vazia** quando o índice do início e o do fim são iguais.

⚫ Fila Cheia

(r + 1) % m == f

Se o próximo índice de inserção (r + 1) cair no mesmo valor de f, significa que não há espaço livre.

👉 Por isso, **uma posição do vetor fica sempre inutilizada** — serve para distinguir "cheia" de "vazia".

📊 Resumo Rápido

Fórmula	**Significado**
r = (r + 1) % m	Próxima posição de inserção
f = (f + 1) % m	Próxima posição de remoção
f == r	Fila vazia
(r + 1) % m == f	Fila cheia

🧠 Resumo Final

Estrutura	**Princípio**	**Inserção**	**Remoção**	**Complexidade Acesso**
Lista	Linear ordenada	O(n)	O(n)	O(1)
Pilha	LIFO	O(1)	O(1)	O(1)
Fila	FIFO	O(1)	O(n)	O(1)
Fila Circular	FIFO otimizado	O(1)	O(1)	O(1)