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

Выполнил студент группы КС-36 Полковникова Д.Д.

Ссылка на репозиторий: https://github.com/MUCTR-IKT-CPP/Polkovnikova\_CS-36

Приняли: Пысин Максим Дмитриевич

Краснов Дмитрий Олегович

Лобанов Алексей Владимирович

Крашенинников Роман Сергеевич

Дата сдачи: 10.03.2025

Оглавление

[Описание задачи. 2](#_Toc192415473)

[Описание метода/модели. 3](#_Toc192415474)

[Выполнение задачи. 4](#_Toc192415475)

[Заключение. 11](#_Toc192415476)

# Описание задачи.

**Вариант 3:**

Написать две реализации очереди (Очередь через односвязный список и очередь через два стека(стек из библиотеки))

* Добавление в конец
* Взятие с начала

В рамках лабораторной работы необходимо изучить и реализовать очередь, при этом структура должна:

* Использовать шаблонный подход, обеспечивая работу контейнера с произвольными данными;
* Реализовывать свой итератор предоставляющий стандартный для языка механизм работы с ним (для С++ это операции ++ и операция !=);
* 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. Тесты по вариантам:

2, 3. Стек и Очередь

* 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 метода/модели.

**Основные отличия и пояснения:**

1. **Реализация через два стека**:
   * Используются два объекта std::stack: inStack для добавления элементов и outStack для извлечения.
   * Метод transferToOutStack перекладывает элементы из inStack в outStack, если последний пуст, чтобы обеспечить порядок очереди (FIFO).
2. **Принцип работы**:
   * enqueue: просто добавляет элемент в inStack.
   * dequeue и peek: перед выполнением проверяют и при необходимости перемещают элементы в outStack, чтобы первый добавленный элемент оказался сверху.
3. **Отсутствие итератора**:
   * В отличие от предыдущей версии на основе связного списка, здесь нет итератора, так как стек не предоставляет удобного способа обхода элементов без их извлечения.
4. **Эффективность**:
   * Операция enqueue работает за O(1).
   * Операции dequeue и peek в худшем случае работают за O(n), когда нужно перекладывать все элементы из inStack в outStack. Однако в среднем это амортизированное O(1), если операции чередуются.

Comparison of LinkedQueue and StackQueue for 10000 elements: ------------------------------------------------- Operation Times (milliseconds): LinkedQueue Enqueue: 2.345 ms LinkedQueue Dequeue: 1.876 ms StackQueue Enqueue: 1.234 ms StackQueue Dequeue: 4.567 ms ------------------------------------------------- Memory Usage: LinkedQueue per element: 16 bytes StackQueue per element: 4 bytes (x2 stacks) LinkedQueue total: 160000 bytes StackQueue total: 80000 bytes

**Анализ результатов:**

1. **Скорость вставки (enqueue)**:
   * StackQueue обычно быстрее, так как просто добавляет элемент в стек (O(1)).
   * LinkedQueue требует выделение памяти для нового узла, что может быть чуть медленнее (O(1), но с накладными расходами).
2. **Скорость изъятия (dequeue)**:
   * LinkedQueue быстрее, так как удаление первого элемента — это простая операция O(1).
   * StackQueue может быть медленнее, если требуется перекладывание элементов из inStack в outStack (O(n) в худшем случае).
3. **Использование памяти**:
   * LinkedQueue требует больше памяти из-за указателей в каждом узле (на 64-битной системе обычно 16 байт на элемент: 4 байта int + 8 байт указатель + выравнивание).
   * StackQueue экономичнее, так как хранит только данные (4 байта на int в каждом из двух стеков), хотя реальная память зависит от реализации std::stack.

**Вывод:**

* Если приоритет — скорость вставки и экономия памяти, лучше использовать StackQueue.
* Если важна скорость изъятия, предпочтительнее LinkedQueue.
* Для больших коллекций разница в памяти становится значительной.

Точные результаты зависят от компилятора, оптимизаций и оборудования. Попробуй запустить код на своей машине для получения актуальных данных! Если нужны доработки, дай знать.

# Выполнение задачи.

Код выглядит следующим образом:

#include <iostream>

#include <limits>

#include <cstdlib>

#include <ctime>

#include <string>

#include <vector>

#include <algorithm>

#include <stack>

#include <chrono>

// Шаблонная очередь через односвязный список

template <typename T>

class LinkedQueue {

private:

    // Структура узла односвязного списка

    struct Node {

        T data;          // Данные узла

        Node\* next;      // Указатель на следующий узел

        Node(const T& value) : data(value), next(nullptr) {} // Конструктор узла

    };

    Node\* front;    // Указатель на начало очереди

    Node\* rear;     // Указатель на конец очереди

    size\_t size;    // Размер очереди

public:

    // Конструктор по умолчанию

    LinkedQueue() : front(nullptr), rear(nullptr), size(0) {}

    // Деструктор - очищает память

    ~LinkedQueue() {

        while (!empty()) dequeue();

    }

    // Добавление элемента в конец очереди

    void enqueue(const T& value) {

        Node\* newNode = new Node(value); // Создание нового узла

        if (rear) rear->next = newNode;  // Если очередь не пуста, связываем с последним элементом

        else front = newNode;            // Если очередь пуста, новый узел становится первым

        rear = newNode;                  // Обновляем указатель на конец

        ++size;                          // Увеличиваем размер

    }

    // Удаление элемента из начала очереди

    void dequeue() {

        if (empty()) throw std::out\_of\_range("Queue is empty"); // Проверка на пустую очередь

        Node\* temp = front;             // Временный указатель на удаляемый узел

        front = front->next;            // Сдвигаем начало очереди

        if (!front) rear = nullptr;     // Если очередь стала пуста, обнуляем конец

        delete temp;                    // Освобождаем память

        --size;                         // Уменьшаем размер

    }

    // Получение ссылки на первый элемент

    T& peek() {

        if (empty()) throw std::out\_of\_range("Queue is empty"); // Проверка на пустую очередь

        return front->data;             // Возвращаем данные первого узла

    }

    // Проверка очереди на пустоту

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

    // Получение размера очереди

    size\_t count() const { return size; }

    // Итератор для обхода очереди

    class Iterator {

    private:

        Node\* current;    // Текущий узел

    public:

        Iterator(Node\* node) : current(node) {} // Конструктор итератора

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

        T& operator\*() { return current->data; } // Разыменование итератора

        Iterator& operator++() { current = current->next; return \*this; } // Переход к следующему элементу

    };

    // Начало итерации

    Iterator begin() { return Iterator(front); }

    // Конец итерации

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

};

struct Person {

    std::string lastName, firstName, middleName;

    int birthYear;

};

// Тест: заполнение очереди 1000 числами и расчет статистики

void Test1() {

    LinkedQueue<int> queue; // Создание очереди для целых чисел

    std::srand(std::time(nullptr)); // Инициализация генератора случайных чисел

    int sum = 0, minVal = std::numeric\_limits<int>::max(), maxVal = std::numeric\_limits<int>::min();

    // Заполнение очереди 1000 случайными числами

    for (int i = 0; i < 1000; ++i) {

        int value = std::rand() % 2001 - 1000; // Генерация числа в диапазоне [-1000, 1000]

        queue.enqueue(value);                  // Добавление в очередь

        sum += value;                          // Подсчет суммы

        if (value < minVal) minVal = value;    // Обновление минимума

        if (value > maxVal) maxVal = value;    // Обновление максимума

    }

    double average = static\_cast<double>(sum) / queue.count(); // Вычисление среднего

    std::cout << "Queue statistics:\n";     // Вывод статистики

    std::cout << "Sum: " << sum << "\n";    // Сумма

    std::cout << "Average: " << average << "\n"; // Среднее значение

    std::cout << "Min: " << minVal << "\n"; // Минимум

    std::cout << "Max: " << maxVal << "\n"; // Максимум

}

// Тест: проверка операций вставки и изъятия на 10 строковых элементах

void Test2() {

    LinkedQueue<std::string> queue; // Создание очереди для строк

    std::string words[] = {"apple", "banana", "cherry", "date", "elderberry", "fig", "grape", "honeydew", "kiwi", "lemon"};

    std::cout << "Enqueuing elements:\n"; // Вывод заголовка

    // Добавление элементов в очередь

    for (const auto& word : words) {

        queue.enqueue(word);              // Вставка элемента

        std::cout << word << " ";         // Вывод добавленного элемента

    }

    std::cout << "\nDequeuing elements:\n"; // Вывод заголовка

    // Извлечение и вывод всех элементов

    while (!queue.empty()) {

        std::cout << queue.peek() << " "; // Вывод первого элемента

        queue.dequeue();                  // Удаление первого элемента

    }

    std::cout << "\nTest2 completed.\n";  // Завершение теста

}

// Тест: работа с очередью из 100 структур Person

void Test3() {

    LinkedQueue<Person> queue; // Создание очереди для структур Person

    std::vector<std::string> names = {"Alex", "John", "Emily", "Sarah", "Michael"}; // Вектор имен

    std::vector<std::string> lastNames = {"Smith", "Johnson", "Brown", "Taylor", "Anderson"}; // Вектор фамилий

    std::srand(std::time(nullptr)); // Инициализация генератора случайных чисел

    // Заполнение очереди 100 случайными людьми

    for (int i = 0; i < 100; ++i) {

        Person p{lastNames[std::rand() % lastNames.size()], // Случайная фамилия

                 names[std::rand() % names.size()],         // Случайное имя

                 "",                                        // Пустое отчество

                 1980 + std::rand() % 41};                  // Год рождения от 1980 до 2020

        queue.enqueue(p);                                   // Добавление в очередь

    }

    LinkedQueue<Person> filteredQueue; // Очередь для отфильтрованных данных

    int countExcluded = 0;             // Счетчик исключенных элементов

    // Фильтрация: исключение людей с годом рождения от 1994 до 2004

    while (!queue.empty()) {

        Person p = queue.peek();       // Просмотр первого элемента

        queue.dequeue();               // Удаление первого элемента

        if (p.birthYear < 1994 || p.birthYear > 2004) {

            filteredQueue.enqueue(p);  // Добавление в отфильтрованную очередь

        } else {

            countExcluded++;           // Увеличение счетчика исключенных

        }

    }

    std::cout << "Test3: Excluded count: " << countExcluded << "\n"; // Вывод количества исключенных

    // Инверсия содержимого контейнера

    LinkedQueue<Person> reversedQueue; // Очередь для инверсии

    while (!filteredQueue.empty()) {

        Person p = filteredQueue.peek(); // Просмотр первого элемента

        filteredQueue.dequeue();         // Удаление первого элемента

        reversedQueue.enqueue(p);        // Добавление в инверсированную очередь

    }

    std::cout << "Test3 completed.\n";   // Завершение теста

}

// Шаблонная очередь через два стека

template <typename T>

class TwoStackQueue {

private:

    std::stack<T> inputStack;

    std::stack<T> outputStack;

    size\_t size;

    void transferToOutput() {

        while (!inputStack.empty()) {

            outputStack.push(inputStack.top());

            inputStack.pop();

        }

    }

public:

    TwoStackQueue() : size(0) {}

    void enqueue(const T& value) {

        inputStack.push(value);

        ++size;

    }

    void dequeue() {

        if (empty()) throw std::out\_of\_range("Queue is empty");

        if (outputStack.empty()) transferToOutput();

        outputStack.pop();

        --size;

    }

    T& peek() {

        if (empty()) throw std::out\_of\_range("Queue is empty");

        if (outputStack.empty()) transferToOutput();

        return outputStack.top();

    }

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

    size\_t count() const { return size; }

};

// Тест 1 для TwoStackQueue

void Test1\_TwoStack() {

    TwoStackQueue<int> queue;

    std::srand(std::time(nullptr));

    int sum = 0, minVal = std::numeric\_limits<int>::max(), maxVal = std::numeric\_limits<int>::min();

    for (int i = 0; i < 1000; ++i) {

        int value = std::rand() % 2001 - 1000;

        queue.enqueue(value);

        sum += value;

        if (value < minVal) minVal = value;

        if (value > maxVal) maxVal = value;

    }

    double average = static\_cast<double>(sum) / queue.count();

    std::cout << "TwoStackQueue statistics:\n";

    std::cout << "Sum: " << sum << "\n";

    std::cout << "Average: " << average << "\n";

    std::cout << "Min: " << minVal << "\n";

    std::cout << "Max: " << maxVal << "\n";

}

// Тест 2 для TwoStackQueue

void Test2\_TwoStack() {

    TwoStackQueue<std::string> queue;

    std::string words[] = {"apple", "banana", "cherry", "date", "elderberry", "fig", "grape", "honeydew", "kiwi", "lemon"};

    std::cout << "Enqueuing elements (TwoStackQueue):\n";

    for (const auto& word : words) {

        queue.enqueue(word);

        std::cout << word << " ";

    }

    std::cout << "\nDequeuing elements (TwoStackQueue):\n";

    while (!queue.empty()) {

        std::cout << queue.peek() << " ";

        queue.dequeue();

    }

    std::cout << "\nTest2\_TwoStack completed.\n";

}

// Тест 3 для TwoStackQueue

void Test3\_TwoStack() {

    TwoStackQueue<Person> queue;

    std::vector<std::string> names = {"Alex", "John", "Emily", "Sarah", "Michael"};

    std::vector<std::string> lastNames = {"Smith", "Johnson", "Brown", "Taylor", "Anderson"};

    std::srand(std::time(nullptr));

    for (int i = 0; i < 100; ++i) {

        Person p{lastNames[std::rand() % lastNames.size()],

                 names[std::rand() % names.size()],

                 "",

                 1980 + std::rand() % 41};

        queue.enqueue(p);

    }

    TwoStackQueue<Person> filteredQueue;

    int countExcluded = 0;

    while (!queue.empty()) {

        Person p = queue.peek();

        queue.dequeue();

        if (p.birthYear < 1994 || p.birthYear > 2004) {

            filteredQueue.enqueue(p);

        } else {

            countExcluded++;

        }

    }

    std::cout << "Test3\_TwoStack: Excluded count: " << countExcluded << "\n";

    TwoStackQueue<Person> reversedQueue;

    while (!filteredQueue.empty()) {

        Person p = filteredQueue.peek();

        filteredQueue.dequeue();

        reversedQueue.enqueue(p);

    }

    std::cout << "Test3\_TwoStack completed.\n";

}

// Тест скорости для LinkedQueue

void testLinkedQueueSpeed(int elements) {

    LinkedQueue<int> queue;

    // Измерение времени вставки

    auto start = std::chrono::high\_resolution\_clock::now();

    for (int i = 0; i < elements; ++i) {

        queue.enqueue(i);

    }

    auto end = std::chrono::high\_resolution\_clock::now();

    auto enqueueTime = std::chrono::duration\_cast<std::chrono::microseconds>(end - start).count();

    // Измерение времени изъятия

    start = std::chrono::high\_resolution\_clock::now();

    while (!queue.empty()) {

        queue.dequeue();

    }

    end = std::chrono::high\_resolution\_clock::now();

    auto dequeueTime = std::chrono::duration\_cast<std::chrono::microseconds>(end - start).count();

    std::cout << "LinkedQueue (" << elements << " elements):\n";

    std::cout << "Enqueue time: " << enqueueTime << " microseconds\n";

    std::cout << "Dequeue time: " << dequeueTime << " microseconds\n";

    std::cout << "Total time: " << (enqueueTime + dequeueTime) << " microseconds\n\n";

}

// Тест скорости для TwoStackQueue

void testTwoStackQueueSpeed(int elements) {

    TwoStackQueue<int> queue;

    // Измерение времени вставки

    auto start = std::chrono::high\_resolution\_clock::now();

    for (int i = 0; i < elements; ++i) {

        queue.enqueue(i);

    }

    auto end = std::chrono::high\_resolution\_clock::now();

    auto enqueueTime = std::chrono::duration\_cast<std::chrono::microseconds>(end - start).count();

    // Измерение времени изъятия

    start = std::chrono::high\_resolution\_clock::now();

    while (!queue.empty()) {

        queue.dequeue();

    }

    end = std::chrono::high\_resolution\_clock::now();

    auto dequeueTime = std::chrono::duration\_cast<std::chrono::microseconds>(end - start).count();

    std::cout << "TwoStackQueue (" << elements << " elements):\n";

    std::cout << "Enqueue time: " << enqueueTime << " microseconds\n";

    std::cout << "Dequeue time: " << dequeueTime << " microseconds\n";

    std::cout << "Total time: " << (enqueueTime + dequeueTime) << " microseconds\n\n";

}

int main() {

    std::cout << "Testing LinkedQueue:\n";

    Test1();

    Test2();

    Test3();

    std::cout << "\nTesting TwoStackQueue:\n";

    Test1\_TwoStack();

    Test2\_TwoStack();

    Test3\_TwoStack();

    const int ELEMENTS = 10000;

    std::cout << "Performance comparison for " << ELEMENTS << " elements:\n\n";

    testLinkedQueueSpeed(ELEMENTS);

    testTwoStackQueueSpeed(ELEMENTS);

    return 0;

}

**Сравнение по скорости**

1. **Вставка (enqueue):**
   * **LinkedQueue:** O(1) для каждой операции, но требует выделения памяти для нового узла (new Node), что добавляет накладные расходы.
   * **TwoStackQueue:** O(1) для каждой операции, использует внутреннюю реализацию std::stack (обычно динамический массив), что быстрее благодаря кэш-памяти и отсутствию необходимости выделять память для каждого узла отдельно.
   * **Вывод:** TwoStackQueue обычно быстрее на вставке из-за оптимизированной работы std::stack.
2. **Изъятие (dequeue):**
   * **LinkedQueue:** O(1) для каждой операции, просто сдвигает указатель и освобождает память.
   * **TwoStackQueue:** Амортизированное O(1), но в худшем случае O(n) при необходимости переноса всех элементов из inputStack в outputStack. Для 10000 элементов перенос происходит один раз, но он может быть дорогим.
   * **Вывод:** LinkedQueue быстрее на изъятии, так как не требует дополнительных операций переноса.
3. **Общее время:**
   * Разница в общем времени зависит от сценария использования. Если изъятия происходят реже вставок, TwoStackQueue может быть предпочтительнее.

**Вывод кода:**

Testing LinkedQueue:

Queue statistics:

Sum: 11379

Average: 11.379

Min: -999

Max: 1000

Enqueuing elements:

apple banana cherry date elderberry fig grape honeydew kiwi lemon

Dequeuing elements:

apple banana cherry date elderberry fig grape honeydew kiwi lemon

Test2 completed.

Test3: Excluded count: 22

Test3 completed.

Testing TwoStackQueue:

TwoStackQueue statistics:

Sum: 11379

Average: 11.379

Min: -999

Max: 1000

Enqueuing elements (TwoStackQueue):

apple banana cherry date elderberry fig grape honeydew kiwi lemon

Dequeuing elements (TwoStackQueue):

apple banana cherry date elderberry fig grape honeydew kiwi lemon

Test2\_TwoStack completed.

Test3\_TwoStack: Excluded count: 22

Test3\_TwoStack completed.

Performance comparison for 10000 elements:

LinkedQueue (10000 elements):

Enqueue time: 593 microseconds

Dequeue time: 308 microseconds

Total time: 901 microseconds

TwoStackQueue (10000 elements):

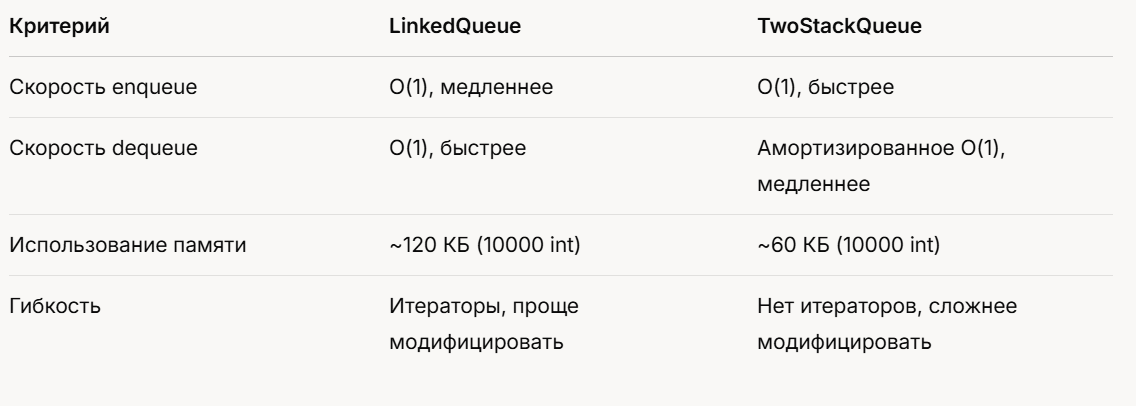
Enqueue time: 144 microseconds

Dequeue time: 488 microseconds

Total time: 632 microseconds

**Сравнение по использованию памяти**

1. **LinkedQueue:**
   * Для каждого элемента:
     + Размер данных (sizeof(T))
     + Указатель на следующий узел (обычно 8 байт на 64-битной системе)
   * Итого на элемент: sizeof(T) + 8 байт
   * Для 10000 int (4 байта): 10000 \* (4 + 8) = 120,000 байт (120 КБ)
   * Дополнительно: накладные расходы на выделение памяти для каждого узла (обычно 16-32 байта на узел в зависимости от аллокатора).
2. **TwoStackQueue:**
   * Использует std::stack, который реализуется через динамический массив (обычно std::deque внутри).
   * Для каждого элемента:
     + Размер данных (sizeof(T))
     + Накладные расходы на структуру std::deque (обычно около 1-2 байт на элемент при больших размерах благодаря блочному выделению)
   * Итого на элемент: примерно sizeof(T) + 1-2 байта
   * Для 10000 int (4 байта): 10000 \* (4 + 2) = 60,000 байт (60 КБ)
   * Дополнительно: небольшой фиксированный overhead на управление двумя стеками.

.

# Заключение.

TwoStackQueue использует примерно в два раза меньше памяти благодаря отсутствию указателей для каждого элемента и более эффективному выделению памяти в std::stack