Implementação de Estruturas de Dados em C++

Lucas Gualtieri F. E.[1]

[1] Pontifical Catholic University of Minas Gerais
Coração Eucarístico – 30535-901 – Belo Horizonte – MG – Brazil

lgualtieri@sga.pucminas.br

Resumo. *Este trabalho apresenta a implementação de diversas estruturas de dados, desenvolvidas em C++ utilizando templates para que as mesmas fossem genéricas. As estruturas desenvolvidas incluem:*

- *Lista*
- *Fila*
- *Pilha*
- *Matriz*

Além das implementações genéricas, uma matriz de inteiros foi implementada, conforme especificado nos requisitos do projeto. Para validar as estruturas implementadas, foram criados arquivos de teste para garantir o correto funcionamento de todas as operações de cada estrutura.

1. Implementação

Nesta seção, detalho a implementação das estruturas de dados desenvolvidas. As estruturas foram divididas em dois grupos principais: *lineares* e *flexíveis*, como ilustrado na Tabela 1.

Lineares	Flexíveis
`LinearList<T>`	`LinkedList<T>`
`LinearStack<T>`	`DoublyLinkedList<T>`
`LinearQueue<T>`	`LinkedStack<T>`
`Matrix<T>`	`LinkedQueue<T>`
`MatrixInt`	

Table 1. Estruturas de dados desenvolvidas.

Cada uma dessas estruturas foi projetada para atender a diferentes necessidades, mas todas compartilham uma interface comum que garante consistência e reusabilidade do código.

As estruturas lineares, com exceção da `Matrix<T>` e `MatrixInt`, possuem métodos adicionais relacionados à manipulação da capacidade de memória reservada. Abaixo, são apresentados esses métodos:

```
void reserve(size_t newCapacity) {
    if (newCapacity > maxSize) {
        resize(newCapacity);
    }
}

void shrink_to_fit() {
    resize(this->_size);
}

size_t capacity() {
    return maxSize;
}
```

1.1. Listas

Nesta subseção, discuto as três variações de listas que foram implementadas: a lista linear, a lista simplesmente encadeada e a lista duplamente encadeada. Embora essas estruturas tenham diferentes organizações internas, todas compartilham uma interface comum chamada `List<T>`. Essa interface define um conjunto de operações básicas que garantem a consistência na manipulação das listas. A seguir, apresento a interface `List<T>`:

```
template <typename T>
class List {
  protected:
    size_t _size;

  public:
    virtual ~List() = default;

    virtual void push_front(T value) = 0;
    virtual void push_back(T value) = 0;
    virtual T pop_front() = 0;
    virtual T pop_back() = 0;
    virtual void add(const T& value, unsigned int pos) = 0;
    virtual T remove(unsigned int pos) = 0;
    virtual T& front() const = 0;
    virtual T& back() const = 0;
    virtual void sort() = 0;

    virtual bool contains(const T& value) const = 0;
    virtual void clear() final { while (!empty()) pop_front(); }
    virtual size_t size() const final { return _size; }
    virtual bool empty() const final { return _size == 0; }
}
```

1.2. Filas

Nesta subseção, discuto as duas variações de filas que foram implementadas: a fila linear e a fila simplesmente encadeada. Embora essas estruturas tenham diferentes organizações internas, todas compartilham uma interface comum chamada `Queue<T>`. Essa interface define um conjunto de operações básicas que garantem a consistência na manipulação das filas. A seguir, apresento a interface `Queue<T>`:

```
template <typename T>
class Queue {
  protected:
        size_t _size;

  public:
        virtual ~Queue() = default;

        virtual void push(const T& value) = 0;
        virtual T pop() = 0;
        virtual T& peek() const = 0;

        virtual bool contains(const T& value) const = 0;
        virtual void clear() { while (!empty()) pop(); }
        virtual size_t size() const final { return _size; }
        virtual bool empty() const final { return _size == 0; }
        virtual std::string str() const = 0;
};
```

1.3. Pilhas

Nesta subseção, discuto as duas variações de pilhas que foram implementadas: a pilha linear e a pilha simplesmente encadeada. Embora essas estruturas tenham diferentes organizações internas, todas compartilham uma interface comum chamada `Stack<T>`. Essa interface define um conjunto de operações básicas que garantem a consistência na manipulação das pilhas. A seguir, apresento a interface `Stack<T>`:

```
template <typename T>
class Stack {
  protected:
        size_t _size;

  public:
        virtual ~Stack() = default;

        virtual void push(const T& value) = 0;
        virtual T pop() = 0;
        virtual T& peek() const = 0;

        virtual bool contains(const T& value) const = 0;
        virtual void clear() { while (!empty()) pop(); }
        virtual size_t size() const final { return _size; }
        virtual bool empty() const final { return _size == 0; }
        virtual std::string str() const = 0;
};
```

1.4. Matrizes

Nesta subseção, discuto as duas variações de matrizes que foram implementadas: a matriz genérica e a matriz de inteiros. A matriz genérica foi implementada usando uma `vector<vector<T>>`. Já a matriz de inteiros usou um simples `int**`. A seguir, apresento os métodos de ambas as classes:

```
template <typename T>
class Matrix {
    std::vector<std::vector<T>> matrix;
  public:
    const int height, width;

    Matrix(int height, int width);
    Matrix(std::initializer_list<std::initializer_list<T>> list);

    bool inBounds(int i, int j);
    bool notInBounds(int i, int j);

    std::vector<T>& operator[](int i);

    std::string str() const;
    std::string str(bool flag) const;

    friend ostream& operator<<ostream& os, const Matrix<T>& m);

    class Iterator;
    Iterator begin();
    Iterator end();
};
```

```
class Matrix {
    int** matrix;
  public:
    const int height, width;

    Matrix(int height, int width);

    bool inBounds(int i, int j);
    bool notInBounds(int i, int j);

    int* operator[](int i);

    std::string str() const;
    std::string str(bool flag) const;

    friend ostream& operator<<ostream& os, const Matrix<T>& m);
};
```

2. Estrutura do Projeto

Os exemplos de uso das estruturas se encontraM nos arquivos de teste na pastas `tests/`.

```
|-- bin
|   |-- doublyLinkedListTest
|   |-- linearListTest
|   |-- linearQueueTest
|   |-- linearStackTest
|   |-- linkedListTest
|   |-- linkedQueueTest
|   |-- linkedStackTest
|   |-- matrixIntTest
|   |-- matrixTest
|-- docs
|   |-- Grafos_Implementação_1.pdf
|   |-- latex
|       |-- main.tex
|-- include
|   |-- cell.hpp
|   |-- list
|   |   |-- doublyLinkedList.hpp
|   |   |-- linearList.hpp
|   |   |-- linkedList.hpp
|   |   |-- list.hpp
|   |-- matrix
|   |   |-- matrix.hpp
|   |   |-- matrixInt.hpp
|   |-- queue
|   |   |-- linearQueue.hpp
|   |   |-- linkedQueue.hpp
|   |   |-- queue.hpp
|   |-- stack
|       |-- linearStack.hpp
|       |-- linkedStack.hpp
|       |-- stack.hpp
|-- tests
|   |-- doublyLinkedListTest.cc
|   |-- linearListTest.cc
|   |-- linearQueueTest.cc
|   |-- linearStackTest.cc
|   |-- linkedListTest.cc
|   |-- linkedQueueTest.cc
|   |-- linkedStackTest.cc
|   |-- matrixIntTest.cc
|   |-- matrixTest.cc
|-- util
    |-- util.hpp
```