**МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ**

**НОВОСИБИРСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСТИЕТ**

Лабораторная работа №8 по дисциплине «Программирование»

**Библиотека шаблонных классов STL**

Группа: **АВТ-342**

Студенты: **Бондаренко А.В., Фадеев В.А.**

НОВОСИБИРСК 2024

1. **Постановка задачи**

**Вариант 4.**

Для встроенного типа и класса из лабораторной работы №1 провести временной анализ заданных шаблонных классов на основных операциях: добавление, удаление, поиск, сортировка. Использовать итераторы для работы с контейнерами. Для получения времени выполнения операции засекать системное время начала и окончания операции и автоматически генерировать большое количество данных.

– стек

– множество с дубликатами

Нужно реализовать программу, которая заполняет типами данных int и matrix (класс из 1 лабораторной работы) стек, пользовательский стек, а также заполнение множествами с дубликатами, выводит время добавления элемента, время заполнения всей структуры, время удаления элемента, время поиска элемента.

1. **Теоретические сведения**

Контейнерные классы представляют собой специальные структуры, предназначенные для хранения и организации данных определённым образом. К числу таких контейнеров можно отнести массивы, линейные списки и стеки. Для каждого контейнера предусмотрен набор методов для работы с его элементами, которые не зависят от конкретного типа данных, находящихся внутри. Благодаря этому, один и тот же контейнер может использоваться для хранения данных разных типов. Такая универсальность достигается с помощью шаблонов классов. Поэтому часть стандартной библиотеки C++, включающая контейнеры, алгоритмы и итераторы, носит название **Standard Template Library (STL)** — стандартная библиотека шаблонов.

STL предоставляет набор контейнеров, реализующих распространённые структуры данных, используемые при разработке программного обеспечения. Сюда входят векторы, двусторонние очереди, списки, словари и множества. Эти контейнеры делятся на два основных типа: **последовательные** и **ассоциативные**.

**Последовательные контейнеры** предназначены для хранения конечного набора элементов одного типа в виде упорядоченной последовательности. К этому типу относятся: **vector** (вектор), **deque** (двусторонняя очередь), **list** (список) и адаптеры контейнеров — **stack** (стек), **queue** (очередь) и **priority\_queue** (очередь с приоритетами).

**Ассоциативные контейнеры** обеспечивают быстрый доступ к элементам по заданному ключу. Их структура основана на сбалансированных деревьях. Среди них выделяют: **map** (словарь), **multimap** (словарь с дубликатами), **set** (множество), **multiset** (множество с дубликатами) и **bitset** (битовое множество).

Контейнерные классы предоставляют единый интерфейс для работы с элементами. Операции с одноимёнными методами работают одинаково для различных типов контейнеров, что упрощает их использование. Однако стандарт описывает только интерфейс контейнеров, и различные реализации могут значительно различаться по производительности.

**Итераторы** позволяют обходить элементы контейнеров, не учитывая их внутреннюю структуру. Итератор можно рассматривать как аналог указателя на элемент контейнера. Он обеспечивает перемещение по контейнеру как в прямом, так и в обратном направлении. Главное требование к итератору — возможность ссылаться на элемент контейнера и переходить к следующему элементу.

1. **Ход работы**

**1)** Создан пользовательский класс Matrix для работы с квадратными матрицами (операции ввода, вывода, сложения, вычитания и транспонирования).

**2)** Создан пользовательский стек на основе двусвязного списка (CustomStack) с поддержкой операций добавления, удаления и просмотра элементов.

**3)** Реализованы функции для генерации тестовых данных:

* Генерация векторов целых чисел.
* Генерация векторов объектов класса Matrix.

**4)** Реализованы функции для тестирования производительности стандартного и пользовательского стека:

* Время заполнения стека.
* Время добавления одного элемента.
* Время удаления элемента.

**5)** Реализованы функции для тестирования производительности множества с дубликатами (std::multiset) на основных операциях:

* Время заполнения множества.
* Время добавления элемента.
* Время удаления элемента.
* Время поиска элемента.

1. **Графики**

График зависимости времени заполнения стека от длины стека для типа данных int.

import matplotlib.pyplot as plt  
stack\_len = [10000, 50000, 100000, 150000, 200000]  
STL\_stack = [157, 740, 1608, 2276, 3046]  
custom\_stack = [152, 749, 1472, 2148, 2927]  
plt.plot(stack\_len, STL\_stack, marker='o', linestyle='-', color='r', label='STL стек')  
plt.plot(stack\_len, custom\_stack, marker='o', linestyle='-', color='b', label='Пользовательский стек')  
plt.xlabel('Длина стека')  
plt.ylabel('Время заполнения стека (мкс)')  
plt.title('int')  
plt.grid(True)  
plt.legend()  
plt.show()

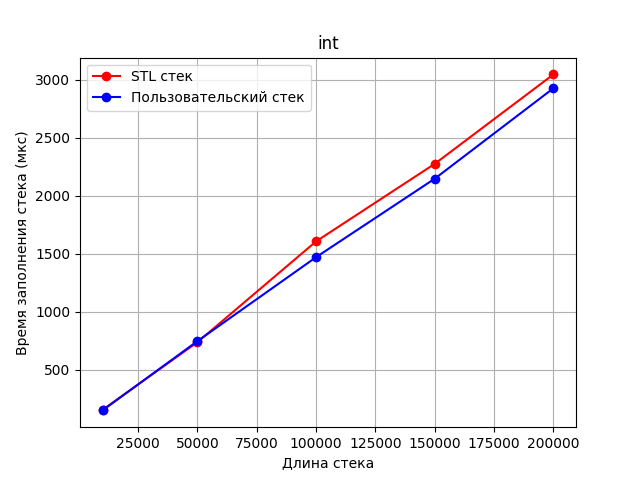


График зависимости времени заполнения стека от длины стека для типа данных Matrix.

import matplotlib.pyplot as plt  
 stack\_len = [10000, 50000, 100000, 150000, 200000]  
STL\_stack = [2980, 12273, 24870, 36084, 49077]  
custom\_stack = [7819, 33058, 68617, 137641, 143111]  
plt.plot(stack\_len, STL\_stack, marker='o', linestyle='-', color='r', label='STL стек')  
plt.plot(stack\_len, custom\_stack, marker='o', linestyle='-', color='b', label='Пользовательский стек')  
plt.xlabel('Длина стека')  
plt.ylabel('Время заполнения стека (мкс)')  
plt.title('Matrix')  
plt.grid(True)  
plt.legend()  
plt.show()

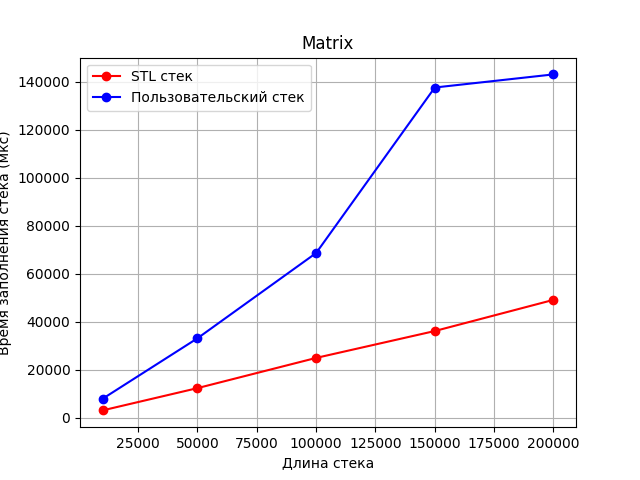


График зависимости времени заполнения стека от длины стека для типа данных int в случае с множеством дубликатов.

import matplotlib.pyplot as plt  
stack\_len = [10000, 50000, 100000, 150000, 200000]  
STL\_stack = [3468, 17911, 39602, 62985, 81786]  
plt.plot(stack\_len, STL\_stack, marker='o', linestyle='-', color='g', label='стек')  
plt.xlabel('Длина стека')  
plt.ylabel('Время заполнения стека (мкс)')  
plt.title('int')  
plt.grid(True)  
plt.legend()  
plt.show()

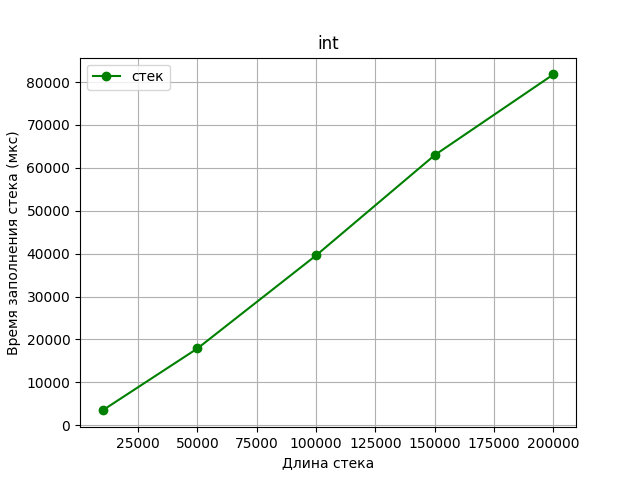
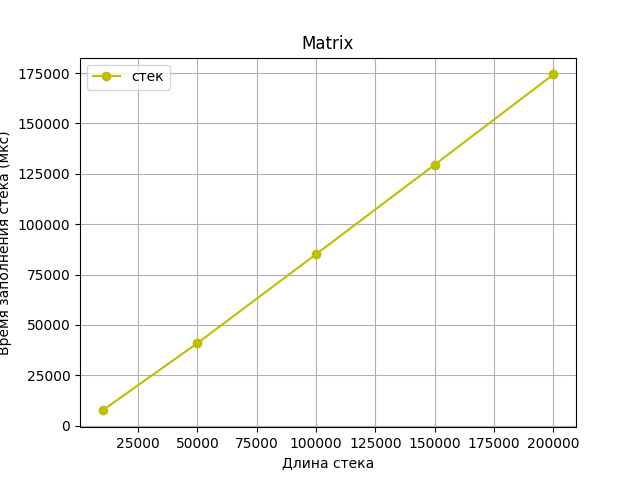


График зависимости времени заполнения стека от длины стека для типа данных Matrix в случае с множеством дубликатов.

import matplotlib.pyplot as plt  
stack\_len = [10000, 50000, 100000, 150000, 200000]  
STL\_stack = [7562, 40960, 85111, 129553, 174343]  
plt.plot(stack\_len, STL\_stack, marker='o', linestyle='-', color='y', label='стек')  
plt.xlabel('Длина стека')  
plt.ylabel('Время заполнения стека (мкс)')  
plt.title('Matrix')  
plt.grid(True)  
plt.legend()  
plt.show()

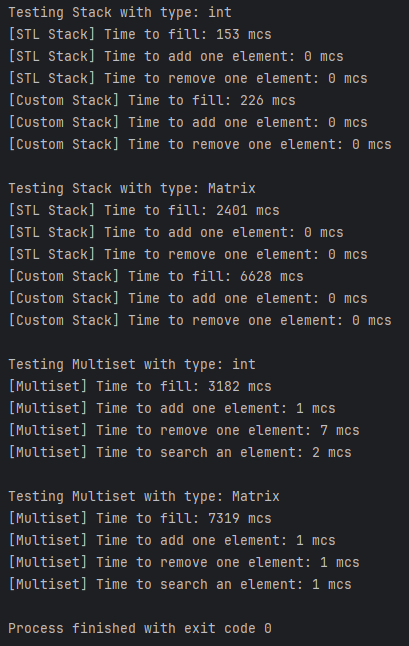
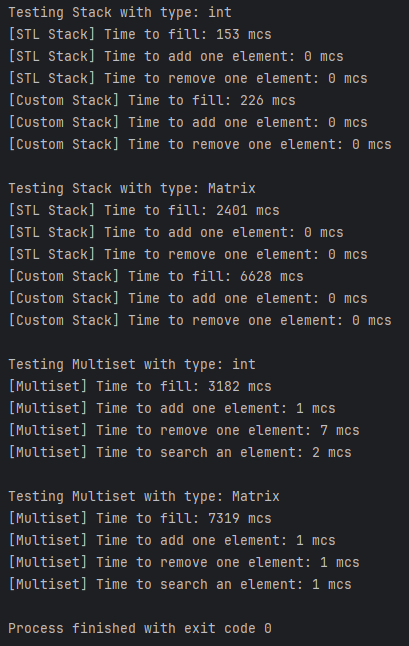


1. **Время добавления элемента к уже существующим**

### ****Анализ времени добавления элемента в стек****

В ходе экспериментов было установлено, что **время добавления элемента в стек не зависит от количества элементов в стеке**. Независимо от того, пуст ли стек или содержит тысячи элементов, операция добавления (push) выполняется за примерно одинаковое время.

Данная особенность объясняется **принципом работы стека как структуры данных**:

1. **Архитектура стека:**
   * Стек представляет собой **структуру данных**, где добавление элемента всегда происходит в **одно и то же фиксированное место** — на вершину стека.
   * Это означает, что нет необходимости проходить по всем предыдущим элементам стека, чтобы добавить новый.
2. **Операция добавления (push):**
   * В стандартной библиотеке C++ (std::stack) стек, как правило, реализуется на основе **контейнера std::deque** или **std::vector**.
   * При добавлении нового элемента стек просто обновляет указатель на вершину, добавляя элемент в конец внутреннего контейнера.
   * В пользовательской реализации на основе **связного списка** новый узел добавляется в начало (или конец, в зависимости от реализации), что также требует фиксированного количества операций.
3. **Отсутствие перераспределения памяти:**
   * Операция добавления в стек не требует **перераспределения памяти для всех элементов.**
   * В случае, если внутренний контейнер, например, std::vector, исчерпал выделенный блок памяти, может произойти перераспределение памяти. Однако, при использовании связного списка такая операция никогда не выполняется, так как каждый новый элемент хранится в отдельной области памяти.
4. **Константная временная сложность:**
   * Теоретическая временная сложность операции push в стеке составляет **O(1)**.
   * Это означает, что время выполнения операции всегда остается постоянным, независимо от размера стека.
5. **Результаты работы программы**

Результаты представлены для стека размером 10000.

1. **Листинг программы**

**matrix.h:**

#ifndef MATRIX\_H  
#define MATRIX\_H  
  
#include <iostream>  
  
class Matrix {  
public:  
 Matrix(); // Конструктор по умолчанию  
 Matrix(int size); // Конструктор с параметром  
 Matrix(const Matrix&); // Конструктор копирования  
 ~Matrix(); // Деструктор  
 bool operator<(const Matrix& other) const;  
 void input();  
 void output() const;  
 Matrix add(const Matrix&) const;  
 Matrix subtract(const Matrix&) const;  
 Matrix multiply(const Matrix&) const;  
 Matrix transpose() const;  
 int determinant() const;  
  
 Matrix& operator=(const Matrix&);  
  
private:  
 int size;  
 int\*\* data;  
 void freeMemory();  
};  
  
#endif

**custom\_stack.h:**

#ifndef CUSTOM\_STACK\_H  
#define CUSTOM\_STACK\_H  
  
#include <iostream>  
#include <vector>  
#include <chrono>  
  
using namespace std;  
using namespace chrono;  
  
template <typename T>  
class CustomStack {  
private:  
 vector<T> stack;  
public:  
 CustomStack() = default;  
 // Добавление элемента  
 void push(const T& elem) {  
 stack.push\_back(elem);  
 }  
  
 // Удаление элемента  
 void pop() {  
 if (!stack.empty()) {  
 stack.pop\_back();  
 }  
 }  
  
 // Получение верхнего элемента  
 T top() const {  
 if (!stack.empty()) {  
 return stack.back();  
 }  
 throw runtime\_error("Stack is empty");  
 }  
  
 // Проверка на пустоту  
 bool empty() const {  
 return stack.empty();  
 }  
  
 // Размер стека  
 size\_t size() const {  
 return stack.size();  
 }  
  
 // Замер времени заполнения стека  
 void measureFillTime(const vector<T>& elements) {  
 auto start = high\_resolution\_clock::now();  
 for (const T& elem : elements) {  
 push(elem);  
 }  
 auto end = high\_resolution\_clock::now();  
 cout << "Time to fill CustomStack: "  
 << duration\_cast<microseconds>(end - start).count() << " mcs" << endl;  
 }  
  
 // Замер времени добавления одного элемента  
 void measurePushTime(const T& element) {  
 auto start = high\_resolution\_clock::now();  
 push(element);  
 auto end = high\_resolution\_clock::now();  
 cout << "Time to add element to CustomStack: "  
 << duration\_cast<microseconds>(end - start).count() << " mcs" << endl;  
 }  
  
 // Замер времени удаления элемента  
 void measurePopTime() {  
 auto start = high\_resolution\_clock::now();  
 pop();  
 auto end = high\_resolution\_clock::now();  
 cout << "Time to remove an element from CustomStack: "  
 << duration\_cast<microseconds>(end - start).count() << " mcs" << endl;  
 }  
};  
  
#endif // CUSTOM\_STACK\_H

**methods.cpp:**

#include "matrix.h"  
  
#include <cmath>  
using namespace std;  
  
//Конструктор по умолчанию  
Matrix::Matrix() : size(0), data(nullptr) {}  
  
//Конструктор с параметром размера  
Matrix::Matrix(int size) : size(size) {  
 data = new int\* [size];  
 for (int i = 0; i < size; i++) {  
 data[i] = new int[size];  
 }  
}  
  
//Конструктор копирования  
Matrix::Matrix(const Matrix& other) : size(other.size) {  
 data = new int\* [size];  
 for (int i = 0; i < size; i++) {  
 data[i] = new int[size];  
 for (int j = 0; j < size; j++) {  
 data[i][j] = other.data[i][j];  
 }  
 }  
}  
  
//Деструктор  
Matrix::~Matrix() {  
 freeMemory();  
}  
  
//Освобождение памяти  
void Matrix::freeMemory() {  
 for (int i = 0; i < size; i++) {  
 delete[] data[i];  
 }  
 delete[] data;  
}  
  
// Ввод матрицы  
void Matrix::input() {  
 cout << "Add matrix numbers " << size << "x" << size << ":" << endl;  
 for (int i = 0; i < size; i++) {  
 for (int j = 0; j < size; j++) {  
 while (true) {  
 int value;  
 cin >> value;  
 if (cin.fail()) {  
 cin.clear();  
 cin.ignore(numeric\_limits<streamsize>::max(), '\n');  
 cout << "n/a" << endl;  
 exit(1);  
 } else {  
 data[i][j] = value;  
 break;  
 }  
 }  
 }  
 }  
}  
  
//Вывод матрицы  
void Matrix::output() const {  
 for (int i = 0; i < size; i++) {  
 for (int j = 0; j < size; j++) {  
 cout << data[i][j] << " ";  
 }  
 cout << endl;  
 }  
}  
  
//Операция сложения  
Matrix Matrix::add(const Matrix& other) const {  
 Matrix result(size);  
 for (int i = 0; i < size; i++) {  
 for (int j = 0; j < size; j++) {  
 result.data[i][j] = data[i][j] + other.data[i][j];  
 }  
 }  
 return result;  
}  
  
//Операция вычитания  
Matrix Matrix::subtract(const Matrix& other) const {  
 Matrix result(size);  
 for (int i = 0; i < size; i++) {  
 for (int j = 0; j < size; j++) {  
 result.data[i][j] = data[i][j] - other.data[i][j];  
 }  
 }  
 return result;  
}  
  
//Операция умножения  
Matrix Matrix::multiply(const Matrix& other) const {  
 Matrix result(size);  
 for (int i = 0; i < size; i++) {  
 for (int j = 0; j < size; j++) {  
 result.data[i][j] = 0;  
 for (int k = 0; k < size; k++) {  
 result.data[i][j] += data[i][k] \* other.data[k][j];  
 }  
 }  
 }  
 return result;  
}  
  
//Транспонирование матрицы  
Matrix Matrix::transpose() const {  
 Matrix result(size);  
 for (int i = 0; i < size; i++) {  
 for (int j = 0; j < size; j++) {  
 result.data[i][j] = data[j][i];  
 }  
 }  
 return result;  
}  
  
//Вычисление детерминанта  
int Matrix::determinant() const {  
 if (size == 1) {  
 return data[0][0];  
 }  
 if (size == 2) {  
 return (data[0][0] \* data[1][1] - data[1][0] \* data[0][1]);  
 } else {  
 int det = 0;  
 for (int x = 0; x < size; x++) {  
 Matrix submatrix(size - 1);  
 int subi = 0;  
 for (int i = 1; i < size; i++) {  
 int subj = 0;  
 for (int j = 0; j < size; j++) {  
 if (j == x) {  
 continue;  
 }  
 submatrix.data[subi][subj] = data[i][j];  
 subj++;  
 }  
 subi++;  
 }  
 det += (pow(-1, x) \* data[0][x] \* submatrix.determinant());  
 }  
 return det;  
 }  
}  
  
//Оператор присваивания  
Matrix& Matrix::operator=(const Matrix& other) {  
 if (this != &other) {  
 freeMemory();  
 size = other.size;  
 data = new int\* [size];  
 for (int i = 0; i < size; i++) {  
 data[i] = new int[size];  
 for (int j = 0; j < size; j++) {  
 data[i][j] = other.data[i][j];  
 }  
 }  
 }  
 return \*this;  
}  
  
bool Matrix::operator<(const Matrix& other) const {  
 if (size != other.size) {  
 return size < other.size;  
 }  
 for (int i = 0; i < size; ++i) {  
 for (int j = 0; j < size; ++j) {  
 if (data[i][j] != other.data[i][j]) {  
 return data[i][j] < other.data[i][j];  
 }  
 }  
 }  
 return false;  
}

**main.cpp:**

#include <iostream>  
#include <stack>  
#include <set>  
#include <random>  
#include <chrono>  
#include <vector>  
#include "matrix.h"  
#include "custom\_stack.h"  
  
using namespace std;  
using namespace chrono;  
  
// Генерация случайных матриц  
vector<Matrix> generateMatrixData(mt19937& gen, int count, int size) {  
 uniform\_int\_distribution<int> valueDist(1, 100);  
 vector<Matrix> data(count);  
  
 for (Matrix& matrix : data) {  
 matrix = Matrix(size);  
 for (int i = 0; i < size; ++i) {  
 for (int j = 0; j < size; ++j) {  
 matrix.add(Matrix(size));  
 }  
 }  
 }  
 return data;  
}  
  
// Генерация случайных чисел типа int  
vector<int> generateIntData(mt19937& gen, int count) {  
 uniform\_int\_distribution<int> dist(1, 1000);  
 vector<int> data(count);  
 for (int& value : data) {  
 value = dist(gen);  
 }  
 return data;  
}  
  
// Тестирование стеков  
template <typename T>  
void testStacks(const string& typeName, const vector<T>& elements) {  
 T newElement = elements.back();  
  
 cout << "\nTesting Stack with type: " << typeName << "\n";  
  
 // STL стек  
 stack<T> stdStack;  
 auto start = high\_resolution\_clock::now();  
 for (const T& elem : elements) {  
 stdStack.push(elem);  
 }  
 auto end = high\_resolution\_clock::now();  
 cout << "[STL Stack] Time to fill: "  
 << duration\_cast<microseconds>(end - start).count() << " mcs" << endl;  
  
 start = high\_resolution\_clock::now();  
 stdStack.push(newElement);  
 end = high\_resolution\_clock::now();  
 cout << "[STL Stack] Time to add one element: "  
 << duration\_cast<microseconds>(end - start).count() << " mcs" << endl;  
  
 start = high\_resolution\_clock::now();  
 stdStack.pop();  
 end = high\_resolution\_clock::now();  
 cout << "[STL Stack] Time to remove one element: "  
 << duration\_cast<microseconds>(end - start).count() << " mcs" << endl;  
  
 // Пользовательский стек  
 CustomStack<T> customStack;  
 start = high\_resolution\_clock::now();  
 for (const T& elem : elements) {  
 customStack.push(elem);  
 }  
 end = high\_resolution\_clock::now();  
 cout << "[Custom Stack] Time to fill: "  
 << duration\_cast<microseconds>(end - start).count() << " mcs" << endl;  
  
 start = high\_resolution\_clock::now();  
 customStack.push(newElement);  
 end = high\_resolution\_clock::now();  
 cout << "[Custom Stack] Time to add one element: "  
 << duration\_cast<microseconds>(end - start).count() << " mcs" << endl;  
  
 start = high\_resolution\_clock::now();  
 customStack.pop();  
 end = high\_resolution\_clock::now();  
 cout << "[Custom Stack] Time to remove one element: "  
 << duration\_cast<microseconds>(end - start).count() << " mcs" << endl;  
}  
  
// Тестирование множества с дубликатами  
template <typename T>  
void testMultiset(const string& typeName, const vector<T>& elements) {  
 T newElement = elements.back();  
  
 cout << "\nTesting Multiset with type: " << typeName << "\n";  
  
 multiset<T> stdMultiset;  
  
 auto start = high\_resolution\_clock::now();  
 for (const T& elem : elements) {  
 stdMultiset.insert(elem);  
 }  
 auto end = high\_resolution\_clock::now();  
 cout << "[Multiset] Time to fill: "  
 << duration\_cast<microseconds>(end - start).count() << " mcs" << endl;  
  
 start = high\_resolution\_clock::now();  
 stdMultiset.insert(newElement);  
 end = high\_resolution\_clock::now();  
 cout << "[Multiset] Time to add one element: "  
 << duration\_cast<microseconds>(end - start).count() << " mcs" << endl;  
  
 start = high\_resolution\_clock::now();  
 stdMultiset.erase(stdMultiset.begin());  
 end = high\_resolution\_clock::now();  
 cout << "[Multiset] Time to remove one element: "  
 << duration\_cast<microseconds>(end - start).count() << " mcs" << endl;  
  
 start = high\_resolution\_clock::now();  
 auto it = stdMultiset.find(newElement);  
 end = high\_resolution\_clock::now();  
 cout << "[Multiset] Time to search an element: "  
 << duration\_cast<microseconds>(end - start).count() << " mcs" << endl;  
}  
  
int main() {  
 const int initialSize = 10000;  
 const int matrixSize = 3;  
  
 mt19937 gen(42);  
  
 // Генерация данных  
 vector<int> intData = generateIntData(gen, initialSize);  
 vector<Matrix> matrixData = generateMatrixData(gen, initialSize, matrixSize);  
  
 testStacks("int", intData);  
 testStacks("Matrix", matrixData);  
  
 testMultiset("int", intData);  
 testMultiset("Matrix", matrixData);  
  
 return 0;  
}

1. **Вывод**

В ходе выполнения 8 лабораторной работы была разработана программа, которая заполняет данными типа int и Matrix пользовательский класс и класс из библиотеки STL. Был проведён временной анализ для различных операций со стеками, а именно время добавления и удаления элемента, время заполнения стека и время поиска элемента.