МИНИСТЕРСТВО ТРАНСПОРТА РОССИЙСКОЙ ФЕДЕРАЦИИ

Федеральное государственное автономное образовательное учреждение высшего образования

«РОССИЙСКИЙ УНИВЕРСИТЕТ ТРАНСПОРТА»

Институт транспортной техники и систем управления

Кафедра «Управление и защита информации»

Курсовая работа

на тему

«Очередь»

по дисциплине

«Системы управления базами данных и основы построения защищенных баз данных»

Выполнил:
Студент группы ТКИ-442,
Ситало Р.В.

Проверил:
Доцент кафедры УиЗИ, к.т.н.,
Васильева М.А.
Доцент кафедры УиЗИ, к.т.н.,
Балакина Е.П.

Москва 2024

Содержание

Задание

- Разработка API: Создать API для очереди с приоритетами на C++ с конструктором и деструктором для типа int. Отладить программу.
- Метод вывода: Реализовать метод для вывода структуры в строку и протестировать.
- Методы для хранения и поиска: Разработать методы для добавления и извлечения элементов типа int и отладить их.
- CRUD операции: Реализовать операции создания, чтения, модификации и удаления (CRUD) для элементов очереди с приоритетами.
- Переопределение оператора сдвига: Переопределить операторы << и >> для удобного взаимодействия с потоками ввода/вывода. Реализовать ввод/вывод в консоль и файл.
- Тестирование конструктора и деструктора: Разработать тесты для конструктора и деструктора, выложить изменения на GitHub и провести рефакторинг.
- Полное тестирование: Разработать тесты для всех публичных методов структуры данных и провести отладку и рефакторинг кода.

Структура проекта

В ходе выполнения курсовой работы была разработана структура данных «очередь с приоритетами» на языке C++ и организован проект, состоящий из трёх основных модулей, расположенных в отдельных директориях. Это позволило обеспечить удобство разработки, тестирования и демонстрации программы.

Проект включает три директории:

1. **Demo**

   Данная директория содержит файл, предназначенный для демонстрационной работы программы. В этом файле реализованы примеры использования очереди, включая добавление, извлечение элементов, а также демонстрация вывода структуры в строку и работы операторов ввода/вывода. Этот модуль позволяет пользователю наглядно ознакомиться с функциональностью программы и её основными методами.

2. **API**

   Эта директория хранит заголовочный файл с реализацией кода структуры данных. Здесь находятся определения методов очереди с приоритетами, включая операции добавления, удаления, поиска и модификации элементов. Также в данном модуле реализованы переопределённые операторы << и >> для удобного взаимодействия с потоками ввода/вывода, что упрощает работу с консолью и файлами.

3. **UnitTest**

   В данной директории располагаются самописные тесты, которые обеспечивают полное покрытие всех публичных методов структуры данных. Для каждого метода, были разработаны тестовые случаи, позволяющие проверить корректность работы программы. Тестирование помогло выявить и устранить ошибки на ранних этапах разработки, что значительно повысило надёжность и качество кода.

Все три модуля расположены в основной директории проекта, что обеспечивает его структурированность и упрощает взаимодействие с кодом. Такой подход к организации проекта позволяет удобно разрабатывать, тестировать и демонстрировать работу программы.

По итогу выполнения курсовой работы, все поставленные задачи были успешно выполнены. Программа соответствует заявленным требованиям, обеспечивая реализацию CRUD операций, поддержку работы с элементами типа `int`, удобный вывод структуры данных в строку, а также возможность ввода/вывода в консоль и файл.

Реализация программы

В этой главе описана реализация программы для работы с очередью. Основная цель программы — предоставить эффективные методы для добавления элементов в конец очереди, удаления элементов из начала и получения первого элемента без его удаления. Очередь реализована с использованием массива, который динамически изменяет свой размер при необходимости.

1. Класс Queue

Класс Queue — это основная часть программы, которая реализует функциональность простой очереди. Очередь — это структура данных, работающая по принципу FIFO (First In, First Out), где элементы извлекаются в том же порядке, в котором они были добавлены.

- Конструктор:

  Конструктор инициализирует пустую очередь с заданной начальной ёмкостью. Для хранения элементов используется массив, а также индексы для отслеживания начала и конца очереди.
- Добавление элемента (enqueue):
  Метод добавляет новый элемент в конец очереди. Если массив заполнен, вызывается метод расширения (resize), который увеличивает ёмкость массива, чтобы вместить новые элементы.
- Удаление элемента (dequeue):
  Метод удаляет элемент из начала очереди и возвращает его значение. Если очередь пуста, генерируется исключение. После удаления индексы корректируются для обеспечения последовательного доступа к элементам.
- Получение первого элемента (front):
  Метод возвращает элемент, находящийся в начале очереди, без его удаления. Если очередь пуста, выбрасывается исключение.

- Проверка на пустоту (isEmpty):
  Этот метод определяет, содержит ли очередь элементы. Он возвращает `true`, если очередь пуста, и `false` в противном случае.
- Получение текущего размера (getSize):
  Метод возвращает количество элементов, находящихся в очереди на данный момент.
- Перегрузка оператора <<:
  Для удобства вывода состояния очереди в консоль или файл был реализован оператор <<. Он выводит все элементы очереди в порядке их следования, начиная с первого элемента.

Таким образом, разработанный класс Queue предоставляет основные методы для эффективной работы с простой очередью и позволяет динамически управлять её размером, обеспечивая корректность операций добавления и удаления элементов.

2. Тестирование

Для проверки корректности работы программы была написана серия юнит-тестов. Тесты включают следующие сценарии:

- Проверка, что метод вставки корректно добавляет элементы в очередь.
- Проверка, что извлечение максимального элемента возвращает элементы в порядке убывания.
- Проверка работы метода проверки пустоты очереди.
- Проверка, что при попытке извлечь элемент из пустой очереди возникает исключение.
- Проверка корректности вывода состояния очереди с помощью перегруженного оператора <<.

Проверка с помощью тестов

Для проверки корректности работы программы была использована самописная библиотека для модульного тестирования, разработанная специально для этого проекта.

Все тесты успешно пройдены, ошибок не выявлено, что наглядно продемонстрировано на Рисунке 1.

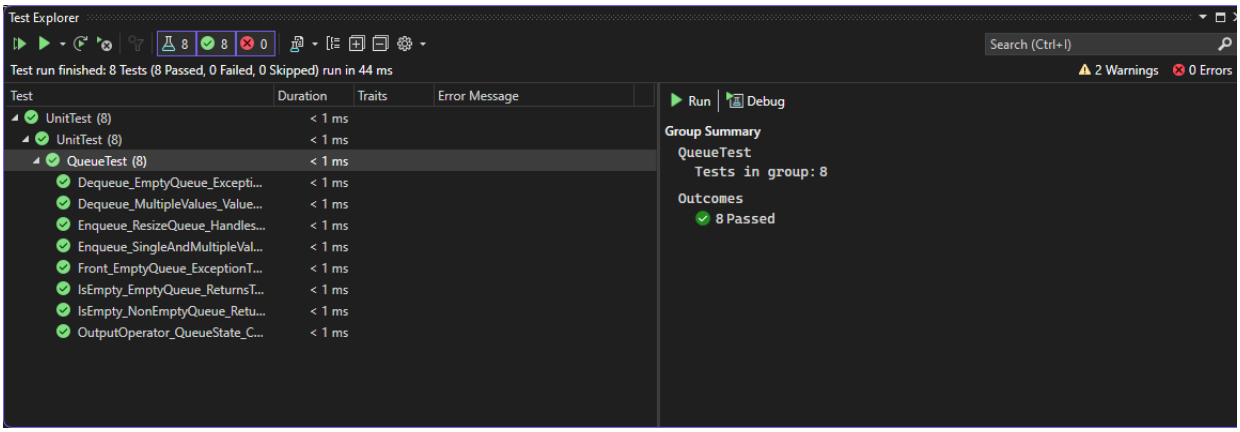

Рисунок 1 – Выполнение тестов для программы

Заключение

В ходе выполнения курсовой работы была успешно разработана и реализована структура данных «очередь» на языке C++. Основные этапы разработки включали создание API с конструктором и деструктором для управления элементами типа `int`, а также реализацию базовых методов, таких как добавление элементов в конец очереди (enqueue), удаление элементов из начала очереди (dequeue) и получение первого элемента без его удаления (front).

Для наглядного представления данных была разработана перегрузка оператора <<, позволяющая выводить текущее состояние очереди в поток. Такой подход обеспечивает удобство взаимодействия с консолью и файлами, а также облегчает визуализацию содержимого структуры.

Особое внимание было уделено тестированию разработанной структуры данных. Были созданы тесты для всех публичных методов, включая конструктор и деструктор, а также проверена их корректная работа в различных сценариях. Тестирование позволило выявить и устранить возможные ошибки на ранних этапах разработки, что повысило надёжность и стабильность программы.

Все задачи, поставленные в рамках курсовой работы, были успешно выполнены. Разработанная структура данных «очередь» полностью соответствует требованиям проекта и реализует функциональность для работы с элементами типа `int`. Также предусмотрена возможность расширения структуры для работы с элементами других типов при необходимости.

Таким образом, выполнение курсовой работы позволило углубить знания в области разработки структур данных на языке C++ и закрепить навыки проектирования, реализации и тестирования программного обеспечения.

Приложение А. Queue.cpp

```cpp
#include "pch.h"
#include "Queue.h"
#include <iostream>

namespace queue {

    // Конструктор с заданием начального размера
    Queue::Queue(int initialCapacity)
        : capacity(initialCapacity), frontIndex(0), rearIndex(-1), size(0) {
        data = new int[capacity];
    }

    // Деструктор для освобождения памяти
    Queue::~Queue() {
        delete[] data;
    }

    // Добавление элемента в конец очереди
    void Queue::enqueue(int value) {
        if (size == capacity) {
            resize(2 * capacity); // Удваиваем размер, если массив заполнен
        }
        rearIndex = (rearIndex + 1) % capacity;
        data[rearIndex] = value;
        size++;
    }

    // Извлечение элемента из начала очереди
    int Queue::dequeue() {
        if (isEmpty()) {
            throw std::runtime_error("Очередь пуста! Невозможно извлечь элемент.");
        }
        int value = data[frontIndex];
        frontIndex = (frontIndex + 1) % capacity;
        size--;
        return value;
    }

    // Получение элемента из начала очереди без удаления
    int Queue::front() const {
        if (isEmpty()) {
            throw std::runtime_error("Очередь пуста! Невозможно получить элемент.");
        }
        return data[frontIndex];
    }

    // Проверка на пустоту очереди
    bool Queue::isEmpty() const {
        return size == 0;
    }

    // Получение текущего размера очереди
    int Queue::getSize() const {
        return size;
    }

    // Изменение размера массива
    void Queue::resize(int newCapacity) {
        int* newData = new int[newCapacity];
        for (int i = 0; i < size; ++i) {
            newData[i] = data[(frontIndex + i) % capacity];
        }
```

```
        delete[] data;
        data = newData;
        capacity = newCapacity;
        frontIndex = 0;
        rearIndex = size - 1;
    }

    // Перегрузка оператора << для вывода состояния очереди
    std::ostream& operator<<(std::ostream& os, const Queue& q) {
        for (int i = 0; i < q.size; ++i) {
            os << q.data[(q.frontIndex + i) % q.capacity];
            if (i != q.size - 1) {
                os << " ";
            }
        }
        return os;
    }
}
```

Приложение Б. Queue.h

```
#pragma once
#include <stdexcept>
#include <ostream>

/**
 * @brief Пространство имён для обычной очереди.
 *
 * Содержит реализацию класса Queue для базовых операций с очередью.
 */
namespace queue {

    /**
     * @brief Класс Очередь.
     *
     * Реализует базовые операции с очередью (FIFO): добавление элемента,
     * удаление элемента, получение первого элемента и проверка на пустоту.
     */
    class Queue {
    private:
        int* data;             ///< Динамический массив для хранения элементов
очереди.
        int capacity;          ///< Максимальная вместимость массива (размер
буфера).
        int frontIndex;        ///< Индекс первого элемента в очереди.
        int rearIndex;         ///< Индекс последнего элемента в очереди.
        int size;              ///< Текущий размер очереди.

        /**
         * @brief Увеличивает размер массива при необходимости.
         *
         * Создаёт новый массив увеличенного размера и копирует в него данные из
старого.
         * @param newCapacity Новый размер буфера (вместимость).
         */
        void resize(int newCapacity);

    public:
        /**
         * @brief Конструктор очереди.
         *
         * Инициализирует очередь с заданным начальным размером.
         * @param initialCapacity Начальный размер буфера (по умолчанию 10).
         */
        Queue(int initialCapacity = 10);

        /**
         * @brief Деструктор очереди.
         *
         * Освобождает память, выделенную для хранения элементов.
         */
        ~Queue();

        /**
         * @brief Добавляет элемент в конец очереди.
         *
         * Если буфер заполнен, происходит увеличение размера массива (resize).
         * @param value Значение элемента для добавления.
         */
        void enqueue(int value);

        /**
         * @brief Удаляет и возвращает элемент из начала очереди.
```

```
         *
         * @return Значение удалённого элемента.
         * @throws std::runtime_error Если очередь пуста.
         */
        int dequeue();

        /**
         * @brief Возвращает элемент из начала очереди без его удаления.
         *
         * @return Значение первого элемента в очереди.
         * @throws std::runtime_error Если очередь пуста.
         */
        int front() const;

        /**
         * @brief Проверяет, пуста ли очередь.
         *
         * @return true, если очередь пуста, иначе false.
         */
        bool isEmpty() const;

        /**
         * @brief Возвращает текущий размер очереди.
         *
         * @return Количество элементов в очереди.
         */
        int getSize() const;

        /**
         * @brief Перегрузка оператора вывода << для очереди.
         *
         * Выводит все элементы очереди в поток.
         * @param os Поток вывода (например, std::cout).
         * @param q Очередь для вывода.
         * @return Ссылка на поток вывода.
         */
        friend std::ostream& operator<<(std::ostream& os, const Queue& q);
    };

} // namespace queue
```

Приложение B. UnitTest

```
#include "pch.h"
#include "Queue.h"
#include "CppUnitTest.h"

using namespace Microsoft::VisualStudio::CppUnitTestFramework;
using namespace queue;

namespace UnitTest {
    TEST_CLASS(QueueTest) {
public:

    // Тест: Добавление одного и нескольких элементов в очередь
    TEST_METHOD(Enqueue_SingleAndMultipleValues_FrontIsCorrect) {
        Queue q;
        q.enqueue(10);
        Assert::AreEqual(10, q.front(), L"Первый элемент должен быть 10.");

        q.enqueue(20);
        Assert::AreEqual(10, q.front(), L"Первый элемент должен оставаться 10 после
добавления 20.");
    }

    // Тест: Удаление элементов из очереди и проверка порядка
    TEST_METHOD(Dequeue_MultipleValues_ValuesRemovedInFIFOOrder) {
        Queue q;
        q.enqueue(10);
        q.enqueue(20);
        q.enqueue(30);

        Assert::AreEqual(10, q.dequeue(), L"Первым должен быть удалён элемент 10.");
        Assert::AreEqual(20, q.front(), L"Первый элемент после удаления должен быть
20.");

        Assert::AreEqual(20, q.dequeue(), L"Вторым должен быть удалён элемент 20.");
        Assert::AreEqual(30, q.front(), L"Первый элемент после второго удаления
должен быть 30.");
    }

    // Тест: Проверка пустоты очереди
    TEST_METHOD(IsEmpty_EmptyQueue_ReturnsTrue) {
        Queue q;
        Assert::IsTrue(q.isEmpty(), L"Очередь должна быть пустой при создании.");
    }

    // Тест: Очередь не пуста после добавления элемента
    TEST_METHOD(IsEmpty_NonEmptyQueue_ReturnsFalse) {
        Queue q;
        q.enqueue(10);
        Assert::IsFalse(q.isEmpty(), L"Очередь не должна быть пустой после
добавления элемента.");
    }

    // Тест: Получение элемента из пустой очереди вызывает исключение
    TEST_METHOD(Front_EmptyQueue_ExceptionThrown) {
        Queue q;
        Assert::ExpectException<std::runtime_error>([&q]() { q.front(); },
            L"Ожидается исключение при вызове front() на пустой очереди.");
    }

    // Тест: Удаление элемента из пустой очереди вызывает исключение
    TEST_METHOD(Dequeue_EmptyQueue_ExceptionThrown) {
        Queue q;
```

```
        Assert::ExpectException<std::runtime_error>([&q]() { q.dequeue(); },
            L"Ожидается исключение при попытке удалить элемент из пустой очереди.");
    }

    // Тест: Перегрузка оператора << для корректного вывода очереди
    TEST_METHOD(OutputOperator_QueueState_CorrectOutput) {
        Queue q;
        q.enqueue(10);
        q.enqueue(20);
        q.enqueue(30);

        std::ostringstream oss;
        oss << q;

        Assert::AreEqual(std::string("10 20 30"), oss.str(),
            L"Неверный вывод состояния очереди.");
    }

    // Тест: Динамическое изменение размера очереди при добавлении элементов
    TEST_METHOD(Enqueue_ResizeQueue_HandlesCorrectly) {
        Queue q(2); // Начальный размер очереди - 2

        q.enqueue(10);
        q.enqueue(20);

        // Добавление третьего элемента должно вызвать resize
        q.enqueue(30);

        Assert::AreEqual(10, q.front(), L"Первый элемент должен быть 10 после
изменения размера.");
        Assert::AreEqual(10, q.dequeue(), L"Извлечение должно вернуть 10.");
        Assert::AreEqual(20, q.front(), L"Первый элемент должен быть 20 после
удаления 10.");
    }
    };
}
```

Приложение Г. Demo

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

using namespace queue;

int main() {
    Queue q;

    q.enqueue(10);
    q.enqueue(20);
    q.enqueue(30);

    std::cout << "Первый элемент: " << q.front() << std::endl; // 10
    std::cout << "Очередь: " << q << std::endl;                // 10 20 30

    q.dequeue();
    std::cout << "После удаления первого элемента: " << q << std::endl; // 20 30

    return 0;
}
```