Министерство науки и высшего образования Российской Федерации
Федеральное государственное бюджетное образовательное учреждение высшего образования
«Российский химико-технологический университет имени Д.И. Менделеева»
Кафедра информационных компьютерных технологий

ОТЧЕТ ПО ЛАБОРАТОРНОЙ РАБОТЕ № 3

Вариант 3

Выполнила студентка группы .....КС-33..... Георгиевская Анастасия Игоревна
Ссылка на репозиторий: .....
https://github.com/MUCTR-IKT-CPP/GeorgievskayaAA_33_alg/tree/main/lab%203

Приняли: .....Пысин Максим Дмитриевич
.....Лобанов Алексей Владимирович

Дата сдачи: 12.03.2025

Оглавление

Описание задачи.

Задание:

Написать две реализации очереди (через односвязный список и через 2 стека). Добавление в конец. Взятие сначала. Все структуры должны:

- Использовать шаблонный подход, обеспечивая работу контейнера с произвольными данными.
- Реализовывать свой итератор предоставляющий стандартный для языка механизм работы с ним (для C++ это операции ++ и !=, для python это)
- Обеспечивать работу стандартных библиотек и конструкции for each если она есть в языке, если их нет, то реализовать собственную функцию использующую итератор.
- Проверку на пустоту и подсчет количества элементов.

Для демонстрации работы структуры необходимо создать набор тестов(под тестом понимается функция, которая создаёт структуру, проводит операцию или операции над структурой и удаляет структуру):

- заполнение контейнера 1000 целыми числами в диапазоне от -1000 до 1000 и подсчет их суммы, среднего, минимального и максимального.
- Провести проверку работы операций вставки и изъятия элементов на коллекции из 10 строковых элементов.
- заполнение контейнера 100 структур содержащих фамилию, имя, отчество и дату рождения (от 01.01.1980 до 01.01.2020) значения каждого поля генерируются случайно из набора заранее заданных. После заполнение необходимо найти всех людей младше 20 лет и старше 30 и создать новые структуры содержащие результат фильтрации, проверить выполнение на правильность подсчётом кол-ва элементов не подходящих под условие в новых структурах

Провести тесты:

- Инверсировать содержимое контейнера заполненного отсортированными по возрастанию элементами не используя операцию перемещения при помощи итератора, а только операторы изъятия и вставки.
- Сравнить две реализации между собой (Сравнить на основании скорости выполнения операции вставки и изъятия на контейнере, использования памяти на все элементы), тестировать для коллекции состоящей из 10000 элементов.

Описание очереди и ее реализаций, предложенных в задании.

Очередь — это структура данных, работающая по принципу "первым пришел — первым ушел" (FIFO).

Очередь на основе односвязного списка:

Реализация очереди через односвязный список представляет собой динамическую структуру данных, в которой каждый элемент (узел) содержит два поля:

- значение (данные);
- ссылка на следующий элемент (или nullptr, если элемент последний).

Вставка в очередь осуществляется путем добавления нового элемента в конец списка, а изъятие — путем удаления элемента с начала списка. Операции вставки и удаления выполняются за время O(1), так как для добавления или удаления элементов достаточно просто изменить указатели на начало и конец списка.

Преимущества:

- Быстрая вставка и удаление элементов.
- Нет необходимости заранее выделять память для очереди — память выделяется по мере добавления элементов.

Недостатки:

- Необходимо хранить указатели для каждого элемента, что незначительно -увеличивает объем памяти.

Очередь на основе двух стеков:

Очередь может быть реализована через два стека, используя два метода:

- Метод вставки: элементы добавляются в первый стек.
- Метод изъятия: элементы извлекаются из второго стека. Если второй стек пуст, элементы из первого стека переносятся во второй, и затем извлекаются из второго стека.

Операции вставки и изъятия обычно имеют сложность O(1) в случае вставки, но изъятие может быть выполнено за время O(n), если второй стек пуст, и все элементы из первого стека переносятся во второй. Однако при амортизированном анализе можно утверждать, что время операции изъятия составляет O(1) в среднем, поскольку каждый элемент переносится во второй стек только один раз.

Преимущества:

- Не нужно хранить дополнительные указатели, как в случае с односвязным списком.
- Очередь может работать быстрее при малом числе элементов, поскольку не требуется управление динамическим выделением памяти.

Недостатки:

- Использует больше памяти, так как необходимо хранить два стека.

Выполнение задачи.

Для выполнения задачи лабораторной был использован язык C++.

Для удобства каждый вариант реализации рассмотрен отдельно.

Очередь через односвязный список

```
#include <iostream>
#include <stdexcept>
#include <random>
#include <limits>
#include <string>
#include <ctime>
#include <chrono>
#include <sstream>
#include <memory>
```

- Подключаемые библиотеки
    - <iostream> — ввод и вывод результатов в консоль.
    - <stdexcept> — обработка исключений. Для выбрасывания исключений при попытке извлечь элемент из пустой очереди.
    - <random> — .генерация случайных чисел.
    - <limits> — работа с предельными значениями типов данных.
    - <string> — работа со строками.
    - <ctime> — работа с датой и временем для подсчета возраста.
    - <chrono> — измерение времени .
    - <sstream> — формирование строки с датой рождения.
    - <memory> — динамическое управление памятью, работа с умными указателями.

```
template <typename T>
class Node {
    public:
        T data;
        Node* next;

        Node(const T& data) : data(data), next(nullptr) {}
};
```

- Шаблонный класс для представления узла односвязного списка
    - Хранит данные типа T и указатель на следующий узел
    - Конструктор инициализирует данные и устанавливает указатель на nullptr (следующий узел отсутствует).

```
template <typename T>
class Queue {
    private:
        Node<T>* front;
        Node<T>* back;
```

```
    size_t size;

public:
    // Конструктор
    Queue() : front(nullptr), back(nullptr), size(0) {}

    //деструктор
    ~Queue() {
        while (!isEmpty()) {
            dequeue();
        }
    }

    //проверка на пустоту
    bool isEmpty() const {
        return size == 0;
    }

    //подсчет элементов
    size_t getSize() const {
        return size;
    }

    //добавление в конец
    void enqueue(const T& value) {
        Node<T>* newNode = new Node<T>(value);
        if (isEmpty()) {
            front = back = newNode;
        }
        else {
            back->next = newNode;
            back = newNode;
        }
        size++;
    }

    //взятие из начала
    T dequeue() {
        if (isEmpty()) {
            throw std::out_of_range("Queue is empty");
        }
        Node<T>* temp = front;
        T data = front->data;
        front = front->next;
        delete temp;
        size--;
        if (isEmpty()) {
            back = nullptr;
        }
        return data;
    }

    //итератор
    class Iterator {
```

```
        private:
            Node<T>* current;

        public:
        Iterator(Node<T>* node) : current(node) {}

        T& operator*() {
            return current->data;
        }

        Iterator& operator++() {
            if (current) {
                current = current->next;
            }
            return *this;
        }

        bool operator!=(const Iterator& other) const {
            return current != other.current;
        }
    };

    //начало итератора
    Iterator begin(){
        return Iterator(front);
    }

    //конец итератора
    Iterator end(){
        return Iterator(nullptr);
    }

    // Функция для обхода очереди
    void forEach(void (*func)(T&)) {
        for (Iterator it = begin(); it != end(); ++it){
            func(*it);
        }
    }
};
```

- Шаблонный класс для реализации очереди на основе связного списка.
    - Конструктор инициализирует пустую очередь.
    - Деструктор очищает очередь, удаляя все элементы.
    - isEmpty() — проверяет, пуста ли очередь.
    - getSize() — возвращает количество элементов в очереди.
    - enqueue() — добавляет элемент в конец очереди.
    - dequeue() — извлекает элемент из начала очереди и возвращает его.
    - Iterator — вложенный класс-итератор для обхода элементов очереди.
    - begin() и end() — возвращают итераторы для начала и конца очереди соответственно.
    - forEach() — выполняет функцию для каждого элемента очереди.

```
void test1() {
    Queue<int> q;
    std::random_device rd;
    std::mt19937 gen(rd());
    std::uniform_int_distribution<int> dis(-1000, 1000);

    //добавление 1000 эл-тов в очередь
    for (int i = 0; i < 1000; ++i) {
        q.enqueue(dis(gen));
    }

    // Ининциализация переменных для подсчета
    long long sum = 0;
    int min = std::numeric_limits<int>::max();
    int max = std::numeric_limits<int>::min();

    // Подсчет суммы, минимума, максимума
    for (auto it = q.begin(); it != q.end(); ++it) {
        sum += *it;
        if (*it < min) min = *it;
        if (*it > max) max = *it;
    }

    double average = static_cast<double>(sum) / q.getSize();

    // Вывод результатов
    std::cout << "====== TEST 1 ======" << std::endl;
    std::cout << "Queue statistics after adding 1000 elements:" << std::endl;
    std::cout << "Sum: " << sum << std::endl;
    std::cout << "Average: " << average << std::endl;
    std::cout << "Min: " << min << std::endl;
    std::cout << "Max: " << max << std::endl;

    // Очистка очереди
    while (!q.isEmpty()) {
        q.dequeue();
    }
}
```

- Функция тестирования очереди 1000 целых чисел
    - Вставляет 1000 случайных чисел в очередь, а затем подсчитывает сумму, минимум, максимум и среднее значение этих чисел, выводя результаты в консоль.

```
void test2() {
    Queue<std::string> q;

    q.enqueue("First");
    q.enqueue("Second");
    q.enqueue("Third");
    q.enqueue("Fourth");
    q.enqueue("Fifth");
    q.enqueue("Sixth");
    q.enqueue("Seventh");
```

```
    q.enqueue("Eighth");
    q.enqueue("Ninth");
    q.enqueue("Tenth");

    std::cout << "====== TEST 2 ======" << std::endl;
    std::cout << "Queue after enqueueing 10 string elements:" << std::endl;

    // Извлечение и вывод строк из очереди
    while (!q.isEmpty()) {
        std::cout << q.dequeue() << std::endl;
    }
}
```

- Функция тестирования очереди из 10 строк
    - Вставляет 10 строк в очередь, затем извлекает их и выводит на экран

```
struct Person {
    std::string lastName;
    std::string firstName;
    std::string patronymic;
    std::string birthDate; // Формат: "ДД.ММ.ГГГГ"
};

class RandomDataGeneration {
    private:
        std::vector<std::string> lastNames;
        std::vector<std::string> firstNames;
        std::vector<std::string> patronymics;
        std::mt19937 gen;

    public:
        RandomDataGeneration() {
            std::random_device rd;
            gen = std::mt19937(rd());

            lastNames   =   {"Иванов",   "Петров",   "Сидоров",   "Кузнецов",
"Новиков"};
            firstNames = {"Иван", "Петр", "Алексей", "Дмитрий", "Максим"};
            patronymics = {"Иванович", "Петрович", "Алексеевич", "Дмитриевич",
"Максимович"};
        }

        std::string getRandomElement(const std::vector<std::string>& vec) {
            std::uniform_int_distribution<int> dis(0, vec.size() - 1);
            return vec[dis(gen)];
        }

        std::string getRandomBirthDate() {
            std::uniform_int_distribution<int> dayDis(1, 28);  // день от 1 до
28 (чтобы избежать сложностей с месяцами)
            std::uniform_int_distribution<int> monthDis(1, 12); // месяц от 1
до 12
            std::uniform_int_distribution<int> yearDis(1980, 2020); // год от
1980 до 2020
```

```
            int day = dayDis(gen);
            int month = monthDis(gen);
            int year = yearDis(gen);

            std::ostringstream oss;
            oss << (day < 10 ? "0" : "") << day << "."
                << (month < 10 ? "0" : "") << month << "."
                << year;
            return oss.str();
        }

        Person generateRandomPerson() {
            Person p;
            p.lastName = getRandomElement(lastNames);
            p.firstName = getRandomElement(firstNames);
            p.patronymic = getRandomElement(patronymics);
            p.birthDate = getRandomBirthDate();
            return p;
        }
};
```

- Класс для генерации случайных данных о людях (имя, фамилия, отчество, дата рождения).
    - Метод generateRandomPerson(), который генерирует случайного человека с использованием случайных значений для фамилии, имени, отчества и даты рождения.

```
int calculateAge(const std::string& birthDate) {
    // Разбираем дату на день, месяц и год
    int day, month, year;
    std::sscanf(birthDate.c_str(), "%d.%d.%d", &day, &month, &year);

    // Получаем текущую дату
    auto now = std::chrono::system_clock::now();
    auto now_tm = std::chrono::system_clock::to_time_t(now);
    std::tm tm_now = *std::localtime(&now_tm);

    int age = tm_now.tm_year + 1900 - year; // текущий год минус год рождения

    // Корректировка по месяцам и дням
    if (tm_now.tm_mon + 1 < month || (tm_now.tm_mon + 1 == month &&
tm_now.tm_mday < day)) {
        age--;
    }

    return age;
}
```

- Функция для вычисления возраста человека на основе его даты рождения.
    - Вход: указатель на строку с датой рождения.
    - Выход: посчитанный на сегодня возраст.

```
void test3() {
```

```
    RandomDataGeneration generator;
    Queue<Person> people;
    Queue<Person> under20, over30;

    // Генерация 100 случайных людей
    for (int i = 0; i < 100; ++i) {
        people.enqueue(generator.generateRandomPerson());
    }

    // Фильтрация людей младше 20 лет и старше 30 лет
    while (!people.isEmpty()) {
        Person person = people.dequeue();
        if (person.age < 20) {
            under20.enqueue(person);
        } else if (person.age > 30) {
            over30.enqueue(person);
        }
    }
}
    // Вывод результатов
    std::cout << "====== TEST 3 ======" << std::endl;
    std::cout << "People under 20 years: " << under20.size() << std::endl;
    std::cout << "People over 30 years: " << over30.size() << std::endl;

    // Подсчет людей, которые не попали в фильтрацию
    int notMatched = people.size() - under20.size() - over30.size();
    std::cout << "People not matched (between 20 and 30 years): " << notMatched
<< std::endl;
}
```

- Функция тестирования очереди из данных о людях, а так же выбор по необходимому критерию.
    - Генерирует 100 случайных людей и выводит информацию о людях, которые младше 20 лет или старше 30 лет.

```
template <typename T>
void invertQueue(Queue<T>& q) {
    size_t n = q.getSize();
    Queue<T> tempQueue;

    for (size_t i = 0; i < n; ++i) {
        T value = q.dequeue();  // Извлекаем элемент
        tempQueue.enqueue(value);  // Вставляем его в новую очередь
    }

    // Теперь элементы в tempQueue инвертированы по порядку
    // Переносим обратно в исходную очередь
    while (!tempQueue.isEmpty()) {
        q.enqueue(tempQueue.dequeue());
    }
}
```

- Шаблонная функция инвертирования порядка элементов в очереди.

```
  template <typename T>
  void testQueueOperations() {
      Queue<T> q;

      // Измеряем время выполнения операции вставки
      auto startInsert = std::chrono::high_resolution_clock::now();
      for (int i = 1; i <= 10000; ++i) {
          q.enqueue(i);  // Заполняем очередь отсортированными элементами
      }
      auto endInsert = std::chrono::high_resolution_clock::now();
      std::chrono::duration<double> insertDuration = endInsert - startInsert;
      std::cout << "Time to insert 10000 elements: " << insertDuration.count()
<< " seconds" << std::endl;

      // Измеряем время выполнения операции изъятия
      auto startDequeue = std::chrono::high_resolution_clock::now();
      for (int i = 1; i <= 10000; ++i) {
          q.dequeue();  // Извлекаем элементы
      }
      auto endDequeue = std::chrono::high_resolution_clock::now();
      std::chrono::duration<double> dequeueDuration = endDequeue - startDequeue;
      std::cout << "Time to dequeue 10000 elements: " << dequeueDuration.count()
<< " seconds" << std::endl;

      // Память, занимаемая очередью
      size_t memoryUsage = sizeof(Queue<T>) + sizeof(Node<T>) * q.getSize();
      std::cout << "Memory usage for 10000 elements: " << memoryUsage << " bytes"
<< std::endl;
  }
```

- Функция тестирования вставки и извлечения элементов из очереди.
    - Измерение времени выполнения операций.
    - Подсчет занимаемой памяти.
    - Вывод результатов

```
  int main(){
      test1();
      test2();
      test3();

      // Заполнение очереди отсортированными элементами и инвертирование
      Queue<int> q;
      for (int i = 1; i <= 10000; ++i) {
          q.enqueue(i);   // Заполняем очередь отсортированными по возрастанию
элементами
      }

      // Инвертируем очередь
      invertQueue(q);

      // Тестирование вставки и изъятия
```

```
    std::cout << "====== TEST 4 ======" << std::endl;
    testQueueOperations<int>();
    return 0;
}
```

- Основная функция
    - test1(), test2(), test3() выполняются для тестирования различных операций с очередью.
    - Вводятся 10 000 элементов в очередь и затем выполняется операция инвертирования очереди с использованием функции invertQueue().
    - Далее тестируются операции с очередью (вставка и извлечение элементов) с замером времени и использованием памяти.

Результаты работы программы

```
====== TEST 1 ======
Queue statistics after adding 1000 elements:
Sum: -6623
Average: -6.623
Min: -999
Max: 997
====== TEST 2 ======
Queue after enqueueing 10 string elements:
First
Second
Third
Fourth
Fifth
Sixth
Seventh
Eighth
Ninth
Tenth
====== TEST 3 ======
People under 20 years: 40
People over 30 years: 36
People not matched (between 20 and 30 years): 24
====== TEST 4 ======
Time to insert 10000 elements: 0.000304752 seconds
Time to dequeue 10000 elements: 0.000133095 seconds
Memory usage for 10000 elements: 24 bytes
```

Очередь через два стека

```
#include <iostream>
#include <stdexcept>
#include <random>
#include <limits>
#include <string>
#include <vector>
#include <ctime>
#include <chrono>
#include <sstream>
#include <memory>
#include <stack>
```

- Подключаемые библиотеки
    - <iostream> — ввод и вывод результатов в консоль.

- <stdexcept> — обработка исключений. Для выбрасывания исключений при попытке извлечь элемент из пустой очереди.
- <random> — .генерация случайных чисел.
- <limits> — работа с предельными значениями типов данных.
- <string> — работа со строками.
- <vector> — работа с динамическими массивами.
- <ctime> — работа с датой и временем для подсчета возраста.
- <chrono> — измерение времени .
- <sstream> — формирование строки с датой рождения.
- <memory> — динамическое управление памятью, работа с умными указателями.
- <stack> — работа с контейнером стека.

```
template <typename T>
class Queue {
private:
    std::stack<T> stack1;  // Стек для входящих элементов
    std::stack<T> stack2;  // Стек для исходящих элементов
    size_t size;           // Размер очереди

public:
    Queue() : size(0) {}

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

    size_t getSize() const {
        return size;
    }

    void enqueue(const T& value) {
        stack1.push(value);  // Элементы добавляются в первый стек
        size++;
    }

    T dequeue() {
        if (isEmpty()) {
            throw std::out_of_range("Queue is empty");
        }

        // Если второй стек пуст, переносим элементы из первого в второй
        if (stack2.empty()) {
            while (!stack1.empty()) {
                stack2.push(stack1.top());
                stack1.pop();
            }
        }

        // Извлекаем элемент из второго стека
```

```
    T data = stack2.top();
    stack2.pop();
    size--;
    return data;
}

// Итератор для обхода очереди
class Iterator {
private:
    std::stack<T> tempStack;  // Временный стек для обхода

public:
    Iterator(std::stack<T> stack) {
        while (!stack.empty()) {
            tempStack.push(stack.top());
            stack.pop();
        }
    }

    bool hasNext() {
        return !tempStack.empty();
    }

    T next() {
        T value = tempStack.top();
        tempStack.pop();
        return value;
    }

    // Перегрузка оператора *
    T operator*() {
        return tempStack.top();
    }

    // Перегрузка оператора ++
    Iterator& operator++() {
        tempStack.pop();
        return *this;
    }

    // Перегрузка оператора !=
    bool operator!=(const Iterator& other) const {
        return !tempStack.empty() || !other.tempStack.empty();
    }
};

Iterator begin() {
    std::stack<T> tempStack1 = stack1;
    std::stack<T> tempStack2 = stack2;
    while (!tempStack2.empty()) {
        tempStack1.push(tempStack2.top());
        tempStack2.pop();
    }
    return Iterator(tempStack1);
```

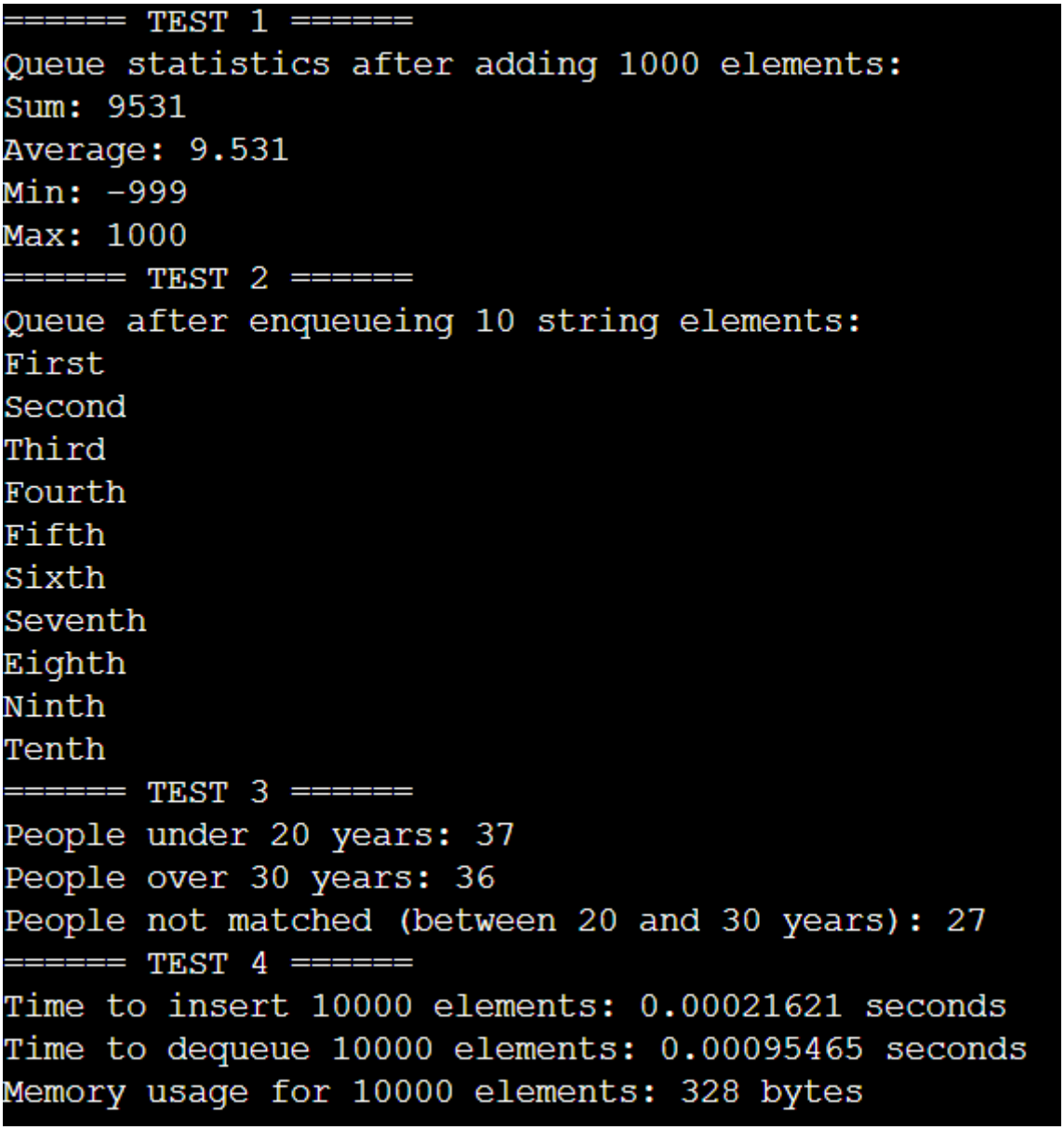
```
    }

    Iterator end() {
        return Iterator({});
    }

    void forEach(void (*func)(T&)) {
        for (Iterator it = begin(); it != end(); ++it) {
            func(*it);
        }
    }
};
```

- Класс, реализующий очередь с использованием двух стеков. В нем используются два стека: один для хранения входящих элементов (stack1), а другой для исходящих (stack2).
    - Queue() — Конструктор, инициализирующий пустую очередь.
    - isEmpty() — Функция, которая возвращает true, если очередь пуста.
    - getSize() — Функция для получения размера очереди.
    - enqueue(const T& value) — Функция для добавления элемента в очередь (элемент добавляется в stack1).
    - dequeue() — Функция для извлечения элемента из очереди. Если stack2 пуст, элементы из stack1 переносятся в stack2, после чего элемент извлекается из stack2.
    - Iterator — Вложенный класс для итерации по очереди. Используется для обхода элементов очереди, переноса их из стеков в временный стек и возвращения элементов через перегруженные операторы.
    - begin() и end() — Функции для получения итераторов, указывающих на начало и конец очереди.
    - forEach() — Функция для применения функции к каждому элементу очереди. Использует итератор для обхода всех элементов.

Функции тестирования работы очереди остались неизменными, За исключением последнего пункта функции testQueueOperations(). Так как теперь очередь реализована с помощью стеков, то выделение памяти происходит по размеру стека.

```
    size_t memoryUsage = sizeof(Queue<int>) + sizeof(std::stack<int>) * 2; // Два стека
```

Результаты работы программы



Данные программы реализуют создание очереди двумя разыми способами и проводят тестирование её работы. Для каждой реализации проводится 4 теста работоспособности.

Заключение.

В ходе выполнения лабораторной работы была реализована очередь двумя способами на языке C++ и приведено ее тестирование на различных данных. Для сравнительного анализа было предложено провести измерение времени работы и объем занимаемой памяти обоих вариантов.

После проведения теста были получены следующие данные:

	Время, с		объем памяти, байт
	вставки	изъятия	
односвязный список	0,00030	0,00013	24
два стека	0,00022	0,00095	328

Анализ полученных данных

1. Время вставки

   Здесь очередь, реализованная через два стека, работает быстрее, чем очередь на основе односвязного списка. Это объясняется тем, что вставка в стек выполняется за O(1) время, и нет необходимости управлять указателями (как в случае с односвязным списком), что позволяет ускорить процесс.

2. Время изъятия

   Изъятие из очереди на основе односвязного списка происходит значительно быстрее. Это связано с тем, что изъятие элемента в списке — операция O(1) (необходимо лишь обновить указатель на начало списка). В случае с двумя стеками, изъятие может занять больше времени, особенно если требуется перенести элементы из первого стека во второй, что увеличивает время операции.

3. Объем памяти

   Очередь на основе двух стеков требует значительно больше памяти. Это объясняется тем, что для каждого стека нужно хранить отдельный массив, который может занимать гораздо больше памяти по сравнению с односвязным списком. В односвязном списке каждый элемент хранит лишь указатель и данные, что экономит память.

Таким образом, получается, что очередь через односвязный список имеет значительно меньшее время изъятия и требует меньше памяти, а очередь через два стека работает быстрее при вставке, но имеет более высокое время изъятия и использует больше памяти.

Можно предположить, что очередь через односвязный список больше подходит для приложений, где важна высокая производительность при извлечении данных и ограничена память, а очередь через два стека может быть более эффективна в случаях, когда важно минимизировать время вставки и изъятие происходит редко.