

# Implementação de Estruturas de Dados em C++: Lista Encadeada, Pilha, Fila e Matriz

Gabriel Felipe Quaresma de Oliveira

<sup>1</sup>Departamento de Computação  
PUC-MG – Belo Horizonte, Brasil

work.gabriel.quaresma@gmail.com

***Resumo.** Este documento descreve a implementação de estruturas de dados fundamentais em C++, incluindo lista encadeada, pilha, fila e matriz de inteiros. Além disso, são apresentados exemplos de inclusão, busca e remoção de elementos dessas estruturas.*

## 1. Introdução

Estruturas de dados são fundamentais para a organização e manipulação de informações em sistemas computacionais, e nesse caso em especial para grafos. Este trabalho apresenta a implementação de uma lista encadeada, uma pilha, uma fila e uma matriz de inteiros em C++.

## 2. Estruturas Implementadas

### 2.1. Lista Encadeada

A lista encadeada é uma estrutura de dados dinâmica que consiste em célula, onde cada célula contém um valor e um ponteiro para a próxima célula.

### 2.2. Pilha

A pilha é uma estrutura de dados linear que segue a política Last In, First Out (LIFO).

### 2.3. Fila

A fila é uma estrutura de dados linear que segue a política First In, First Out (FIFO).

### 2.4. Matriz de Inteiros

Uma matriz é uma estrutura bidimensional de dados, onde cada elemento é acessível através de índices.

## 3. Implementação

### 3.1. Lista Encadeada

```
template <typename T>
```

```
class Cell
{
    public :
```

```

    T value;
    Cell *next;

    Cell() : value(T {}), next(nullptr) {}

    Cell(Cell* next) : value(T {}), next(next) {}

    Cell(T value) : value(value), next(nullptr) {}

    Cell(T value, Cell* next) : value(value), next(next) {}
};

```

```

#include "../Cell/Cell.cpp"
#include <iostream>

```

```

template <typename T>

```

```

class LinkedList{
    private:
        Cell<T>* first;
        Cell<T>* last;
        int N; //Number of elements that are in the List

    public:
        LinkedList() {
            first = last = nullptr;
            N = 0;
        }

        ~LinkedList() {
            Cell<T>* current = first;
            while (current != nullptr) {
                Cell<T>* nextCell = current->next;
                delete current;
                current = nextCell;
            }
        }

        void pushFront(T value) {
            Cell<T>* newCell = new Cell<T>(value);
            newCell->next = first;
            first = newCell;

            if (N == 0) last = newCell;
            N++;
        }
}

```

```

void pushBack(T value) {
    Cell<T>* newCell = new Cell<T>(value);

    if(last == nullptr) first = last = newCell;
    else{
        last->next = newCell;
        last = newCell;
    }
    N++;
}

void insert(T value , int position) {
    if(position < 0 || position > N){
        throw std::runtime_error("Error: Invalid position in insert");
    }

    Cell<T>* temp = first;
    Cell<T>* newCell = new Cell<T>(value);

    for (int i = 0; i < position - 1; i++) temp = temp->next;

    if(position == 0){
        newCell->next = first;
        first = newCell;
    } else {
        newCell->next = temp->next;
        temp->next = newCell;
    }

    if (position == N) last = newCell;
    N++;
}

T popFront( ) {
    if(first == nullptr){
        throw std::runtime_error("Error: Empty List.");
    }

    Cell<T>* temp = first;
    T value = first->value;
    first = first->next;

    delete temp;
    N--;
}

```

```

        if (first == nullptr) last = first;

        return value;
    }

    T popBack() {
        if (first == nullptr){
            throw std::runtime_error("Error: ■Empty■Queue.");
        }

        T value = last->value;

        if (first == last){
            delete last;
            first = last = nullptr;
        } else {
            Cell<T>* temp = first;
            while (temp->next != last) temp = temp->next;
            temp->next = nullptr;
            delete last;
            last = temp;
        }

        N--;
        return value;
    }

    T front() const {
        if (first == nullptr) throw std::runtime_error("Error: ■Empty■List");
        return first->value;
    }

    T remove(int position) {
        if (position < 0 || position > N) throw std::runtime_error("Error: ■Invalid position");

        if (first == nullptr) throw std::runtime_error("Error: ■Empty■List");

        Cell<T>* temp = first;
        T removedValue;

        // Caso especial: remo o do primeiro elemento
        if (position == 0) {
            removedValue = first->value;
            first = first->next;

```

```

        delete temp;

        if (first == nullptr) last = nullptr;

    } else {
        // Percorre at a c lula anterior remover
        for (int i = 0; i < position - 1; i++) temp = temp->next;

        Cell<T>* cellToRemove = temp->next;
        removedValue = cellToRemove->value;
        temp->next = cellToRemove->next;

        // Caso especial: remo o do ltimo elemento
        if (cellToRemove == last) last = temp;

        delete cellToRemove;
    }

    N--;
    return removedValue;
}

bool search(T value) const {
    if (first == nullptr) throw std::runtime_error("Error: ■Empty■Li");

    Cell<T>* temp = first;
    bool findIt = (temp->value == value); // Est aqui para testar
    while(temp->next != nullptr && findIt == false){
        if(temp->value == value) findIt = true;
        temp = temp->next;
    }
    return findIt;
}

int size() const { return N; }

bool isEmpty() const { return N == 0; }

void print(int position) const {
    Cell<T>* temp = first;
    for(int i = 0; i < position; i++) temp = temp->next;

    std::cout << temp->value << "\n";
}

void printList() const {

```

```

        Cell<T>* temp = first;
        for(int i = 0; i < N; i++){
            std::cout << temp->value;
            if(i < N - 1) std::cout << ",■";
            temp = temp->next;
        }

        std::cout << "\n";

    }

};

```

### 3.2. Pilha

```
#include "../List/LinkedList.cpp"
```

```

template <typename T>
class Stack {
private:
    LinkedList<T> list;

public:
    Stack() = default;

    void push(T value) {
        list.pushFront(value);
    }

    T pop() {
        if (isEmpty()) throw std::runtime_error("Error: Empty Stack.");
        return list.popFront();
    }

    T top() const {
        if (isEmpty()) throw std::runtime_error("Error: Empty Stack.");
        return list.front();
    }

    int size() const {
        return list.size();
    }

    bool isEmpty() const {
        return list.isEmpty();
    }
}

```

```

        void printStack() const { list.printList(); }

};

```

### 3.3. Fila

```

#include "../List/LinkedList.cpp"
template <typename T>

class Queue {
private:
    LinkedList<T> list;

public:
    Queue() = default;

    void enqueue(T value) {
        list.pushBack(value);
    }

    T dequeue() {
        if (isEmpty()) throw std::runtime_error("Error: Empty Queue.");
        return list.popFront();
    }

    bool search (T value) const {
        if (isEmpty()) throw std::runtime_error("Error: Empty Queue.");
        return list.search(value);
    }

    int size() const {
        return list.size();
    }

    bool isEmpty() const {
        return size() == 0;
    }

    void printQueue() const { list.printList(); }
};

```

### 3.4. Matriz de Inteiros

```

#include <iostream>
#include <stdexcept>

template <typename T>
class FlexibleMatrix {

```

```

private:
    T** data;
    int rows;
    int cols;

public:
    FlexibleMatrix(int rows, int cols) : rows(rows), cols(cols) {
        data = new T*[rows]; // N o      necess rio parenteses pq n o e
        for (int i = 0; i < rows; i++) data[i] = new T[cols]();
    }

    ~FlexibleMatrix() {
        for (int i = 0; i < rows; i++) delete[] data[i];
        delete[] data; // Necess rio por causa do new[] (pra eu n o es
    }

    // Copia o construtor
    FlexibleMatrix(const FlexibleMatrix& other) : rows(other.rows), col
        data = new T*[rows];
        for (int i = 0; i < rows; i++) {
            data[i] = new T[cols];
            for (int j = 0; j < cols; j++) data[i][j] = other.data[i][j]
        }
    }

    // Atribui o de operador
    FlexibleMatrix& operator=(const FlexibleMatrix& other) {
        if (this == &other) return *this;

        for (int i = 0; i < rows; i++) delete[] data[i];

        delete[] data;

        // Alocar nova mem ria
        rows = other.rows;
        cols = other.cols;
        data = new T*[rows];
        for (int i = 0; i < rows; i++) {
            data[i] = new T[cols];
            for (int j = 0; j < cols; j++) data[i][j] = other.data[i][j]
        }

        return *this; //Retornando a referencia
    }

    // M todo para acessar elementos

```



```

T& at(int row, int col) {
    if (row < 0 || row >= rows || col < 0 || col >= cols) {
        throw std::out_of_range("Index out of bounds");
    }
    return data[row][col];
}

// M todo para acessar elementos (const version)
const T& at(int row, int col) const {
    if (row < 0 || row >= rows || col < 0 || col >= cols) {
        throw std::out_of_range("Index out of bounds");
    }
    return data[row][col];
}

void print() const {
    for (int i = 0; i < rows; ++i) {
        for (int j = 0; j < cols; ++j) {
            std::cout << data[i][j] << " | ";
        }
        std::cout << "\n";
    }
}

bool search(T value) const{
    bool findIt = false;
    for(int i = 0; i < rows; i++){
        for(int j = 0; j < cols; j++){
            if(data[i][j] == value){
                findIt = true;
                i = rows;
                j = cols;
            }
        }
    }
    return findIt;
}

int getRows() const { return rows; }

int getCols() const { return cols; }
};

```

#### 4. Aplicação

Para demonstrar as estruturas de dados implementadas, foram realizados testes de inclusão, busca e remoção de elementos.

#### 4.1. Lista Encadeada

```
#include "LinkedList.cpp"
#include <iostream>

int main() {
    LinkedList<int> list;

    // Testando inserções
    list.pushFront(10);
    list.pushFront(20);
    list.pushBack(30);
    list.insert(25, 2);

    std::cout << "Lista após inserções: ";
    list.printList(); // Deve mostrar: 20, 10, 25, 30

    // Testando remoções
    list.popFront();
    std::cout << "Lista após popFront(): ";
    list.printList(); // Deve mostrar: 10, 25, 30

    list.popBack();
    std::cout << "Lista após popBack(): ";
    list.printList(); // Deve mostrar: 10, 25

    list.remove(0);
    std::cout << "Lista após remover a posição 0: ";
    list.printList(); // Deve mostrar: 25

    // Testando busca
    bool found = list.search(25);
    std::cout << "O valor 25 está na lista? " << (found ? "Sim" : "Não");

    found = list.search(10);
    std::cout << "O valor 10 está na lista? " << (found ? "Sim" : "Não");

    // Testando tamanho
    std::cout << "Tamanho da lista: " << list.size() << std::endl;
    // Deve mostrar: 1

    // Testando se a lista está vazia
    std::cout << "A lista está vazia? " << (list.isEmpty() ? "Sim" : "Não");
    // Deve mostrar: Não

    return 0;
}
```

```
Lista após inserções: 20, 10, 25, 30
Lista após popFront(): 10, 25, 30
Lista após popBack(): 10, 25
Lista após remover a posição 0: 25
O valor 25 está na lista? Sim
O valor 10 está na lista? Não
Tamanho da lista: 1
A lista está vazia? Não
```

Figure 1. Teste da classe da Lista

## 4.2. Pilha

```
#include <iostream>
#include "Stack.cpp"

int main() {
    Stack<int> stack;

    // Testando inserções
    stack.push(10);
    stack.push(20);
    stack.push(30);

    std::cout << "Pilha após inserções: ";
    stack.printStack(); // Deve mostrar: 30, 20, 10

    // Testando o topo da pilha
    std::cout << "Topo da pilha: " << stack.top() << std::endl;
    // Deve mostrar: 30

    // Testando remoções
    std::cout << "Desempilhando: " << stack.pop() << std::endl;
    // Deve mostrar: 30
    std::cout << "Desempilhando: " << stack.pop() << std::endl;
    // Deve mostrar: 20

    std::cout << "Pilha após pop(): ";
    stack.printStack(); // Deve mostrar: 10

    // Testando tamanho
    std::cout << "Tamanho da pilha: " << stack.size() << std::endl;
    // Deve mostrar: 1

    // Testando se a pilha está vazia
    std::cout << "A pilha está vazia? " << (stack.isEmpty() ? "Sim" :
    // Deve mostrar: Não
```

```

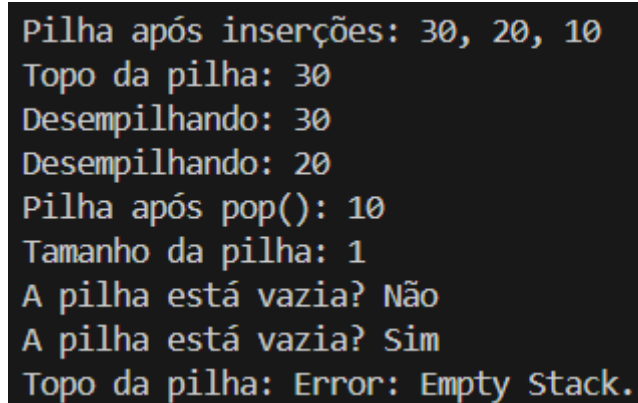
// Testando a pilha vazia
stack.pop(); // Remove o ltimo elemento

std::cout << "A pilha est vazia? " << (stack.isEmpty() ? "Sim" :
// Deve mostrar: Sim

// Testando erro ao acessar o topo de uma pilha vazia
try {
    std::cout << "Topo da pilha: " << stack.top() << std::endl;
} catch (const std::runtime_error& e) {
    std::cerr << e.what() << std::endl;
// Deve mostrar erro: Empty Stack.
}

return 0;
}

```



```

Pilha após inserções: 30, 20, 10
Topo da pilha: 30
Desempilhando: 30
Desempilhando: 20
Pilha após pop(): 10
Tamanho da pilha: 1
A pilha está vazia? Não
A pilha está vazia? Sim
Topo da pilha: Error: Empty Stack.

```

Figure 2. Teste da classe da Pilha

#### 4.3. Fila

```

#include <iostream>
#include "Queue.cpp"

int main() {
    // Cria uma fila de inteiros
    Queue<int> queue;

    // Teste do metodo isEmpty (deve retornar true)
    std::cout << "Fila vazia: "
    << (queue.isEmpty() ? "Sim" : "Nao") << std::endl;

    // Enfileira alguns elementos
    queue.enqueue(10);
    queue.enqueue(20);
    queue.enqueue(30);
}

```

```

// Teste do metodo size (deve retornar 3)
std::cout << "Tamanho da fila: " << queue.size() << std::endl;

// Testa o metodo search
(deve retornar true para 20 e false para 40)
std::cout << "Elemento 20 na fila: "
<< (queue.search(20) ? "Sim" : "Nao") << std::endl;
std::cout << "Elemento 40 na fila: "
<< (queue.search(40) ? "Sim" : "Nao") << std::endl;

// Desenfileira e imprime os elementos
std::cout << "Desenfileirando: "
<< queue.dequeue() << std::endl; // Deve retornar 10
std::cout << "Desenfileirando: "
<< queue.dequeue() << std::endl; // Deve retornar 20

// Teste do metodo size apos desenfileirar (deve retornar 1)
std::cout << "Tamanho da fila: " << queue.size() << std::endl;

// Testa o metodo isEmpty novamente (deve retornar false)
std::cout << "Fila vazia: "
<< (queue.isEmpty() ? "Sim" : "Nao") << std::endl;

// Desenfileira o ultimo elemento
std::cout << "Desenfileirando: "
<< queue.dequeue() << std::endl; // Deve retornar 30

// Testa o metodo isEmpty apos desenfileirar todos os element
std::cout << "Fila vazia: "
<< (queue.isEmpty() ? "Sim" : "Nao") << std::endl;

// Tentativa de desenfileirar de uma fila vazia
try {
    queue.dequeue();
} catch (const std::exception& e) {
    std::cerr << "Erro: " << e.what() << std::endl;
}

return 0;
}

```

#### 4.4. Matriz de Inteiros

```

#include <iostream>
#include "FlexibleMatrix.h" // Supondo que o codigo da classe esteja ne

```

```

int main() {
    // Criando uma matriz 3x3 de inteiros
    FlexibleMatrix<int> matrix(3, 3);

    // Atribuindo valores a matriz
    matrix.at(0, 0) = 1;
    matrix.at(0, 1) = 2;
    matrix.at(0, 2) = 3;
    matrix.at(1, 0) = 4;
    matrix.at(1, 1) = 5;
    matrix.at(1, 2) = 6;
    matrix.at(2, 0) = 7;
    matrix.at(2, 1) = 8;
    matrix.at(2, 2) = 9;

    // Imprimindo a matriz
    std::cout << "Matrix:" << std::endl;
    matrix.print();

    // Testando o metodo de busca
    int searchValue = 5;
    bool found = matrix.search(searchValue);
    std::cout << "\nSearch for value " << searchValue << ": "
    << (found ? "Found" : "Not Found") << std::endl;

    // Testando o construtor de copia
    FlexibleMatrix<int> copiedMatrix = matrix;
    std::cout << "\nCopied Matrix:" << std::endl;
    copiedMatrix.print();

    // Modificando a matriz original e verificando
    se a copia nao foi afetada
    matrix.at(0, 0) = 10;
    std::cout << "\nModified Original Matrix:" << std::endl;
    matrix.print();
    std::cout << "\nUnchanged Copied Matrix:" << std::endl;
    copiedMatrix.print();

    // Testando o operador de atribuicao
    FlexibleMatrix<int> assignedMatrix(3, 3);
    assignedMatrix = matrix;
    std::cout << "\nAssigned Matrix:" << std::endl;
    assignedMatrix.print();

    // Verificando o acesso fora dos limites
    try {

```

```
        int value = matrix.at(3, 3); // Acessando uma posicao
        fora dos limites
    } catch (const std::out_of_range& e) {
        std::cout << "\nException caught: " << e.what() << std::endl;
    }

    return 0;
}
```